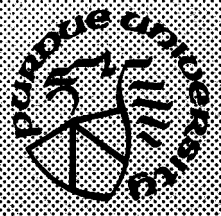
12-1-1984

A Study of a Semi-Direct Method for Computer Analysis of Large-Scale Circuits

Marwan M. Hassoun
Purdue University

P. M. Lin
Purdue University



A Study of a Semi-Direct Method for Computer Analysis of Large-Scale Circuits

M.M. Hassoun
P.M. Lin

TR-EE 84-46
December 1984

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

This work was supported by: International Business Machines Corporation

A Study of a Semi-Direct Method for Computer Analysis of Large-Scale Circuits

M.M. Hassoun
P.M. Lin

TR-EE 84-46
December 1984

This work was supported by: International Business Machines Corporation

ACKNOWLEDGEMENTS

We wish to thank the Computer Aided Circuit Design Department, at IBM East Fishkill Facility, for the financial support of the research which has resulted in a master's thesis for M. M. Hassoun. The thesis is hereby reproduced as a technical report for the third year effort of the research project.

TABLE OF CONTENTS

Page

ABSTRACT

Hassoun, Marwan M. M.S., Purdue University. December 1984. A STUDY OF A SEMI-DIRECT METHOD FOR COMPUTER ANALYSIS OF LARGE-SCALE CIRCUITS. Major Professor: P. M. Lin.

In this thesis a study of a Semi-Direct method for the solution of large scale circuits is presented. The method combines features from the Newton-Raphson method and the Gauss-Seidel method. These two methods are both illustrated. The Semi-Direct method is presented both theoretically and empirically using three programs developed for this purpose. The Semi-Direct method and the Newton-Raphson method are compared. The comparison includes speed (number of iterations and execution time) and storage requirements. The Semi-Direct method definitely has storage advantages over the Newton-Raphson method at all circuit levels of 2 or more nodes. If some set conditions are met in the circuit, the Semi-Direct method will require less CPU time to reach the solution than the Newton-Raphson method.

CHAPTER 1
INTRODUCTION

The size of circuits has been growing at a rapid rate in the past two decades. The new technologies and advances in integrated circuits has supported this growth. The task of analyzing such large circuits using computer analysis programs involves two basic considerations, speed and storage. The larger the circuit, the more variables: node voltages, and branch currents. Therefore, memory storage requirements are increased, and grow quadratically with the number of variables in the circuit. To solve a circuit with n variables we need n characteristic equations that describe the circuit. A lumped nonlinear circuit can be characterized in the time domain by a set of differential algebraic equations of the form

$$\mathbf{f}(\mathbf{x},\dot{\mathbf{x}},t) = 0 \tag{1.1}$$

where $\mathbf{x}$ is, as considered in this report, the vector of node voltages and branch currents. These variables are functions of time t (the case where x(t) = constant is possible). The $\dot{\mathbf{x}}$ vector is the derivative of $\mathbf{x}$ with respect to t.

The formulation of these equations from a large scale circuit involves the task of decomposing it into several smaller subcircuits. This is done because it is easier and less time consuming to solve these subcircuits individually and then interconnect the solutions. Other advantages are that similar subcircuits have to be solved only once and the latency and near-latency of the subcircuits

can be exploited [14]. Several tearing techniques have been proposed [9,17]. The node tearing technique [17] has become the most popular one used.

The solution of the equations eq. (1.1) can be reached using two approaches: direct and semi-direct. The latter methods are sometimes referred to as relaxation methods [15]. The solution of eq. (1.1) at successive time points would produce a wave for each variable with respect to time. The process is called transient analysis. A special case of eq. (1.1) would be when all variables are independent of time (all input sources are constant). For the majority of practical circuits, there is only one solution point, the process is called dc analysis. The variable t would not be involved in the process.

In the transient analysis case the variables in $\dot{\mathbf{x}}$ are replaced by an implicit and stiffly stable integration formulas. A widely used method is the backward differentiation formula (BDF) introduced in [2]. After substituting the BDF formula in eq. (1.1) for $\dot{\mathbf{x}}$ the solution at times $t_0, t_1, t_2 \cdots$ is to be obtained. The process involves setting $t = t_i$ and then using an iterative scheme to solve the resulting nonlinear algebraic equations. The solution at t_i is obtained and stored away or outputted. The values of $\mathbf{x}$, $\dot{\mathbf{x}}$ and t are then updated for t_{i+1} and the iterative scheme is repeated. The process at a certain time point resembles a dc analysis case. The collection of these dc-like solutions produce the waveform with respect to time. This approach is called "incremental in time". An early termination of the process does not affect the validity of the solutions at previous time steps. Another approach called "Global in time" or "waveform relaxation" [10] gradually updates the waveform over the whole time interval. The acceptable solution reached is for the entire time interval. Termination of the process before its completion does not validate the solution for any time point. Methods using the global in time approach have only

recently been used successfully with MOS digital circuits. Most simulation programs use the incremental in time approach for transient analysis.

In this thesis an incremental in time semi-direct method proposed by Odeh and Zein [15] is studied. It is referred to from this point on as the Semi-Direct method. The algorithm and its convergence properties are discussed in Chapter 4. In Chapter 5 a dc analysis comparison between the conventional Newton-Raphson method and the Semi-Direct method is studied. Chapter 6 includes a similar comparison for the transient analysis case where the advantages of the Semi-Direct method are clear. Five programs written in FORTRAN where developed for the purpose of this study, two for the dc analysis case, one implementing the Newton-Raphson method and the other implementing the Semi-Direct method, two more for the transient case and the last for the convergence properties of the Semi-Direct method. These programs are listed in the appendix. Chapters 1 and 2 are dedicated to the illustration of the basic building blocks of the Semi-Direct method, the Newton-Raphson method and the Gauss-Seidel method.

CHAPTER 2
THE NEWTON-RAPHSON METHOD

The Newton-Raphson method or simply the Newton method is one of the most well-known and widely used numerical methods to solve a set of (linear or nonlinear) n equations (where n = 1,2,3 ...). The case where n = 1 is the scalar case, that is, one equation in one unknown. The solution to such an equation can be obtained analytically if it was linear, quadratic, cubic or even quartic. A solution formula does exist for these cases, but by the time we get to the cubic case the equation becomes too long and cumbersome. We resort to an iterative scheme to get the solution, usually for equations of degree three or higher. The scalar case (n = 1) gives an intuitive way of how the Newton method works.

2.1 Scalar Equations

Any equation can be written in the form

$$f(x) = 0 \tag{2.1}$$

where the only unknown is x. Let r be a valid solution for equation (1.1), that is, r satisfies f(r) = 0. The Newton algorithm to find r consists of repeated solutions of the following iterative formula [5]:

$$x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)} \tag{2.2}$$

where i = 0,1,2, ... x^o is the initial guess and x^i is the value of x in the i^{th} iteration.

This formula can be derived from the Taylor polynomial expansion for f(x) about r (see [3]), or by a geometric interpretation of the iterations as shown below and illustrated in Figure 2.1.

$$\tan\theta = f'(x^1) = \frac{f(x^1)}{x^1 - x^2} \tag{2.3}$$

equation (2.3) can be written

$$x^2 = x^1 - \frac{f(x^1)}{f'(x^1)} \tag{2.4}$$

Sequential application of equation (2.4) can be produced in the form of equation (2.2).

For an initial guess x^o the Newton iteration can converge, diverge or oscillate between two points. This is illustrated in Figures (2.2), (2.3) and (2.4) respectively.

The termination of the Newton iterations is based on two criteria:

1) The absolute increment, and
2) The number of iterations

The absolute increment is defined as

$$e^i = \left| x^{i+1} - x^i \right| \tag{2.5}$$

where i = 0,1,2,... is the iteration number.

So for a user specified absolute increment criterion "e" the iterations can be terminated when $e^i < e$. A problem may arise though if the iterations are

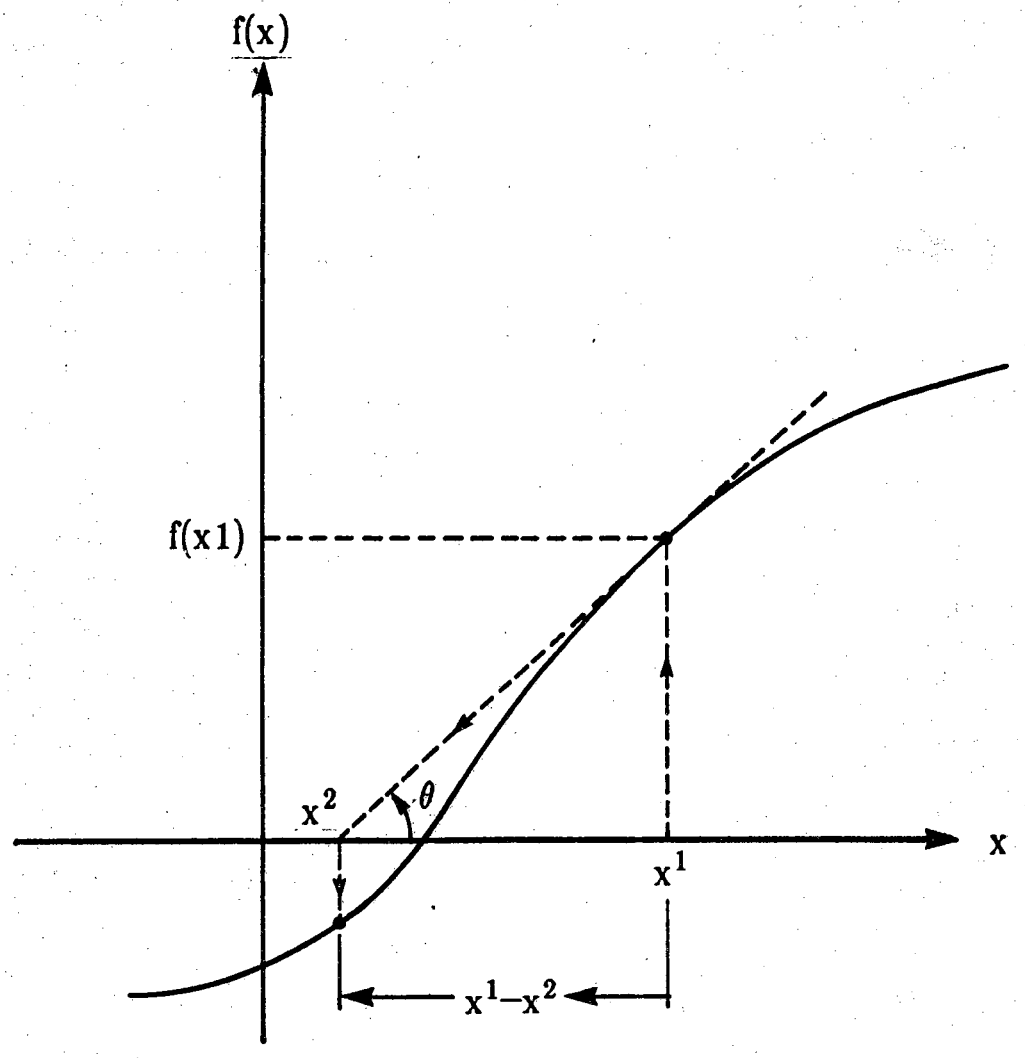


Figure 2.1: Geometric interpretation of scalar case.

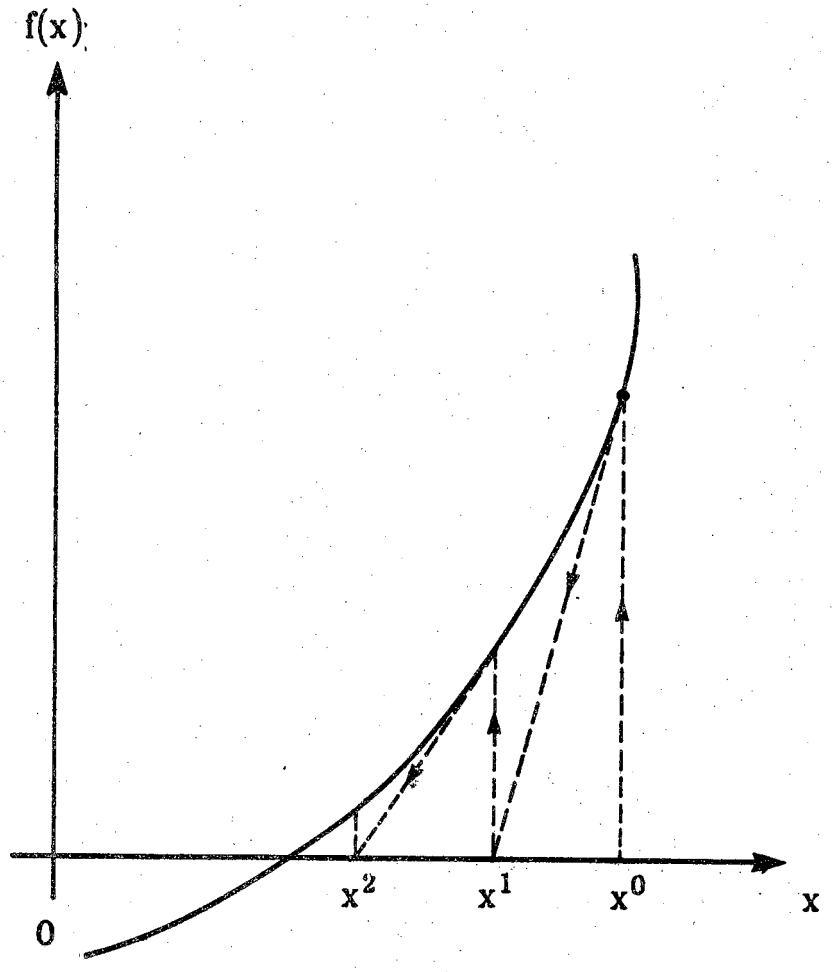


Figure 2.2: Convergent case.

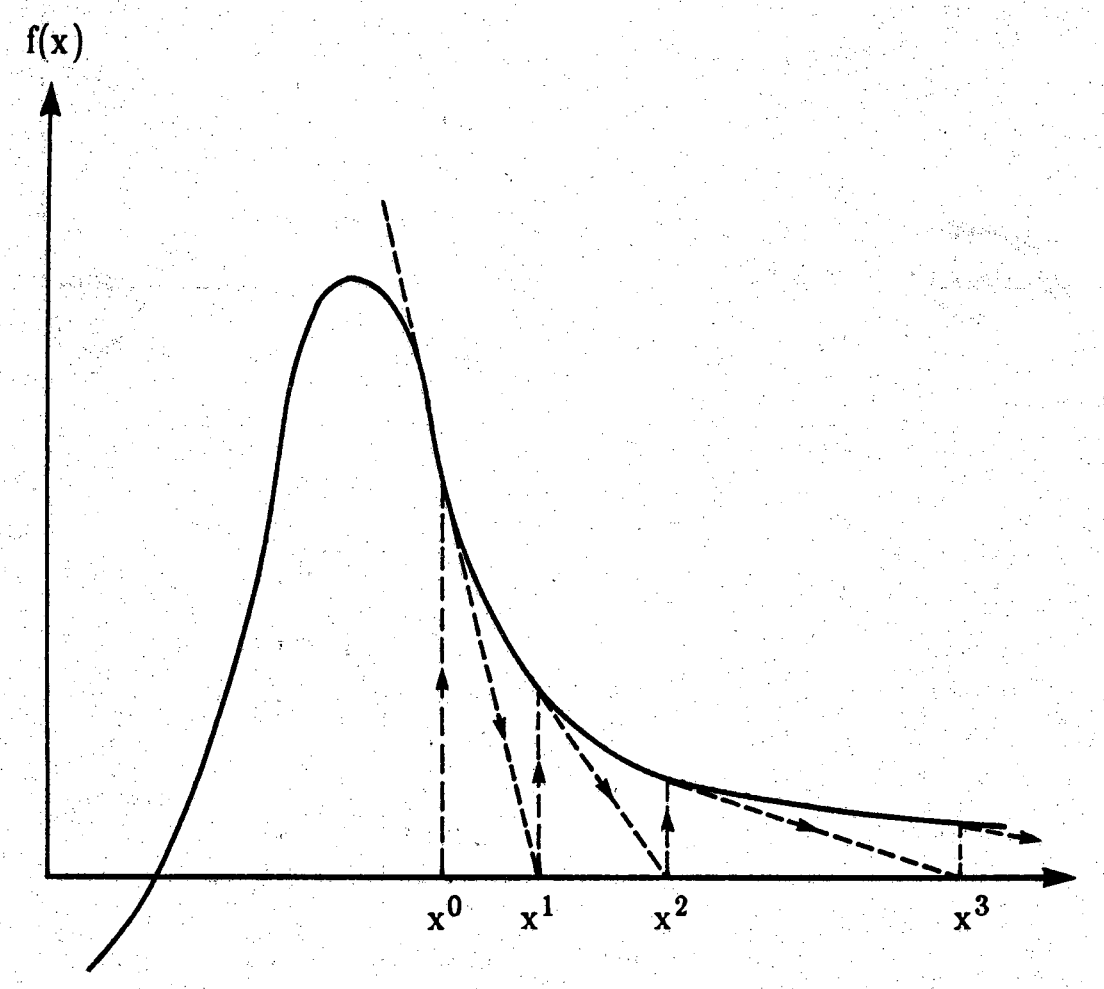


Figure 2.3: Divergent case.

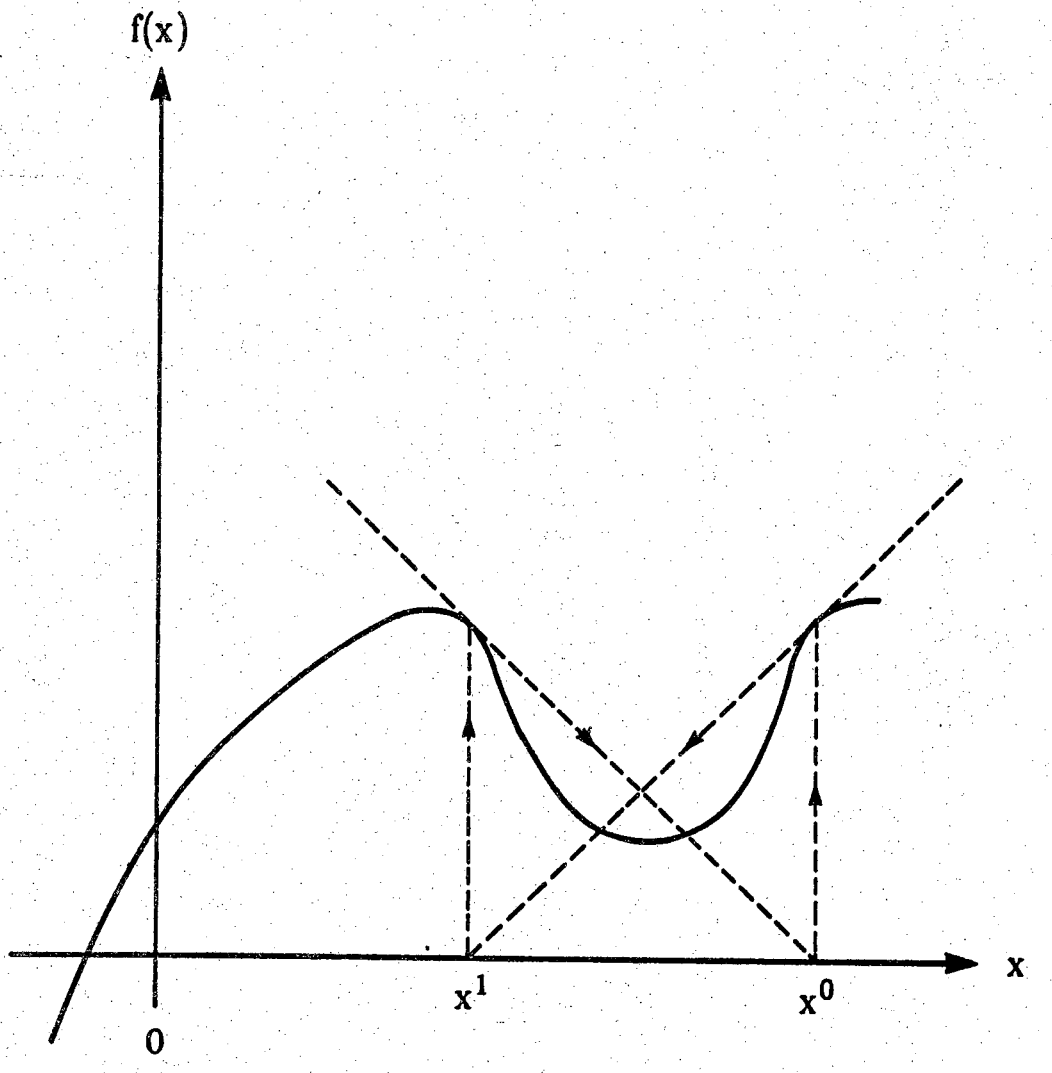


Figure 2.4: Oscillatory case.

diverging or oscillating, the absolute increment would be increasing or stay constant, respectively. In this case, the second criterion takes effect. A user specified maximum number of iterations would terminate the process. Since the Newton algorithm in general has a quadratic rate of convergence (see sec. 2.3), if the algorithm does not converge after a number of iterations that depends on the problem, the initial guess and the machine on which the algorithm is running, then the iterations are not converging to the solution or is converging but not fast enough.

2.2 Vector Equations

The Newton algorithm is also used to solve a set of n equations in n unknowns provided there is a solution to the system. An intuitive geometric interpretation of the method does not exist on this level ($n > 1$) like in the scalar case ($n = 1$). The solution scheme is obtained by the use of Linear Algebra techniques. Consider the equations in the form:

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ f_2(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, \ldots, x_n) &= 0 \end{aligned}$$

which in vector form can be written as

$$\mathbf{f}(\mathbf{x}) = 0 \tag{2.6}$$

The solution using Newton's algorithm is obtained by [11]:

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \mathbf{J}^{-1}(\mathbf{x}^i)\mathbf{f}(\mathbf{x}^i) \tag{2.7}$$

where $i = 1,2,\ldots$, $\mathbf{x}^o$ is the initial guess vector, and

$$\mathbf{J(x)} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \cdots \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \\ \vdots & \\ \dfrac{\partial f_n}{\partial x_1} & \cdots \dfrac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{2.8}$$

is the Jacobian matrix of the system of equations. When implementing this scheme on a digital computer, finding an inverse of a matrix is quite cumbersome. It takes time and a lot of storage. To get around this we manipulate equation (2.7) moving x^i to the left hand side and multiplying by $\mathbf{J(x)}$, assuming $\mathbf{J}(x)$ is nonsingular

$$\mathbf{J}(\mathbf{x}^i)\,[\mathbf{x}^{i+1} - \mathbf{x}^i] = -\mathbf{f}(\mathbf{x}^i) \tag{2.9}$$

The above equation cannot be derived geometrically as explained earlier, but the use of the Taylor polynomial expansion would lead to the same iterative equation [11]. Let us define Δx^i as:

$$\Delta \mathbf{x}^i = \mathbf{x}^{i+1} - \mathbf{x}^i \tag{2.10}$$

So equation (2.9) becomes

$$\mathbf{J}(\mathbf{x}^i)\,\Delta \mathbf{x}^i = -\mathbf{f}(\mathbf{x}^i) \tag{2.11}$$

This is in the form $\mathbf{Ay} = \mathbf{b}$ which can be solved at each iteration using several numerical methods like Gaussian-back substitution or LU factorization methods. The LU factorization method was used in implementing the programs for this thesis. The method can be illustrated as follows [4]:

$$\mathbf{J(x)} = \mathbf{LU} \tag{2.12}$$

and hence

$$\mathbf{L}(\mathbf{x}^i)\mathbf{U}(\mathbf{x}^i)\Delta\mathbf{x}^i = -\mathbf{f}(\mathbf{x}^i) \tag{2.13}$$

call

$$\mathbf{U}(\mathbf{x}^i)\Delta\mathbf{x}^i = \mathbf{y} \tag{2.14}$$

So we first solve

$$\mathbf{L}(\mathbf{x}^i)\mathbf{Y} = -\mathbf{f}(\mathbf{x}^i) \tag{2.15}$$

for $\mathbf{y}$, and then solve equation (2.14) for $\Delta\mathbf{x}^i$. The last step to compute x^{i+1} we get from equation (2.10)

$$x^{i+1} = x^i + \Delta x^i \tag{2.16}$$

The same stopping criterions are used as in the scalar case (sec. 2.2) but we have to define a different absolute increment. We use the 2nd norm of the vector $\Delta\mathbf{x}^i$, which is defined as [13]

$$||\Delta\mathbf{x}^i||_2 = ((\Delta x_1^i)^2 + (\Delta x_2^i)^2 + \cdots + (\Delta x_n^i)^2)^{1/2} \tag{2.17}$$

as the absolute increment and we check if $||\Delta x^i||_2 < e$, where e is the user specified absolute increment.

2.3 Convergence

One of the properties that makes the Newton method so widely used is that if the initial guess is sufficiently close to the true solution, then the algorithm converges and the rate of convergence is in general quadratic.

We show the quadratic convergence of the Newton algorithm using the Taylor series expansion for the case where $f'(r) \neq 0$. Let r be a solution for f(x) = 0 and let

$$x^{i+1} = g(x^i) \triangleq x^i - \frac{f(x^i)}{f'(x^i)} \tag{2.18}$$

so $g(r) = r$ and we can write

$$x^{i+1} - r = g(x^i) - g(r) \tag{2.19}$$

Now expanding $g(x^i)$ as a Taylor series in terms of $(x^i - r)$ with the second-derivative term as the remainder:

$$g(x^i) = g(r) + g'(r)(x^i - r) + \frac{g''(\epsilon)}{2}(x^i - r)^2 \tag{2.20}$$

where ϵ lies in the interval from x^i to r. Since

$$g'(r) = \frac{f(r)f''(r)}{[f'(r)]^2} = 0 \tag{2.21}$$

because $f(r) = 0$, we have

$$g(x^i) = g(r) + \frac{g''(\epsilon)}{2}(x^i - r)^2 \tag{2.22}$$

but $g(x^i) = x^{i+1}$ and $g(r) = r$ so by substituting back into equation (2.22) and letting $x^i - r = \delta^i$ we get, (assuming $g''(r) \neq 0$),

$$\delta^{i+1} = x^{i+1} - r = g(x^i) - g(r) = \frac{g''(\epsilon)}{2}(\delta^i)^2 \tag{2.23}$$

Let $k = \frac{g''(\epsilon)}{2}$,

$$\delta^{i+1} = k(\delta^i)^2 \tag{2.24}$$

This says that the error in the $i+1^{th}$ iteration is directly proportional to the square of the error at the i^{th} iteration. So when the iterations get close to the solution and δ^i is less than 1, $(\delta^i)^2$ becomes less than δ^i and the quadratic convergence prevails.

Example 2.1:

As an example consider a system of two nonlinear equations $f_1(x)$ and $f_2(x)$ in two unknowns x_1 and x_2.

$$f_1(x) = x_1^3 + 10\,x_1 + x_2 - 31$$

$$f_2(x) = 2(x_1 - x_2 - 1)^3 - x_2 + 19$$

the solution to this system is

$$x_1 = 2,\ x_2 = 3$$

Using the Newton algorithm we first find the Jacobian matrix of the system

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \end{bmatrix}$$

where

$$\frac{\partial f_1}{\partial x_1} = 3x_1^2 + 10$$

$$\frac{\partial f_1}{\partial x_1} = 1$$

$$\frac{\partial f_2}{\partial x_1} = 6(x_1 - x_2 - 1)^2$$

$$\frac{\partial f_2}{\partial x_2} = -6(x_1 - x_2 - 1)^2 - 1$$

Solving by the use of equations (2.11), (2.16) and an initial guess $x_1^o = x_2^o = 1$ produces the following results: where i is the iteration number and deltax is the increment (delta x $= x^{i+1} - x^i$).

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 120619e+01	0. 100000e+01	0. 331959e+01
0. 220619e+01	-0. 209006e+00	0. 431959e+01	-0. 977583e+00
0. 199718e+01	0. 782370e-03	0. 334200e+01	-0. 297179e+00
0. 199796e+01	0. 199497e-02	0. 304483e+01	-0. 438442e-01
0. 199996e+01	0. 435395e-04	0. 300098e+01	-0. 981709e-03
0. 200000e+01	0. 214871e-07	0. 300000e+01	-0. 484090e-06
0. 200000e+01	0. 520838e-14	0. 300000e+01	-0. 117693e-12

To see the quadratic convergence rate we look at the delta x1 and delta x2 exponents starting at iteration 3. The exponents almost double after each iteration which is a quadratic relationship. We say almost because of the effect of the constant k in equation (2.24) and the fact that deltax $= (x^{i+1}-x^i)$ not $(x^{i+1}-r)$.

CHAPTER 3
THE GAUSS-SEIDEL METHOD

The Gauss-Seidel method is an iterative method designed specifically for solving a set of n linear equations, where (n=2,3,4...) [17]. The case where n=1, that is the scalar case, we have one linear equation in one unknown in the form

$$ax = b \tag{3.1}$$

The solution is simply

$$x = b/a \tag{3.2}$$

and no iterative scheme is needed.

3.1 The Algorithm

Let us consider the set of n linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad \vdots \qquad\quad \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{3.3}$$

which can be written in matrix form as

$$\mathbf{Ax} = \mathbf{B} \tag{3.4}$$

The system will have a unique solution if and only if det $\mathbf{A} \neq 0$. That matrix $\mathbf{A}$ is called the coefficient matrix and its determinant, the coefficient determinant. There exist many iterative techniques to solve such a system of equations [3], but the Gauss-Seidel method is specifically useful if the coefficient matrix has many zero terms.

It is essential to assume that the diagonal elements of $\mathbf{A}$ be non-zero for reasons we will see in the following. Let us solve the j^{th} equation of (3.3) for x_j.

$$\begin{aligned}
x_1 &= \frac{1}{a_{11}} (b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\
x_2 &= \frac{1}{a_{22}} (b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\
&\vdots \\
x_n &= \frac{1}{a_{nn}} (b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})
\end{aligned} \tag{3.5}$$

The necessity for the diagonal of $\mathbf{A}$ be non-zero is now obvious. The terms $\frac{1}{a_{11}}, \frac{1}{a_{22}}, \cdots \frac{1}{a_{nn}}$ have to exist in order for the scheme to work. In the case where one or more of the diagonal elements is zero, a rearrangement of the order of the equations is possible in order to satisfy this condition.

The iteration process begins by a choice of an initial guess vector $\mathbf{x}^0 = (x_1^0, x_2^0, \ldots, x_n^0)^T$. The first equation is evaluated using this initial guess and a new value for x_1 is obtained, x_1^1. Now the second equation is evaluated using $\mathbf{x}^0$ but with the x_1^0 replaced by the updated value x_1^1. Therefore the choice of x_1^0 has no effect or the iteration on the evaluation processes because it is actually never used. Now we have two updated values x_1^1 and x_2^1 resulting from the evaluations of the first two equations of (3.5). These updated values

and the rest of $\mathbf{x}^0$ are used to evaluate x_3^1 from the third equation, and the process continues. So when we reach the n^{th} equation the $\mathbf{x}^0$ vector would be $\mathbf{x}^o = (x_1^1, x_2^1, x_3^1, \ldots, x_n^0)^T$. x_n^0 is then evaluated and we have a new value for $\mathbf{x}$ called $\mathbf{x}^1 = (x_1^1, x_2^1, \ldots, x_n^1)^T$, and the first iteration is completed. The process is then repeated starting with the first equation again to obtain $\mathbf{x}^2, \mathbf{x}^3, \ldots$. The termination of the process depends on two criteria:

1) The number of iteration equals or exceeds a user specified maximum number of iterations.

2) The second norm of $\mathbf{x}^k - \mathbf{x}^{k-1}$ is less than a user specified increment e [13]. That is

$$||\mathbf{x}^{i+1} - \mathbf{x}^i|| \triangleq ((x_1^{i+1} - x_1^i)^2 + (x_2^{i+1} - x_2^i)^2 + \ldots + (x_n^{i+1} - x_n^i)^2)^{1/2} \qquad (3.6)$$

less than e. Termination of the process under the first criterion means that the iterations are either diverging or converging at a very slow rate. Termination under the second criterion implies that the iterations are close enough to the true solution, that is, they are converging. The rate of convergence of the Gauss-Seidel method is linear. This is illustrated in the next section.

3.2 Convergence

The greatest advantage of the Newton-Raphson method is its unconditional convergence if the initial guess is chosen close enough to the true solution. This does not apply to the Gauss-Seidel method. The method is not always convergent. It will converge if, in the coefficient matrix, each term on the main diagonal is larger (in absolute value) than the sum of the absolute values of all the other terms in the same row [19]. That is,

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \quad \text{for all } i=1,2,\ldots,n \tag{3.7}$$

will guarantee convergence of the iterations to the true solution. A matrix that satisfies equation (3.7) is said to have a dominant diagonal. So by rearranging the equations we can try to create a diagonal dominant coefficient matrix.

