CANONICAL PIECEWISE - LINEAR ANALYSIS

by

L. O. Chua and R. L. P. Ying

Memorandum No. UCB/ERL M82/25

23 February 1982

CANONICAL PIECEWISE - LINEAR ANALYSIS

by

Leon O. Chua and Robin L. P. Ying

Memorandum UCB/ERL M82/25

23 February 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CANONICAL PIECEWISE – LINEAR ANALYSIS[†]

Leon O. Chua and Robin L.P. Ying[§]

ABSTRACT

Any resistive nonlinear circuit can be approximated to any desired accuracy by a global piecewise-linear equation in the *canonical form*

$$\mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{i=1}^{p} \mathbf{c}_i \mid < \alpha_i, \mathbf{x} > - \beta_i \mid = 0$$

All conventional circuit analysis methods (nodal, mesh, cut set, loop, hybrid, modified nodal, tableau) are shown to always yield an equation of this form, provided the only *non-linear* elements are 2-terminal resistors and controlled sources, each modeled by a 1-dimensional piecewise-linear function.

The well-known Katzenelson algorithm when applied to this equation yields an efficient algorithm which requires only a minimal computer storage. In the important special case when the canonical equation has a *lattice structure* (which always occur in the hybrid analysis), the algorithm is further refined to achieve a dramatic reduction in computation time.

† Research supported in part by the Joint Services Electronics Program Contract F49620-79-C-0178, by the Office of Naval Research under Contract N00014-76-C-0572, and the National Science Foundation under Grants ECS 80-20-640/ENG-7722745.

§ The authors are with the Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, CA 94720.

1. INTRODUCTION

The Katzenelson algorithm [1] for solving piecewise-linear resistive circuits has been refined and extended to the most general case of a system of piecewise-linear equations $\mathbf{f}(\mathbf{x}) = 0$ in a series of excellent papers during the last decade [2-6]. In these papers, the n-dimensional piecewise-linear functions are specified by

$$\mathbf{f}(\mathbf{x}) = \mathbf{J}_i\,\mathbf{x} + \mathbf{s}_i\ , \quad i = 1, 2, \ldots, K \tag{1.1}$$

where $\mathbf{J}_i$ is a constant $n \times n$ matrix and $\mathbf{s}_i$ is a constant n-vector, both defined in region $R_i \subset \mathbb{R}^n$. The whole space $\mathbb{R}^n$ is divided into a finite number (K) of polyhedral regions by a finite number (p) of hyperplanes

$$< \alpha_i\ , \mathbf{x} > = \beta_i\ , \quad i = 1, 2, \ldots, p\ , \quad \mathbf{x} \in \mathbb{R}^n \tag{1.2}$$

where α_i is the ***normal vector*** to the i-th hyperplane, β_i is a constant and $< , >$ denote the usual vector "dot" product in $\mathbb{R}^n$. For illustration, see *Example 1* (n=2, k=4) in *Section 2.3*.

In order to solve (1.1) and (1.2) in a computer, it is necessary to store an $n \times n$ matrix $\mathbf{J}_i$, two n-vectors $\mathbf{s}_i$ and α_i and a scalar β_i for *each* of the K polyhedral regions. The number K depends on the number of regions required to approximate a nonlinear function $\mathbf{f}(\mathbf{x})$ to within acceptable accuracy. For $n > 100$, K is generally an extremely large number. Hence for large n, it becomes extremely tedious and error prone for a user to input all the data necessary to specify (1.1) and (1.2). Moreover, this data also requires an excessive amount of computer memory. For most piecewise-linear functions of practical interest, the above objection can be overcome by representing (1.1) and (1.2) by a single piecewise-linear equation in the canonical form [8]

$$\mathbf{f}(\mathbf{x}) = \mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{i=1}^{p} \mathbf{c}_i \mid < \alpha_i, \mathbf{x} > - \beta_i \mid = 0 \tag{1.3}$$

where $\mathbf{B}$ is a constant $n \times n$ matrix, $\mathbf{a}$, $\mathbf{c}_i$, and α_i are constant n-vectors, and β_i is a constant.

Our first objective is to show, in *section 2*, that any piecewise-linear function defined by (1.1) and (1.2) which satisfies the *linear partition* assumption in [8] has an equivalent *canonical representation* (1.3), where $\mathbf{a}$, $\mathbf{B}$, $\mathbf{c}_i$, α_i and β_i can be calculated via *explicit formulas*.

Our second objective in this paper is to show, in *section 3*, that if the only *nonlinear* elements in a resistive circuit are 2-terminal resistors and/or controlled sources (all 4 types) modeled by *arbitrary* 1-dimensional piecewise-linear functions, then any conventional circuit analysis method (nodal, mesh, loop, cut set, hybrid, modified nodal, tableau, etc.) *always* gives rise to an equation having the canonical form (1.3). Since the above repertoire of nonlinear elements are sufficient to model all resistive n-terminal or coupled elements [9-10], we conclude that *all resistive nonlinear circuits* can be modeled, to within any prescribed accuracy, by an equation in the canonical form (1.3).