Example 3.1:

Consider the equations

$$x_1 + 3x_2 = 9$$
$$2x_1 + x_2 = 8$$

The solution to the system is known

$$x_1 = 3, \quad x_2 = 2$$

The coefficient matrix is $\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}$, obviously the diagonal elements do not satisfy equation (3.7). That is the diagonal is not dominant. Performing the iteration with $\mathbf{x}^0 = (1,1)$

$$x_1 = 9 - 3x_2$$
$$x_2 = 8 - 2x_1$$

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 500000e+01	0. 100000e+01	-0. 500000e+01
0. 600000e+01	0. 150000e+02	-0. 400000e+01	-0. 300000e+02
0. 210000e+02	0. 900000e+02	-0. 340000e+02	-0. 180000e+03
0. 111000e+03	0. 540000e+03	-0. 214000e+03	-0. 108000e+04
0. 651000e+03	0. 324000e+04	-0. 129400e+04	-0. 648000e+04
0. 389100e+04	0. 194400e+05	-0. 777400e+04	-0. 388800e+05
0. 233310e+05	0. 116640e+06	-0. 466540e+05	-0. 233280e+06
0. 139971e+06	0. 699840e+06	-0. 279934e+06	-0. 139968e+07
0. 839811e+06	0. 419904e+07	-0. 167961e+07	-0. 839808e+07
0. 503885e+07	0. 251942e+08	-0. 100777e+08	-0. 503885e+08
0. 302331e+08	0. 151165e+09	-0. 604662e+08	-0. 302331e+09
0. 181399e+09	0. 906993e+09	-0. 362797e+09	-0. 181399e+10
0. 108839e+10	0. 544196e+10	-0. 217678e+10	-0. 108839e+11
0. 653035e+10	0. 326517e+11	-0. 130607e+11	-0. 653035e+11
0. 391821e+11	0. 195910e+12	-0. 783642e+11	-0. 391821e+12
0. 235092e+12	0. 117546e+13	-0. 470185e+12	-0. 235092e+13
0. 141055e+13	0. 705277e+13	-0. 282111e+13	-0. 141055e+14
0. 846333e+13	0. 423166e+14	-0. 169267e+14	-0. 846333e+14
0. 507800e+14	0. 253900e+15	-0. 101560e+15	-0. 507800e+15
0. 304680e+15	0. 152340e+16	-0. 609360e+15	-0. 304680e+16
0. 182808e+16	0. 914040e+16	-0. 365616e+16	-0. 182808e+17
0. 109685e+17	0. 548424e+17	-0. 219370e+17	-0. 109685e+18
0. 658109e+17	0. 329054e+18	-0. 131622e+18	-0. 658109e+18
0. 394865e+18	0. 197433e+19	-0. 789730e+18	-0. 394865e+19

Clearly the iterations are diverging. If we rearrange the equations as follows:

$$2x_1 + x_2 = 8$$
$$x_1 + 3x_2 = 9$$

so that $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$ has a dominant diagonal, we have

$$x_1 = \frac{8 - x_2}{2}$$
$$x_2 = \frac{9 - x_1}{3}$$

Use $\mathbf{x}^o = (1,1)$ we get

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 250000e+01	0. 100000e+01	0. 833333e+00
0. 350000e+01	-0. 416667e+00	0. 183333e+01	0. 138889e+00
0. 308333e+01	-0. 694444e-01	0. 197222e+01	0. 231481e-01
0. 301389e+01	-0. 115741e-01	0. 199537e+01	0. 385802e-02
0. 300231e+01	-0. 192901e-02	0. 199923e+01	0. 643004e-03
0. 300039e+01	-0. 321502e-03	0. 199987e+01	0. 107167e-03
0. 300006e+01	-0. 535837e-04	0. 199998e+01	0. 178612e-04
0. 300001e+01	-0. 893061e-05	0. 200000e+01	0. 297687e-05
0. 300000e+01	-0. 148844e-05	0. 200000e+01	0. 496145e-06
0. 300000e+01	-0. 248073e-06	0. 200000e+01	0. 826909e-07
0. 300000e+01	-0. 413454e-07	0. 200000e+01	0. 137818e-07
0. 300000e+01	-0. 689090e-08	0. 200000e+01	0. 229697e-08

The iterations converge to the true solution on 11th iteration (accurate to 8 digits that is e = 10^{-9}).

The condition of diagonal dominance is a sufficient but not a necessary one. A set of equations with a coefficient matrix **A** not satisfying equation (3.7) could converge. Example 3.2 illustrates that.

Example 3.2:

Consider the equations with solution $x_1 = 2$ and $x_2 = 3$

$$8x_1 + 3x_2 = 25$$
$$7x_1 + 7x_2 = 26$$

The coefficient matrix $\mathbf{A} = \begin{bmatrix} 8 & 3 \\ 7 & 4 \end{bmatrix}$ does not have a dominant diagonal since $a_{22} < a_{21}$. Solving the system using the Gauss-Seidel method with initial guess $\mathbf{x}^0 = (1,1)$ produces

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 175000e+01	0. 100000e+01	0. 687500e+00
0. 275000e+01	-0. 257813e+00	0. 168750e+01	0. 451172e+00
0. 249219e+01	-0. 169189e+00	0. 213867e+01	0. 296082e+00
0. 232300e+01	-0. 111031e+00	0. 243475e+01	0. 194304e+00
0. 221197e+01	-0. 728638e-01	0. 262906e+01	0. 127512e+00
0. 213910e+01	-0. 478169e-01	0. 275657e+01	0. 836795e-01
0. 209129e+01	-0. 313798e-01	0. 284025e+01	0. 549147e-01
0. 205991e+01	-0. 205930e-01	0. 289516e+01	0. 360378e-01
0. 203931e+01	-0. 135142e-01	0. 293120e+01	0. 236498e-01
0. 202580e+01	-0. 886867e-02	0. 295485e+01	0. 155202e-01
0. 201693e+01	-0. 582006e-02	0. 297037e+01	0. 101851e-01
0. 201111e+01	-0. 381942e-02	0. 298056e+01	0. 668398e-02
0. 200729e+01	-0. 250649e-02	0. 298724e+01	0. 438636e-02
0. 200479e+01	-0. 164489e-02	0. 299163e+01	0. 287855e-02
0. 200314e+01	-0. 107946e-02	0. 299450e+01	0. 188905e-02
0. 200206e+01	-0. 708393e-03	0. 299639e+01	0. 123969e-02
0. 200135e+01	-0. 464883e-03	0. 299763e+01	0. 813545e-03
0. 200089e+01	-0. 305080e-03	0. 299845e+01	0. 533889e-03
0. 200058e+01	-0. 200208e-03	0. 299898e+01	0. 350365e-03
0. 200038e+01	-0. 131387e-03	0. 299933e+01	0. 229927e-03
0. 200025e+01	-0. 862226e-04	0. 299956e+01	0. 150890e-03
0. 200016e+01	-0. 565836e-04	0. 299971e+01	0. 990212e-04
0. 200011e+01	-0. 371330e-04	0. 299981e+01	0. 649827e-04
0. 200007e+01	-0. 243685e-04	0. 299988e+01	0. 426449e-04
0. 200005e+01	-0. 159918e-04	0. 299992e+01	0. 279857e-04
0. 200003e+01	-0. 104946e-04	0. 299995e+01	0. 183656e-04
0. 200002e+01	-0. 688711e-05	0. 299996e+01	0. 120524e-04
0. 200001e+01	-0. 451966e-05	0. 299998e+01	0. 790941e-05
0. 200001e+01	-0. 296603e-05	0. 299998e+01	0. 519055e-05
0. 200001e+01	-0. 194646e-05	0. 299999e+01	0. 340630e-05
0. 200000e+01	-0. 127736e-05	0. 299999e+01	0. 223538e-05
0. 200000e+01	-0. 838269e-06	0. 300000e+01	0. 146697e-05
0. 200000e+01	-0. 550114e-06	0. 300000e+01	0. 962700e-06
0. 200000e+01	-0. 361012e-06	0. 300000e+01	0. 631772e-06
0. 200000e+01	-0. 236914e-06	0. 300000e+01	0. 414600e-06
0. 200000e+01	-0. 155475e-06	0. 300000e+01	0. 272081e-06
0. 200000e+01	-0. 102031e-06	0. 300000e+01	0. 178553e-06
0. 200000e+01	-0. 669575e-07	0. 300000e+01	0. 117176e-06
0. 200000e+01	-0. 439409e-07	0. 300000e+01	0. 768965e-07
0. 200000e+01	-0. 288362e-07	0. 300000e+01	0. 504633e-07
0. 200000e+01	-0. 189238e-07	0. 300000e+01	0. 331166e-07
0. 200000e+01	-0. 124187e-07	0. 300000e+01	0. 217327e-07
0. 200000e+01	-0. 814978e-08	0. 300000e+01	0. 142621e-07
0. 200000e+01	-0. 534829e-08	0. 300000e+01	0. 935951e-08
0. 200000e+01	-0. 350982e-08	0. 300000e+01	0. 614218e-08

$$x_1 = \frac{25-3x_2}{8}$$

$$x_2 = \frac{26-7x_1}{4}$$

The iterations converge to the time solution although **A** does not have a dominant diagonal.

To find a more strict convergence condition than that of equation (3.7), that is a sufficient and necessary one, we partition the coefficient matrix **A** of equation (3.4) into two matrices **P** and **Q** so that

$$\mathbf{A} = \mathbf{P} + \mathbf{Q} \tag{3.8}$$

P consists of the diagonal elements of **A** in addition to the lower triangular elements. **Q** consists of the upper triangular elements of **A**. The Gauss-Seidel method for solving equation (3.4) may be described by the following difference equation

$$\mathbf{P}\mathbf{x}^{i+1} = -\mathbf{Q}\mathbf{x}^{i} + \mathbf{B} \tag{3.9}$$

where $i = 0,1,2,...$ and $\mathbf{x}^0$ is the initial guess vector.

Example 3.3:

Consider the following system of equations

$$\begin{bmatrix} 4 & 2 & 1 \\ -1 & 6 & 2 \\ 3 & -2 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18 \\ 24 \\ 32 \end{bmatrix}$$

the matrices **P** and **Q** are

$$\mathbf{P} = \begin{bmatrix} 4 & 0 & 0 \\ -1 & 6 & 0 \\ 3 & -2 & 8 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Substituting in equation (3.9) we get

$$\begin{bmatrix} 4 & 0 & 0 \\ -1 & 6 & 0 \\ 3 & -2 & 8 \end{bmatrix} \mathbf{x}^{i+1} = -\begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{x}^{i} + \begin{bmatrix} 18 \\ 24 \\ 32 \end{bmatrix}$$

which represents the iterations of the Gauss-Seidel method.

The matrix **P** is nonsingular because its diagonal elements which are the diagonal elements of the coefficient matrix **A** must not contain a zero (see section 3.1 and equations (3.5)). So equation (3.9) can be written in the form

$$\mathbf{x}^{i+1} = -\mathbf{P}^{-1}\mathbf{Q}\ \mathbf{x}^{i} + \mathbf{P}^{-1}\mathbf{B} \tag{3.10}$$

The Gauss-Seidel method described by equation (3.10) will converge to the solution of $\mathbf{Ax} = \mathbf{B}$ if an only if all eigenvalues of the matrix $\mathbf{P}^{-1}\mathbf{Q}$ have magnitude less than unity [20].

For a given matrix **C** a way to determine whether all its eigenvalues have magnitudes less than unity is by the use of Gersgorin circle theorem [20]. The theorem states:

> The eigenvalues of a matrix **C** lie in the union of the circles 1 to n, where each circle j is centered at c_{jj} with a radius $= \sum_{m \neq j} |c_{jm}|$.

So the Gauss-Seidel method will converge for equation (3.4) if the union of the Gersgorin circles for the matrix $\mathbf{P}^{-1}\mathbf{Q}$ lies within the unit circle.

The convergence rate of equation (3.10) (the Gauss-Seidel method) can be determined using the eigenvalue with largest magnitude, λ_m, of the matrix $\mathbf{P}^{-1}\mathbf{Q}$ (see equation (3.8) and (3.4)). Defining the error e^i at iteration i as follows

$$e_j^i = (x_j^i - r_j) \tag{3.11}$$

where x_j^i is the value of the j^{th} variable at iteration i, and r_j is the true value of x_j. The convergence rate of the Gauss-Seidel method is linear and can be given by [3]

$$|e_j^{i+1}| = |\lambda_m| \cdot |e_j^i| \tag{3.12}$$

The case where $|\lambda_m| \geq 1 \Rightarrow |e_j^{i+1}| \geq |e_j^i|$ and the iterations diverge. The case where $|\lambda_m| < 1 \Rightarrow |e_j^{i+1}| < |e_j^i|$ and the iterations converge.

Example 3.4:

For the first case of example 3.1 the iterations diverged. To confirm that we compute the eigenvalues of $\mathbf{P}^{-1}\mathbf{Q}$.

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} = \mathbf{P} + \mathbf{Q}$$

$$\mathbf{P}^{-1}\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 3 \\ 0 & -6 \end{bmatrix}$$

$\mathbf{P}^{-1}\mathbf{Q}$ has only one eigenvalue $\lambda = -6$ which has magnitude greater than unity confirming the divergence of the iterations.

For the second case of example 3.1 where the iterations converged we had

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \mathbf{P} + \mathbf{Q}$$

$$\mathbf{P}^{-1}\mathbf{Q} = \begin{bmatrix} 0.5 & 0 \\ 0.1667 & 0.3333 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0.5 \\ 0 & 0.1667 \end{bmatrix}$$

$\mathbf{P}^{-1}\mathbf{Q}$ has one eigenvalue $\lambda = 0.1667$ which has magnitude less than unity confirming the convergence of the iterations.

To show the linear rate of convergence, equation (3.12), we look at the results of the iterations of example 3.1, the convergent case. At iterations 9 and 10 we have

$$\text{delta } x_1^9 = -0.248073 \times 10^{-6}$$
$$\text{delta } x_1^{10} = -0.413454 \times 10^{-7}$$
$$\text{delta } x_2^9 = 0.826909 \times 10^{-7}$$
$$\text{delta } x_2^{10} = 0.137818 \times 10^{-7}$$

where delta $x_j^i = x_j^{i+1} - x_j^i$ applying equation (3.12) but using delta x_j^i instead of e_j^i we get

$$\frac{\text{delta } x_1^{10}}{\text{delta } x_1^9} = \frac{-0.413454 \times 10^{-7}}{-0.24807 \times 10^{-6}} = 0.1667 = \lambda$$

$$\text{and} \quad \frac{\text{delta } x_2^{10}}{\text{delta } x_2^9} = \frac{0.137818 \times 10^{-7}}{0.826909 \times 10^{-7}} = 0.1667 = \lambda$$

As another case consider example 3.2. The maximum eigenvalue of $\mathbf{P}^{-1}\mathbf{Q}$ is $\lambda_m = 0.6563$, which implies a slower convergence rate than the above case. That is why it took 44 iterations to get within $\sim 10^{-8}$ of the true solution. We check this out

$$\frac{\text{delta } x_1^{42}}{\text{delta } x_1^{41}} = \frac{-0.814978\times10^{-8}}{-0.124187\times10^{-7}} = 0.6563 = \lambda_m$$

$$\frac{\text{delta } x_2^{42}}{\text{delta } x_2^{41}} = \frac{0.142621\times10^{-7}}{0.217327\times10^{-7}} = 0.6563 = \lambda_m$$

To determine the convergence condition of a system of equations in practical applications the diagonal dominance condition is much easier to evaluate then the eigenvalues magnitudes condition. The calculation of the eigenvalues of a matrix requires a fair amount of computer execution time and storage. The diagonal dominance condition can only confirm if the Gauss-Seidel method will converge but cannot determine if a system will diverge. The eigenvalues condition can state precisely if a system will converge or diverge.

CHAPTER 4
TEARING AND EQUATION FORMULATION

In any circuit analysis program the fundamental process is to solve a set of nonlinear algebraic equations. The way in which a circuit is translated into a set of equations has a significant affect on several aspects of the solution like the time it takes to reach it and the amount of storage required. Solving large-scale circuits starts with some decomposition technique to break the circuit into several smaller subcircuits. The purpose is to obtain a Jacobian matrix, referred to from now on as a "dependency matrix" [6] of a bordered block-diagonal form as in Fig. 4.1. The advantages of such a form are:

1) The diagonal blocks can either be processed in parallel for savings in processing time or in sequence for savings in storage requirements.

2) The repetitiveness of some diagonal blocks will have speed and storage advantages. Only one block has to be processed and saved. A set of pointers can relate the similar blocks.

3) The latency of some parts of the circuit may be exploited for more saving in time during transient analysis.

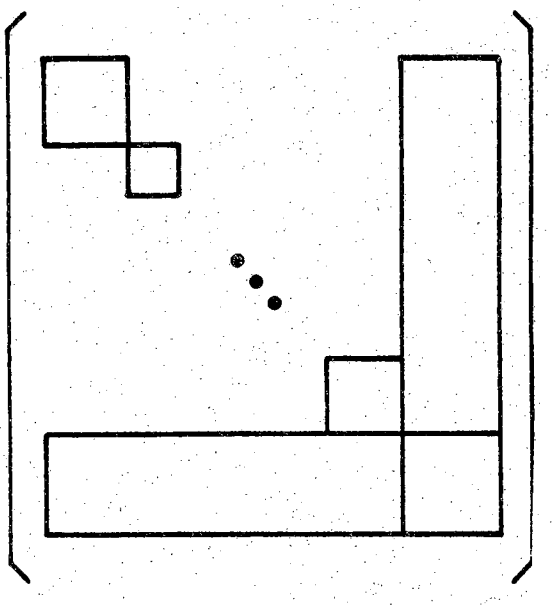


Figure 4.1 Bordered block-diagonal matrix form.

4.1 Tearing Techniques

The process of decomposing a circuit into subcircuits is referred to as tearing or diakoptics [9]. Several tearing techniques would result in a dependency matrix of the form of Figure 4.1. We will use a node tearing technique [17,21] that produces the dependency matrix of Figure 4.1.

The circuit is partitioned into m subcircuits as shown in Fig. 4.2.

The blocks labeled $\mathbf{F}^1$ through $\mathbf{F}^m$ correspond to the subcircuits we have chosen and the **H** block, which is a subcircuit itself, correspond to the rest of the circuit elements. There are nu nodes contained in block **G** in Figure 4.2. These nodes connect the subcircuits together, they are called the tearing nodes [17]. The tearing nodes are labeled as u_j's, $(1 \leq j \leq nu)$, and referred to as the vector **u** $(\mathbf{u} = (u_1, u_2, \ldots, u_{nu})^T)$. The **G** block can be thought of as a subcircuit with no internal nodes. The node-to-datum voltages of the tearing nodes are used as variables in the equation formulation process. Each of the other subcircuits has a number of internal nodes not shared with any other subcircuits. The node-to-datum voltages of these nodes in addition to selected branch currents are used as variables in the equation formulation process. The **H** subcircuit variables are labeled $\mathbf{z}_k'$s, $(1 \leq k \leq nz)$, where nz is the number of variables in **H**. The vector **z** consists of all the z variables $(\mathbf{z} = (z_1, z_2, \ldots, z_{nz})^T)$. The $\mathbf{F}^i$ subcircuit variables are labeled x_ℓ^i's, $(1 \leq \ell \leq nf^i)$, where nf^i is the total number of variables in subcircuit $\mathbf{F}^i$. The vector $\mathbf{x}^i$ consists of all the x^i variables $(\mathbf{x}^i = (x_1^i, x_2^i, \ldots, x_{nf_i}^i)^T)$. The total number of x variables in all the **F** subcircuits is nx.

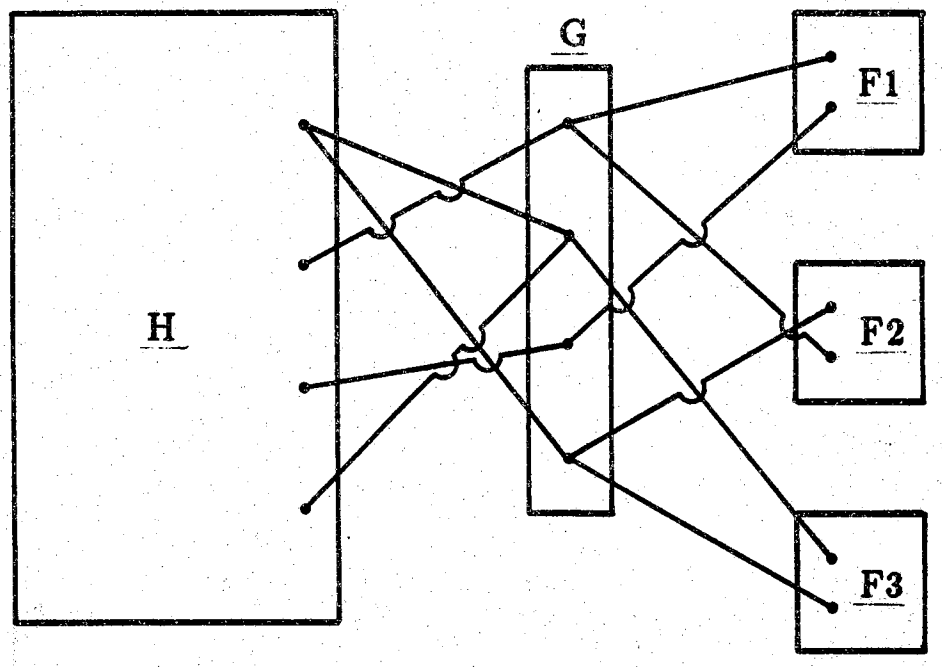


Figure 4.2 Tearing the circuit.

4.2 Modified Nodal Analysis

In formulating the equations for a circuit we use the modified nodal analysis approach, abbreviated MNA [7]. We start with the node-to-datum voltages and then write an equation for each node using Kirchhoff's current law such that the summation of all currents leaving the node is equal to zero. For a circuit containing only linear conductances and capacitances, and independent current sources, the first portion of the MNA equations at a certain time instance t would have the form [7]

$$\mathbf{Y}\mathbf{V} = \mathbf{J} \tag{4.1}$$

where **Y** is the node admittance matrix, **V** the node-to-datum voltage vector and **J** the current source vector. Capacitors are replaced by a linear model, Figure 4.3 [4]. The value of v(t-h), where t is the time variable and h is the time step taken, is the voltage at the previous time point. The value of v(t-h) would be readily available from the solution of equation (4.1) at t-h. The next step is introducing some branch currents as additional variables and the corresponding voltage-current branch relationships as additional equations. These additional variables are chosen to be the currents through voltage sources and any controlling currents (e.g. inductor currents). Adding voltage sources and linear inductances to the above circuit, the MNA equations would have the form [7]

$$\begin{bmatrix} \mathbf{Y}* & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{J} \\ \mathbf{K} \end{bmatrix} \tag{4.4}$$

where **Y*** excludes contribution due to branches whom currents are variables. The contributions are covered in **B** as ± 1's. **C** and **D** represent the branch voltage-current relationships. **J** and **K** are excitations (voltage and current

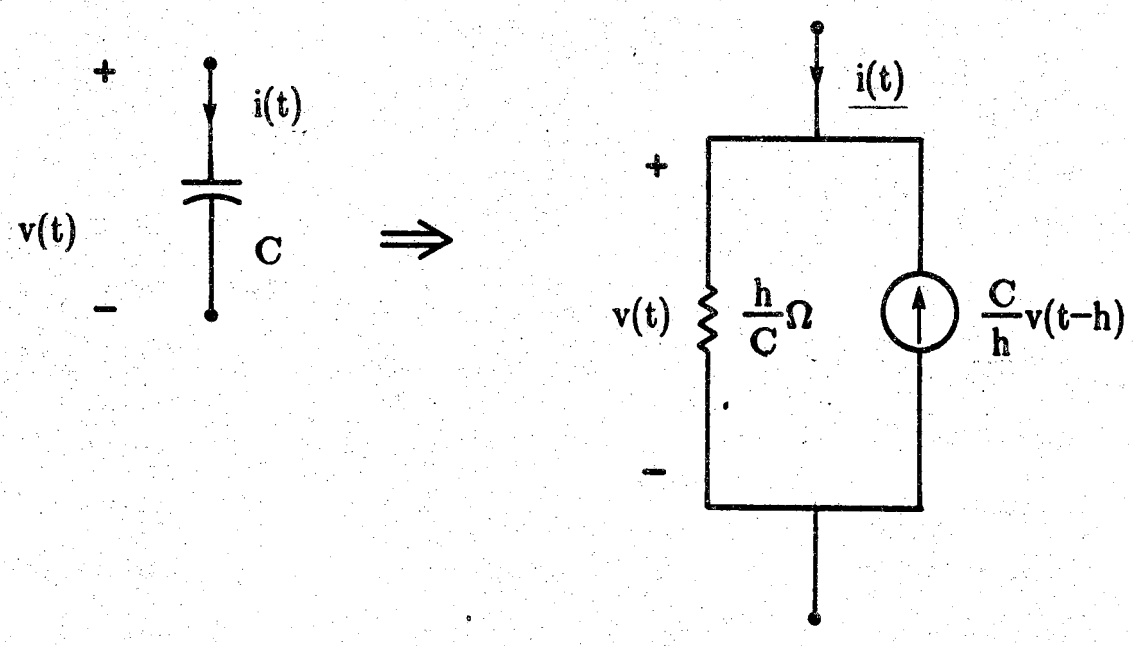


Figure 4.3 Linear model for a capacitor using 1st order BDF (see section 7.1)

sources values). Inductors are replaced by a linear model, Figure 4.4 [4]. It is obvious that a previous value for the inductor current is needed at time t-h. That is why it is introduced as a variable in MNA equations.

4.3 Equation Formulation

The MNA approach is used to formulate the equations for each of the subcircuits described in section 4.1. Each subcircuit is treated as an individual circuit in the formulation process. Given a circuit, the equation formulation process proceeds as follows:

1) Replace all devices which have three terminals or more by their appropriate models (e.g. replace a BJT-transistor by its Ebers-Moll model [4]).
2) Tear the circuit into subcircuits labeling the node and current variables as described in section 4.1; (refer to Figure 4.2).
3) Write the nonlinear algebraic equations using MNA for the circuit treating each block in Figure 4.2 as an individual circuit with some constraints as illustrated below:
   a) The **G** block has nu variables, so we write nu nonlinear equations by applying Kirchhoff's current law at each tearing node u_j. The equations will be functions of **u** in addition to $\mathbf{x}^i$'s and **z**. In the solution process for **u**, using the Semi-Direct method, **z** and $\mathbf{x}^i$ are treated as constants. The equations can be expressed in the following form

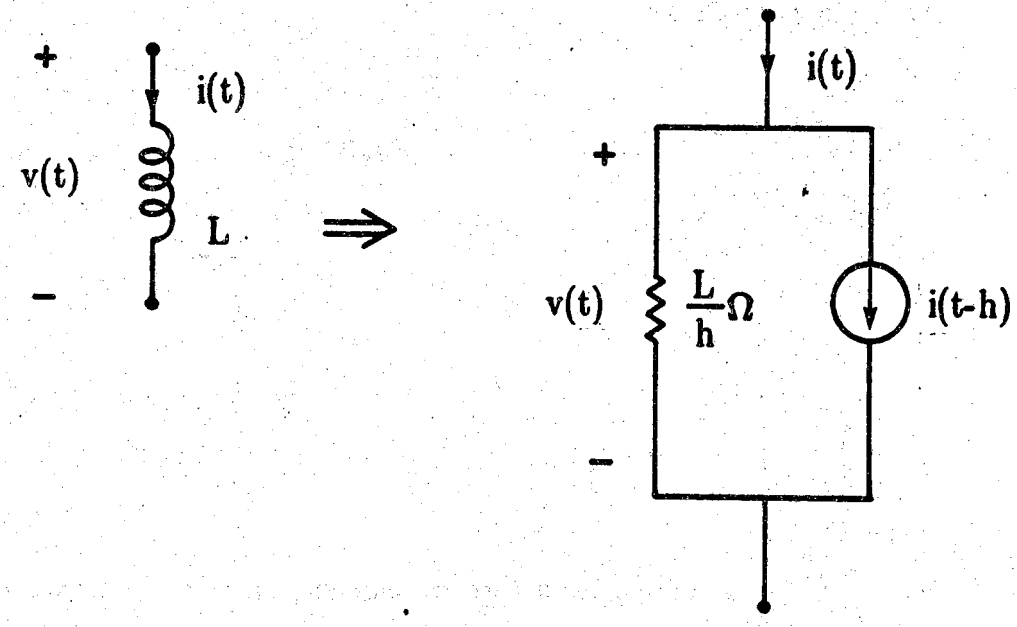


Figure 4.4 Linear model for an inductor using 1st order BDF (see section 7.1)

$$\mathbf{G}(\mathbf{x}^1,\mathbf{x}^2,...,\mathbf{x}^m,\mathbf{z},\mathbf{u}) = \mathbf{0} \tag{4.5}$$

b) The **H** block has nz variables. At the node variables we apply Kirchhoff's current law, and for the current variables we apply the voltage-current branch relationships. The result is nz equations of the form of equation (4.4). These equations are a function of **z** and **u** only. They are not connected to any of the $\mathbf{F}^i$ subcircuits. In the solution process for **z** using the Semi-Direct method, **u** will be treated as a constant vector. This block of equations can be written as

$$\mathbf{H}(\mathbf{z},\mathbf{u}) = \mathbf{0} \tag{4.6}$$

c) Each $\mathbf{F}^i$ block has nf^i variables. At the node variables we apply Kirchhoff's current law, and for the current variables we apply the voltage-current branch relationships. The result is nf^i equations for each block F^i in the form of equation (4.4). These equations are functions of $\mathbf{x}^i$ and **u** only. They are not connected to the **H** subcircuit. In the solution process for $\mathbf{x}^i$, using the Semi-Direct method, **u** will be treated as a constant vector. The blocks of equations can be written as

$$\begin{aligned}
\mathbf{F}^1(\mathbf{x}^1,\mathbf{u}) &= \mathbf{0} \\
\mathbf{F}^2(\mathbf{x}^2,\mathbf{u}) &= \mathbf{0} \\
&\vdots \\
\mathbf{F}^m(\mathbf{x}^m,\mathbf{u}) &= \mathbf{0}
\end{aligned}$$

or in a more compact form

$$\mathbf{F}^i(\mathbf{x}^i,\mathbf{u}) = 0 \tag{4.7}$$

The result of step 3 is m+2 blocks of equations, each has an equal number of assigned unknowns (variables) and equations. That is, for

G we have nu equations in nu unknowns, for **H** nz equations in nz unknowns, and for each $\mathbf{F}^i$, nf^i equations in nf^i unknowns.

4) Write the Jacobian Matrix (equation 2.6) for the system as follows:

a) For the Newton-Raphson method: Equations (4.5), (4.6) and (4.7) are all considered as one system of equations. Let the total number of variables in the system, which is the same as the total number of equations be

$$n^{tot} = \sum_{i=1}^{m} nf^i + nz + nu \tag{4.8}$$

We have a system of equations that has n^{tot} equations in n^{tot} unknowns and is to be solved using the Newton-Raphson method described in section 2.2. The Jacobian matrix will have dimensions $n^{tot} \times n^{tot}$ and have a bordered block-diagonal form (Figure 4.1). We partition it as follows:

$$\mathbf{J}(\mathbf{x}^i,\mathbf{z},\mathbf{u}) = \begin{bmatrix} \mathbf{A}_{ss1} & & & & & \mathbf{A}_{sr1} \\ & \mathbf{A}_{ss2} & & & & \mathbf{A}_{sr2} \\ & & \bullet & & & \bullet \\ & & & \bullet & & \bullet \\ & & & & \mathbf{A}_{ssm} & \mathbf{A}_{srm} \\ & & & & \mathbf{A}_{ss(m+1)} & \mathbf{A}_{sr(m+1)} \\ \mathbf{A}_{rs1} & \mathbf{A}_{rs2} & \bullet & \bullet & \mathbf{A}_{rsm} & \mathbf{A}_{rsm} \quad \mathbf{A}_{rr} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{A}_{ss} & \mathbf{A}_{sr} \\ \mathbf{A}_{rs} & \mathbf{A}_{rr} \end{bmatrix} \tag{4.9}$$

The algorithm used is described in section 4.4 which is a special case of the Newton-Raphson method designed to make use of the above bordered block-diagonal form of the Jacobian matrix.

b) For the Semi-Direct Method:

Each block is treated as an independent system of equations. So we write a Jacobian matrix for each block taking into consideration which variables are to be treated as a constant in each block as described in step 3. These smaller Jacobian matrices can be related to equation (4.9) as follow:

$$\mathbf{J}_{f^i}(\mathbf{x}^i) = \mathbf{A}_{ssi} \tag{4.10}$$

$$\mathbf{J}_h(\mathbf{Z}) = \mathbf{A}_{ss(m+1)} \tag{4.11}$$

$$\mathbf{J}_g(\mathbf{u}) = \mathbf{A}_{rr} \tag{4.12}$$

The Semi-Direct method is then used to solve the system. The algorithm is described in chapter 5.

5) Input the equations into the appropriate algorithm.

From the above discussion we can see an advantage for the Semi-Direct method. There is no need to calculate the partial derivatives in the submatrices $\mathbf{A}_{rs}$ and $\mathbf{A}_{sr}$. This means they do not have to be stored either which accounts for computer storage space savings. The relationship between the subcircuits and the equations can be easily illustrated by labeling the blocks of equation (4.9) (Figure 4.5).

To illustrate the above procedure, consider this example:

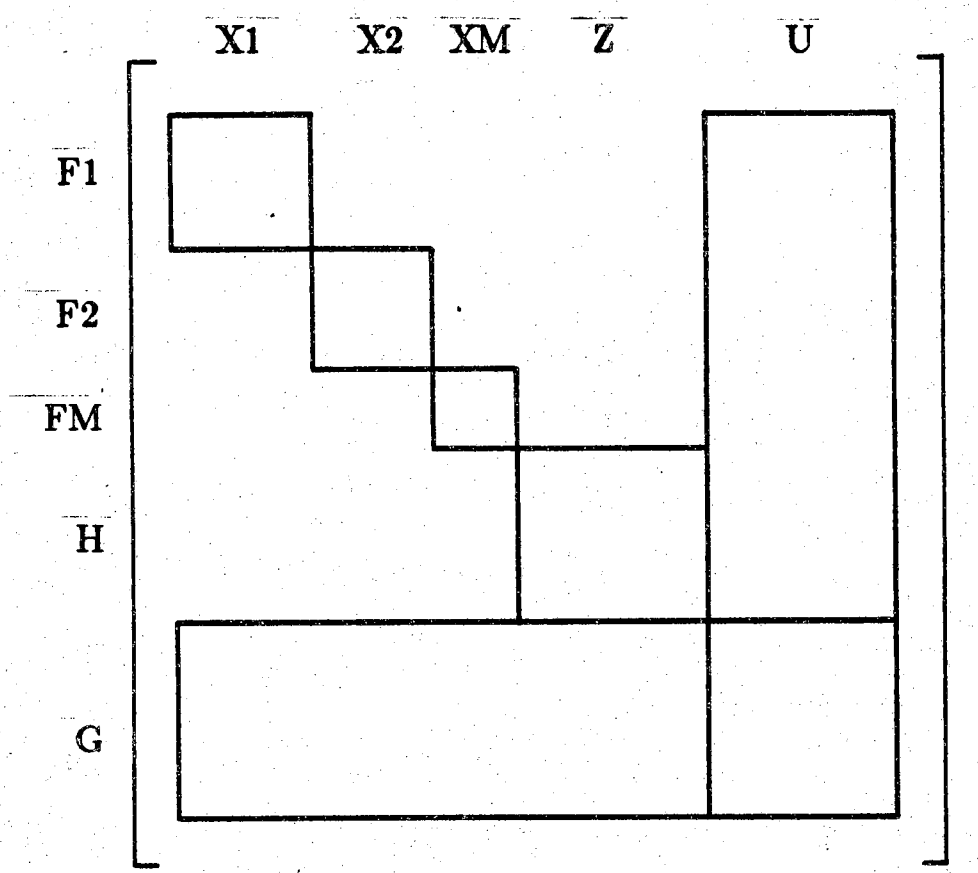


Figure 4.5 The Jacobian matrix.

Example 4.1

Consider the circuit in Figure 4.6 which has nonlinear resistors as illustrated. The circuit is partitioned into three subcircuits using two tearing nodes.

Applying Kirchhoff current laws at the node variables and voltage current relationships for the current variables we would obtain 4 $\mathbf{F}^1$ equations, $3\mathbf{F}^2$ equations, 4 **H** equations and 2 **G** equations. The Jacobian matrix will have the form of Figure 4.5 where $\mathbf{A}_{ss1}$ is a 4×4 matrix, $\mathbf{A}_{ss2}$ is a 3×3 matrix, A_{ss3} is a 4×4 matrix and $\mathbf{A}_{rr}$ is a 2×2 matrix. (See example 5.5).

4.4 A Newton-Raphson Algorithm for Bordered Block-Diagonal Matrices

We have seen in sections 4.1-3 how the dependency matrix is formulated in bordered block-diagonal form. We must note that a common reference node is assumed to be shared by all the torn subcircuits (each subcircuit should have a branch connected to the reference node). The reason for such an assumption, which is a very practical one, is to simplify the bordered block-diagonal form of the dependency matrix. If floating reference nodes are used by each subcircuit the dependency matrix would have a form as in Figure 4.7. An algorithm using the Newton-Raphson method to solve a system with such a dependency matrix was proposed in [21]. A simplified version is extracted to serve the simpler dependency matrix form of equation (4.9).

The regular Newton-Raphson algorithm illustrated in section 2.2 would be sufficient to obtain a solution for any general case, but to make use of the sparsity of the dependency matrix we have to utilize the simplified algorithm.

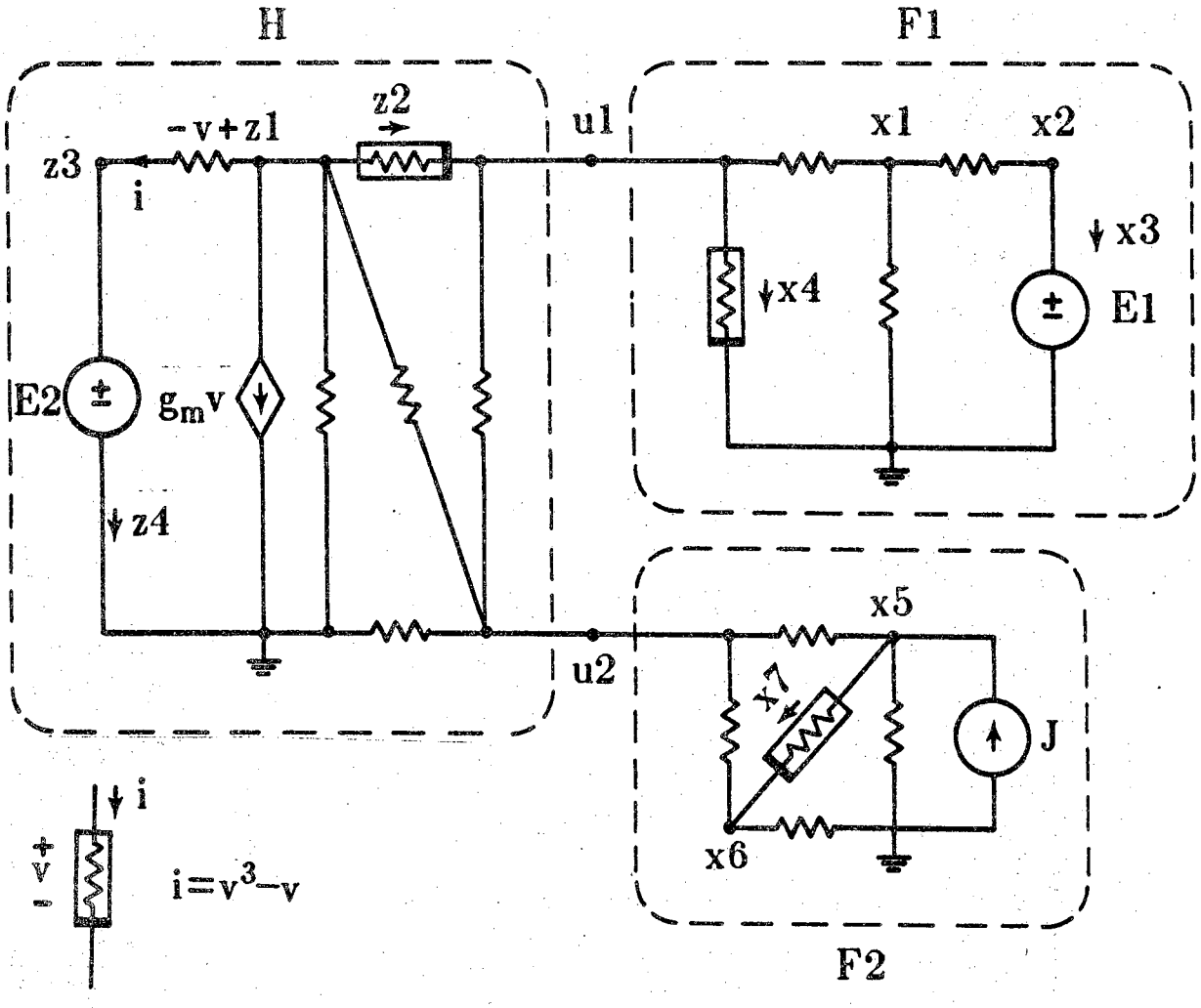


Figure 4.6 Tearing a circuit

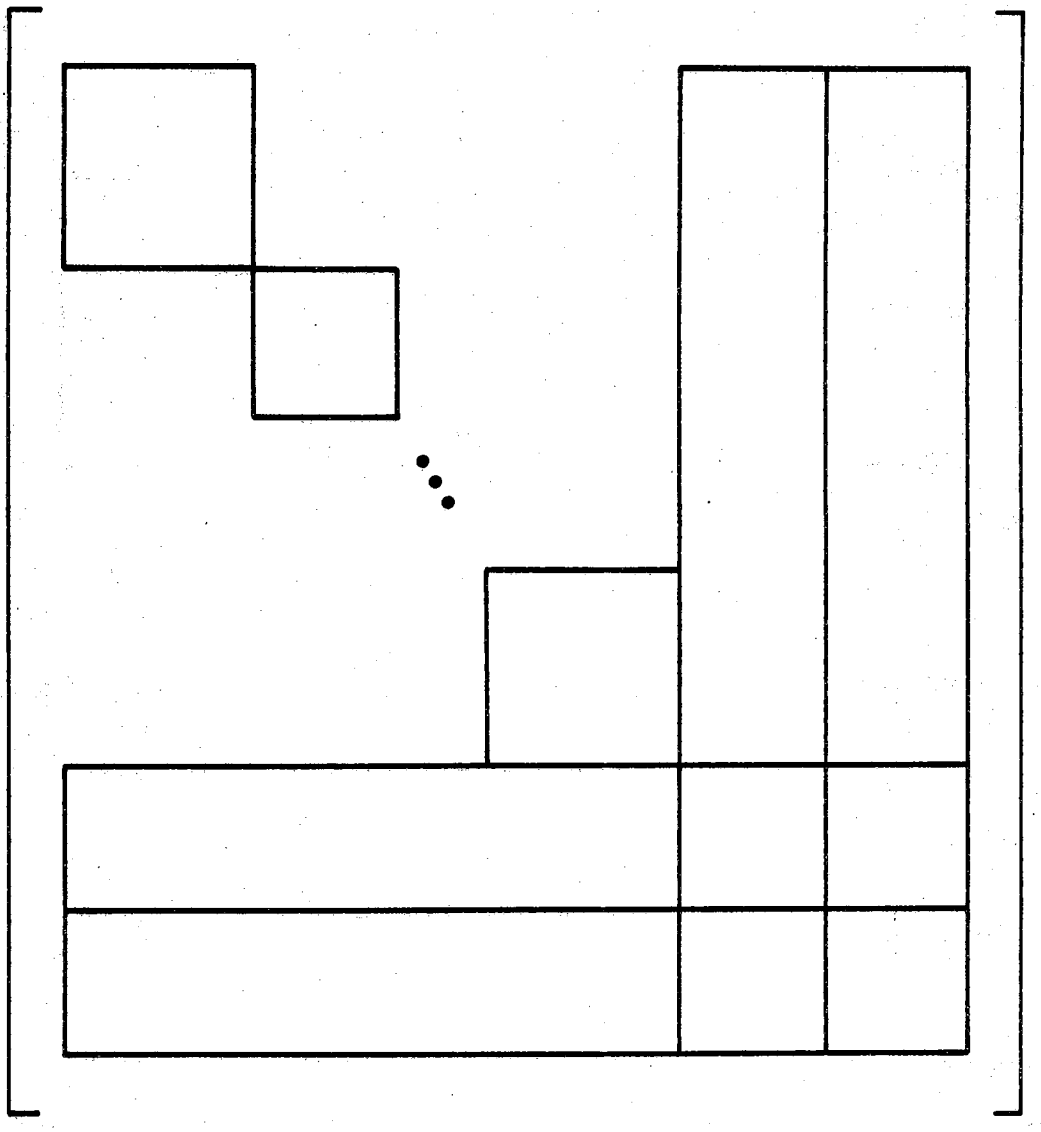


Figure 4.7 Bordered block-diagonal form of a dependency matrix for a circuit with a floating reference node.

Referring to Section 2.2 the equation to be solved at a time instant t is equation (2.9) which is in the form

$$\mathbf{Ax} = \mathbf{b} \tag{4.13}$$

We are assuming **A** has the form of the dependency matrix in equation (4.9) so

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{A}_{ss} & \mathbf{A}_{sr} \\ \mathbf{A}_{rs} & \mathbf{A}_{rr} \end{bmatrix} \begin{bmatrix} \mathbf{x}_s \\ \mathbf{u}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_s \\ \mathbf{b}_r \end{bmatrix} \tag{4.14}$$

where the **z**'s are included in $\mathbf{x}_s$.

The restriction on the block diagonal matrices is that they be nonsingular or else this algorithm fails. We solve equations (4.5) by solving for **u** from

$$(\mathbf{A}_{rr} - \mathbf{A}_{rs}\,\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr})\mathbf{u}_r = (\mathbf{b}_r - \mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{b}_s) \tag{4.15}$$

then for $\mathbf{x}_s$ from

$$\mathbf{A}_{ss}\,\mathbf{x}_s = (\mathbf{b}_s - \mathbf{A}_{sr}\mathbf{u}_r) \tag{4.16}$$

The inverse of A_{ss} is simply the direct sum of the inverse of the diagonal blocks. The procedure can be restated in terms of **LU** factors which is convenient for programming. There are five steps to the procedure:

1) **LU** decomposition

$$\mathbf{A}_{ssi} = \mathbf{L}_{ssi}\,\mathbf{U}_{ssi} \tag{4.17}$$

2) Solve for

$\boldsymbol{\phi}_i$ from $\quad \mathbf{L}_{ssi}\,\boldsymbol{\phi}_i = \mathbf{A}_{sri}$ (4.18)

$\boldsymbol{\psi}_i$ from $\quad \boldsymbol{\psi}_i\,\mathbf{U}_{ssi} = \mathbf{A}_{rsi}$ (4.19)

ξ_i from $\quad \mathbf{L}_{ssi}\,\xi_i = \mathbf{b}_{si}$ (4.20)

3) Solve for $\mathbf{u}$ from

$$\left[\mathbf{A}_{rr} - \sum_{i=1}^{m+1} \psi_i \phi_i\right] \mathbf{u}_r = \left[\mathbf{b}_r - \sum_{i=1}^{m+1} \psi_i \xi_i\right] \tag{4.21}$$

4) Solve for $\mathbf{Y}_{si}$ from

$$\mathbf{L}_{ssi}\mathbf{Y}_{si} = [\mathbf{b}_{si} - \mathbf{A}_{sri}\,\mathbf{u}_r] \,. \tag{4.22}$$

5) Solve for $\mathbf{x}_{si}$ from

$$\mathbf{U}_{ssi}\,\mathbf{x}_{si} = \mathbf{Y}_{si} \tag{4.23}$$

The advantages are clearly the same as stated in the opening of this chapter.

4.5 Remarks

The tearing and equation formulation techniques in addition to the Newton-Raphson algorithm explained in this chapter have some restrictions. Such restrictions are common and practical ones. They are listed below with a justification of their practicality.

1) Common reference node: Almost all Integrated Circuits fulfill this requirement.
2) Nonsingularity of the block-diagonal matrices of the dependency matrix. Each block represents a subcircuit, the possibility of it being singular is very slim in real circuit application. Even if it occurs, an addition of a small conductance in parallel to some element in the circuit would take care of that problem.

3) Non-zero diagonal entries on the diagonal of the dependency matrix. This is for LU factorization purposes. This can be satisfied by exchanging one or more rows.

4) An ideal voltage source connected to one of the **u** nodes. This would require the introduction of the current through the source as an extra **u** variable. The ideal voltage source though is nonexistent in reality so the introduction of a small resistance in series (an extra node variable) would enable us to keep the tearing configuration as in Figure 4.2.

CHAPTER 5
THE SEMI-DIRECT METHOD

The Semi-Direct method proposed in [15] makes use of the two numerical analysis methods illustrated in Chapters 1 & 2, namely the Newton-Raphson and the Gauss-Siedel methods. The equations to be solved are in the form of Eq. (4.5), (4.6) and (4.7) which are

$$\mathbf{F}_i(\mathbf{x}^i,\mathbf{u}) = \mathbf{0} \quad 1 \leq i \leq m \tag{5.1}$$

$$\mathbf{H}(\mathbf{z},\mathbf{u}) = \mathbf{0} \tag{5.2}$$

$$\mathbf{G}(\mathbf{x}^1,\ldots,\mathbf{x}^m,\mathbf{z},\mathbf{u}) = \mathbf{0} \tag{5.3}$$

and the dependency matrix has the form of equation (4.9), where $\mathbf{F}_i$, $\mathbf{H}$, $\mathbf{G}$, $\mathbf{x}^i$, $\mathbf{z}$ and $\mathbf{u}$ are all as defined in Chapter 4.

5.1 Algorithm

The Semi-Direct method if used to solve equations (5.1), (5.2) and (5.3) would take the following steps:

1) Make an initial guess of the vector $\mathbf{u}$.
2) Solve equations (5.1) and (5.2) for all $\mathbf{x}^i$ and $\mathbf{z}$ vectors, (sequentially or in parallel).

3) Using the new values of $\mathbf{x}_i$'s and **z** solve for an updated value of **u** using equations (5.3).

4) Repeat steps 2 and 3 until some termination criterion is met. Each iteration of steps 2 and 3 is called a sweep.

Let us consider a special case where the **u** vector has length 1 and there is only one $\mathbf{x}^i$ vector and it is of length 1. So we only have two equations in two unknowns x and u such that

$$f(x,u) = 0 \tag{5.4}$$

$$g(x,u) = 0 \tag{5.5}$$

Using the above Semi-Direct algorithm to solve equations (5.4) and (5.5) we take a look at two cases:

1) f and g are linear equations in which case they can be reduced to the explicit forms

$$x = f^*(u) \tag{5.6}$$

$$u = g^*(x) \tag{5.7}$$

Applying the Semi-Direct algorithm in this case would reduce to the Gauss-Seidel method. Steps 2 and 3 can be directly evaluated from equations (5.6) and (5.7).

2) f and g are nonlinear equations in which case steps 2 and 3 have to be executed using some numerical analysis technique. (We are assuming a general case where f and g are of order ≥ 2). The algorithm in this case is sometimes referred to as a nonlinear Gauss-Seidel method [15].

The Newton-Raphson method is used to execute steps 2 and 3. It was chosen because of its convergence characteristics advantages (Chapter 1).

A geometric interpretation for the 2-dimensional case discussed above does exist. The two equations (5.4) and (5.5) can be plotted as shown in Figure 5.1. The initial guess is u^o. Then tracing steps 2 through 4 of the previous algorithm would produce the dotted line which is converging to the solution point p. There are cases however that the system can diverge or oscillate as illustrated in Figures 5.2 and 5.3, respectively. The case where the graphs in Figure 5.1, 5.2, and 5.3 are straight lines is the linear case of the Semi-Direct method which is simply the Gauss-Seidel method.

The stopping criteria for the algorithm are the same as for the Newton-Raphson method and the Gauss-Seidel method discussed in Chapters 1 and 2 respectively. They are

1) The increment Δx_{j+1} becomes less than a user specified increment e. Which implies the sweeps are converging.
2) The number of sweeps exceed a user specified maximum number in which case the sweeps are diverging, oscillating or converging too slow for a satisfactory performance.

Example 5.1:

Let f and g in equations (5.4) and (5.5) be

$$f(x,u) = x^3+10x+u-31 = 0 \tag{5.8}$$

$$g(x,u) = 2(x-u-1)^3-u+19 = 0 \tag{5.9}$$

Using the Newton-Raphson algorithm and an initial guess of $x^o = 1$ and $u^o = 1$

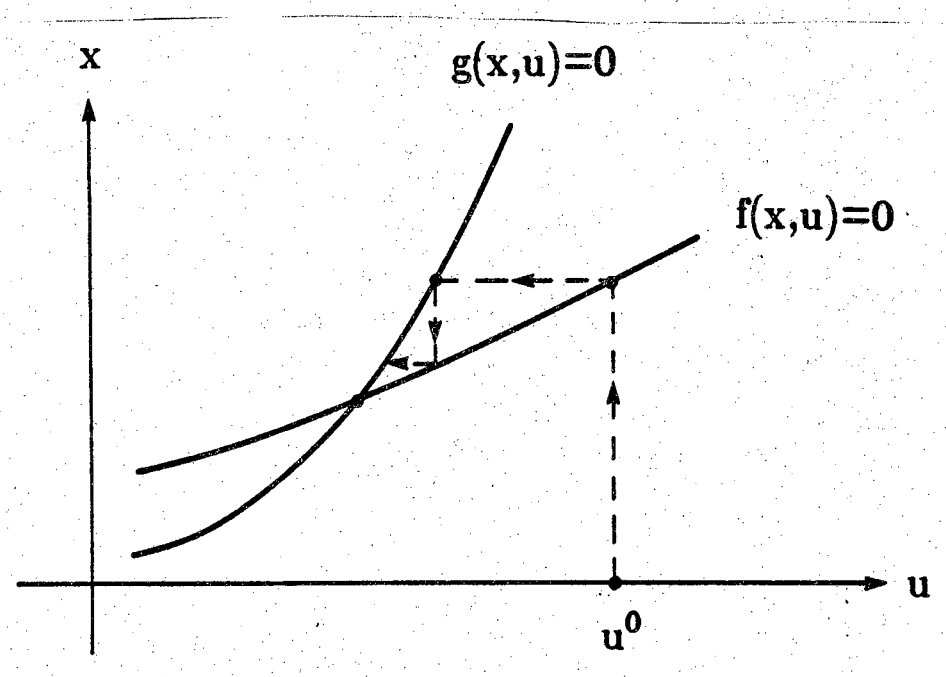


Figure 5.1: Convergent case (Semi-Direct)

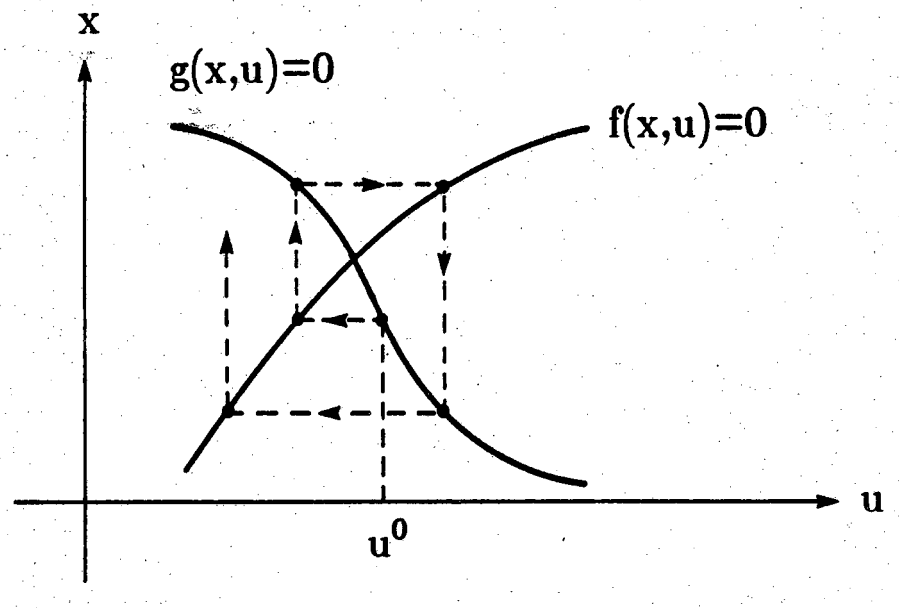


Figure 5.2: Divergent case (Semi-Direct)

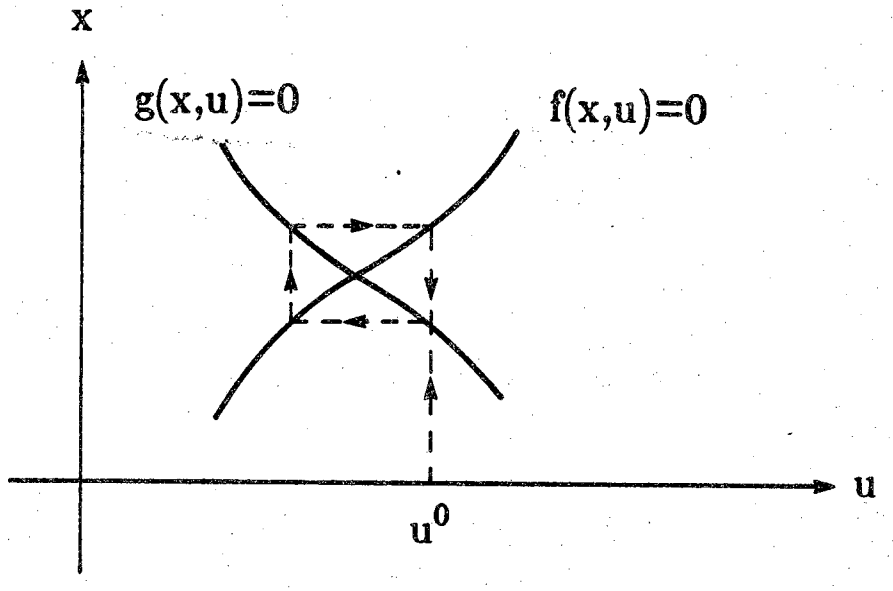


Figure 5.3: Oscillatory case (Semi-Direct)

we get the following results:

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 120619e+01	0. 100000e+01	0. 331959e+01
0. 220619e+01	-0. 209006e+00	0. 431959e+01	-0. 977583e+00
0. 199718e+01	0. 782370e-03	0. 334200e+01	-0. 297179e+00
0. 199796e+01	0. 199497e-02	0. 304483e+01	-0. 438442e-01
0. 199996e+01	0. 435395e-04	0. 300098e+01	-0. 981709e-03
0. 200000e+01	0. 214871e-07	0. 300000e+01	-0. 484090e-06
0. 200000e+01	0. 520838e-14	0. 300000e+01	-0. 117693e-12

The algorithm converges to the solution x = 2 and u = 3 in 7 iterations. To get the solution using the Semi-Direct method. Let us define u_j and x_j as the jth sweep results. We first treat u as a constant in equation (5.8) and using equation (2.2) we can write

$$x_j^{k+1} = x_i^k - \frac{f(x_j^k, u_j)}{\left.\frac{\partial f}{\partial x}\right|_{x=x_j^k}} \tag{5.10}$$

where k = 0,1,2,..., is the iteration number for x. We evaluation the partial derivative from

$$\frac{\partial f(x,u)}{\partial x} = 3x^2 + 10 \tag{5.11}$$

After obtaining a solution from (5.10) for x_i we substitute that value into the iterative expression for equation (5.9)

$$u_j^{k+1} = u_j^k - \frac{g(x_i, u_j^k)}{\left.\frac{\partial g}{\partial u}\right|_{u=u_j^k}} \tag{5.12}$$

where

$$\frac{\partial g}{\partial u} = -6(x-u-1)^2-1 \tag{5.13}$$

The result of this iteration process would produce u_{j+1} which is then used in place of u_j (updating the value of u) in equation (5.10) to obtain x_{j+1} and the process continues. The following are the sweep results, it took the Semi-Direct method 8 sweeps to reach the solution x = 2 and u = 3.

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 108873e+01	0. 100000e+01	0. 208517e+01
0. 208873e+01	-0. 926058e-01	0. 308517e+01	-0. 888955e-01
0. 199612e+01	0. 404479e-02	0. 299628e+01	0. 388301e-02
0. 200017e+01	-0. 176493e-03	0. 300016e+01	-0. 169433e-03
0. 199999e+01	0. 770152e-05	0. 299999e+01	0. 739346e-05
0. 200000e+01	-0. 336066e-06	0. 300000e+01	-0. 322624e-06
0. 200000e+01	0. 146647e-07	0. 300000e+01	0. 140781e-07
0. 200000e+01	-0. 639914e-09	0. 300000e+01	-0. 614318e-09

The dependency matrix of example 5.1 is a degenerate form of eq. (4.9), it has the form

$$J(x,u) = \begin{bmatrix} 3x^2+10 & 1 \\ +6(x-u-1)^2 & -6(x-u-1)^2-1 \end{bmatrix} \tag{5.14}$$

so the submatrices of eq. (4.9) would be

$$A_{ss1} = [3x^2+10] , \tag{5.15}$$

$$A_{rr} = [-6(x-u-1)^2] , \tag{5.16}$$

$$A_{sr} = [1], \tag{5.17}$$

and

$$A_{rs} = [6(x-u-1)^2-1] \tag{5.18}$$

The Semi-Direct method only needed the two submatrices $\mathbf{A}_{ss1}$ and $\mathbf{A}_{rr}$ (equations (5.15) and (5.16)) to calculate the solution for x and u in equations (5.8) and (5.9). There was no need to calculate or store the expressions for $\mathbf{A}_{sr}$ and $\mathbf{A}_{rs}$ which in this example are half the entries to the dependency matrix. These savings may be further amplified in a more general case of the dependency matrix as in example 5.3 in the next section.

5.2 Convergence

We expect the Semi-Direct or the nonlinear Gauss-Seidel method to have convergence properties similar to the regular linear Gauss-Seidel method (Chapter 3).

The global convergence properties of the Semi-Direct method is discussed in [2]. They are derived and stated in terms of the contractions requirement on the function. These conditions are extremely difficult to apply in practice, e.g., a computer program. A more practical approach would be to study the local convergence properties of the method. Since we will use the Semi-Direct method mainly in transient analysis applications, the solution at a certain time point using a sufficiently small time step, which usually is the case in transient analysis, can be considered within the local area of the solution at the next time point. So the initial guess at a time point will be relatively close to the time solution if the result at the previous time point is used and even closer if a predictor formula in time is used to approximate the next solution.

The Gauss-Seidel method was designed to solve a set of linear equations (equations (3.3)) of the form

$$\mathbf{Ax} = \mathbf{B} \tag{5.19}$$

The coefficient matrix **A** in this case is simply the same as the Jacobian matrix equations (3.3). As the sweeps of the Semi-Direct method approaches the true solution the nonlinear equations can be approximated by a linear version. In which case the Jacobian Matrix of the nonlinear system at the solution point can be considered as the coefficient matrix of the linearized system. So, the convergence conditions for the linear Gauss-Seidel method can be applied to the Jacobian matrix evaluated at the solution point.

Consider the 2 nonlinear equation case of equations (5.4) and (5.5).

The Jacobian matrix at the solution **x** and **u**

$$\mathbf{J}(\mathbf{x},\mathbf{u}) = \left.\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial u} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial u} \end{bmatrix}\right|_{\substack{x=\mathbf{x} \\ u=\mathbf{u}}} \tag{5.20}$$

which can be written as

$$\mathbf{J} = \mathbf{P}+\mathbf{Q} = \begin{bmatrix} \frac{\partial f}{\partial x} & 0 \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial u} \end{bmatrix} + \begin{bmatrix} 0 & \frac{\partial f}{\partial u} \\ 0 & 0 \end{bmatrix} \tag{5.21}$$

The system will converge if and only if the eigenvalues of $\mathbf{P}^{-1}\mathbf{Q}$ have magnitude less than 1. That is the eigenvalues of

$$\begin{bmatrix} \frac{\partial f}{\partial x} & 0 \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial u} \end{bmatrix}^{-1} \begin{bmatrix} 0 & \frac{\partial f}{\partial u} \\ 0 & 0 \end{bmatrix}, \text{ or } \begin{bmatrix} 0 & \frac{\partial g}{\partial u} \cdot \frac{\partial f}{\partial u} \\ 0 & -\frac{\partial g}{\partial x} \cdot \frac{\partial f}{\partial u} \end{bmatrix} \frac{1}{\frac{\partial f}{\partial x} \cdot \frac{\partial g}{\partial u}} \tag{5.22}$$

evaluated at $x=\mathbf{x}$ and $u=\mathbf{u}$ should have magnitudes less than 1

$$|\lambda| = \left| \frac{\frac{\partial f}{\partial u} \cdot \frac{\partial g}{\partial x}}{\frac{\partial f}{\partial x} \cdot \frac{\partial g}{\partial u}} \right| < 1 \tag{5.23}$$

The convergence rate is linear and is given by

$$|e_{j+1}| = \max |\lambda| \cdot |e_j| \tag{5.24}$$

where

$$e_j = a_j - \mathbf{a} \tag{5.25}$$

and a can be any x, y or z variables. (see section 3.2). As an approximation for (5.24) when the sweeps are close to the time solution. We can write

$$|\Delta a_{j+1}| = \max |\lambda| \cdot |\Delta a_j| \tag{5.26}$$

where

$$\Delta a_j = a_{j+1} - a_j \tag{5.27}$$

Example 5.2:

For equation (5.8) and (5.9) in example 5.1 we can evaluate their Jacobian matrix (equation (5.14)) at the solution x=2 u=3

$$\mathbf{J}(2,3) = \begin{bmatrix} 22 & 1 \\ 24 & -25 \end{bmatrix}$$

Separating into **P** and **Q** we have

$$\mathbf{J} = \mathbf{P} + \mathbf{Q} = \begin{bmatrix} 22 & 0 \\ 24 & -25 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

The convergence condition then is that the magnitude of the eigenvalues of $\mathbf{P}^{-1}\mathbf{Q}$ be less than 1.

$$\begin{bmatrix} 22 & 0 \\ 24 & -25 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0.0455 \\ 0 & 0.04364 \end{bmatrix}$$

The eigenvalue λ of $\mathbf{P^{-1}Q}$ is 0.04364 whose magnitude is less than 1.

So the system is convergent and the results were obtained in example 5.1

Checking values of x and u close to the solution and using equation (5.24) we should obtain this approximate relationship,

$$\left| \frac{\Delta x_{j+1}}{\Delta x_j} \right| \simeq |\lambda| \tag{5.28}$$

and

$$\left| \frac{\Delta u_{j+1}}{\Delta u_j} \right| \simeq |\lambda| \tag{5.29}$$

From example 5.1 Semi-Direct results we have

$$\left| \frac{\Delta x_7}{\Delta x_6} \right| = \left| \frac{-0.6399140 \times 10^{-9}}{0.146647 \times 10^{-7}} \right| = 0.04364 = |\lambda|$$

$$\left| \frac{\Delta u_7}{\Delta u_y} \right| = \left| \frac{-0.614318 \times 10^{-9}}{0.140781 \times 10^{-7}} \right| = 0.04364 = |\lambda|$$

$$\left| \frac{\Delta x_5}{\Delta x_6} \right| = \left| \frac{+0.146647 \times 10^{7}}{-0.336066 \times 10^{-6}} \right| = 0.04364 = |\lambda|$$

$$\left| \frac{\Delta u_5}{\Delta u_6} \right| = \left| \frac{0.140781 \times 10^{-7}}{-0.322624 \times 10^{-6}} \right| = 0.04364 = |\lambda|$$

For the general case of the dependency matrix in the form of equation (4.9) the Semi-Direct method converges if and only if the eigenvalues of $\mathbf{P^{-1}Q}$

(see sec. 4.2) evaluated at the solution point have magnitude less than unity. So as we get close to the solution point the equations can be written in the form of equation (5.19), where **A** is the Jacobian matrix of the nonlinear system evaluated at the solution point. It can be rewritten as

$$\begin{bmatrix} \mathbf{A}_{ss} & \mathbf{A}_{sr} \\ \mathbf{A}_{rs} & \mathbf{A}_{rr} \end{bmatrix} \begin{bmatrix} \mathbf{x}_s \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{B}_s \\ \mathbf{B}_r \end{bmatrix} \tag{5.30}$$

The Semi-Direct iterations can be expressed in the form of a difference equation (see section 4.2)

$$\mathbf{x}_{j+1} = -\mathbf{P}^{-1}\mathbf{Q}\,\mathbf{x}_j + \mathbf{P}^{-1}\mathbf{B} \tag{5.31}$$

where

$$\mathbf{P} = \begin{bmatrix} \mathbf{A}_{ss} & 0 \\ \mathbf{A}_{rs} & \mathbf{A}_{rr} \end{bmatrix} \text{ and } \mathbf{Q} = \begin{bmatrix} 0 & \mathbf{A}_{sr} \\ 0 & 0 \end{bmatrix}$$

we have

$$\mathbf{P}^{-1} = \begin{bmatrix} \mathbf{A}_{ss}^{-1} & 0 \\ -\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1} & \mathbf{A}_{rr}^{-1} \end{bmatrix}$$

so $\mathbf{P}^{-1}\mathbf{Q}$ has the form

$$\mathbf{P}^{-1}\mathbf{Q} = \begin{bmatrix} 0 & \mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} \\ 0 & \mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} \end{bmatrix} \tag{5.32}$$

The eigenvalues of $\mathbf{P}^{-1}\mathbf{Q}$ (equation (5.32)) are 0 and the eigenvalues of $\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr}$. So, the Semi-Direct iterations converge if and only if the magnitude of the eigenvalues of

$$\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} \tag{5.33}$$

are all less than unity.

The rate of convergence is linear and is equal to the eigenvalue of (5.33) with the largest magnitude as explained earlier in this section.

Example 5.3:

Consider this system of 10 equations in 10 unknowns.

```
fx(1)= x(1)**3 + 10*x(1) + x(2) -u(1)**2+2*u(2)**2-60
fx(2)= 2*(x(1)-x(2)-1)**3 -x(2) +6*u(1)*u(2) -16
 jac(1,1,1)= 3*x(1)**2 + 10
 jac(1,1,2)= 1
 jac(1,2,1)= 6*(x(1)-x(2)-1)**2
 jac(1,2,2)= -6*(x(1)-x(2)-1)**2 -1
fx(3)=x(3)**3+10*x(3)+x(4) -u(1)**2+2*u(2)**2+27
fx(4)= 2*(x(3)-x(4)-1)**3 -x(4) +6*u(1)*u(2) +216
 jac(2,1,1)= 3*x(3)**2 + 10
 jac(2,1,2)= 1
 jac(2,2,1)= 6*(x(3)-x(4)-1)**2
 jac(2,2,2)= -6*(x(3)-x(4)-1)**2 -1
fx(5)=x(5)+ 4*x(6)**3 +u(1)**3-u(2)**3 +84
fx(6)=-x(6)**2+ x(5)**2+x(6) +2*u(1)**3+u(2)-69
 jac(3,1,1) = 1
 jac(3,1,2)= 12*x(6)**2
 jac(3,2,1)=x(5)*2
 jac(3,2,2) = -2*x(6) + 1
fx(7) = u(1)**3 -u(2) +z(1)**2 +z(1) +z(2) -29
fx(8)=z(1)+u(1)**3+u(2)**3+4*(z(1)*z(2))**2-176
 jac(4,1,1) = 2*z(1) +1
 jac(4,1,2)=1
 jac(4,2,1)=1+8*z(1)*z(2)**2
 jac(4,2,2)=8*z(2)*z(1)**2
fx(9)=x(1)**2+x(3)**2+x(6)**2+10*u(1)**2+u(2)*u(1)
     +2*z(1)*z(2)+120*u(1)-490+120*u(1)-490
fx(10)=x(2)**3+x(4)**2-x(5)**2+u(1)-150*u(2)+z(1)**3
       +z(2)**3+289
 jac(5,1,1)=120+u(2)+20*u(1)
 jac(5,1,2)=u(1)
 jac(5,2,1)=1
 jac(5,2,2)=-150
```

The system is partitioned as follows:

$$\mathbf{F}^1 = \begin{bmatrix} fx(1) \\ fx(2) \end{bmatrix} \quad \mathbf{x}^1 = \begin{bmatrix} x(1) \\ x(2) \end{bmatrix}$$

$$\mathbf{F}^2 = \begin{bmatrix} fx(3) \\ fx(4) \end{bmatrix} \quad \mathbf{x}^2 = \begin{bmatrix} x(3) \\ x(4) \end{bmatrix}$$

$$\mathbf{F}^3 = \begin{bmatrix} fx(5) \\ fx(6) \end{bmatrix} \quad \mathbf{x}^3 = \begin{bmatrix} x(5) \\ x(6) \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} fx(7) \\ fx(8) \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} z(1) \\ z(2) \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} fx(9) \\ fx(10) \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} u(1) \\ u(2) \end{bmatrix}$$

The solution to the system is

$$\mathbf{x}^1 = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \quad \mathbf{x}^2 = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \quad \mathbf{x}^3 = \begin{bmatrix} 5 \\ -3 \end{bmatrix}$$

$$\mathbf{z} = \begin{bmatrix} -3 \\ -2 \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

The matrix of equation (5.33) is

$$\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} = \begin{bmatrix} -0.018 & 0.004 \\ 1.184 & 0.152 \end{bmatrix}$$

which has eigenvalues of

$$\lambda_1 = -0.04099$$
$$\text{and} \quad \lambda_2 = 0.17465$$

Since $|\lambda_2| > |\lambda_1|$ then the convergence rate of the system is λ_2. This can be seen from the output of the Semi-Direct method for some chosen variables. (The stopping criterion is $e = 10^{-8}$).

x 1	deltax 1	x 2	deltax 2
0. 100000e+01	0. 204499e+01	0. 100000e+01	-0. 682876e+00
0. 304499e+01	-0. 940797e-01	0. 317124e+00	0. 376632e+01
0. 295091e+01	0. 450155e-01	0. 408344e+01	-0. 737360e-01
0. 299592e+01	0. 316198e-02	0. 400970e+01	-0. 767571e-02
0. 299909e+01	0. 762710e-03	0. 400203e+01	-0. 168599e-02
0. 299985e+01	0. 124398e-03	0. 400034e+01	-0. 281423e-03
0. 299997e+01	0. 220868e-04	0. 400006e+01	-0. 497403e-04
0. 300000e+01	0. 384271e-05	0. 400001e+01	-0. 866470e-05
0. 300000e+01	0. 671750e-06	0. 400000e+01	-0. 151429e-05
0. 300000e+01	0. 117299e-06	0. 400000e+01	-0. 264439e-06
0. 300000e+01	0. 204876e-07	0. 400000e+01	-0. 461868e-07
0. 300000e+01	0. 357820e-08	0. 400000e+01	-0. 806662e-08
0. 300000e+01	0. 624947e-09	0. 400000e+01	-0. 140887e-08

x 5	deltax 5	x 6	deltax 6
0. 100000e+01	0. 777409e+01	0. 100000e+01	-0. 385182e+01
0. 877409e+01	-0. 377328e+01	-0. 285182e+01	-0. 121813e+00
0. 500081e+01	-0. 967315e-02	-0. 297363e+01	-0. 244918e-01
0. 499114e+01	0. 787259e-02	-0. 299813e+01	-0. 143163e-02
0. 499901e+01	0. 792014e-03	-0. 299956e+01	-0. 370123e-03
0. 499981e+01	0. 160807e-03	-0. 299993e+01	-0. 595356e-04
0. 499997e+01	0. 270943e-04	-0. 299999e+01	-0. 106037e-04
0. 499999e+01	0. 477082e-05	-0. 300000e+01	-0. 184343e-05
0. 500000e+01	0. 831594e-06	-0. 300000e+01	-0. 322308e-06
0. 500000e+01	0. 145307e-06	-0. 300000e+01	-0. 562779e-07
0. 500000e+01	0. 253756e-07	-0. 300000e+01	-0. 982972e-08
0. 500000e+01	0. 443205e-08	-0. 300000e+01	-0. 171677e-08
0. 500000e+01	0. 774068e-09	-0. 300000e+01	-0. 299841e-09

x 9	deltax 9	x10	deltax10
0. 100000e+01	0. 199275e+01	0. 100000e+01	0. 119863e+01
0. 299275e+01	0. 830794e-02	0. 219863e+01	-0. 180298e+00
0. 300106e+01	-0. 100535e-02	0. 201833e+01	-0. 143212e-01
0. 300005e+01	-0. 368315e-04	0. 200401e+01	-0. 333959e-02
0. 300001e+01	-0. 118950e-04	0. 200067e+01	-0. 548953e-03
0. 300000e+01	-0. 184083e-05	0. 200012e+01	-0. 973105e-04
0. 300000e+01	-0. 330865e-06	0. 200002e+01	-0. 169375e-04
0. 300000e+01	-0. 573924e-07	0. 200000e+01	-0. 296060e-05
0. 300000e+01	-0. 100396e-07	0. 200000e+01	-0. 516981e-06
0. 300000e+01	-0. 175280e-08	0. 200000e+01	-0. 902966e-07
0. 300000e+01	-0. 306159e-09	0. 200000e+01	-0. 157705e-07
0. 300000e+01	-0. 534707e-10	0. 200000e+01	-0. 275438e-08
0. 300000e+01	-0. 933892e-11	0. 200000e+01	-0. 481061e-09

Example 5.4:

Consider another system of equations in 10 equations and 10 unknowns with the same partitions as example 5.3. The equations are:

```
fx(1)= x(1)**3 + 10*x(1) + x(2) -u(1)**2+2*u(2)**2-60
fx(2)= 2*(x(1)-x(2)-1)**3 -x(2) +6*u(1)*u(2) -16
*************************************************************
fx(3)=x(3)**3+10*x(3)+x(4) -u(1)**2+2*u(2)**2+27
fx(4)= 2*(x(3)-x(4)-1)**3 -x(4) +6*u(1)*u(2) +216
*************************************************************
fx(5)= x(5)**3 + 10*x(5) + x(6) -u(1)**2+2*u(2)**2-171
fx(6)= 2*(x(5)-x(6)-1)**3 -x(6) +6*u(1)*u(2) -725
*************************************************************
fx(7)= z(1)**3 + 10*z(1) + z(2) -u(1)**2+2*u(2)**2+60
fx(8)= 2*(z(1)-z(2)-1)**3 -z(2) +6*u(1)*u(2) -22
*************************************************************
fx(9)=-16+x(1)**2 + x(3)**2 + x(6)**2-u(1)+u(2)
      +z(1)+z(2)
fx(10)=-36+x(2)**3 + x(4)**2-x(5)**2+2*u(1)-4*u(2)
      +z(1)+z(2)
*************************************************************
```

The matrix of equation (5.33) is

$$\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} = \begin{bmatrix} 15.5 & 13.1 \\ 15.5 & 12.6 \end{bmatrix}$$

which has eigenvalues

$$\lambda_1 = 28.4 \qquad \lambda_2 = -0.295$$

but $|\lambda_1| > 1$ so the Semi-Direct method should diverge. A sample output for some variables confirms that.

x 1	deltax 1	x 2	deltax 2
0.100000e+01	0.204499e+01	0.100000e+01	-0.682876e+00
0.304499e+01	-0.128632e+02	0.317124e+00	0.433097e+01
-0.981824e+01	0.523756e+02	0.464810e+01	0.976279e+02
0.425574e+02	-0.666501e+04	0.102276e+03	0.307951e+04

x 3	deltax 3	x 4	deltax 4
0.100000e+01	-0.307643e+01	0.100000e+01	0.717040e+00
-0.207643e+01	-0.802425e+01	0.171704e+01	0.280868e+01
-0.101007e+02	0.526421e+02	0.452572e+01	0.977447e+02
0.425414e+02	-0.666500e+04	0.102270e+03	0.307952e+04

x 5	deltax 5	x 6	deltax 6
0.100000e+01	0.397782e+01	0.100000e+01	-0.412236e+01
0.497782e+01	-0.144102e+02	-0.312236e+01	0.764576e+01
-0.943236e+01	0.520101e+02	0.452340e+01	0.977409e+02
0.425777e+02	-0.666503e+04	0.102264e+03	0.307952e+04

x 7	deltax 7	x 8	deltax 8
0.100000e+01	-0.395403e+01	0.100000e+01	-0.668195e+01
-0.295403e+01	-0.724920e+01	-0.568195e+01	0.994114e+01
-0.102032e+02	0.527385e+02	0.425919e+01	0.979945e+02
0.425353e+02	-0.666499e+04	0.102254e+03	0.307954e+04

x 9	deltax 9	x10	deltax10
0.100000e+01	-0.368239e+02	0.100000e+01	-0.355206e+02
-0.358239e+02	0.424696e+03	-0.345206e+02	0.226454e+03
0.388873e+03	0.567326e+06	0.191934e+03	0.553316e+06
0.567715e+06	0.162841e+11	0.553508e+06	0.161863e+11

Example 5.5:

Consider the circuit of figure 4.6. Formulating the equations for the system produces the following equations:

```
fx(1) = 3*x(1) - u(1) -x(2)
fx(2) = x(2) - 2
fx(3) = x(2) -x(1) + x(3)
fx(4) = x(4) - u(1)**3 + u(1)
 jac(1,4,4) = 1
 jac(1,1,1) = 3
 jac(1,1,2) = -1
 jac(1,2,2) = 1
 jac(1,3,2) = 1
 jac(1,3,1) = -1
 jac(1,3,3) = 1
*****************************************************.
fx(6) = 2*x(6) -u(2) - x(7)
fx(7) = x(7) - (x(5) - x(6) )**3 + (x(5)- x(6))
 jac(2,3,3) = 1
 jac(2,3,1) = - 3*(x(5) - x(6))**2 +1
 jac(2,3,2) = 3*(x(5)-x(6))**2 -1
 jac(2,1,1) = 2
 jac(2,1,3)= 1
 jac(2,2,2) = 2
 jac(2,2,3) = -1
*****************************************************.
fx(8) = 3.5*z(1) -1.5*z(3) +z(2) - u(2)
fx(9) = z(2) - (z(1)-u(1))**3 + (z(1)-u(1))
fx(10) = z(3) - 2
fx(11) = z(3) - z(1) + z(4)
 jac(3,1,1)=3.5
 jac(3,1,2)=1
 jac(3,1,3)=-1.5
 jac(3,2,1)=-3*(z(1)-u(1))**2+1
 jac(3,2,2)=1
 jac(3,3,3)=1
 jac(3,4,1)=-1
 jac(3,4,3)=1
 jac(3,4,4)=1
*****************************************************.
fx(12) = x(4) +2*u(1) -x(1) -u(2) -z(2)
fx(13) = 5*u(2) -z(1) -u(1) -x(5) -x(6)
 jac(4,1,1) = 2
 jac(4,1,2) = -1
 jac(4,2,1) = -1
 jac(4,2,2) = 5
*****************************************************.
```

The rate of convergence is 0.2728 and the solution is:

$$x_1 = 0.9210 \quad x_2 = 2.0 \quad x_3 = -1.079$$
$$x_4 = -0.3189 \quad x_5 = 0.9441 \quad x_6 = 0.1504$$
$$x_7 = -0.2937 \quad z_1 = 1.115 \quad z_2 = -0.3085$$
$$z_3 = 2.0 \quad z_4 = -0.8848 \quad u_1 = 0.7629$$
$$u_2 = 0.5945$$

A sample output looks like this.

x 3	deltax 3	x 4	deltax 4
0. 100000e+01	-0. 200000e+01	0. 100000e+01	-0. 100000e+01
-0. 100000e+01	-0. 833179e-01	0. e+00	-0. 328093e+00
-0. 108332e+01	0. 491768e-02	-0. 328093e+00	0. 106387e-01
-0. 107840e+01	-0. 264129e-03	-0. 317454e+00	-0. 596618e-03
-0. 107866e+01	-0. 235310e-03	-0. 318051e+00	-0. 529096e-03
-0. 107890e+01	-0. 898942e-04	-0. 318580e+00	-0. 201525e-03
-0. 107899e+01	-0. 289776e-04	-0. 318782e+00	-0. 648911e-04
-0. 107902e+01	-0. 868616e-05	-0. 318847e+00	-0. 194447e-04
-0. 107903e+01	-0. 259601e-05	-0. 318866e+00	-0. 560933e-05
-0. 107903e+01	-0. 707341e-06	-0. 318872e+00	-0. 158323e-05
-0. 107903e+01	-0. 197058e-06	-0. 318873e+00	-0. 441068e-06
-0. 107903e+01	-0. 544608e-07	-0. 318874e+00	-0. 121897e-06
-0. 107903e+01	-0. 149766e-07	-0. 318874e+00	-0. 335215e-07
-0. 107903e+01	-0. 410570e-08	-0. 318874e+00	-0. 918961e-08
-0. 107903e+01	-0. 112333e-08	-0. 318874e+00	-0. 251429e-08
-0. 107903e+01	-0. 306960e-09	-0. 318874e+00	-0. 687055e-09

x11	deltax11	x12	deltax12
0. 100000e+01	-0. 180306e+01	0. 100000e+01	-0. 249954e+00
-0. 803056e+00	-0. 468189e-01	0. 750046e+00	0. 147530e-01
-0. 849874e+00	-0. 262227e-01	0. 764799e+00	-0. 792387e-03
-0. 876097e+00	-0. 646319e-02	0. 764007e+00	-0. 705929e-03
-0. 882560e+00	-0. 167506e-02	0. 763301e+00	-0. 269683e-03
-0. 884235e+00	-0. 442428e-03	0. 763031e+00	-0. 869327e-04
-0. 884678e+00	-0. 118137e-03	0. 762944e+00	-0. 260585e-04
-0. 884796e+00	-0. 317728e-04	0. 762918e+00	-0. 751804e-05
-0. 884828e+00	-0. 858639e-05	0. 762911e+00	-0. 212202e-05
-0. 884836e+00	-0. 232777e-05	0. 762909e+00	-0. 591175e-06
-0. 884839e+00	-0. 632356e-06	0. 762908e+00	-0. 163382e-06
-0. 884839e+00	-0. 172013e-06	0. 762908e+00	-0. 449298e-07
-0. 884840e+00	-0. 468306e-07	0. 762908e+00	-0. 123171e-07
-0. 884840e+00	-0. 127566e-07	0. 762908e+00	-0. 336998e-08
-0. 884840e+00	-0. 347612e-08	0. 762908e+00	-0. 920880e-09
-0. 884840e+00	-0. 947436e-09	0. 762908e+00	-0. 251440e-09

Example 5.6:

Consider the two stage transistor circuits of figure 5.4. The two transistors are replaced by their Ebers-Moll dc model and the circuit is partitioned as in figure 5.5. The current variables x_4, x_5, x_9, x_{10}, z_2, z_4 and z_6 are all added optionally. We may choose not to include some or all of them in the formulation if there is not a need for their values as an output. Assuming the diodes are ideal, their current i_d can be expressed as

$$i_d \cong I_s \, e^{\frac{qv_d}{kT}} \tag{5.34}$$

where v_d is the voltage across the diode, q is the electron charge, k is Boltzmann's constant, T is the absolute temperature and I_s is the reverse saturation current. Using room temperature (T=300 ° K) and $I_s = 10^{-14}$ A, equation (5.34) becomes

$$i_d \cong 10^{-14} \, e^{38.65 V_d} \tag{5.35}$$

The equations for the circuit are:

```
     F1:  fx(1) = (x(1)-u(2))*.001 - x(4) - .5*x(5)
          fx(2) = x(2)*.01 - x(5) + .98*x(4)
          fx(3) = (x(3)-u(1))*.001 + .02*x(4) + .5*x(5)
          fx(4) = x(4) - 1.e-14*exp(38.65*(x(3)-x(1)))
          fx(5) = x(5) - 1.e-14*exp(38.65*(x(3)-x(2)))
*************************************************************
     F2:  fx(6) = (x(6)-u(2))*.001 - x(9) - .5*x(10)
          fx(7) = x(7)*.01 - x(10) + .98*x(9)
          fx(8) = (x(8)-u(1))*.001 + .02*x(9) + .5*x(10)
          fx(9) = x(9) - 1.e-14*exp(38.65*(x(8)-x(6)))
          fx(10) = x(10) - 1.e-14*exp(38.65*(x(8)-x(7)))
*************************************************************
```

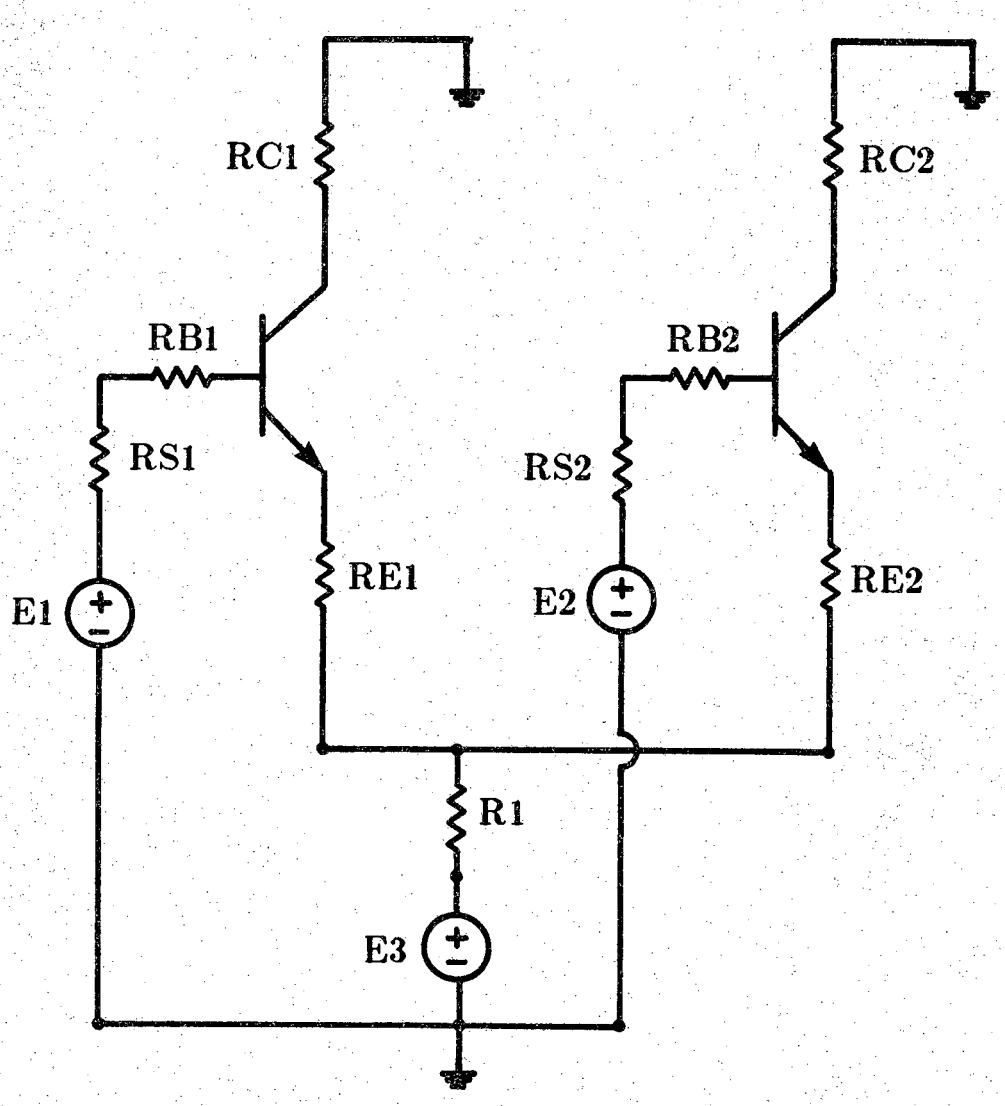


Figure 5.4: 2-stage transistor circuit

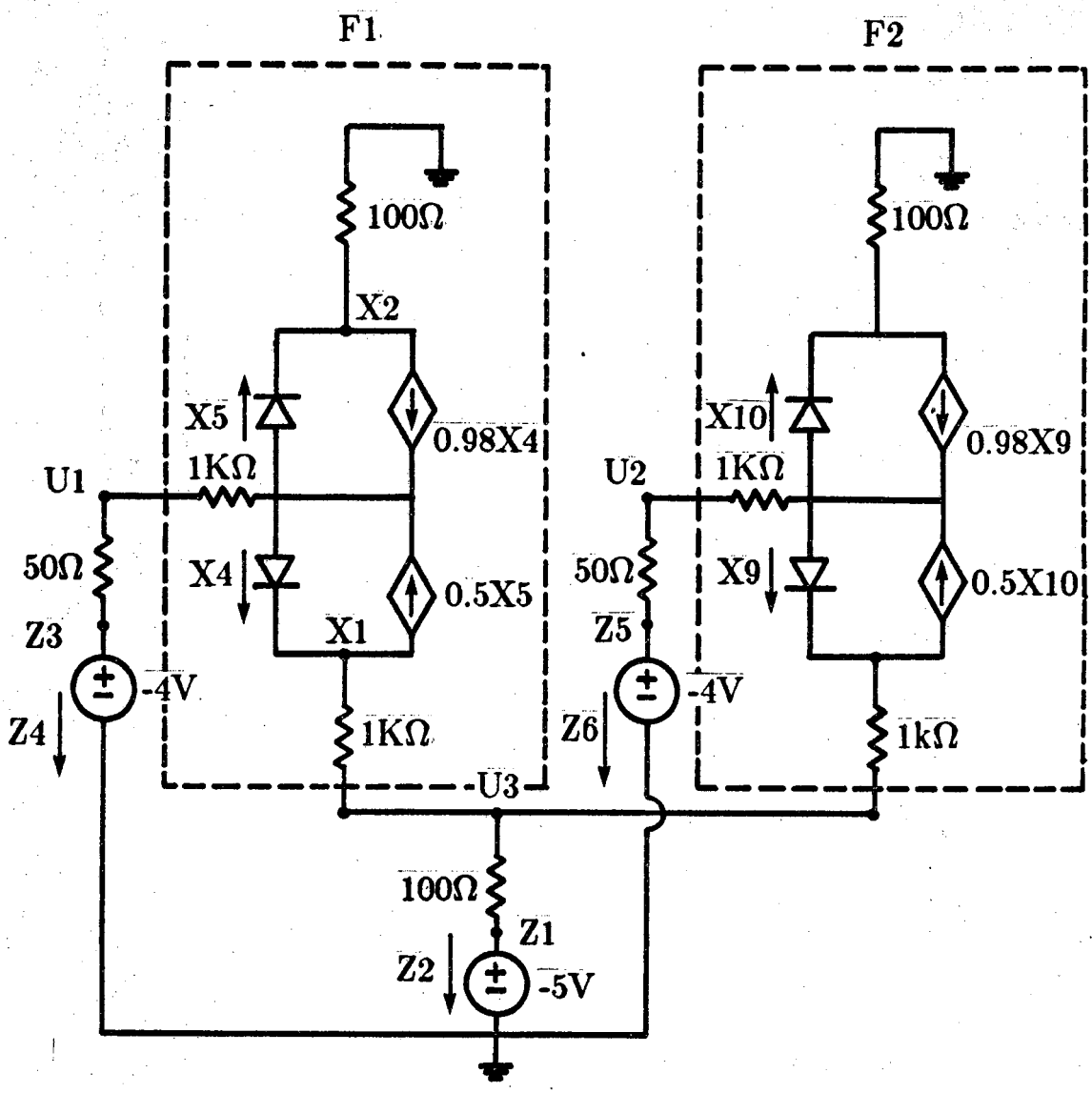


Figure 5.5: Labeled circuit

```
     H :  fx(11) = z(1) + 5
          fx(12) = (z(1)-u(3))/100.  + z(2)
          fx(13) = z(3) + 4
          fx(14) = (z(3)-u(1))/50.  + z(4)
          fx(15) = z(5) + 4
          fx(16) = (z(5)-u(2))/50.  + z(6)
*************************************************************
     G :  fx(17) = (u(1)-x(3))/1000.  + (u(1)-z(3))/50.
          fx(18) = (u(2)-x(8))/1000.  + (u(2)-z(5))/50.
          fx(19) = (u(3)-x(1))/1000.  + (u(3)-x(6))/1000.  +
                   (u(3)-z(1))/100.
*************************************************************
```

The analysis was done using a set of equations that combined the two resistors R_{B1} and R_{S1} as one 1050 ohm resistor, and the two resistors R_{B2} and R_{S2} as one 1050 ohm resistor. This is a case where an ideal voltage source is connected to a tearing node. This forces the currents through the source to be introduced as u variables. Figure 5.6 illustrates the new labeling. We now have 5 u variables and 2 z variables. The equations take the form:

```
     F1:  fx(1) = (x(1)-u(2))*.001 - x(4) - .5*x(5)
          fx(2) = x(2)*.01 - x(5) + .98*x(4)
          fx(3) = (x(3)-u(1))/1050.  + .02*x(4) + .5*x(5)
          fx(4) = x(4) - 1.e-14*exp(38.65*(x(3)-x(1)))
          fx(5) = x(5) - 1.e-14*exp(38.65*(x(3)-x(2)))
*************************************************************
     F1:  fx(6) = (x(6)-u(2))*.001 - x(9) - .5*x(10)
          fx(7) = x(7)*.01 - x(10) + .98*x(9)
          fx(8) = (x(8)-u(3))/1050.  + .02*x(9) + .5*x(10)
          fx(9) = x(9) - 1.e-14*exp(38.65*(x(8)-x(6)))
          fx(10) = x(10) - 1.e-14*exp(38.65*(x(8)-x(7)))
*************************************************************
     H :  fx(11) = z(1) +5
          fx(12) = (z(1)-u(2))*.01 +z(2)
*************************************************************
     G :  fx(13) = u(1) +4
          fx(14) = (u(2)-z(1))*.01 + (u(2)-x(1))*.001
                   + (u(2)-x(6))*.001
          fx(15) = u(3) +4
          fx(16) = (u(1) - x(3))/1050.  +u(4)
          fx(17) = (u(3) - x(8))/1050.  + u(5)
*************************************************************
```

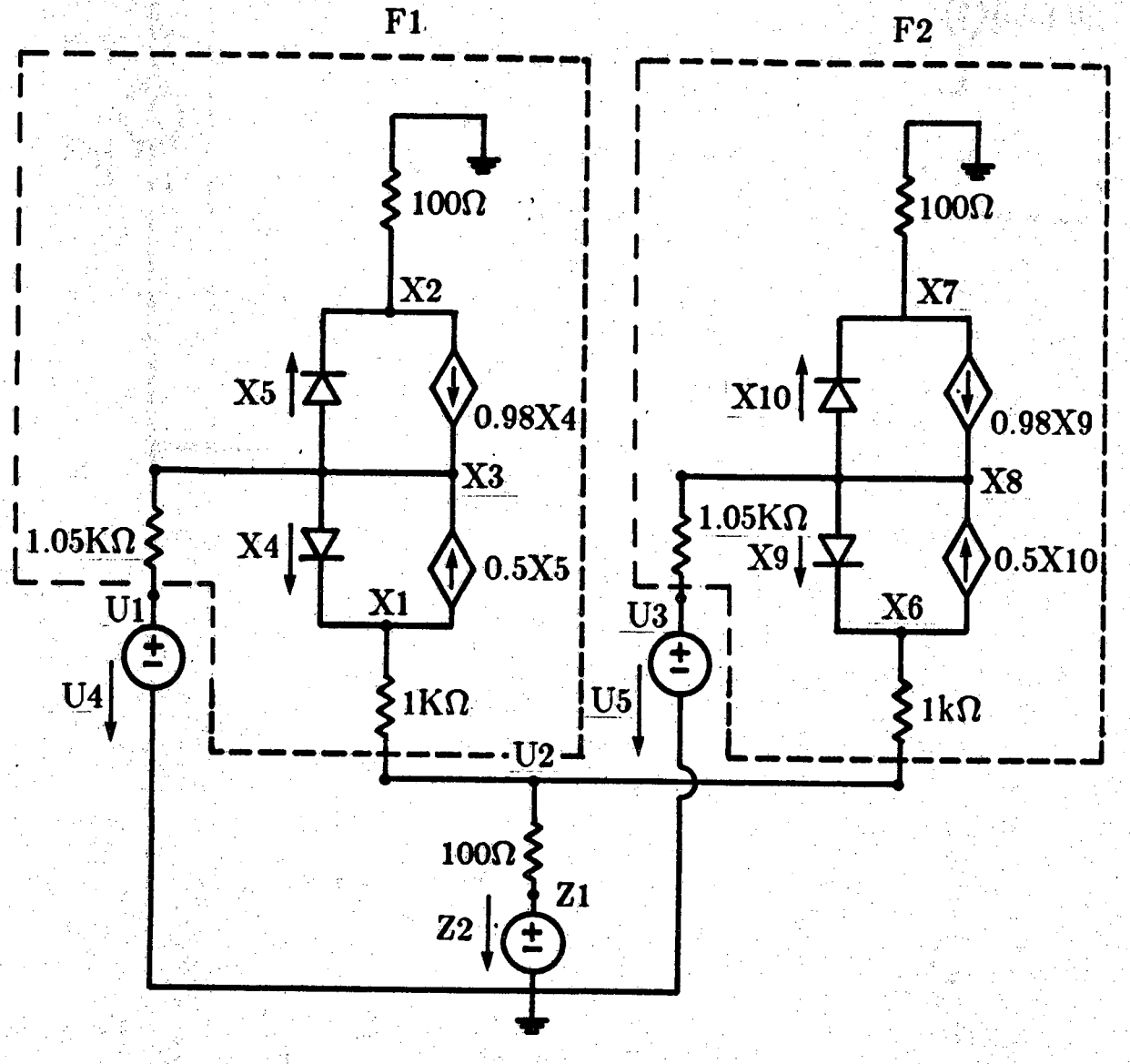


Figure 5.6: Model used in the analysis

The matrix equation (5.33) becomes

$$\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.01508 & 0.0159 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0.93\times10^{-3} & 0.16\times10^{-4} & 0 & 0 & 0 \\ 0.93\times10^{-3} & 0.18\times10^{-4} & 0 & 0 & 0 \end{bmatrix}$$

The only eigenvalue for the above matrix is

$$\lambda = 0.0158636$$

which has magnitude less than 1. The solution is as follows (apply to figure 5.6)

$$x_6 = x_1 = -4.631 \qquad x_7 = x_2 = -0.0301 \qquad x_8 = x_3 = -4.006$$
$$x_9 = x_4 = 0.00031 \qquad x_{10} = x_5 = 0.0$$
$$z_1 = -5.0 \qquad z_2 = 0.000612$$
$$u_1 = -4.0 \qquad u_2 = -4.939 \qquad u_3 = -4.0$$
$$u_5 = u_4 = -0.615\times10^{-5}$$

The output for a few selected variables are listed below.

x 6	deltax 6	x 7	deltax 7
0.100000e+01	0.287212e+00	0.100000e+01	-0.942558e+00
0.128721e+01	-0.107691e+02	0.574425e-01	0.484474e+00
-0.948189e+01	0.123014e+01	0.541917e+00	-0.296449e+00
-0.825176e+01	0.127127e+01	0.245468e+00	-0.104492e+00
-0.698049e+01	0.127141e+01	0.140976e+00	-0.103834e+00
-0.570907e+01	0.106323e+01	0.371414e-01	-0.834305e-01
-0.464584e+01	0.143581e-01	-0.462891e-01	0.159590e-01
-0.463148e+01	0.227182e-03	-0.303301e-01	0.212252e-03
-0.463126e+01	0.360390e-05	-0.301178e-01	0.335746e-05
-0.463125e+01	0.571726e-07	-0.301145e-01	0.532607e-07
-0.463125e+01	0.906994e-09	-0.301144e-01	0.844934e-09

x12	deltax12	x14	deltax14
0.100000e+01	-0.940000e+00	0.100000e+01	-0.495213e+01
0.600000e-01	-0.495213e-01	-0.395213e+01	-0.179485e+01
0.104787e-01	-0.179485e-01	-0.574698e+01	0.205023e+00
-0.746982e-02	0.205023e-02	-0.554196e+01	0.211878e+00
-0.541959e-02	0.211878e-02	-0.533008e+01	0.211902e+00
-0.330081e-02	0.211902e-02	-0.511818e+01	0.177206e+00
-0.118179e-02	0.177206e-02	-0.494097e+01	0.239302e-02
0.590265e-03	0.239302e-04	-0.493858e+01	0.378637e-04
0.614195e-03	0.378637e-06	-0.493854e+01	0.600649e-06
0.614574e-03	0.600649e-08	-0.493854e+01	0.952877e-08
0.614580e-03	0.952877e-10	-0.493854e+01	0.151166e-09

Example 5.7:

The transistor-switch circuit in figure 5.7 is considered in this example. This is a case where the convergence condition is marginally satisfied. So although convergence does occurs, the number of sweeps it takes becomes very large because of the slow convergence rate. The transistor is replaced by its Ebers-Moll model, and the ideal diode equation (equation (5.35)) is used in the formulation. The circuit is partitioned as illustrated in figure 5.8. The equations extracted from the circuit are:

```
F1: fx(1) = x(1)/100. + 1.e-4*dxdt(1) + .5*x(5) -x(4)
    fx(2) = (x(2)-u(2))/1000. + .98*x(4) - x(5)
    fx(3) = (x(3)-u(1))/50. + .02*x(4) + .5*x(5)
    fx(4)= x(4) - 1.e-14*exp(38.65*(x(3)-x(1)))
    fx(5)= x(5) - 1.e-14*exp(38.65*(x(3)-x(2)))
   ***************************************************
H : fx(6) = 2.e-6*(dzdt(1)-dudt(1)) + (z(1)-z(2))/1000.
    fx(7) = z(2) -.01
    fx(8) = z(3)  + (z(2)-z(1))/1000.
   ***************************************************
G : fx(9) = (u(1)-x(3))/50. + (u(1)-u(2))/100000.
          + u(1)/27000. + 2.e-6*(dudt(1)-dzdt(1))
    fx(10) = u(2) - 10
    fx(11) = (u(2)-u(1))/100000. + (u(2)-x(2))/1000.
           + u(3)
   ***************************************************
```

The matrix of equation (5.33) becomes (at t=0, dc solution)

$$\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}\mathbf{A}_{ss}^{-1}\mathbf{A}_{sr} = \begin{bmatrix} 0.9886 & 0 & 0 \\ 0 & 0 & 0 \\ -0.001 & 0.001 & 0 \end{bmatrix}$$

which has an eigenvalue

$$\lambda = 0.9886$$

The convergence rate of this circuit is very slow and reaching a dc-solution would take an access of 100 sweeps (where $e = 10^{-8}$). The Semi-Direct method in this case is very impractical.

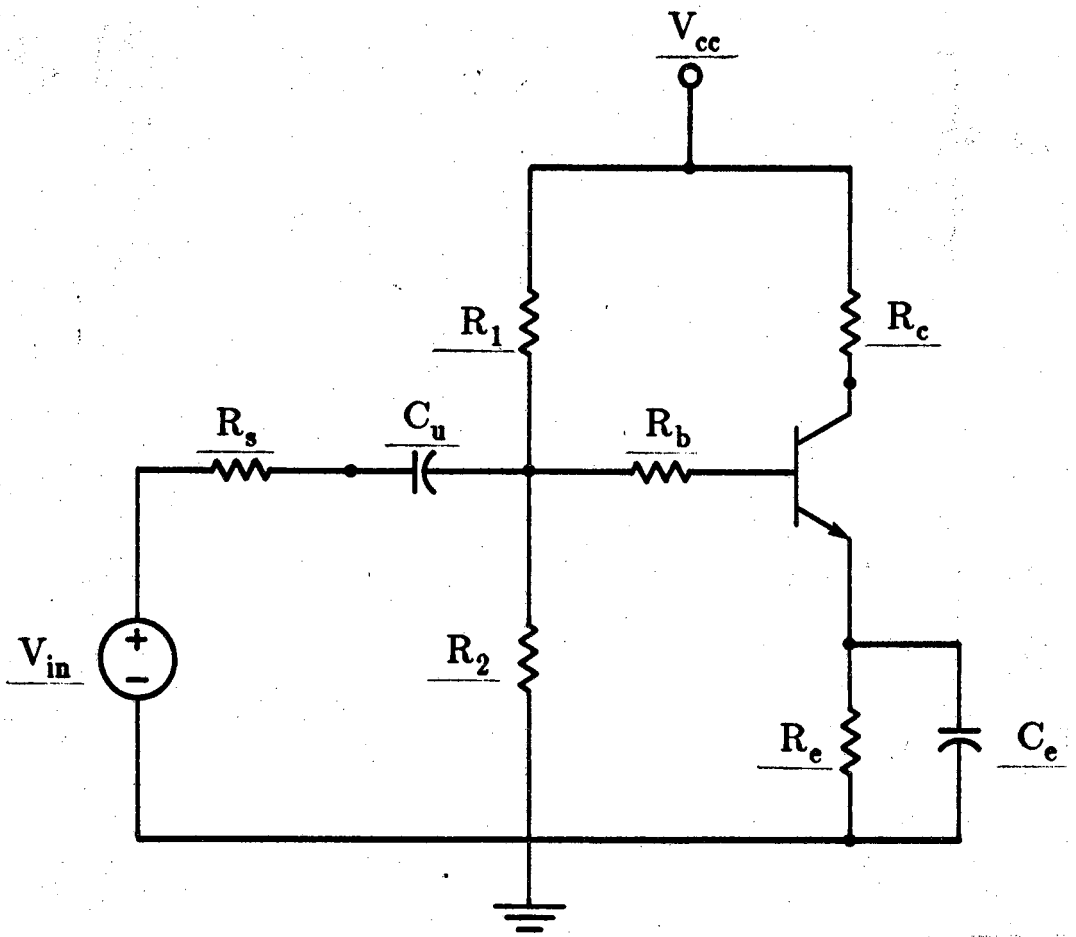


Figure 5.7: Transistor switch [4]

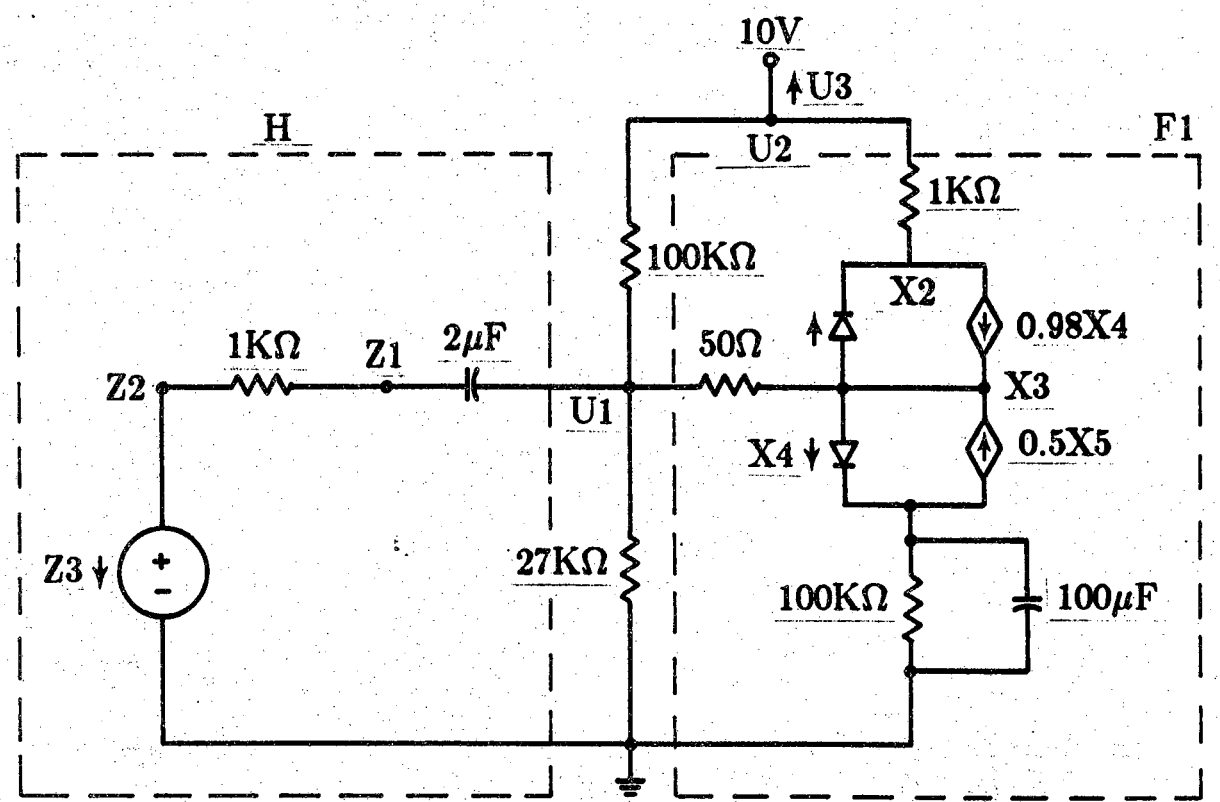


Figure 5.8: Labeled transistor switch circuit.

CHAPTER 6
COMPARISON BETWEEN THE NEWTON-RAPHSON AND THE SEMI-DIRECT METHODS

The previous chapters were devoted to the explanation of the features and theoretical aspects of the Semi-Direct method. The application of the method in circuit analysis is explored in the next two chapters. The dc-case is studied first because, as explained earlier in chapter 1 the Semi-Direct method is an incremental in time method. A transient analysis is a collection of dc-analysis cases at successive time points. The comparison with respect to the conventional (Newton-Raphson) method, which is currently used in most computer-aided analysis programs, is made through the use of the Fortran programs we developed for this study (Appendix). The comparison is made in two categories, computer storage requirements and execution time (CPU time).

6.1 Storage

The main storage requirement is storing the Jacobian matrix elements that result from tearing the circuit (Chapter 4). The matrix has the bordered block-diagonal form of figure 4.1. To solve the system using the Newton-Raphson algorithm for bordered block-diagonal dependency matrices (section 4.4), which is used throughout this thesis, we need only store the elements of the submatrices $\mathbf{A}_{ssi}$ $(1 \leq i, \leq m+1)$, $\mathbf{A}_{rr}$, $\mathbf{A}_{rs}$ and $\mathbf{A}_{sr}$. To solve the system

using the Semi-Direct method we only need to store the $\mathbf{A}_{ssi}$ and $\mathbf{A}_{rr}$ submatrices. Each $\mathbf{A}_{ssi}$ submatrix corresponds to a subcircuit and $\mathbf{A}_{rr}$ corresponds to the tearing node variables which can be considered as a subcircuit with no internal nodes and will be in the next discussion.

Assume we have m $\mathbf{F}^i$ subcircuits. The total number of subcircuits (including **H** and **G**, see figure 4.2) is then

$$k = m+2 \tag{6.1}$$

At this point let us consider equal size subcircuits each with n variables for a total number of variables for the torn circuit of kn. The dimensions of the Jacobian matrix would then be $(kn)^2$. The number of elements that are needed for the Semi-Direct method would be kn^2. For the Newton-Raphson method we would need to store the same kn^2 elements in addition to the elements of the submatrices $\mathbf{A}_{rs}$ and $\mathbf{A}_{sr}$ each of size n(kn-n). So the total number of elements needed for the Newton-Raphson method is

$$kn^2 + 2n(kn-n)$$

$$= kn^2 + 2n^2(k-1)$$

$$= n^2(k+2k-2)$$

$$= n^2(3k-2) \quad \text{elements}$$

The savings ratio that the Semi-Direct method exhibits over the Newton-Raphson method is calculated from

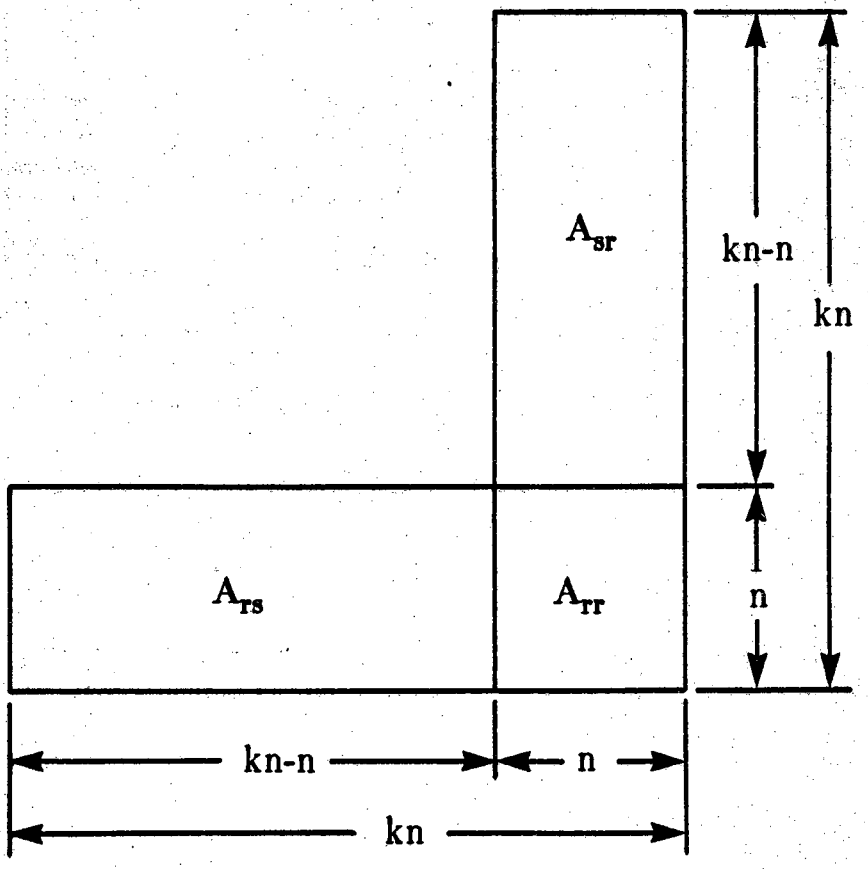


Figure 6.1: $\mathbf{A_{sr}}$ and $\mathbf{A_{rs}}$ dimensions.

$$\text{savings ratio} = \left(1 - \frac{\text{number of elements in Semi-Direct}}{\text{number of elements in Newton-Raphson}}\right) \tag{6.2}$$

So

$$\text{savings ratio} = 1 - \frac{kn^2}{n^2(3k-2)}$$

which can be written

$$\text{savings ratio} = 1 - \frac{k}{3k-2} \tag{6.3}$$

The savings for a large scale circuit where it is torn into a large number of subcircuits so that $3k >> 2$ we have

$$\text{savings ratio} \simeq 1 - \frac{1}{3} = 0.6667 \tag{6.4}$$

or 66.67%.

Now let us do an analysis on the savings ratio for a more general case where the number of variables per subcircuit varies. Let n_{max} be the number of variables in the largest subcircuit and n_{min} the number of variables in the smallest subcircuit. So for the Semi-Direct method we have

$$\text{Total number of elements to be stored} \leq kn_{max}^2 \tag{6.5}$$

and for the Newton-Raphson method we need to find the minimum size for the submatrices $\mathbf{A}_{rs}$ and $\mathbf{A}_{sr}$. From figure 6.1 we see that the size is minimal when $\mathbf{A}_{rr}$ is $n_{min} \times n_{min}$ because we have in the case of equal size subcircuits

$$\text{size of } \mathbf{A}_{rs} = (kn-n)n$$

taking the derivative of the above expression we get

$$\frac{d(kn^2-n^2)}{dn} = 2n(k-1) \tag{6.6}$$

from equation (6.1) we know $k \geq 2$, so equation (6.6) is always positive for positive n. Which means the size of $\mathbf{A}_{rs}$ increases as n increases. So the minimum size of $\mathbf{A}_{rs}$ occurs when n is minimal. The same argument applies to $\mathbf{A}_{sr}$ since they always have the same dimensions. So for the Newton-Raphson method we have

$$\text{Total number of elements to be stored} \geq kn_{max}^2 + 2n_{min}^2(k-1) \tag{6.7}$$

So the savings ratio for a certain circuit can be approximated by

$$\text{Savings Ratio} \cong 1 - \frac{kn_{max}^2}{kn_{max}^2 + 2n_{min}^2(k-1)} \tag{6.8}$$

for the case where $k >> 1$ we have

$$\text{Savings Ratio} \cong 1 - \frac{n_{max}^2}{n_{max}^2 + 2n_{min}^2}$$

or

$$\text{Savings Ratio} \cong \frac{2n_{min}^2}{n_{max}^2 + 2n_{min}^2} \tag{6.9}$$

Equation (6.9) is optimized when $n_{min} = n_{max}$ where the savings ratio is 0.6667 which is the same case where all subcircuits are of the same size. For the application in a computer program we have to dimension the arrays that store the elements of the submatrices of the dependency matrix. These arrays have a fixed dimension that is equal for all the submatrices (subcircuits). So the users instructions would include a maximum subcircuit size (number of variables/subcircuit) that can be inputted. The storage savings ratio is then

fixed for a certain computer program and approaches 66.67% as the number of subcircuits becomes large.

for a certain computer program to implement the above algorithms a constant number of variables should be declared to store the values of the functions in equations (4.5), (4.6) and (4.7). In addition to some intermediate result variables and loop counters. This constant number of variables is approximately equal for implementation of both methods. The effect of the constant on the savings ratio decreases as the programs are designed to solve large circuits, but it will have a significant effect in decreasing it for programs designed for smaller circuits.

The following examples will illustrate the above. The discussions are applied to IBM 360 series computers [8,16].

Example 6.1.

First, let us consider a program designed to handle a maximum of 10 subcircuits at a maximum of 5 variables per subcircuit for a total of 50 variables. Using double precision floating point operations, each floating point variable would need 64-bits to store, that is 8-bytes. The integers would require 32-bits that is 4-bytes.

The storage space needed, that is common in both programs can be approximated as follows:

50 circuit variables at 8-bytes for a total of	400-bytes
50 function values at 8-bytes for a total of	400-bytes
50 error variables at 8-bytes for a total of	400-bytes

20 miscellaneous at 8-bytes for a total of	160-bytes
40 integers at 4-bytes each	160-bytes
Total	1520-bytes

Including Jacobian matrix elements:

The Semi-Direct method needs

10×5^2 function values at 8-bytes for a total of	2200-bytes
So total bytes needed	3520-bytes

The Newton-Raphson method needs:

5^2 (3 x 10-2) function values at 8-bytes for a total of	5600-bytes
So Total bytes needed	7120-bytes

which gives us a savings ratio of

$$1 - \frac{3520}{7120} = 0.5056$$

instead of 0.6667 as expected earlier due to the 1520-bytes added to both computations. The programs developed for this thesis, (see Appendix), has the limitations described in the above example. The actual storage the programs reserve for the data are as follows:

Semi-Direct	3772-bytes
Newton-Raphson	7212-bytes

So the savings ratio is

$$\text{savings ratio} = 1 - \frac{3772}{7212} = 0.4770$$

which is very close to the value predicted above.

Example 6.2:

Now let us increase the maximum number of subcircuits allowed to 1000 and the number of variables per subcircuit allowed to 100 for a total of 100,000 variables. The storage space needed can be approximated as follows:

100,000	circuit variables	800,000-bytes
100,000	function values	800,000-bytes
100,000	error variables	800,000-bytes
20	miscellaneous	160-bytes
100,030	integers	400,120-bytes
	Total	2,800,280-bytes

Including the Jacobian matrix elements:

The Semi-Direct methods needs:

$1{,}000 \times 100^2$ function values	80,000,000-bytes
Total needed	82,800,280-bytes

The Newton-Raphson method needs:

100^2 (3×1000−2) function values	239,840,000-bytes
Total needed	242,640,280-bytes

which gives us a savings ratio of

$$1 - \frac{82,800,280}{242,640,280} = 0.6588$$

This figure is a lot closer to 0.6667 as predicted earlier.

The two examples above illustrate the savings advantage that the Semi-Direct would have in an ideal case. From a more practical point of view we can study the results of analyzing the circuits of figures 6.1, 6.2 and 6.3 in section 6.3 using the programs developed during this study.

6.2 Execution Time

There are two criteria to be considered when studying execution time. One that depends on the circuit properties and the other on the computer program structure. They are:

1) The number of iterations or sweeps, and

2) Execution time of each iteration or sweep, respectively.

In the dc case the number of iterations or sweeps depends on the convergence rate of the method and the initial guess. For the Newton-Raphson method the rate of convergence is quadratic for almost all practical circuits as long as the initial guess is within a certain range of the solution. If the initial

guess is somewhat far from the solution quadratic convergence will not take effect until the iteration reach some local range of the solution. An example of that is the circuit of figure 5.6. Results from the Newton-Raphson solution is illustrated in example 6.3. Examining the exponent of the delta x column we noticed that the quadratic convergence starts taking effect at the 17th iteration, (initial guess is 0 for all variables). The Semi-Direct method has a linear convergence rate determined by the eigenvalue of the matrix $\mathbf{P^{-1}Q}$ (see section 5.2). Looking at example 6.3 the convergence rate is 0.01572 and takes effect at the 4th sweep of the Semi-Direct method. In general the convergence rate we have discussed for both methods is a local one and takes effect when Δx^i became less than 0.01 where $\Delta x^i = x^{i+1} - x^i$ and x^i is the value of the slowest convergent variable of a given system. So the main consideration is how close does the initial guess have to be in order that the linearly convergent Semi-Direct method take as many sweeps (or even less) as the quadratically convergent Newton-Raphson method would take in iterations? The answer depends on the circuit and its rate of convergence. For the case of example 6.3 the Semi-Direct method converges in 8 sweeps while the Newton-Raphson takes 20 iterations to reach the same solution. For the case of example 6.4 the Semi-Direct method converges in 17 sweeps while the Newton-Raphson converges in 13 iterations. For local convergence the Newton-Raphson method is a faster convergent method than the Semi-Direct, but they can be compatible if we start at the very close initial guess where only one or two iterations are needed to reach a solution within an error limit of the true solution. This is the case in transient analysis, where for a small enough time step the solution at one time point is within the local area of the solution at the next time point. This criterion is illustrated through examples 6.3 and 6.4.

The time that each iteration takes depends more on the structure of the algorithm and the problem size and is independent of the convergence rate.

We have to note that the time the Semi-Direct method consumes during one sweep could vary from sweep to sweep in the same problem. The reason is in steps 2 and 3 of the algorithm (see section 5.1), we use the Newton-Raphson method to solve for the variables the subcircuits. These subiterations could vary depending on the closeness of the initial guess. It was found that in general a Semi-Direct sweep and a Newton-Raphson iterations take the same amount of execution time when steps 2 and 3 of the Semi-Direct method take on the average 1.5 subiterations. So if the Semi-Direct method takes the same number of sweeps as the Newton-Raphson would take in iterations, with only one subiterations per sweep, the Semi-Direct method would have a time savings advantage, because of the added calculations associated with $\mathbf{A}_{rs}$ and $\mathbf{A}_{sr}$ (see section 4.4).

Example 6.3:

Consider the 2-stage transistor circuit of figure 5.6. If we start at an initial guess of 0 for all the variables, the Semi-Direct sweeps would converge to the solution ($e=10^{-8}$) after 8 sweeps. On the other hand it would take the Newton-Raphson method 20 iterations with the same initial guess. Each Newton-Raphson iteration takes ~0.0966 seconds. A Semi-Direct sweep takes ~0.1625 seconds with no limit on the number of subiterations. So for this case

Total CPU time Semi-Direct $\simeq$	1.300 seconds
Newton-Raphson $\simeq$	1.932 seconds

For the case where we allow only 2 subiterations per sweep, each sweep would take ~0.1167 seconds, but now it takes 12 sweeps to converge to the solution. So

Total CPU time, Semi-Direct $\simeq$ 1.400 seconds

For the case where we allow only 1 subiteration per sweep, each sweep would take ~0.0582 seconds, but it takes 18 sweeps to converge to the solution.

Total CPU time, Semi-Direct $\simeq$ 1.0476 seconds

If we choose an initial guess close enough to the true solution so that both methods start their local convergence rate properties. The initial guess in this case is 0.1% away from the true solution. The Semi-Direct method results are listed in table 6.1 below.

Table 6.1: Semi-Direct results (1).

# substitutions	# sweeps	Total time (seconds)
unlimited	6	0.9750
2	6	0.7002
1	6	0.3492

The Newton-Raphson method on the other hand takes 3 iterations to converge, and the

Total CPU time, Newton-Raphson $\simeq$ 0.2898 seconds

which is less than any of the time figures for the Semi-Direct method.

Now we examine another aspect. Let us reduce the incremental stopping criterion from $e = 10^{-8}$ to $e = 10^{-4}$, which for most analysis purposes is a sufficient figure. The initial guess is still 0.1% away from the time solution. The Semi-Direct results are listed in table 6.2 below.

Table 6.2: Semi-Direct results (2).

# substitutions	# sweeps	Total time (seconds)
unlimited	3	0.4875
2	3	0.3501
1	3	0.1746

The Newton-Raphson method takes 2 iterations in this case for a total CPU time of 0.1932 seconds. In this case the Semi-Direct method is faster.

A summary of the results for example 6.3 are listed in table 6.3. The local convergence rate does not take effect for either method until the initial guess is within ~0.2% of the true solution.

For an initial guess outside the local area of convergence for a system, either method could have the speed advantage. That depends very much on the problem itself. In the case of example 6.3 the Semi-Direct method had the edge while in the next example the Newton-Raphson method will be faster. This thesis is concerned with the convergence in a local area of the solution, since that is the case in transient analysis. Let us study another example.

Table 6.3: CPU times for example 6.3

		Incremental stopping criterion		
Initial Guess (app)	Method	10^{-8}	10^{-6}	10^{-4}
1%	S-D	0.3492	0.2910	0.2328
	N-R	0.3864	0.2898	0.2898
0.1%	S-D	0.3492	0.2910	0.1746
	N-R	0.2989	0.2898	0.1932

Example 6.4:

Consider the following 10 equation in 10 unknowns.

```
fx(1)= x(1)**3 + 10*x(1) + x(2) -u(1)**2+2*u(2)**2-60
fx(2)= 2*(x(1)-x(2)-1)**3 -x(2) +6*u(1)*u(2) -16
*************************************************************
fx(3)=x(3)**3+10*x(3)+x(4) -u(1)**2+2*u(2)**2+27
fx(4)= 2*(x(3)-x(4)-1)**3 -x(4) +6*u(1)*u(2) +216
*************************************************************
fx(5)= x(5)**3 + 10*x(5) + x(6) -u(1)**2+2*u(2)**2-171
fx(6)= 2*(x(5)-x(6)-1)**3 -x(6) +6*u(1)*u(2) -725
*************************************************************
fx(7)= z(1)**3 + 10*z(1) + z(2) -u(1)**2+2*u(2)**2+60
fx(8)= 2*(z(1)-z(2)-1)**3 -z(2) +6*u(1)*u(2) -22
*************************************************************
fx(9)=281+x(1)**2 + x(3)**2 + x(6)**2-100*u(1)+u(2)
      +z(1)+z(2)
fx(10)=156+x(2)**3 + x(4)**2-x(5)**2+2*u(1)-100*u(2)
      +z(1)+z(2)
*************************************************************
```

The system is partitioned as illustrated in example 5.3.

For the case where e = 10^{-8} and we start at a far initial guess (0's), the Semi-Direct method converges in 17 sweeps while the Newton-Raphson method converges in 13 iterations.

Total CPU time, Semi-Direct $\simeq$ 1.5 seconds

Newton Raphson $\simeq$ 0.4 seconds

Any attempt to reduce the number of subiterations did not improve the time for the Semi-Direct method. The first two sweeps needed at least 3 subiterations in order to reach convergence. Let us consider a local area of the solution, where the Semi-Direct method only needs 1 subiteration during a sweep. CPU times are

CPU time/sweep $\simeq$ 0.0281

CPU time/iteration $\simeq$ 0.0332

The results from this example are listed in table 6.4. For the Semi-Direct method to exhibit faster convergence, we needed an initial guess and an incremental criterion where it would only take both methods only one iteration or sweep to converge to the true solution.

We know that the Newton-Raphson method has, in general, quadratic convergence. Then for an initial guess 10^{-q} away from the solution, the ith iteration will be $k^{(2^i-1)}\cdot 10^{-2^i q}$ away from the solution, (see Section 2.3). The convergence rate for the Semi-Direct method is the magnitude of the largest eigenvalue (in magnitude) of the matrix of equation (5.33), call it λ. Then for an initial guess 10^{-q} away from the solution the jth sweep will be $|\lambda|^j 10^{-q}$ away from the true solution. Assuming a specified increment $e=10^{-\ell}$, the iterations or sweeps will stop when

Table 6.4: CPU times for example 6.4.

		Incremental stopping criterion		
Initial Guess (app)	Method	10^{-3}	10^{-5}	10^{-4}
1%	S-D	0.3653	0.2529	0.1967
	N-R	0.1328	0.0996	0.0996
0.1%	S-D	0.3372	0.1967	0.1405
	N-R	0.0996	0.0996	0.0664
0.01%	S-D	0.2810	0.1405	0.0843
	N-R	0.0996	0.0664	0.0064
0.001%	S-D	0.2810	0.0281	0.0281
	N-R	0.0332	0.0332	0.0332

$$k^{(2^i-1)} \cdot 10^{-2^i q} \leq 10^{-\ell} \tag{6.10}$$

and

$$|\lambda|^j \cdot 10^{-q} \leq 10^{-\ell} \tag{6.11}$$

respectively. If each iteration takes t_i seconds and each sweep takes t_j seconds of CPU time to run, then for the Semi-Direct method to be faster than the Newton-Raphson method we need

$$t_j j < t_i i \tag{6.12}$$

where j = number of sweeps and i = number of iterations. We can rewrite this requirement as

$$j < \frac{t_i}{t_j} i \tag{6.13}$$

Since we don't want to give up any accuracy we have to require that

$$|\lambda|^{j}10^{-q} \leq k^{(2^i-1)}10^{-2^iq} \tag{6.14}$$

that is

$$|\lambda|^{j} \leq \frac{k^{(2^i-1)}10^{-2^iq}}{10^{-q}} \tag{6.15}$$

which finally reduces to

$$|\lambda|^{j} \leq k^{(2^i-1)}10^{(1-2^i)q} \tag{6.16}$$

If equations (6.13) and (6.16) are satisfied, and the initial guess is within the local convergence area, the Semi-Direct method will converge faster than the Newton-Raphson method.

Equations (6.13) and (6.16) involve variables that are related to the Newton-Raphson method. An expression that only involves Semi-Direct method variables may be derived.

Assume we have an absolute maximum error requirement of $10^{-\ell}$, and an initial guess that is 10^{-q} away from the true solution. If we require that

1) the initial guess is within the local convergence area for the Semi-Direct method,

2) only one subiteration per block in each sweep is allowed, and

3) the method converges to the solution in only one sweep,

the Semi-Direct method guarantees a faster convergence than the Newton-Raphson method. The reason is the time one Semi-Direct sweep takes is less than the time of one Newton-Raphson iteration, and either method needs at least one sweep or iteration to reach the solution if $10^{-q} > 10^{-\ell}$. After one sweep the absolute error becomes $|\lambda| \cdot 10^{-q}$, which has to be within the error limit $10^{-\ell}$, that is

$$|\lambda| \cdot 10^{-q} < 10^{-\ell} \tag{6.17}$$

or

$$|\lambda| < 10^{q-\ell} \tag{6.18}$$

As an easy test, we choose to use the Semi-Direct method, if the initial guess is such that $10^{q-\ell}$ is in the order of magnitude of $|\lambda|$. An example case is when $q-\ell = -1$, that is when equation's (5.33) largest (in magnitude) eigenvalue has a magnitude in the order of 10^{-1}.

CHAPTER 7
TRANSIENT ANALYSIS

For performing a transient analysis on a circuit we need to model the dynamic elements (capacitors and inductors) of the circuit at each time point. Two approaches can be taken:

1) Replacing each of the dynamic elements by a model that incorporates static elements (resistors and sources). The values of these elements are then updated at each successive time point. Section 4.2 and figures 4.3 and 4.4 explain and show the models used for capacitors and inductors.

2) Expressing the inductor voltages and the capacitor currents in the equations using

$$v_L = L \frac{di_L}{dt} \tag{7.1}$$

and

$$i_c = C \frac{dv_c}{dt} \tag{7.2}$$

Then the computer algorithm would incorporate some numerical integration formula to replace the two derivatives in equations (7.1) and (7.2). This approach is more user oriented and is implemented in

the programs developed for this thesis.

7.1 Integration Methods

The numerical integration formulae used are the first order and second order backward differentiation formulae (BDF). A predicator formula is also used to improve the initial guess at the next time point, and a truncation error estimation formula [1].

So for a system characterized by

$$\mathbf{f}(\mathbf{x},\dot{\mathbf{x}},t) = 0 \tag{7.3}$$

$\dot{\mathbf{x}}$ can be approximated using the BDF. The equation to be solved at a certain time point t_n then becomes

$$\mathbf{f}(\mathbf{x}(t_n)) = 0 \tag{7.4}$$

A suitable numerical analysis method like the Newton-Raphson method or the Semi-Direct method could then be used to solve equation (7.4) for $\mathbf{x}(t_n)$.

The first order BDF, sometime referred to as backward Euler formula, estimates $\dot{x}(t)$ at some time point t_n as

$$\dot{x}(t_n) \simeq \frac{1}{t_n - t_{n-1}} \left[x(t_n) - x(t_{n-1}) \right] \tag{7.5}$$

where t_{n-1} is the previous time point. $(t_n - t_{n-1})$ is the time step "h" taken. The second order BDF has the form

$$\dot{x}(t_n) \simeq \left[\frac{1}{t_n - t_{n-1}} + \frac{1}{t_n - t_{n-2}} \right] x(t_n)$$

$$- \left[\frac{t_n - t_{n-2}}{(t_n - t_{n-1})(t_{n-1} - t_{n-2})}\right] x(t_{n-1})$$

$$+ \left[\frac{t_n - t_{n-1}}{(t_n - t_{n-2})(t_{n-1} - t_{n-2})}\right] x(t_{n-2}) \qquad (7.6)$$

For the case where we are taking uniform time steps, that is, $t_n - t_{n-1} = t_{n-1} - t_{n-2} = h$, equation (7.6) becomes

$$\dot{x}(t_n) \simeq \frac{1}{h}\left[1.5x(t_n) - 2x(t_{n-1}) + 0.5x(t_{n-2})\right] \qquad (7.7)$$

To help the numerical analysis algorithm used to solve equation (7.4) a predicator formula can be used to predict the value of $\mathbf{x}(t_{n+1})$. This would produce a closer initial guess which implies a faster convergence.

The first order BDF predicator formula can be written as

$$x^p(t_n) = \frac{t_n - t_{n-2}}{t_{n-1} - t_{n-2}} x(t_{n-1}) + \frac{t_n - t_{n-1}}{t_{n-2} - t_{n-1}} x(t_{n-2}) \qquad (7.8)$$

For the uniform time step case this becomes.

$$x^p(t_n) = 2x(t_{n-1}) - x(t_{n-2}) \qquad (7.9)$$

The second order BDF predicator formula is

$$x^p(t_n) = \frac{(t_n - t_{n-2})(t_n - t_{n-3})}{(t_{n-1} - t_{n-2})(t_{n-1} - t_{n-3})} x(t_{n-1})$$

$$+ \frac{(t_n - t_{n-1})(t_n - t_{n-3})}{(t_{n-2} - t_{n-1})(t_{n-2} - t_{n-3})} x(t_{n-2})$$

$$+ \frac{(t_n - t_{n-1})(t_n - t_{n-2})}{(t_{n-3} - t_{n-1})(t_{n-3} - t_{n-2})} x(t_{n-3}) \qquad (7.10)$$

which reduces, for uniform time steps, to

$$x^{p}(t_n) = 3x(t_{n-1}) - 3x(t_{n-2}) + x(t_{n-3}) \tag{7.11}$$

The truncation error may be estimated using first order BDF as

$$e(t_n) \simeq \frac{t_n - t_{n-1}}{t_n - t_{n-2}} \left[x(t_n) - x^{p}(t_n) \right] \tag{7.12}$$

or for uniform time steps as

$$e(t_n) \cong \frac{1}{2} \left[x(t_n) - x^{p}(t_n) \right] \tag{7.13}$$

For the second order BDF, the truncation error estimate is

$$e(t_n) \cong \frac{t_n - t_{n-1}}{t_n - t_{n-3}} \left[x(t_n) - x^{p}(t_n) \right] \tag{7.14}$$

or for the uniform time step

$$e(t_n) \cong \frac{1}{3} \left[x(t_n) - x^{p}(t_n) \right] \tag{7.15}$$

This estimation of the error at a time point n is very helpful in getting a better estimate of the derivative using the BDF formulae. The aspect to notice in these formula is that the smaller the time step is the better the estimate is.

7.2 Applications of the Semi-Direct Method

The transient analysis algorithm developed using the Semi-Direct method implements the above formula to estimate the capacitor currents and the inductor voltages.

Example 7.1:

Consider the RC circuit in figure 7.1. The circuit has essentially two constants, one effective for the rise part and one for the fall part. Figure 7.2 illustrates the labeled variables and the values of the elements. The capacitor currents are expressed using equation (7.2). The derivative is approximated at each time point by the 2nd order BDF as in equation (7.7). The equations extracted from the circuit are:

```
F1: fx(1) = 3.e-9*dxdt(1)+x(1)/500. +2.e-6*(dxdt(1)-dxdt(2))
    fx(2) = 2.e-6*(dxdt(2)-dxdt(1)) + (x(2)-u(1))/1500.
    ************************************************************
H : fx(3) = z(1) - .1
    fx(4) = .02*(z(1)-u(1)) + z(2)
    ************************************************************
G : fx(5) = .02*(u(1) - z(1)) +(u(1) - x(1))/1500.
    ************************************************************
```

The rise time constant is determined using

$$R_2C_2 = 1.5 \text{ ms}$$

while the fall time constant is approximated using

$$(R_1+R_S)C_1 = 3.1 \text{ ms}$$

The output for the transient analysis of this circuit using both, the Newton-Raphson method and the Semi-Direct method are listed in the appendix. The rise analysis is made from t = 0 up to t = 10 μs using a time step h = 0.1 μs. The fall analysis is made from t = 0 up to t = 20 ms using a time step h = 0.2

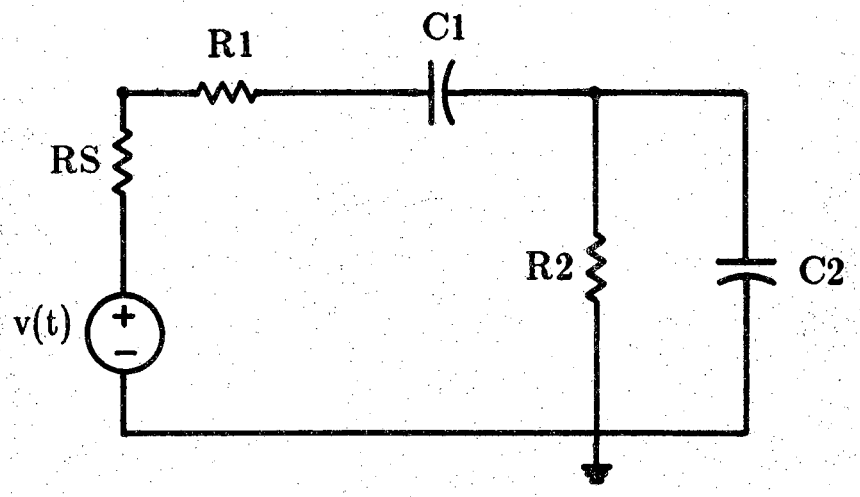


Figure 7.1. 2-time constant RC circuit

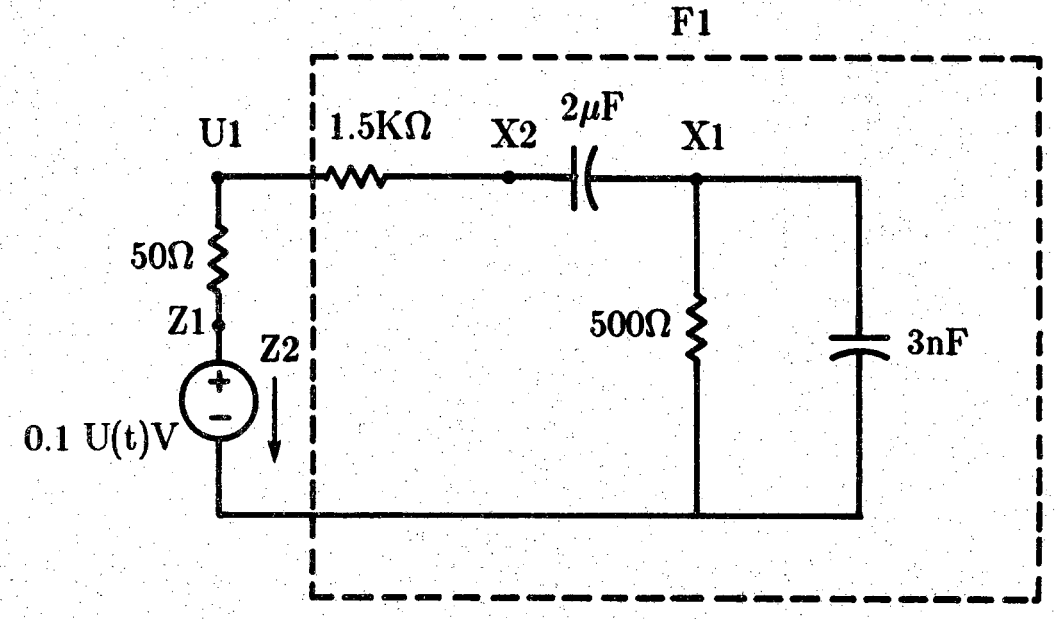
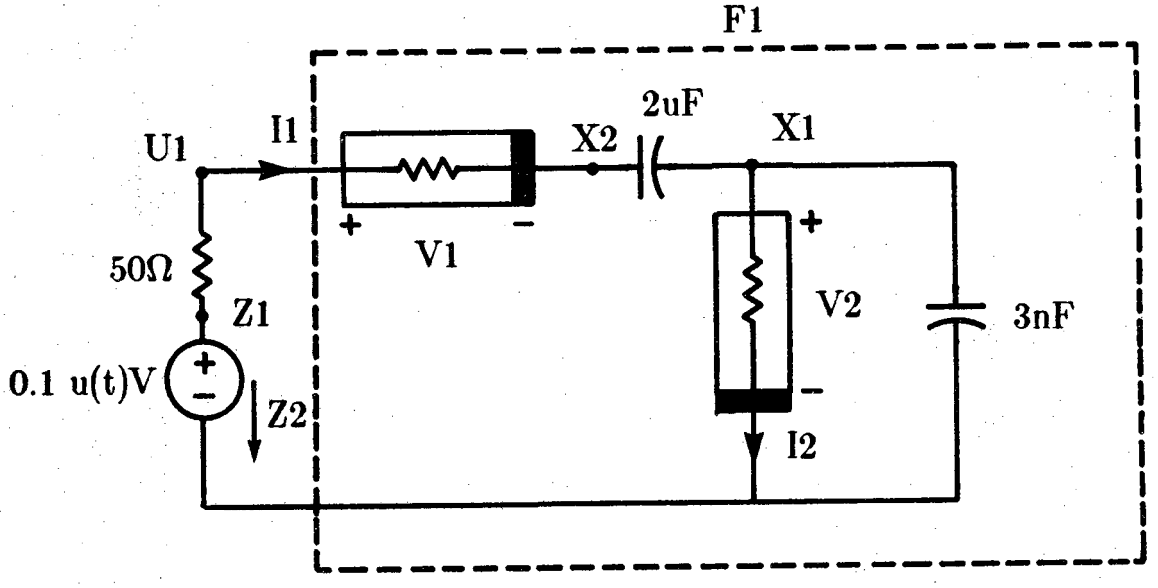


Figure 7.2. RC circuit with variables labeled and element values



$I1 = (1500)^{-1}(V1-V1^3)$
$I2 = (500)^{-1}(V2-V2^3)$

Figure 7.3. RC circuit with nonlinear elements

ms. Since the equations are linear at any time point, the Semi-Direct method takes less time to execute than the Newton-Raphson method. Only one sweep is needed at each time point and one subiteration internally. Only one iteration is needed, also, but a sweep takes less time than an iteration (1 subiteration/sweep is the case here).

Example 7.2:

Now let us introduce some nonlinearity in the RC circuit of figure 7.1. The new nonlinear circuit is introduced in figure 7.3. The currents through the nonlinear resistors are given by

$$i = R^{-1}(v_1 - v_1^3).$$

The element values are illustrated in figure 7.3. The output using both, the Newton-Raphson method and the Semi-Direct method are listed in the appendix. The equations for the circuit are:

```
F1: fx(1) = 3.e-9*dxdt(1) + (x(1)-x(1)**3)/500
          + 2.e-6*(dxdt(1)-dxdt(2))
    fx(2) = 2.e-6*(dxdt(2)-dxdt(1)) - ((u(1)-x(2))
          - (u(1)-x(2))**3)/1500.
    ***************************************************
H : fx(3) = z(1) - .1
    fx(4) = .02*(z(1)-u(1)) + z(2)
    ***************************************************
G : fx(5) = .02*(u(1)-z(1)) + ((u(1)-x(2))
          - (u(1)-x(2))**3)/1500.
    ***************************************************
```

The time step for which the Semi-Direct method takes as many sweeps as the

Newton-Raphson iterations per time point is $h \leq 0.15\ \mu s$ for the rise part of the response (1 subiteration/sweep). We can assume our time step guarantees that the solution at (t-h) is within the local convergence area of the solution at t. The time step would have to become very large in order to contradict the above assumption. For example 7.2 the time step could be 3 times the time constant and we still would be within the local convergence area.

The constraint on the time step is as follows: We want h such that $|x^{\rho}(t_n)-x(t_n)| \leq 10^{-q}$, so that 10^{q-f} is in the order of $|\lambda(t_n)|$, (see section 6.2 and equation (6.20)), where 10^{-f} is the incremental stopping criterion for the sweeps or iterations at certain time point. If we assume $e = 10^{-f}$ we would require a prediction at t_n that is at most 10^{-q} away from the solution at t_n. From equations (7.12) and (7.14) we want for 1st order BDF

$$|e(t_n)| \leq 10^{-q} \frac{t_n - t_{n-1}}{t_n - t_{n-2}} \tag{7.16}$$

and for 2nd order BDF.

$$|e(t_n)| \leq 10^{-q} \frac{t_n - t_{n-1}}{t_n - t_{n-3}} \tag{7.17}$$

In Example 7.2 the value of h for which the Semi-Direct method would do the transient analysis faster than the Newton-Raphson method was experimentally determined to be $\sim 0.15\ \mu s$ for the rise time response of the circuit.

CHAPTER 8
CONCLUSIONS

We have studied in this thesis the algorithm and the properties of the Semi-Direct method. Its application in computer-aided circuit analysis was illustrated using several examples. It was found that the method has a linear convergence rate. The method was then compared to the Newton-Raphson method which is widely used in most computer analysis programs nowadays. The Semi-Direct method has computer-storage advantages over the Newton-Raphson method. As the circuits get larger the savings ratio approaches ~66.67%. Therefore, the method is most suitable in the analysis of large scale circuits. The execution time comparison revealed that the Semi-Direct method can be faster than the Newton-Raphson method under certain conditions. These conditions are usually satisfied in transient analysis applications. Therefore, the Semi-Direct method would be very applicable in the transient analysis of large scale circuits. The Semi-Direct method can be studied further in these areas:

1) The local convergence condition of the Semi-Direct method is that the eigenvalues of the matrix of equation (5.33) all have magnitudes less than unity. The matrices $\mathbf{A}_{rr}$, $\mathbf{A}_{rs}$, $\mathbf{A}_{ss}$ and $\mathbf{A}_{sr}$, in the case of linear circuits are submatrices of the modified nodal analysis matrix, equation (4.4). How is the matrix of equation (5.33) and its eigenvalues related to the circuit parameters?

That is, can we write the convergence condition in terms of the physical properties of the circuit?

2) The Semi-Direct method will converge faster than the Newton-Raphson method if the initial guess satisfies equations (6.14) and (6.16). In transient analysis, the initial guess at time t_{n+1} is the solution at time t_n (or a predicted value based on it). What size time step can be taken in order to guarantee the comparable or faster convergence of the Semi-Direct method? An equation to compute the time step in terms of the other parameters could then be included in the transient analysis program. A time step that would guarantee a faster convergence could then be computed before the analysis at each time point.

3) Modify the programs developed in this thesis to accommodate larger circuits and experiment with transient analysis of some large-scale circuits.

LIST OF REFERENCES

LIST OF REFERENCES

[1] "Advanced Statistical Analysis Program (ASTAP)," reference guide, IBM, 1973.

[2] R. K. Brayton, F. Gustavson and G. D. Hachtel, "A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas," in Proc. IEEE, Vol. 60, pp. 98-108, Jan. 1972.

[3] R. L. Burden, J. D. Faires, et al., Numerical Analysis, Weber and Schmidt, Inc., 1978.

[4] L. O. Chua and P. M. Lin, Computer-Aided Analysis of Electronic Circuits, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

[5] C. F. Gerald, Applied Numerical Analysis, Addison-Wesley Publishing Co., 1978.

[6] G. Hachtel and A. L. Gangiovanni-Vincentelli, "A Survey of Third Generation Simulation Techniques," Proc. IEEE, Vol. 69, pp. 1264-1280, Oct. 1981.

[7] C-W. Ho, A. E. Ruehli and P. A. Brennan, "The Modified Nodal Approach to Network Analysis," IEEE Trans. Circuits Syst., Vol. CAS-25, pp. 504-509, June 1975.

[8] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, Inc., 1984.

[9] G. Kron, Diakoptics: The Piecewise Solution of Large-Scale Systems, MacDonald, London, 1963.

[10] E. L. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time-Domain Analysis of Large-Scale Integrated Circuits," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-1, pp. 131-145, July 1982.

[11] P. M. Lin and A. W. Nordsieck, "A Study of the Convergence Rate of a Multilevel Newton Algorithm," TR-EE 82-24, Purdue University, Nov. 1982.

[12] P. M. Lin, A. W. Nordsieck, and H. Y. Hsieh, "A Convergence Rate Investigation of a Multilevel Newton Algorithm for Large-Scale Analysis," Proc. 1983 Midwest Symp. on Circuits and Systems, August 1983.

[13] B. Nobel and J. W. Daniel, Applied Linear Algebra, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[14] A. W. Nordsieck and P. M. Lin, "Exploitation of Latency and Near-Latency in a Multilevel Newton Algorithm for Transient Analysis of Large-Scale Circuits," TR-EE 83-41, Purdue University, November 1983.

[15] F. Odeh and D. Zein, "A Semi-Direct method for Modular Circuits," Proc. 1983 Int'l. Symp. on Circuits and Systems, pp. 226-229, May 1983.

[16] W. G. Rudd, Assembly Language Programming and the IBM 360 and 370 Computers, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[17] P. A. Stark, Introduction to Numerical Methods, The Macmillan Company, 1970.

[18] A. Sangiovanni-Vincentelli, L-K. Chen and L. Chua, "A New Tearing Approach -- Node-Tearing Nodal Analysis," Proc. 1977 Int'l. Symp. on Circuits and Systems, pp. 143-147, 1977.

[19] W. A. Smith, Elementary Numerical Analysis, Harper and Row, Publishers, 1979.

[20] J. Todd (editor), Survey of Numerical Analysis, McGraw-Hill Book Company, Inc., 1962.

[21] F. F. Wu, "Solution of Large-Scale Networks by Tearing," IEEE Trans. on Circuits and Systems, Vol. CAS-23, pp. 706-713, Dec. 1976.

APPENDICES

APPENDIX A

Program Listings

```
*****************************************************************
************   USERS' INSTRUCTIONS   ************************
*****************************************************************
* For "semi-direct.dc.f" and "semi-direct.transient.f":
* The equations:
* Limit: 50 equations (10 blocks at 5 equations/block)
* Write in a file "out2" as FORTRAN statements (all Fortran
* rules apply) in the form:
*  fx(i) = ....  (all listed sequentially,Fi,H then G equations)
*  jac(a,b,c)=... (jac is db/dc where b and c are the equation #
*                  and variable # within the block a)
*
* Program Parameters:
* Create a file "initial2" with the following FORTRAN
* statements (all FORTRAN rules apply):
*   nblks = ..   # of blocks (10 max)
*   nequ = ..    total # of equations (50 max)
*   nf(i) = ..   # of equations/block (5 max)
*   nu = ..      # of u variables
*   nz = ..      # of z varilbles
*   msteps = .. max # of sweeps (default 20)
*   ssteps = .. max # of subiterations (default 20)
*   merror = .. sweep increment, for termination (default 1e-8)
*   serror = .. subiteration termination (default 1e-8)
*   miter = 1 or 0 sweep print control (yes or no)
*   siter = 1 or 0 subiteration print control (yes or no)
*   x(i) = .. initial guess for ith x (default 0)
*   z(i) = .. initial guess for ith x (default 0)
*   u(i) = .. initial guess for ith x (default 0)
* (for "semi-direct.transient.f" only)
*   h = ..       time step
*   limit = .. upper time limit
*   subout= 1 or 0   print sweeps or only time point result
*   order = 1 or 2   1st or 2nd order BDF
*   predict= 1 or 0  use predictor formula or not
*
* Results:
*   Final result is always in file "result". Other requested
*   data are in files "result1","result2",..,"result6")
*.
*****************************************************************
* For "newton.dc.f" and "newton.transient.f":
* The equations:
* Limit: 50 equations (10 blocks at 5 equations/block)
* Write in a file "out" as FORTRAN statements (all Fortran
* rules apply). Same as "out2" above plus
*   asr(p,q) = .. asr is d(fx(p))/du(q)
*   ars(q,p) = .. ars is d(fx(nvar-nu+q))/dx(p)
*
* Program Parameters:
* Create a file "initial" with the following FORTRAN
* statements (all FORTRAN rules apply):
*   nblks = ..   # of blocks (10 max)
*   nvar = ..    total # of equations (50 max)
*   nf(i) = ..   # of equations/block (5 max)
*   nu = ..      # of u variables
```

```
*    nz = ..       # of z varilbles
*    steps = ..    max # of sweeps (default 20)
*    error = ..    iteration termination (default 1e-8)
*    iter = 1 or 0  iteration print control (yes or no)
*    x(i) = .. initial guess for ith x (default 0)
*    z(i) = .. initial guess for ith x (default 0)
*    u(i) = .. initial guess for ith x (default 0)
* (for "newton.transient.f" only)
*  Same as for "semi-direct.transient.f" above.
*
* Results:
*    Final result is always in file "res". Other requested
*    data are in files "res1","res2",..,"res6")
*.
*************************************************************************
* For "convergance.f":
* The equations:
*    File "out2" from above and file solution that is
*    created by program "newton.dc.f". The eigenvalues of
*    the matrix of equation (5.33) are computed. The matix
*    elements are printed,too.
*************************************************************************
```

semi-direct.dc.f

```
1   ************************************************************
2   * This is a program that implements the Semi-Direct method  *
3   * to solve a set of nonlinear (or linear) equations.        *
4   *  INPUT: files out2 that contains the equations and their  *
5   *         dirivatives, and initial2 that contains the       *
6   *         partitioning information and the initial guess.   *
7   *                                                           *
8   *  OUTPUT: final solution in a file called result. The sweep*
9   *          are in files result1,result2,..(4 variables/file)*
10  ************************************************************
11  c
12        double precision norm,merror,serror,x(50),u(5),z(5),
13       +                 xi(50),sum,fx(50)
14        integer miter,siter,msteps,ssteps
15       +        ,nblks,nequ,nf(10),nz,nu,pos
16  c
17        common /data/nequ,nu,nz
18  c
19  c This is to set the default values for the print control
20  c and the initial guesses.
21        data miter,siter/1,0/
22        data msteps,ssteps,merror,serror/2*20,2*1e-8/
23        data x,u,z/60*0/
24  c
25  cThe file "initial2" includes the initial print control
26  c variables, the initial guesses and the partitions.
27  c
28        include "initial2"
29         nf(nblks-1) = nz
30         nf(nblks)  = nu
31  c
32  cxi saves the previous sweep values for error and
33  cnorm calculation purposes.
34        nn = nequ - nu - nz
35        do 13 i=1,nn
36   13     xi(i) = x(i)
37  c Concatenate u's and z's to the end of xi array
38        do 14 j=1,nz
39          xi(i) = z(j)
40           x(i) = z(j)
41   14     i = i + 1
42        do 16 j=1,nu
43          xi(i) = u(j)
44          x(i)  = u(j)
45   16     i = i + 1
46  c
47  c The title for the final result.
48        open ( unit=11,file="result",status="new")
49           write(11,800)
50           write(11,301) nequ
51           write(11,302) msteps
52        write(11,303)merror
53        write (11,310)
54        write(11,305) ( i,x(i),i=1,nequ )
55         write(11,307)
56   800  format(/'The Semi-Direct Method (dc analysis)')
```

```
 57    301  format(//'Number of Equations: ',i2)
 58    302  format(//'Max Number of Iterations: ',i2)
 59    303  format(//'Error Criterion: ',e10.4)
 60    310  format(//'Initial Conditions:')
 61   c
 62   c The headings of the sweep results
 63   c
 64            if (miter.eq.1) then
 65   c
 66           open(unit=2,file="result1",status="new")
 67               write(2,309)(i,i,i=1,4)
 68           if (nequ.gt.4) then
 69           open(unit=3,file="result2",status="new")
 70               write(3,309)(i,i,i=5,8)
 71           end if
 72           if (nequ.gt.8) then
 73           open(unit=4,file="result3",status="new")
 74               write(4,309)(i,i,i=9,12)
 75           end if
 76           if (nequ.gt.12) then
 77           open(unit=7,file="result4",status="new")
 78               write(7,309)(i,i,i=13,16)
 79           end if
 80           if (nequ.gt.16) then
 81           open(unit=8,file="result5",status="new")
 82               write(8,309)(i,i,i=17,20)
 83           end if
 84           if (nequ.gt.20) then
 85           open(unit=9,file="result6",status="new")
 86               write(9,309)(i,i,i=21,24)
 87           end if
 88           end if
 89   c
 90    309    format(/" i  ",2x,4("    x",i2,7x,"   deltax",i2,6
 91   c
 92             nsave=ssteps
 93           do 700 i=1,msteps
 94   c
 95   c This to control the number of Newton subiterations
 96            if (n1.eq.1) then
 97             if (i.le.n2) then
 98                 ssteps=nsteps
 99             else
100                 ssteps=nsave
101             end if
102            end if
103   c
104   cThis is to solve each block using the Newton method
105            pos = 1
106            do 710 j=1,nblks
107           call newton(j,x,fx,nf(j),serror,ssteps,siter,pos,u
108    710     pos = pos + nf(j)
109   c
110   c This calcultates the 2nd norm of the vector x
111            sum = 0
112            do 720 j=1,nequ
```

```
113              fx(j) = x(j) - xi(j)
114    720       sum = sum + fx(j)**2
115            norm = sqrt(sum)
116    c
117    c Sweep print statements
118    c
119            if (miter.eq.1) then
120    c
121          if (nequ.gt.4) then
122           write(2,110) i,(xi(j),fx(j),j=1,4)
123             if (nequ.gt.8) then
124              write(3,110) i,(xi(j),fx(j),j=5,8)
125                if (nequ.gt.12) then
126                 write(4,110) i,(xi(j),fx(j),j=9,12)
127                   if (nequ.gt.16) then
128                    write(7,110) i,(xi(j),fx(j),j=13,16)
129                     if (nequ.gt.20) then
130    3                  write(8,110) i,(xi(j),fx(j),j=17,20)
131                       if (nequ.gt.24) then
132                        write(9,110) i,(xi(j),fx(j),j=21,24)
133                       else
134                        write(9,110) i,(xi(j),fx(j),j=21,nequ)
135                       end if
136                     else
137                      write(8,110) i,(xi(j),fx(j),j=17,nequ)
138                     end if
139                   else
140                    write(7,110) i,(xi(j),fx(j),j=13,nequ)
141                   end if
142                 else
143                  write(4,110) i,(xi(j),fx(j),j=9,nequ)
144                 end if
145               else
146                write(3,110) i,(xi(j),fx(j),j=5,nequ)
147               end if
148          else
149           write(2,110) i,(xi(j),fx(j),j=1,nequ)
150          end if
151    c
152    c
153     110    format(i2,2x,4(e13.6,2x,e13.6,3x))
154    c
155             end if
156    c
157    c Check if increment stopping criterion has been acheived
158             if (norm.lt.merror) go to 900
159             do 910 j=1,nequ
160     910        xi(j) = x(j)
161     700   continue
162    c
163    c This is to print the final solution
164    c
165     900     write(11,307)
166           write(11,300)
167           write(11,305) (i,x(i),i=1,nequ)
168     307   format(//"*****************************************
```

```
169         +*********************************************")
170     300   format(//"Solution:")
171     305   format(/4("      x",i2,"=",e10.4))
172               end
173   c
174   c
175   c
176   c
177   c
178           subroutine newton (num,x,fx,nvar,error,steps,iter,pc
179         +                      u,z)
180   c
181           common /data/nequ,nu,nz
182           double precision jac(10,5,5),fx(50),error,norm,temp(2
183         +                  ,sum,z(nz),u(nu),x(nequ)
184            integer nvar,steps,count,nu,nz,nequ,c1,c2,pos
185   c
186           if (iter.eq.1) then
187           write(6,320)num
188     320   format (//"Block Number ",i2)
189   c
190           end if
191   c
192   c
193            do 10 count=1,steps
194   c
195   c here we insert the equations and their derivatives
196             include "out2"
197   c
198   c Transfer the jacobian of a block to lu subroutine.
199           c2 = 1
200           do 500 i=1,nvar
201              do 500  j=1,nvar
202              temp(c2)=jac(num,j,i)
203     500   c2 = c2+1
204   c
205           call lu(fx(pos),temp,nvar)
206   c
207   c this will output the subiterations
208           if (iter.eq.1) then
209             write(6,110) count,(x(i),fx(i),i=pos,nn)
210     110   format(i2,2x,4(e13.6,2x,e13.6,3x),5(/4x,4(e13.6
211         +,2x,e13.6,3x)))
212           end if
213   c
214   c
215   c
216   c Compute  second norm of x and calculate new x
217           nn=pos+nvar-1
218            sum = 0
219            do 100 i=pos,nn
220                sum = sum+fx(i)**2
221     100    x(i) = x(i) + fx(i)
222           norm = sqrt(sum)
223   c
224   c this is to compensate for the u's and z's in the original
```

```
225  c equations in the file out2
226           nn =nequ - nu - nz +1
227         if (pos.eq.nn) then
228           c1 = pos
229           do 311 i=1,nz
230             z(i) = x (c1)
231    311    c1 = c1 + 1
232         end if
233         if (pos.eq.nn+nz) then
234           c1=pos
235           do 312 i=1,nu
236             u(i) = x (c1)
237    312    c1 = c1 + 1
238         end if
239  c
240  c Check for increment stopping criterion
241             if (norm.lt.error)  go to 200
242  c      otherwise go do another subiteration
243    10         continue
244    200   end
245  c
246  c
247  c this is the LU subroutine called once each subiteration
248  c It factors the jacobian into lower and upper triangular
249  c matrices and then solves for deltax. (result returned
250  c in fx vector).
251  c.
252          subroutine lu(fx,jac,nvar)
253  c
254          double precision sumlu1,sumlu2,jac(nvar,nvar),fx(nvar)
255         +                  ,l(5,5),u(5,5)
256          equivalence(l,u)
257            integer s,nvar
258             do 11 i=1,nvar
259              fx(i)=-fx(i)
260    11        continue
261          s=1
262    15     j1=s
263          do 30  i1=s,nvar
264            sumlu1=0
265              do 20  k=1,j1-1
266    20           sumlu1= sumlu1+l(i1,k)*u(k,j1)
267    30     l(i1,j1)=jac(i1,j1) - sumlu1
268  c
269          if (j1.ge.nvar) go to 55
270  c
271          i2=s
272          s = s + 1
273          do 50  j2=s,nvar
274            sumlu2 = 0
275              do 40  k=1,i2-1
276    40           sumlu2 = sumlu2 + l(i2,k)*u(k,j2)
277    50     u(i2,j2) = (jac(i2,j2) - sumlu2) / l(i2,i2)
278          go to 15
279    55    do 70 i=1,nvar
280             fx(i) = fx(i) / l(i,i)
```

```
281            do 70  j=1,i-1
282    60       l(i,j) = l(i,j) / l(i,i)
283    70       continue
284    c  solve for y answer stored in fx
285             do 80 j=1,nvar
286            do 80 i=j+1,nvar
287               fx(i) = fx(j) * (-l(i,j))+ fx(i)
288    80       continue
289    c solve for delta x answer is in fx
290             do 90 js=1,nvar
291            j = nvar - js + 1
292            do 90 k=js+1,nvar
293               i = nvar - k + 1
294               fx(i) = fx(j) * (-u(i,j)) + fx(i)
295    90       continue
296    c
297          end
```

semi-direct.transient.f

```
1  ****************************************************************
2  * This is a program that implements the Semi-Direct method     *
3  * to perform a transient analysis on a set of time varying     *
4  * set of equations                                             *
5  *  INPUT: files out2 that contains the equations and their     *
6  *         dirivatives, and initial2 that contains the          *
7  *           partitioning information and the dc solution.      *
8  *                                                              *
9  *  OUTPUT: Solutions for the whole time interval are in the    *
10 *          files result1,result2,..(7 variables/file)          *
11 ****************************************************************
12 c
13       double precision norm,merror,serror,x(50),u(5),z(5),
14      +xi(50),sum,time,h,secd,limit,x1(50),x2(50),xp(50)
15       integer uniform,miter,siter,equ,incon,msteps,ssteps
16      + ,flag,nblks,nequ,nf(10),nz,nu,pos,order,reads,predict
17      + ,subout
18 c
19       common /data/nequ,nu,nz,secd
20       common order,x,x2,x1,h,flag
21 c
22 c  This is to set the default values for the print control
23 c  and program options.
24       data subout,siter,miter/2*0,1/
25       data msteps,ssteps,merror,serror/2*20,2*1e-16/
26       data x,reads,predict/50*0,2*1/
27       data u,z/10*0/
28       data x2/50*0/
29 c
30 cThe file "initial2" includes the initial print control
31 c variables plus the initial guesses.
32 c
33       include "initial2"
34 c
35 c  This is to read the solution at t=0 (dc solution) stored is
36 c  a file called "solution" on unit 10. We get solution from
37 c  running the original dc prgram once with capacitors replaced
38 c by voltage sorces.
39       if (reads.eq.1) then
40        nn = nequ - nu - nz
41        open(unit=10,file="solution",status="old")
42        read(10,43) (x(i),i=1,nn),(z(i),i=1,nz),(u(i),i=1,nu)
43        read(10,43) h,limit
44        read(10,44) subout,order,predict
45        read(10,45) ssteps
46        read(10,43) merror,serror
47       end if
48   45   format(i2)
49   44   format(i1)
50   43   format(e13.6)
51        nf(nblks-1) = nz
52        nf(nblks)  = nu
53 c
54 c xi saves the previous main iterations values for error and
55 c norm calculation  purposes. x2 and x1 are used for the
56 c calculation of the derivatives using BDF formulas.
```

```
57  c
58          do 13 i=1,nn
59            xp(i) = x(i)
60            x1(i) = x(i)
61            x2(i) = x(i)
62   13       xi(i) = x(i)
63  c Concatenate u's and z's to the end of xi array
64          do 14 j=1,nz
65            xi(i) = z(j)
66             x(i) = z(j)
67            x2(i) = x(i)
68            xp(i) = x(i)
69            x1(i) = x(i)
70   14       i = i + 1
71          do 16 j=1,nu
72            xi(i) = u(j)
73            x(i)  = u(j)
74            x2(i) = x(i)
75            xp(i) = x(i)
76            x1(i) = x(i)
77   16       i = i + 1
78  c
79  c The program parameters used in file result
80  c
81  c
82  c
83          open ( unit=11,file="result",status="new")
84             write(11,800)
85           if(order.eq.1) then
86             write(11,801)
87           else
88             write(11,802)
89           end if
90             write(11,301) nequ
91             write(11,302) msteps
92          write(11,303)merror
93          write(11,304)limit
94           write(11,314) h
95          write (11,310)
96          write(11,305) ( i,x(i),i=1,nequ )
97           write(11,307)
98    800   format(/'The Time Solution Using The Semi-Direct Meth
99    801   format(/'Transient Analysis (using First Order BDF)')
100   802   format(/'Transient Analysis (using Second Order BDF)'
101   301   format(/'Number of Equations: ',i2)
102   302   format(/'Max Number of Iterations: ',i2)
103   303   format(/'Error Criterion: ',e10.4)
104   304   format('Time Limit: ',e10.4)
105   314   format('Time Step: ',e10.4,' (uniform time step)')
106   310   format(/'Initial Conditions:')
107 c
108          if (miter.eq.1) then
109          if (subout.eq.1) then
110           open(unit=2,file="result1",status="new")
111               write(2,309)(i,i,i=1,4)
112          if (nequ.gt.4) then
```

```
113          open(unit=3,file="result2",status="new")
114              write(3,309)(i,i,i=5,8)
115          end if
116          if (nequ.gt.8) then
117           open(unit=4,file="result3",status="new")
118              write(4,309)(i,i,i=9,12)
119          end if
120          if (nequ.gt.12) then
121           open(unit=7,file="result4",status="new")
122              write(7,309)(i,i,i=13,16)
123          end if
124          if (nequ.gt.16) then
125           open(unit=8,file="result5",status="new")
126              write(8,309)(i,i,i=17,20)
127          end if
128          if (nequ.gt.20) then
129           open(unit=9,file="result6",status="new")
130              write(9,309)(i,i,i=21,24)
131          end if
132           else
133           open(unit=2,file="result1",status="new")
134              write(2,333)(i,i=1,7)
135          if (nequ.gt.7) then
136           open(unit=3,file="result2",status="new")
137              write(3,333)(i,i=8,14)
138          end if
139          if (nequ.gt.14) then
140           open(unit=4,file="result3",status="new")
141              write(4,333)(i,i=15,21)
142          end if
143          if (nequ.gt.21) then
144           open(unit=7,file="result4",status="new")
145              write(7,333)(i,i=22,28)
146          end if
147          end if
148          end if
149   c
150    309    format(/"t*h ",2x,4("     x",i2,7x,"    deltax",i2,6x))
151    333    format(/6x,"    time ",7x,7("     x",i2,8x))
152   c
153   c     The Time loop starts here.
154          flag=1
155            time =h
156   c  Compute the derivative of the BDF
157    701    if ((order.eq.1).or.(flag.eq.1)) then
158             secd = 1./h
159           else
160             secd = 1.5/h
161           end if
162   c
163   c This is the Semi-Direct method to solve the equations at
164   c a certain time point.(dc program).
165   c
166          do 700 i=1,msteps
167            pos = 1
168            do 710 j=1,nblks
```

```
169             call newton(j,x,nf(j),serror,ssteps,siter,pos,u,z,h)
170    710      pos = pos + nf(j)
171    c
172             sum = 0
173             do 720 j=1,nequ
174               xp(j) = x(j) - xi(j)
175    720        sum = sum + xp(j)**2
176             norm = sqrt(sum)
177    c
178            if (subout.eq.1) then
179             if (miter.eq.1) then
180    c
181    c
182    c
183    c          print statements
184           if (nequ.gt.4) then
185            write(2,110) i,(xi(j),xp(j),j=1,4)
186              if (nequ.gt.8) then
187               write(3,110) i,(xi(j),xp(j),j=5,8)
188                if (nequ.gt.12) then
189                 write(4,110) i,(xi(j),xp(j),j=9,12)
190                  if (nequ.gt.16) then
191                   write(7,110) i,(xi(j),xp(j),j=13,16)
192                    if (nequ.gt.20) then
193                     write(8,110) i,(xi(j),xp(j),j=17,20)
194                      if (nequ.gt.24) then
195                       write(9,110) i,(xi(j),xp(j),j=21,24)
196                      else
197                       write(9,110) i,(xi(j),xp(j),j=21,nequ)
198                      end if
199                    else
200                     write(8,110) i,(xi(j),xp(j),j=17,nequ)
201                    end if
202                  else
203                   write(7,110) i,(xi(j),xp(j),j=13,nequ)
204                  end if
205                else
206                 write(4,110) i,(xi(j),xp(j),j=9,nequ)
207                end if
208              else
209               write(3,110) i,(xi(j),xp(j),j=5,nequ)
210              end if
211           else
212            write(2,110) i,(xi(j),xp(j),j=1,nequ)
213           end if
214    c
215    c
216     110   format(i2,2x,4(e13.6,2x,e13.6,3x))
217    c
218    c
219             end if
220             end if
221             if (norm.lt.merror) go to 900
222             do 910 j=1,nequ
223     910        xi(j) = x(j)
224     700   continue
```

```
c
 900       if (subout.ne.1) then
            write(11,307)
           write(11,300) time
           write(11,305) (i,x(i),i=1,nequ)
           write(11,307)
           else
          if (nequ.gt.7) then
           write(2,111) time,(x(j),j=1,7)
             if (nequ.gt.14) then
              write(3,111) time,(x(j),j=8,14)
               if (nequ.gt.21) then
                write(4,111) time,(x(j),j=15,21)
                 if (nequ.gt.28) then
                  write(7,111) time,(x(j),j=22,28)
                     else
                      write(7,111) time,(x(j),j=22,nequ)
                     end if
                   else
                    write(4,111) time,(x(j),j=15,nequ)
                   end if
                 else
                  write(3,111) time,(x(j),j=8,nequ)
                 end if
               else
                write(2,111) time,(x(j),j=1,nequ)
               end if
           end if
 111     format(4x,8(e13.6,2x))
 307    format(//"*****************************************
     +*****************************************")
 300    format(//"Solution at t= ",e14.9," :")
 305    format(/4("      x",i2,"=",e10.4))
          if (time.gt.limit) go to 703
          time = time + h
c
c Update the initial guess for the next time step by
c using the solution obtained or use of a predictor
c formula xh
          if ((order.eq.1).or.(flag.eq.1)) then
           flag = 0
           do 569 i=1,nequ
             xp(i) = 2*x(i) - x2(i)
             x1(i) = x2(i)
             x2(i) = x(i)
           if (predict.eq.1) x(i) = xp(i)
          xi(i) = x(i)
 569        continue
           else
          do 571 i=1,nequ
             xp(i) = 3*x(i) - 3*x2(i) + x1(i)
             x1(i) = x2(i)
             x2(i) = x(i)
           if (predict.eq.1) x(i) = xp(i)
            xi(i) = x(i)
 571        continue
```

```
281           end if
282            nn = nequ-nu-nz+1
283            do 311 i=1,nz
284              z(i) = x(nn)
285    311     nn = nn + 1
286            do 312 i=1,nu
287              u(i) = x(nn)
288    312     nn = nn + 1
289           go to 701
290    703     end
291   c
292   c
293   c
294   c
295   c
296         subroutine newton(num,x,nvar,error,steps,iter,pos,u,z,
297   c
298   c
299           common /data/nequ,nu,nz,secd
300           double precision jac(10,5,5),fx(50),error,norm,temp(25
301        +                    ,h,secd,dxdt,sum,z(nz),u(nu),x(nequ)
302            integer nvar,steps,count,nu,nz,nequ,c1,c2,pos
303            if (iter.eq.1) then
304           write(6,320)num
305    320    format(//"Block Number",i2)
306         end if
307   c
308   c this is to compensate for the u's and z's in the original
309   c equations in the file out2
310   c
311            do 10 count=1,steps
312            nn =nequ - nu - nz +1
313         if (pos.eq.nn) then
314           c1 = pos
315           do 311 i=1,nz
316             z(i) = x(c1)
317    311     c1 = c1 + 1
318         end if
319         if (pos.eq.nn+nz) then
320           c1=pos
321           do 312 i=1,nu
322             u(i) = x(c1)
323    312     c1 = c1 + 1
324         end if
325   c
326   c
327   c here we insert the equations and their derivatives
328               include "out2"
329   c
330   c
331         c2 = 1
332         do 500 i=1,nvar
333            do 500  j=1,nvar
334            temp(c2)=jac(num,j,i)
335    500   c2 = c2+1
336   c
```

```
337           call lu(fx(pos),temp,nvar)
338   c
339   c
340           nn=pos+nvar-1
341            sum = 0
342            do 100 i=pos,nn
343                sum = sum+fx(i)**2
344    100      x(i) = x(i) + fx(i)
345            norm = sqrt(sum)
346   c  this will output the data
347           if (iter.eq.1) then
348              write(6,110) count,(x(i),fx(i),i=pos,nn)
349    110    format(/i2,2x,4(e13.6,2x,e13.6,3x),
350           end if              5(/4x,4(x,4(e13.6,2x,e13.6,3x)))
351   c
352              if (norm.lt.error)  go to 200
353   c       otherwise go do another iteration
354    10          continue
355    200    end
356   c
357   c
358   c Soluiton of blocks using LU factorization.
359   c
360   c.
361           subroutine lu(fx,jac,nvar)
362   c
363           double precision sumlu1,sumlu2,jac(nvar,nvar),fx(nvar)
364          +,l(5,5),u(5,5)
365           equivalence(l,u)
366             integer s,nvar
367              do 11 i=1,nvar
368               fx(i)=-fx(i)
369    11         continue
370           s=1
371    15      j1=s
372           do 30  i1=s,nvar
373             sumlu1=0
374               do 20  k=1,j1-1
375    20            sumlu1= sumlu1+l(i1,k)*u(k,j1)
376    30      l(i1,j1)=jac(i1,j1) - sumlu1
377   c
378           if (j1.ge.nvar) go to 55
379   c
380           i2=s
381           s = s + 1
382           do 50  j2=s,nvar
383             sumlu2 = 0
384               do 40  k=1,i2-1
385    40            sumlu2 = sumlu2 + l(i2,k)*u(k,j2)
386    50      u(i2,j2) = (jac(i2,j2) - sumlu2) / l(i2,i2)
387           go to 15
388    55     do 70 i=1,nvar
389             fx(i) = fx(i) / l(i,i)
390             do 70  j=1,i-1
391    60        l(i,j) = l(i,j) / l(i,i)
392    70        continue
```

```
c   solve for y answer stored in fx
          do 80 j=1,nvar
          do 80 i=j+1,nvar
             fx(i) = fx(j) * (-l(i,j))+ fx(i)
 80       continue
c  solve for delta x answer is in fx
          do 90 js=1,nvar
          j = nvar - js + 1
          do 90 k=js+1,nvar
             i = nvar - k + 1
             fx(i) = fx(j) * (-u(i,j)) + fx(i)
 90       continue
c
       end
c
c
c this function is to evaluate the dirivatives using BDF of
c order 1 or 2 .
c
         function dxdt(i)
         double precision x(50),x2(50),x1(50),h,dxdt
         integer flag,order
        common order,x,x2,x1,h,flag
c
         if ((order.eq.1).or.(flag.eq.1)) then
          dxdt = (x(i) - x2(i) )/h
         else
          dxdt = (1.5*x(i) - 2*x2(i) + .5*x1(i) )/h
         end if
         return
         end
```

newton.dc.f

```
      ************************************************************
      * This is a program that implements the Newton-Raphson method*
      * to solve a set of nonlinear (or linear) equations.         *
      *  INPUT: files out that contains the equations and their    *
      *         dirivatives, and initial that contains the         *
      *         partitioning information and the initial guess.    *
      *                                                            *
      *  OUTPUT: final solution in file called res. The iterations *
      *          in files res1,res2,....(4 variables per file).    *
      ************************************************************
      c
            double precision norm,sum,b(5),phi(25),error,x(50),u(5),z
           +,ars(5,45),asr(45,5),move(25),fx(50),jac(10,5,5)
            integer k,pos,c,nblks,nvar,steps,nu,nf(10)
      c
      c
      c
      c  This is to set the default values for the print control
      c  and the initial guesses.
            data steps,error,iter/20,1.e-8,1/
            data x,asr,ars,jac/50*0,450*0,250*0/
            data z,u/10*0/
      c
      cThe file "initial" includes the initial print control
      c variables, the initial guesses and the partitions.
      c
            include "initial"
             nf(nblks-1) = nz
             nf(nblks) = nu
      c
      c Concatenate u's and z's to the x array.
                k=nvar -nu - nz+1
                do 13 j=1,nz
                 x(k) = z(j)
       13        k=k+1
                do 16 j=1,nu
                 x(k) = u(j)
       16        k=k+1
      c
      c The title for the main result
            open ( unit=11,file="res",status="new")
               write(11,800)
               write(11,301) nvar
               write(11,302) steps
            write(11,303)error
            write (11,310)
            write(11,305) ( i,x(i),i=1,nvar )
             write(11,307)
        800 format(//'Modified Newton-Raphson Method (block diagonal)
        301 format(//'Number of Equations: ',i2)
        302 format(//'Max Number of Iterations: ',i2)
        303 format(//'Error Criterion: ',e10.4)
        310 format(//'Initial Conditions: ')
      c
      c Headings for the iteration results
              if (iter.eq.1) then
```

```
      open(unit=2,file="res1",status="new")
          write(2,309)(i,i,i=1,4)
      if (nvar.gt.4) then
       open(unit=3,file="res2",status="new")
          write(3,309)(i,i,i=5,8)
      end if
      if (nvar.gt.8) then
       open(unit=4,file="res3",status="new")
          write(4,309)(i,i,i=9,12)
      end if
      if (nvar.gt.12) then
       open(unit=7,file="res4",status="new")
          write(7,309)(i,i,i=13,16)
      end if
      if (nvar.gt.16) then
       open(unit=8,file="res5",status="new")
          write(8,309)(i,i,i=17,20)
      end if
      if (nvar.gt.20) then
       open(unit=9,file="res6",status="new")
          write(9,309)(i,i,i=21,24)
      end if
      end if
c
 309   format(/" i  ",2x,4("    x",i2,7x,"   deltax",i2,6x
c
c This is the iteration loop
c
      do 700 i=1,steps
c
c these do loops zero the arrays jac,asr and ars because t
c are used for storage purposes
c.
       do 702 j=1,nblks
        do 701 k=1,nf(j)
         do 701 l=1,nf(j)
 701      jac(j,k,l)=0
       do 702 k=1,nvar
       asr(k,j)=0
 702   ars(j,k)=0
c
c the file out contains the equations and their derivative
      include "out"
c
     do 777 j=1,nvar
 777   fx(j) = -fx(j)
c
c This part makes use of the boardered block diagonal form
c of the jacobian. The algorithm is explained in section 4.
c
         pos = 1
         do 70 j =1,nblks-1
         c=1
            do 330 l=1,nf(j)
            do 330 k=1,nf(j)
               move(c) = jac(j,k,l)
```

```
113   330         c=c+1
114               call lu(nf(j),move)
115               c=1
116                 do 331 l=1,nf(j)
117                   lzz=l+pos-1
118                   b(l) = fx(lzz)
119                 do 331 k=1,nf(j)
120                 jac(j,k,l) = move(c)
121   331         c=c+1
122               call lsolve(move,b,nf(j),1)
123               c=1
124                 do 345 l=1,nu
125                    do 345 k=pos,pos+nf(j)-1
126                   phi(c)=ars(l,k)
127   345         c=c+1
128               c=1
129                do 344 l=1,nf(j)
130                 do 344 k=1,nf(j)
131                 if(k.eq.l) then
132                    move(c) = 1
133                 else
134                  move(c) = jac(j,l,k)
135                 end if
136   344           c=c+1
137              call lsolve(move,phi,nf(j),nu)
138             c=1
139             do 668 l=1,nu
140               do 668 k=1,nf(j)
141             ars(l,k) = phi(c)
142   668       c=c+1
143               c=1
144                 do 320 l=1,nu
145                    do 320 k=pos,pos+nf(j)-1
146                      phi(c)=asr(k,l)
147    320        c=c+1
148              c=1
149              do 667 l=1,nf(j)
150               do 667 k=1,nf(j)
151                 move(c)=jac(j,k,l)
152   667       c=c+1
153                call lsolve(move,phi,nf(j),nu)
154               do 888 l=1,nu
155                lzz= l+nvar-nu
156                do 889 k=1,nu
157                do 889 m=1,nf(j)
158               jac(nblks,l,k)=jac(nblks,l,k) - ars(l,m)
159    889          continue                          *phi(m+(k-1)*nf(j))
160                 do 888 k=1,nf(j)
161                 fx(lzz)=fx(lzz)-ars(l,k)*b(k)
162   888           continue
163   70          pos=pos+nf(j)
164                 c=1
165                 do 910 l=1,nu
166                 do 910 k=1,nu
167                  move(c)=jac(nblks,k,l)
```

```
169              call lu(nu,move)
170              call lusolve(move,fx(nvar-nu+1),nu)
171              pos=1
172              do 950 j=1,nblks-1
173               c=1
174               do 920 l=1,nf(j)
175               do 920 k=1,nf(j)
176                move(c)=jac(j,k,l)
177   920         c=c+1
178               do 921 l=pos,pos+nf(j)-1
179                do 921 k=1,nu
180                  ku=k+nvar-nu
181                fx(l)=fx(l)-asr(l,k)*fx(ku)
182   921         continue
183               call lusolve(move,fx(pos),nf(j))
184   950       pos = pos+nf(j)
185   c
186   c     Iteration print statement.
187   c
188            if (iter.eq.1) then
189   c
190          if (nvar.gt.4) then
191           write(2,110) i,(x(j),fx(j),j=1,4)
192             if (nvar.gt.8) then
193              write(3,110) i,(x(j),fx(j),j=5,8)
194               if (nvar.gt.12) then
195                write(4,110) i,(x(j),fx(j),j=9,12)
196                 if (nvar.gt.16) then
197                  write(7,110) i,(x(j),fx(j),j=13,16)
198                   if (nvar.gt.20) then
199                    write(8,110) i,(x(j),fx(j),j=17,20)
200                     if (nvar.gt.24) then
201                      write(9,110) i,(x(j),fx(j),j=21,24)
202                     else
203                      write(9,110) i,(x(j),fx(j),j=21,nvar)
204                     end if
205                   else
206                    write(8,110) i,(x(j),fx(j),j=17,nvar)
207                   end if
208                 else
209                  write(7,110) i,(x(j),fx(j),j=13,nvar)
210                 end if
211               else
212                write(4,110) i,(x(j),fx(j),j=9,nvar)
213               end if
214             else
215              write(3,110) i,(x(j),fx(j),j=5,nvar)
216             end if
217          else
218           write(2,110) i,(x(j),fx(j),j=1,nvar)
219          end if
220   c
221   c
222    110   format(i2,2x,4(e13.6,2x,e13.6,3x))
223   c
224            end if
```

```
225  c
226  c Calculate the second norm of the vector x
227  c and update the value of x
228           sum = 0
229           do 720 j=1,nvar
230             sum = sum + fx(j)**2
231   720          x(j) = x(j) + fx(j)
232           norm = sqrt(sum)
233  c     Update u and z.
234            k=nvar -nu - nz+1
235            do 14 j=1,nz
236             z(j) = x(k)
237   14        k=k+1
238            do 17 j=1,nu
239             u(j) = x(k)
240   17        k=k+1
241  c Check if error stopping criterion has been acheived
242           if (norm.lt.error) go to 900
243   700  continue
244  c
245  c This is to print the final solution
246   900    write(11,307)
247         write(11,300)
248         write(11,305) (i,x(i),i=1,nvar)
249   307   format(//"*************************************************
250       +*********************************")
251   300   format(//"Solution:")
252   305   format(/4("      x",i2,"=",e10.4))
253  c  the file solution contains the solution for the use of the
254  c program convergance.f .
255        open ( unit=10,file="solution",status="new")
256         nn= 1
257         write(10,332)nn,nvar,nu,nblks,(nf(i),i=1,nblks)
258   332   format(i2)
259          write(10,333)(x(i),i=1,nvar)
260   333    format(e13.6)
261          end
262  c
263  c
264  c
265  c
266  c
267  c   LU factorization
268  c.
269         subroutine lu(nvar,jac)
270  c
271         double precision sumlu1,sumlu2,jac(nvar,nvar),l(5,5)
272         equivalence(l,u)                                    ,u(5,5)
273          integer s,k,nvar
274        s=1
275   15    j1=s
276        do 30  i1=s,nvar
277         sumlu1=0
278           do 20  k=1,j1-1
279   20         sumlu1= sumlu1+l(i1,k)*u(k,j1)
280   30    l(i1,j1)=jac(i1,j1) - sumlu1
```

```
c
      if (j1.ge.nvar) go to 55
c
      i2=s
      s = s + 1
      do 50  j2=s,nvar
        sumlu2 = 0
          do 40  k=1,i2-1
 40          sumlu2 = sumlu2 +l(i2,k)*u(k,j2)
 50    u(i2,j2) = (jac(i2,j2) - sumlu2) /l(i2,i2)
      go to 15
 55     do 1 i=1,nvar
          do 1 j=1,nvar
 1      jac(i,j)=l(i,j)
        end
c
c
c    Solve an LU factorized jacobian for deltax.
c
          subroutine lusolve(jac,fx,nvar)
c
        double precision jac(nvar,nvar),fx(nvar)
c
c   solve LY = FX for Y storing it in fx
c   normalize pivot
 55   do 70 i=1,nvar
        fx(i) = fx(i) / jac(i,i)
        do 70  j=1,i-1
         jac(i,j) = jac(i,j) / jac(i,i)
 70      continue
c   solve for y answer stored in fx
        do 80 j=1,nvar
        do 80 i=j+1,nvar
           fx(i) = fx(j) * (-jac(i,j))+ fx(i)
 80      continue
c  solve for  x answer is in fx
         do 90 js=1,nvar
        j = nvar - js + 1
        do 90 k=js+1,nvar
           i = nvar - k + 1
           fx(i) = fx(j) * (-jac(i,j)) + fx(i)
 90     continue
       end
c
c     Solve a jacobian that is in lower triangular form.
c
         subroutine lsolve(jac,fx,nvar,nu)
c
       double precision jac(nvar,nvar),fx(nvar,nu)
c   solve LY = FX for Y storing it in fx
c   normalize pivot
 55   do 70 i=1,nvar
c       take care of whole row in fx
       do 56 k=1,nu
 56     fx(i,k) = fx(i,k) / jac(i,i)
        do 70  j=1,i-1
```

```
337             jac(i,j) = jac(i,j) / jac(i,i)
338    70       continue
339    c    solve for y answer stored in fx
340             do 80 j=1,nvar
341             do 80 i=j+1,nvar
342            do 80 k=1,nu
343                fx(i,k) = fx(j,k) * (-jac(i,j))+ fx(i,k)
344    80       continue
345    c
346            end
```

newton.transient.f

```
 1  ************************************************************
 2  * This is a program that implements the Newton-Raphson method
 3  * to perform a transient analysis on a set of time varying
 4  * set of equations.
 5  *  INPUT: files out that contains the equations and their
 6  *         dirivatives, and initial that contains the
 7  *         partitioning information and the dc solution.
 8  *
 9  *  OUTPUT: Solutions for the whole time interval are in the
10  *          files res1,res2,....(7 variables per file).
11  ************************************************************
12  c
13        double precision norm,sum,b(5),phi(25),error,x(50),x1(5
14       + ,x2(50),limit,secd,h,time,xp(50),u(5),z(5),jac(10,5,5)
15       + ars(5,45),asr(45,5),move(25),fx(50)
16        integer k,pos,c,nblks,nvar,steps,nu,nf(10),order,unifor
17       + predict,subout
18          common order,x,x2,flag,x1,h
19  c
20  c
21  c
22  c  This is to set the default values for the print control
23  c  and the initial guesses.
24        data steps,error,iter/20,1.e-16,0/
25        data x,asr,ars,jac/50*1,450*0,250*0/
26        data reads,predict/2*1/
27  c
28  cThe file "initial" includes the initial print control
29  c variables plus the initial guesses.
30  c
31        include "initial"
32         nf(nblks) = nu
33         nf(nblks-1) = nz
34  c This is to read the soluiton at t=0 (dc solution) stored in
35  c a file called "soluiton" on unit 10.
36  c
37         if (reads.eq.1) then
38        open ( unit=10,file="solution",status="old")
39        read(10,43)(x(i),i=1,nvar)
40         read(10,43)h,limit
41         read(10,44) subout,order,predict
42         read(10,44) i
43         read(10,43) error
44          end if
45   44    format(i1)
46   43    format(e13.6)
47  c
48  c Initial x1 and x2 that are used to store the previous
49  c values of x for the use in BDF.
50          do 13 i=1,nvar
51        xp(i) = x(i)
52        x1(i) = x(i)
53   13     x2(i) = x(i)
54  c
55  c Write program parameters used in file res
56  c
```

```
      open ( unit=11,file="res",status="new")
         write(11,800)
       if(order.eq.1) then
         write(11,801)
         else
        write(11,802)
         end if
          write(11,301) nvar
          write(11,302) steps
      write(11,303)error
      write(11,304) limit
          write(11,314) h
      write (11,310)
      write(11,305) ( i,x(i),i=1,nvar )
       write(11,307)
800   format(/'Newton-Raphson Method (block diagonal)')
801   format(/'Transient Analysis (using First Order BDF)')
802   format(/'Transient Analysis (using Second Order BDF)')
301   format(/'Number of Equations:  ',i2)
302   format(/'Max Number of Iterations: ',i2)
303   format(/'Error Criterion:  ',e10.4)
304   format(/'Time Limit :  ',e10.4)
314   format(/'Time Step :  ',e10.4)
310   format(/'Initial Conditions:')
c
      if (iter.eq.1) then
       if (subout.eq.1) then
       open(unit=2,file="res1",status="new")
           write(2,309)(i,i,i=1,4)
      if (nvar.gt.4) then
       open(unit=3,file="res2",status="new")
           write(3,309)(i,i,i=5,8)
      end if
      if (nvar.gt.8) then
       open(unit=4,file="res3",status="new")
           write(4,309)(i,i,i=9,12)
      end if
      if (nvar.gt.12) then
       open(unit=7,file="res4",status="new")
           write(7,309)(i,i,i=13,16)
      end if
      if (nvar.gt.16) then
       open(unit=8,file="res5",status="new")
           write(8,309)(i,i,i=17,20)
      end if
      if (nvar.gt.20) then
       open(unit=9,file="res6",status="new")
           write(9,309)(i,i,i=21,24)
      end if
      else
       open(unit=2,file="res1",status="new")
           write(2,333)(i,i=1,7)
      if (nvar.gt.7) then
       open(unit=3,file="res2",status="new")
           write(3,333)(i,i=8,14)
      end if
```

```
113         if (nvar.gt.14) then
114          open(unit=4,file="res3",status="new")
115              write(4,333)(i,i=15,21)
116         end if
117         if (nvar.gt.21) then
118          open(unit=7,file="res4",status="new")
119              write(7,333)(i,i=22,28)
120         end if
121         end if
122         end if
123   c
124    309    format(/" j  ",2x,4("     x",i2,7x,"   deltax",i2,6x)
125    333    format(/6x,"   time ",7x,7("     x",i2,8x))
126   c
127   c The time loop starts here
128           flag=1
129          time = h
130   c Compute the derivative of the BDF
131    766     if ((order.eq.1).or.(flag.eq.1)) then
132              secd=1./h
133           else
134            secd = 1.5/h
135           end if
136   c This is the Newton method to solve the equations at
137   c a certain time point (dc program).
138   c
139         do 700 i=1,steps
140   c
141   c these do loops zero the arrays jac,asr and ars because th
142   c are used for storage purposes
143   c.
144          do 702 j=1,nblks
145           do 701 k=1,nf(j)
146            do 701 l=1,nf(j)
147    701     jac(j,k,l)=0
148          do 702 k=1,nvar
149          asr(k,j)=0
150    702   ars(j,k)=0
151   c
152            k=nvar -nu - nz+1
153            do 14 j=1,nz
154             z(j) = x(k)
155    14       k=k+1
156            do 17 j=1,nu
157             u(j) = x(k)
158    17       k=k+1
159   c the file out contains the equations and their derivatives
160         include "out"
161   c
162         do 777 j=1,nvar
163    777    fx(j) = -fx(j)
164   c
165   c Steps 1 and 2 are executed using the subroutine sums
166   c
167            pos = 1
168            do 70 j =1,nblks-1
```

```
169             c=1
170                do 330 l=1,nf(j)
171                do 330 k=1,nf(j)
172                   move(c) = jac(j,k,l)
173   330             c=c+1
174              call lu(nf(j),move)
175             c=1
176               do 331 l=1,nf(j)
177                 lzz=l+pos-1
178                 b(l) = fx(lzz)
179               do 331 k=1,nf(j)
180               jac(j,k,l) = move(c)
181   331       c=c+1
182              call lsolve(move,b,nf(j),1)
183             c=1
184               do 345 l=1,nu
185                  do 345 k=pos,pos+nf(j)-1
186                 phi(c)=ars(l,k)
187   345       c=c+1
188             c=1
189              do 344 l=1,nf(j)
190               do 344 k=1,nf(j)
191               if(k.eq.l) then
192                  move(c) = 1
193               else
194                move(c) = jac(j,l,k)
195               end if
196   344         c=c+1
197            call lsolve(move,phi,nf(j),nu)
198           c=1
199           do 668 l=1,nu
200             do 668 k=1,nf(j)
201           ars(l,k) = phi(c)
202   668     c=c+1
203             c=1
204               do 320 l=1,nu
205                  do 320 k=pos,pos+nf(j)-1
206                    phi(c)=asr(k,l)
207    320      c=c+1
208            c=1
209            do 667 l=1,nf(j)
210             do 667 k=1,nf(j)
211               move(c)=jac(j,k,l)
212   667     c=c+1
213              call lsolve(move,phi,nf(j),nu)
214             do 888 l=1,nu
215              lzz= l+nvar-nu
216              do 889 k=1,nu
217              do 889 m=1,nf(j)
218         jac(nblks,l,k)=jac(nblks,l,k)-ars(l,m)*phi(m+(k-1)*nf(j)
219    889       continue
220              do 888 k=1,nf(j)
221              fx(lzz)=fx(lzz)-ars(l,k)*b(k)
222   888        continue
223   70        pos=pos+nf(j)
224               c=1
```

```
225               do 910 l=1,nu
226               do 910 k=1,nu
227                move(c)=jac(nblks,k,l)
228   910         c=c+1
229               call lu(nu,move)
230               call lusolve(move,fx(nvar-nu+1),nu)
231               pos=1
232               do 950 j=1,nblks-1
233                c=1
234                do 920 l=1,nf(j)
235                do 920 k=1,nf(j)
236                 move(c)=jac(j,k,l)
237   920          c=c+1
238                do 921 l=pos,pos+nf(j)-1
239                 do 921 k=1,nu
240                   ku=k+nvar-nu
241                 fx(l)=fx(l)-asr(l,k)*fx(ku)
242   921          continue
243                call lusolve(move,fx(pos),nf(j))
244   950       pos = pos+nf(j)
245   c
246   c
247             if (subout.eq.1) then
248             if (iter.eq.1) then
249   c
250   c
251   c
252   c          print statements
253           if (nvar.gt.4) then
254            write(2,110) i,(x(j),fx(j),j=1,4)
255              if (nvar.gt.8) then
256               write(3,110) i,(x(j),fx(j),j=5,8)
257                if (nvar.gt.12) then
258                 write(4,110) i,(x(j),fx(j),j=9,12)
259                  if (nvar.gt.16) then
260                   write(7,110) i,(x(j),fx(j),j=13,16)
261                    if (nvar.gt.20) then
262                     write(8,110) i,(x(j),fx(j),j=17,20)
263                      if (nvar.gt.24) then
264                       write(9,110) i,(x(j),fx(j),j=21,24)
265                      else
266                       write(9,110) i,(x(j),fx(j),j=21,nvar)
267                      end if
268                    else
269                     write(8,110) i,(x(j),fx(j),j=17,nvar)
270                    end if
271                  else
272                   write(7,110) i,(x(j),fx(j),j=13,nvar)
273                  end if
274                else
275                 write(4,110) i,(x(j),fx(j),j=9,nvar)
276                end if
277              else
278               write(3,110) i,(x(j),fx(j),j=5,nvar)
279              end if
280           else
```

```
281          write(2,110) i,(x(j),fx(j),j=1,nvar)
282         end if
283   c
284   c
285    110   format(i2,2x,4(e13.6,2x,e13.6,3x))
286   c
287   c
288           end if
289           end if
290           sum = 0
291           do 720 j=1,nvar
292             sum = sum + fx(j)**2
293    720          x(j) = x(j) + fx(j)
294           norm = sqrt(sum)
295   c Check if error stopping criterion has been acheived
296           if (norm.lt.error) go to 900
297    700   continue
298   c
299    900    if (subout.eq.1) then
300           write(11,307)
301           write(11,300) time
302          write(11,305) (i,x(i),i=1,nvar)
303           write(11,307)
304          else
305         if (nvar.gt.7) then
306          write(2,111) time,(x(j),j=1,7)
307            if (nvar.gt.14) then
308             write(3,111) time,(x(j),j=8,14)
309              if (nvar.gt.21) then
310               write(4,111) time,(x(j),j=15,21)
311                if (nvar.gt.28) then
312                 write(7,111) time,(x(j),j=22,28)
313                    else
314                     write(7,111) time,(x(j),j=22,nvar)
315                    end if
316                  else
317                   write(4,111) time,(x(j),j=15,nvar)
318                  end if
319                else
320                 write(3,111) time,(x(j),j=8,nvar)
321                end if
322              else
323               write(2,111) time,(x(j),j=1,nvar)
324              end if
325           end if
326    111     format(4x,8(e13.6,2x))
327    307    format(//"*******************************************
328         +*****************************************")
329    300    format(//"Solution at t= ",e14.9," : ")
330    305    format(/4("      x",i2,"=",e10.4))
331                if (time.gt.limit) go to 703
332              time = time + h
333   c
334   c Update the initial guess for the next time point
335   c by using the solution obtained or a predictor
336   c formula.
```

```
c
      if ((order.eq.1).or.(flag.eq.1)) then
        flag = 0
       do 569 i=1,nvar
          xp(i) = 2*x(i) - x2(i)
        x1(i) = x2(i)
        x2(i) = x(i)
        if (predict.eq.1) x(i) = xp(i)
569     continue
       else
        do 571 i=1,nvar
          xp(i) = 3*x(i) - 3*x2(i) + x1(i)
         x1(i) = x2(i)
         x2(i) = x(i)
       if (predict.eq.1) x(i) = xp(i)
571     continue
       end if
       go to 766
703    end
c
c
c
c
c
c
c  LU factorization
c.
       subroutine lu(nvar,jac)
c
       double precision sumlu1,sumlu2,jac(nvar,nvar),
     + l(5,5),u(5,5)
       equivalence(l,u)
        integer s,k,nvar
      s=1
15     j1=s
      do 30  i1=s,nvar
        sumlu1=0
          do 20  k=1,j1-1
20           sumlu1= sumlu1+l(i1,k)*u(k,j1)
30     l(i1,j1)=jac(i1,j1) - sumlu1
c
      if (j1.ge.nvar) go to 55
c
      i2=s
      s = s + 1
      do 50  j2=s,nvar
        sumlu2 = 0
          do 40  k=1,i2-1
40           sumlu2 = sumlu2 +l(i2,k)*u(k,j2)
50     u(i2,j2) = (jac(i2,j2) - sumlu2) /l(i2,i2)
      go to 15
55     do 1 i=1,nvar
          do 1 j=1,nvar
1       jac(i,j)=l(i,j)
       end
c
```

```
c
c       Solve lu factorized jacobian
c
        subroutine lusolve(jac,fx,nvar)
c
        double precision jac(nvar,nvar),fx(nvar)
c
c   solve LY = FX for Y storing it in fx
c   normalize pivot
 55   do 70 i=1,nvar
        fx(i) = fx(i) / jac(i,i)
        do 70  j=1,i-1
         jac(i,j) = jac(i,j) / jac(i,i)
 70      continue
c   solve for y answer stored in fx
        do 80 j=1,nvar
        do 80 i=j+1,nvar
           fx(i) = fx(j) * (-jac(i,j))+ fx(i)
 80      continue
c  solve for  x answer is in fx
         do 90 js=1,nvar
        j = nvar - js + 1
        do 90 k=js+1,nvar
           i = nvar - k + 1
           fx(i) = fx(j) * (-jac(i,j)) + fx(i)
 90     continue
       end
c
c Solve a lower triangular jacobian
c
        subroutine lsolve(jac,fx,nvar,nu)
c
       double precision jac(nvar,nvar),fx(nvar,nu)
c   solve LY = FX for Y storing it in fx
c   normalize pivot
 55   do 70 i=1,nvar
c       take care of whole row in fx
       do 56 k=1,nu
 56     fx(i,k) = fx(i,k) / jac(i,i)
        do 70  j=1,i-1
         jac(i,j) = jac(i,j) / jac(i,i)
 70      continue
c   solve for y answer stored in fx
        do 80 j=1,nvar
        do 80 i=j+1,nvar
       do 80 k=1,nu
           fx(i,k) = fx(j,k) * (-jac(i,j))+ fx(i,k)
 80      continue
       end
c
c this function is to evaluate the dirivatives using BDF of
c order 1 or 2 .
c
        function dxdt(i)
        double precision x(50),x2(50),x1(50),h,dxdt
        integer order
```

```
c
      common order, x, x2, flag, x1, h
      if ((order.eq.1).or.(flag.eq.1)) then
       dxdt = (x(i) - x2(i) )/h
      else
       dxdt = (1.5*x(i) - 2*x2(i) + .5*x1(i))/h
      end if
      return
      end
```

conversance.f

```
C*********************************************************************
C* This program computes the eigenvalues of the matrix               *
C*              inv(Arr).Asr.inv(Ass).Asr                            *
C* The input is the file out and the the file solution.              *
C* The file solution is set by the program newton.dc.f.              *
C* The output is the eigenvalues and optionally the above            *
C* matrix element. IMSL (International Mathematical and              *
C* Statistical Library) routines are used.                           *
C*********************************************************************
C
        double precision jac(10,5,5),x(50),fx(50),arr(5,5)
     +,z(5),u(5),ans(5,5),ars(5,20),ass(5,5),assinv(5,5)
     +,asr(20,5),eigen(10)
       integer nf(10)
        data ans,jac/275*0/
        open(unit=10,file="solution",status="old")
        read(10,10) n,nvar,nu,nblks,(nf(i),i=1,nblks)
         write(6,10) nvar,nu,nblks,(nf(i),i=1,nblks)
 10     format(i2)
        nn=nvar-nu-nz+1
        read(10,20)(x(i),i=1,nvar)
        do 22 i=1,nz
         z(i) = x(nn)
 22      nn=nn+1
       do 23 i=1,nu
         u(i) = x(nn)
 23     nn=nn+1
c        write(6,11)(x(i),i=1,nvar)
 11      format(e13.6)
 20     format(e13.6)
        include "out"
c
       pos = 1
       do 1 i=1,nblks-1
       do 15 l=1,nf(i)
         l1=l+pos-1
         do 25 k=1,nf(i)
 25      ass(l,k)=jac(i,l,k)
       do 15 k=1,nu
       asr(l,k)=asr(l1,k)
       ars(k,l)=ars(k,l1)
 15    continue
       call linv1f(ass,nf(i),5,assinv,0,arr,ier)
       call vmulff(ars,assinv,nu,nf(i),nf(i),5,5,ass,5,ier)
       call vmulff(ass,asr,nu,nf(i),nu,5,20,arr,5,ier)
       do 35 l=1,nu
        do 35 k=1,nu
 35    ans(l,k)=ans(l,k)+arr(l,k)
       write(6,343)
 343   format("ok")
 1     pos = pos+nf(i)
       do 40 i=1,nu
        do 40 j=1,nu
 40     arr(i,j)=jac(nblks,i,j)
       call linv1f(arr,nu,5,assinv,0,asr,ier)
       call vmulff(assinv,ans,nu,nu,nu,5,5,ass,5,ier)
```

```
c this statement write the resulting matrix from the
c product inv(Arr).Ars.inv(Ass).Asr.
      if (n.eq.1) write(6,600)((ass(i,j),j=1,nu),i=1,nu)
600   format(4e13.6)
      call eigrf(ass,nu,5,0,eigen,assinv,5,asr,ier)
70    format(/"eigenvalue #",i2," =",e13.6," + j ",e13.6)
       do 43 i=2,2*nu,2
         write(6,70) i/2,eigen(i-1),eigen(i)
        eigen(i)=sqrt(eigen(i-1)**2+eigen(i)**2)
         write(6,71) eigen(i)
71       format("Magnitude = ",e13.6)
43     continue
      end
```

APPENDIX B

Results for Examples 7.1 and 7.2

(Semi-Direct and Newton-Raphson have identical listings, therefore only one version is listed)

example 7.1 (rise)

time	x 1	x 2	x 5
0.100000e-06	0. e+00	0. e+00	0.967742e-01
0.200000e-06	0.135468e-02	0.135680e-02	0.968179e-01
0.300000e-06	0.305937e-02	0.306429e-02	0.968729e-01
0.400000e-06	0.478035e-02	0.478824e-02	0.969284e-01
0.500000e-06	0.641075e-02	0.642165e-02	0.969810e-01
0.600000e-06	0.792205e-02	0.793593e-02	0.970297e-01
0.700000e-06	0.931190e-02	0.932872e-02	0.970746e-01
0.800000e-06	0.105863e-01	0.106060e-01	0.971157e-01
0.900000e-06	0.117536e-01	0.117762e-01	0.971533e-01
0.100000e-05	0.128222e-01	0.128477e-01	0.971878e-01
0.110000e-05	0.138005e-01	0.138288e-01	0.972194e-01
0.120000e-05	0.146960e-01	0.147270e-01	0.972483e-01
0.130000e-05	0.155157e-01	0.155494e-01	0.972747e-01
0.140000e-05	0.162659e-01	0.163023e-01	0.972989e-01
0.150000e-05	0.169526e-01	0.169917e-01	0.973211e-01
0.160000e-05	0.175811e-01	0.176229e-01	0.973413e-01
0.170000e-05	0.181564e-01	0.182008e-01	0.973599e-01
0.180000e-05	0.186829e-01	0.187300e-01	0.973769e-01
0.190000e-05	0.191649e-01	0.192145e-01	0.973924e-01
0.200000e-05	0.196060e-01	0.196582e-01	0.974066e-01
0.210000e-05	0.200097e-01	0.200645e-01	0.974197e-01
0.220000e-05	0.203792e-01	0.204366e-01	0.974316e-01
0.230000e-05	0.207173e-01	0.207773e-01	0.974425e-01
0.240000e-05	0.210268e-01	0.210894e-01	0.974525e-01
0.250000e-05	0.213101e-01	0.213752e-01	0.974616e-01
0.260000e-05	0.215693e-01	0.216370e-01	0.974700e-01
0.270000e-05	0.218066e-01	0.218767e-01	0.974776e-01
0.280000e-05	0.220237e-01	0.220964e-01	0.974846e-01
0.290000e-05	0.222224e-01	0.222976e-01	0.974910e-01
0.300000e-05	0.224042e-01	0.224819e-01	0.974969e-01
0.310000e-05	0.225706e-01	0.226508e-01	0.975023e-01
0.320000e-05	0.227228e-01	0.228055e-01	0.975072e-01
0.330000e-05	0.228621e-01	0.229473e-01	0.975117e-01
0.340000e-05	0.229896e-01	0.230772e-01	0.975158e-01
0.350000e-05	0.231062e-01	0.231964e-01	0.975196e-01
0.360000e-05	0.232129e-01	0.233055e-01	0.975230e-01
0.370000e-05	0.233106e-01	0.234056e-01	0.975261e-01
0.380000e-05	0.233999e-01	0.234974e-01	0.975290e-01
0.390000e-05	0.234816e-01	0.235816e-01	0.975317e-01
0.400000e-05	0.235563e-01	0.236588e-01	0.975341e-01
0.410000e-05	0.236247e-01	0.237296e-01	0.975363e-01
0.420000e-05	0.236872e-01	0.237946e-01	0.975383e-01
0.430000e-05	0.237444e-01	0.238542e-01	0.975401e-01
0.440000e-05	0.237967e-01	0.239090e-01	0.975418e-01
0.450000e-05	0.238445e-01	0.239592e-01	0.975434e-01
0.460000e-05	0.238882e-01	0.240054e-01	0.975448e-01
0.470000e-05	0.239281e-01	0.240478e-01	0.975461e-01
0.480000e-05	0.239647e-01	0.240868e-01	0.975472e-01
0.490000e-05	0.239981e-01	0.241226e-01	0.975483e-01
0.500000e-05	0.240286e-01	0.241556e-01	0.975493e-01
0.510000e-05	0.240564e-01	0.241859e-01	0.975502e-01
0.520000e-05	0.240819e-01	0.242138e-01	0.975510e-01
0.530000e-05	0.241051e-01	0.242395e-01	0.975518e-01
0.540000e-05	0.241264e-01	0.242632e-01	0.975525e-01

0.550000e-05	0.241458e-01	0.242850e-01	0.975531e-01
0.560000e-05	0.241635e-01	0.243051e-01	0.975537e-01
0.570000e-05	0.241796e-01	0.243237e-01	0.975542e-01
0.580000e-05	0.241943e-01	0.243409e-01	0.975547e-01
0.590000e-05	0.242078e-01	0.243568e-01	0.975551e-01
0.600000e-05	0.242200e-01	0.243714e-01	0.975555e-01
0.610000e-05	0.242312e-01	0.243850e-01	0.975558e-01
0.620000e-05	0.242413e-01	0.243976e-01	0.975562e-01
0.630000e-05	0.242506e-01	0.244093e-01	0.975565e-01
0.640000e-05	0.242590e-01	0.244201e-01	0.975567e-01
0.650000e-05	0.242666e-01	0.244302e-01	0.975570e-01
0.660000e-05	0.242736e-01	0.244396e-01	0.975572e-01
0.670000e-05	0.242799e-01	0.244484e-01	0.975574e-01
0.680000e-05	0.242856e-01	0.244565e-01	0.975576e-01
0.690000e-05	0.242908e-01	0.244641e-01	0.975578e-01
0.700000e-05	0.242955e-01	0.244713e-01	0.975579e-01
0.710000e-05	0.242997e-01	0.244780e-01	0.975581e-01
0.720000e-05	0.243036e-01	0.244842e-01	0.975582e-01
0.730000e-05	0.243070e-01	0.244901e-01	0.975583e-01
0.740000e-05	0.243101e-01	0.244957e-01	0.975584e-01
0.750000e-05	0.243129e-01	0.245009e-01	0.975585e-01
0.760000e-05	0.243154e-01	0.245058e-01	0.975586e-01
0.770000e-05	0.243177e-01	0.245105e-01	0.975586e-01
0.780000e-05	0.243197e-01	0.245150e-01	0.975587e-01
0.790000e-05	0.243215e-01	0.245192e-01	0.975588e-01
0.800000e-05	0.243230e-01	0.245232e-01	0.975588e-01
0.810000e-05	0.243244e-01	0.245270e-01	0.975589e-01
0.820000e-05	0.243257e-01	0.245307e-01	0.975589e-01
0.830000e-05	0.243267e-01	0.245342e-01	0.975589e-01
0.840000e-05	0.243276e-01	0.245375e-01	0.975590e-01
0.850000e-05	0.243284e-01	0.245408e-01	0.975590e-01
0.860000e-05	0.243291e-01	0.245439e-01	0.975590e-01
0.870000e-05	0.243297e-01	0.245469e-01	0.975590e-01
0.880000e-05	0.243302e-01	0.245498e-01	0.975590e-01
0.890000e-05	0.243305e-01	0.245526e-01	0.975590e-01
0.900000e-05	0.243308e-01	0.245553e-01	0.975591e-01
0.910000e-05	0.243310e-01	0.245580e-01	0.975591e-01
0.920000e-05	0.243312e-01	0.245605e-01	0.975591e-01
0.930000e-05	0.243313e-01	0.245631e-01	0.975591e-01
0.940000e-05	0.243313e-01	0.245655e-01	0.975591e-01
0.950000e-05	0.243313e-01	0.245679e-01	0.975591e-01
0.960000e-05	0.243312e-01	0.245703e-01	0.975591e-01
0.970000e-05	0.243311e-01	0.245726e-01	0.975591e-01
0.980000e-05	0.243309e-01	0.245748e-01	0.975591e-01
0.990000e-05	0.243307e-01	0.245771e-01	0.975591e-01
0.100000e-04	0.243304e-01	0.245793e-01	0.975590e-01

example 7.1 (fall)

time	x 1	x 2	x 5
0.200000e-03	0. e+00	0. e+00	0.967742e-01
0.400000e-03	0.233962e-01	0.265508e-01	0.975289e-01
0.600000e-03	0.226451e-01	0.298578e-01	0.975047e-01
0.800000e-03	0.213323e-01	0.327404e-01	0.974623e-01
0.100000e-02	0.204702e-01	0.360055e-01	0.974345e-01
0.120000e-02	0.194843e-01	0.389921e-01	0.974027e-01
0.140000e-02	0.185219e-01	0.418226e-01	0.973717e-01
0.160000e-02	0.176122e-01	0.445246e-01	0.973423e-01
0.180000e-02	0.167464e-01	0.470946e-01	0.973144e-01
0.200000e-02	0.159225e-01	0.495383e-01	0.972878e-01
0.220000e-02	0.151393e-01	0.518620e-01	0.972626e-01
0.240000e-02	0.143945e-01	0.540714e-01	0.972385e-01
0.260000e-02	0.136864e-01	0.561722e-01	0.972157e-01
0.280000e-02	0.130131e-01	0.581696e-01	0.971940e-01
0.300000e-02	0.123729e-01	0.600688e-01	0.971733e-01
0.320000e-02	0.117642e-01	0.618745e-01	0.971537e-01
0.340000e-02	0.111855e-01	0.635914e-01	0.971350e-01
0.360000e-02	0.106352e-01	0.652238e-01	0.971173e-01
0.380000e-02	0.101120e-01	0.667759e-01	0.971004e-01
0.400000e-02	0.961453e-02	0.682517e-01	0.970843e-01
0.420000e-02	0.914154e-02	0.696549e-01	0.970691e-01
0.440000e-02	0.869182e-02	0.709890e-01	0.970546e-01
0.460000e-02	0.826423e-02	0.722575e-01	0.970408e-01
0.480000e-02	0.785767e-02	0.734636e-01	0.970277e-01
0.500000e-02	0.747111e-02	0.746104e-01	0.970152e-01
0.520000e-02	0.710357e-02	0.757007e-01	0.970033e-01
0.540000e-02	0.675411e-02	0.767374e-01	0.969921e-01
0.560000e-02	0.642184e-02	0.777231e-01	0.969813e-01
0.580000e-02	0.610592e-02	0.786604e-01	0.969712e-01
0.600000e-02	0.580554e-02	0.795515e-01	0.969615e-01
0.620000e-02	0.551993e-02	0.803987e-01	0.969523e-01
0.640000e-02	0.524838e-02	0.812043e-01	0.969435e-01
0.660000e-02	0.499018e-02	0.819703e-01	0.969352e-01
0.680000e-02	0.474469e-02	0.826986e-01	0.969272e-01
0.700000e-02	0.451128e-02	0.833910e-01	0.969197e-01
0.720000e-02	0.428934e-02	0.840494e-01	0.969126e-01
0.740000e-02	0.407833e-02	0.846754e-01	0.969058e-01
0.760000e-02	0.387770e-02	0.852706e-01	0.968993e-01
0.780000e-02	0.368693e-02	0.858365e-01	0.968931e-01
0.800000e-02	0.350555e-02	0.863746e-01	0.968873e-01
0.820000e-02	0.333310e-02	0.868862e-01	0.968817e-01
0.840000e-02	0.316913e-02	0.873727e-01	0.968764e-01
0.860000e-02	0.301322e-02	0.878352e-01	0.968714e-01
0.880000e-02	0.286498e-02	0.882749e-01	0.968666e-01
0.900000e-02	0.272404e-02	0.886930e-01	0.968621e-01
0.920000e-02	0.259003e-02	0.890906e-01	0.968577e-01
0.940000e-02	0.246262e-02	0.894686e-01	0.968536e-01
0.960000e-02	0.234147e-02	0.898280e-01	0.968497e-01
0.980000e-02	0.222628e-02	0.901697e-01	0.968460e-01
0.100000e-01	0.211676e-02	0.904946e-01	0.968425e-01
0.102000e-01	0.201262e-02	0.908035e-01	0.968391e-01
0.104000e-01	0.191361e-02	0.910973e-01	0.968359e-01
0.106000e-01	0.181947e-02	0.913765e-01	0.968329e-01
0.108000e-01	0.172996e-02	0.916421e-01	0.968300e-01

0. 110000e-01	0. 164486e-02	0. 918946e-01	0. 968273e-01
0. 112000e-01	0. 156394e-02	0. 921346e-01	0. 968246e-01
0. 114000e-01	0. 148700e-02	0. 923629e-01	0. 968222e-01
0. 116000e-01	0. 141385e-02	0. 925799e-01	0. 968198e-01
0. 118000e-01	0. 134429e-02	0. 927862e-01	0. 968176e-01
0. 120000e-01	0. 127816e-02	0. 929824e-01	0. 968154e-01
0. 122000e-01	0. 121528e-02	0. 931689e-01	0. 968134e-01
0. 124000e-01	0. 115549e-02	0. 933463e-01	0. 968115e-01
0. 126000e-01	0. 109865e-02	0. 935149e-01	0. 968096e-01
0. 128000e-01	0. 104460e-02	0. 936753e-01	0. 968079e-01
0. 130000e-01	0. 993212e-03	0. 938277e-01	0. 968062e-01
0. 132000e-01	0. 944351e-03	0. 939727e-01	0. 968047e-01
0. 134000e-01	0. 897894e-03	0. 941105e-01	0. 968032e-01
0. 136000e-01	0. 853722e-03	0. 942415e-01	0. 968017e-01
0. 138000e-01	0. 811723e-03	0. 943661e-01	0. 968004e-01
0. 140000e-01	0. 771790e-03	0. 944846e-01	0. 967991e-01
0. 142000e-01	0. 733822e-03	0. 945972e-01	0. 967979e-01
0. 144000e-01	0. 697721e-03	0. 947043e-01	0. 967967e-01
0. 146000e-01	0. 663397e-03	0. 948062e-01	0. 967956e-01
0. 148000e-01	0. 630761e-03	0. 949030e-01	0. 967945e-01
0. 150000e-01	0. 599731e-03	0. 949950e-01	0. 967935e-01
0. 152000e-01	0. 570227e-03	0. 950826e-01	0. 967926e-01
0. 154000e-01	0. 542175e-03	0. 951658e-01	0. 967917e-01
0. 156000e-01	0. 515502e-03	0. 952449e-01	0. 967908e-01
0. 158000e-01	0. 490142e-03	0. 953201e-01	0. 967900e-01
0. 160000e-01	0. 466030e-03	0. 953917e-01	0. 967892e-01
0. 162000e-01	0. 443103e-03	0. 954597e-01	0. 967885e-01
0. 164000e-01	0. 421305e-03	0. 955243e-01	0. 967878e-01
0. 166000e-01	0. 400579e-03	0. 955858e-01	0. 967871e-01
0. 168000e-01	0. 380872e-03	0. 956443e-01	0. 967865e-01
0. 170000e-01	0. 362135e-03	0. 956999e-01	0. 967859e-01
0. 172000e-01	0. 344320e-03	0. 957527e-01	0. 967853e-01
0. 174000e-01	0. 327381e-03	0. 958030e-01	0. 967848e-01
0. 176000e-01	0. 311275e-03	0. 958508e-01	0. 967842e-01
0. 178000e-01	0. 295962e-03	0. 958962e-01	0. 967837e-01
0. 180000e-01	0. 281402e-03	0. 959394e-01	0. 967833e-01
0. 182000e-01	0. 267559e-03	0. 959805e-01	0. 967828e-01
0. 184000e-01	0. 254396e-03	0. 960195e-01	0. 967824e-01
0. 186000e-01	0. 241881e-03	0. 960566e-01	0. 967820e-01
0. 188000e-01	0. 229982e-03	0. 960919e-01	0. 967816e-01
0. 190000e-01	0. 218668e-03	0. 961255e-01	0. 967812e-01
0. 192000e-01	0. 207910e-03	0. 961574e-01	0. 967809e-01
0. 194000e-01	0. 197682e-03	0. 961877e-01	0. 967806e-01
0. 196000e-01	0. 187957e-03	0. 962166e-01	0. 967803e-01
0. 198000e-01	0. 178711e-03	0. 962440e-01	0. 967800e-01
0. 200000e-01	0. 169919e-03	0. 962701e-01	0. 967797e-01

examples 7.2

time	x 1	x 2	x 5
0.100000e-06	0.195933e-02	0.196247e-02	0.968651e-01
0.200000e-06	0.381049e-02	0.381672e-02	0.969233e-01
0.300000e-06	0.552536e-02	0.553464e-02	0.969774e-01
0.400000e-06	0.710259e-02	0.711487e-02	0.970272e-01
0.500000e-06	0.854935e-02	0.856458e-02	0.970728e-01
0.600000e-06	0.987510e-02	0.989323e-02	0.971147e-01
0.700000e-06	0.110895e-01	0.111105e-01	0.971531e-01
0.800000e-06	0.122017e-01	0.122255e-01	0.971883e-01
0.900000e-06	0.132202e-01	0.132469e-01	0.972206e-01
0.100000e-05	0.141530e-01	0.141824e-01	0.972502e-01
0.110000e-05	0.150072e-01	0.150393e-01	0.972773e-01
0.120000e-05	0.157894e-01	0.158242e-01	0.973021e-01
0.130000e-05	0.165057e-01	0.165432e-01	0.973249e-01
0.140000e-05	0.171616e-01	0.172018e-01	0.973457e-01
0.150000e-05	0.177623e-01	0.178051e-01	0.973648e-01
0.160000e-05	0.183123e-01	0.183578e-01	0.973823e-01
0.170000e-05	0.188160e-01	0.188641e-01	0.973983e-01
0.180000e-05	0.192772e-01	0.193279e-01	0.974130e-01
0.190000e-05	0.196995e-01	0.197528e-01	0.974265e-01
0.200000e-05	0.200862e-01	0.201421e-01	0.974388e-01
0.210000e-05	0.204403e-01	0.204987e-01	0.974501e-01
0.220000e-05	0.207646e-01	0.208255e-01	0.974605e-01
0.230000e-05	0.210614e-01	0.211249e-01	0.974700e-01
0.240000e-05	0.213333e-01	0.213992e-01	0.974787e-01
0.250000e-05	0.215821e-01	0.216506e-01	0.974867e-01
0.260000e-05	0.218100e-01	0.218810e-01	0.974940e-01
0.270000e-05	0.220187e-01	0.220922e-01	0.975007e-01
0.280000e-05	0.222097e-01	0.222857e-01	0.975068e-01
0.290000e-05	0.223846e-01	0.224631e-01	0.975124e-01
0.300000e-05	0.225447e-01	0.226256e-01	0.975176e-01
0.310000e-05	0.226912e-01	0.227747e-01	0.975223e-01
0.320000e-05	0.228254e-01	0.229114e-01	0.975267e-01
0.330000e-05	0.229482e-01	0.230367e-01	0.975306e-01
0.340000e-05	0.230607e-01	0.231516e-01	0.975343e-01
0.350000e-05	0.231636e-01	0.232569e-01	0.975376e-01
0.360000e-05	0.232578e-01	0.233536e-01	0.975407e-01
0.370000e-05	0.233440e-01	0.234423e-01	0.975435e-01
0.380000e-05	0.234229e-01	0.235237e-01	0.975461e-01
0.390000e-05	0.234952e-01	0.235983e-01	0.975485e-01
0.400000e-05	0.235613e-01	0.236669e-01	0.975507e-01
0.410000e-05	0.236217e-01	0.237298e-01	0.975526e-01
0.420000e-05	0.236771e-01	0.237876e-01	0.975545e-01
0.430000e-05	0.237277e-01	0.238406e-01	0.975562e-01
0.440000e-05	0.237740e-01	0.238894e-01	0.975577e-01
0.450000e-05	0.238163e-01	0.239342e-01	0.975591e-01
0.460000e-05	0.238551e-01	0.239754e-01	0.975604e-01
0.470000e-05	0.238905e-01	0.240132e-01	0.975616e-01
0.480000e-05	0.239229e-01	0.240481e-01	0.975628e-01
0.490000e-05	0.239525e-01	0.240801e-01	0.975638e-01
0.500000e-05	0.239796e-01	0.241096e-01	0.975647e-01
0.510000e-05	0.240044e-01	0.241368e-01	0.975656e-01
0.520000e-05	0.240270e-01	0.241619e-01	0.975664e-01
0.530000e-05	0.240476e-01	0.241850e-01	0.975671e-01
0.540000e-05	0.240665e-01	0.242063e-01	0.975678e-01

0.550000e-05	0.240837e-01	0.242259e-01	0.975684e-01
0.560000e-05	0.240995e-01	0.242441e-01	0.975690e-01
0.570000e-05	0.241138e-01	0.242609e-01	0.975695e-01
0.580000e-05	0.241269e-01	0.242764e-01	0.975700e-01
0.590000e-05	0.241389e-01	0.242908e-01	0.975705e-01
0.600000e-05	0.241498e-01	0.243041e-01	0.975709e-01
0.610000e-05	0.241597e-01	0.243165e-01	0.975713e-01
0.620000e-05	0.241687e-01	0.243279e-01	0.975716e-01
0.630000e-05	0.241770e-01	0.243386e-01	0.975720e-01
0.640000e-05	0.241845e-01	0.243485e-01	0.975723e-01
0.650000e-05	0.241913e-01	0.243578e-01	0.975726e-01
0.660000e-05	0.241974e-01	0.243664e-01	0.975729e-01
0.670000e-05	0.242031e-01	0.243744e-01	0.975731e-01
0.680000e-05	0.242082e-01	0.243819e-01	0.975734e-01
0.690000e-05	0.242128e-01	0.243890e-01	0.975736e-01
0.700000e-05	0.242169e-01	0.243956e-01	0.975738e-01
0.710000e-05	0.242207e-01	0.244018e-01	0.975740e-01
0.720000e-05	0.242241e-01	0.244076e-01	0.975742e-01
0.730000e-05	0.242272e-01	0.244131e-01	0.975743e-01
0.740000e-05	0.242300e-01	0.244183e-01	0.975745e-01
0.750000e-05	0.242325e-01	0.244232e-01	0.975747e-01
0.760000e-05	0.242347e-01	0.244279e-01	0.975748e-01
0.770000e-05	0.242367e-01	0.244323e-01	0.975750e-01
0.780000e-05	0.242384e-01	0.244365e-01	0.975751e-01
0.790000e-05	0.242400e-01	0.244405e-01	0.975752e-01
0.800000e-05	0.242414e-01	0.244443e-01	0.975753e-01
0.810000e-05	0.242426e-01	0.244479e-01	0.975755e-01
0.820000e-05	0.242437e-01	0.244514e-01	0.975756e-01
0.830000e-05	0.242446e-01	0.244548e-01	0.975757e-01
0.840000e-05	0.242454e-01	0.244580e-01	0.975758e-01
0.850000e-05	0.242461e-01	0.244611e-01	0.975759e-01
0.860000e-05	0.242467e-01	0.244641e-01	0.975760e-01
0.870000e-05	0.242472e-01	0.244670e-01	0.975761e-01
0.880000e-05	0.242476e-01	0.244698e-01	0.975761e-01
0.890000e-05	0.242479e-01	0.244726e-01	0.975762e-01
0.900000e-05	0.242481e-01	0.244752e-01	0.975763e-01
0.910000e-05	0.242483e-01	0.244778e-01	0.975764e-01
0.920000e-05	0.242484e-01	0.244804e-01	0.975765e-01
0.930000e-05	0.242484e-01	0.244828e-01	0.975766e-01
0.940000e-05	0.242484e-01	0.244852e-01	0.975766e-01
0.950000e-05	0.242484e-01	0.244876e-01	0.975767e-01
0.960000e-05	0.242483e-01	0.244899e-01	0.975768e-01
0.970000e-05	0.242481e-01	0.244922e-01	0.975769e-01
0.980000e-05	0.242479e-01	0.244944e-01	0.975769e-01
0.990000e-05	0.242477e-01	0.244967e-01	0.975770e-01
0.100000e-04	0.242475e-01	0.244988e-01	0.975771e-01