Our final objective is to apply the Katzenelson algorithm developed in [6] to the canonical form (1.3). The resulting method -- called the *canonical Katzenelson algorithm* -- given in *section 4* is applicable to *any* system of piecewise-linear equations represented in the canonical form (1.3), regardless of whether it comes from a circuit or not.

2. CANONICAL FORM REPRESENTATION OF CONTINUOUS PIECEWISE-LINEAR FUNCTIONS

In this section, we discuss the canonical form representation of continuous piecewise-linear functions. First, we review briefly the single-variable ($f : \mathbb{R}^1 \to \mathbb{R}^1$) case in *section 2.1* and the scalar multi-variable ($f : \mathbb{R}^n \to \mathbb{R}^1$, $n > 1$) case in *section 2.2*. We also present a new formula for determining the coefficients of f in the multi-variable case which is more general than that given in [8]. In *section 2.3* we propose a general compact form for representing an n-dimensional piecewise-linear function ($f : \mathbb{R}^n \to \mathbb{R}^n$, $n > 1$).

2.1 1-dimensional scalar function: $f : \mathbb{R}^1 \to \mathbb{R}^1$

A function $f : \mathbb{R}^1 \to \mathbb{R}^1$ is said to be *continuous piecewise-linear* if (1) it is continuous, and (2) it is composed of finitely many linear segments. Points common to two segments of different slopes are called *break points*.

Consider an arbitrary continuous piecewise-linear function f with p distinct break-points $x_1 < x_2 < \ldots. < x_p$ as shown in Fig. 1. Let m_i , $i = 0, 1, 2, \ldots. , p$ denote the slope of each segment. It is proved in [7] that f can be represented globally by the following canonical form :

$$\boxed{f(x) = a + bx + \sum_{i=1}^{p} c_i \mid x - x_i \mid} \tag{2.1}$$

where $x \in \mathbb{R}^1$ and the coefficients can be calculated explicitly by :

$$b = \frac{1}{2}(m_0 + m_p) \tag{2.2}$$

$$c_i = \frac{1}{2}(m_i - m_{i-1}), \qquad i = 1, 2, \ldots. , p \tag{2.3}$$

$$a = f(0) - \sum_{i=1}^{p} c_i \mid x_i \mid \tag{2.4}$$

A simplified derivation of these relationships are given in *Appendix A*. It should be noted that since (2.2)–(2.4) defines a *unique* set of coefficients for each piecewise-linear function, the representation (2.1) is also unique.

2.2 n-dimensional scalar function: $f : \mathbb{R}^n \to \mathbb{R}^1$

A *linear partition* in $\mathbb{R}^n$ is a finite set of hyperplanes characterized by the equation

$$< \alpha_i , \mathbf{x} > = \beta_i, \quad i = 1, 2, \ldots. , p, \quad \mathbf{x} \in \mathbb{R}^n \tag{2.5}$$

where $\alpha_i \in \mathbb{R}^n$ denotes the normal vector to the i-th hyperplane. A linear partition is said to be *non-degenerate* if for every set of linearly-dependent $\alpha_{j1}, \alpha_{j2}, \ldots. , \alpha_{j_m}$, $1 \le j_1, j_2, \ldots. , j_m \le p$, the rank of $[\, \alpha_{j_1}, \alpha_{j_2}, \ldots. , \alpha_{j_m} \,]$ is strictly less than the rank of $\begin{bmatrix} \alpha_{j_1} & \alpha_{j_2} & \ldots. & \alpha_{j_m} \\ \beta_{j_1} & \beta_{j_2} & \ldots. & \beta_{j_m} \end{bmatrix}$. Geometrically, this means that the dimension of the intersection among any j of the p hyperplanes must be less than or equal to $n - j$ where n is the dimension of the space. For example, if three lines intersect at a common point in $\mathbb{R}^2$, then any partition in $\mathbb{R}^2$ containing those three lines is degenerate.

A linear partition in $\mathbb{R}^n$ separates $\mathbb{R}^n$ into many polyhedral regions. There are unbounded regions as well as bounded regions. The unbounded regions can be divided into two classes. Suppose some of the hyperplanes in the linear partition, or their intersections, are parallel to each other. As we translate these hyperplanes (or their intersections) until they coincide into a single hyperplane (or intersection) some of the unbounded regions may vanish. We call them the *pseudo-unbounded regions*. The remaining regions which remain unbounded in spite of the above translations are called *essentially-unbounded regions*. For example, in Fig. 2(a), regions R_1 and R_5 are pseudo-unbounded since they vanish upon merging L_1 and L_2. Regions R_2, R_3, R_4, R_6, R_7 and R_8 are essentially-unbounded. In Fig. 2(b), R_7 is pseudo-unbounded since it vanishes upon merging the parallel intersections L_1, L_2 and L_3. Note that we can always eliminate bounded regions as well as pseudo-unbounded regions by merging the parallel hyperplanes and/or their intersections.

Consider a linear partition in $\mathbb{R}^n$ defined by (2.5) and an arbitrary polyhedral region $R_k \in \mathbb{R}^n$. Let $\mathbf{x}$ be an interior point of R_k. Then we have $< \alpha_i , \mathbf{x} > \neq \beta_i$ for $i = 1, 2, \dots, p$. If we define

$$\omega_{k_i}(\mathbf{x}) = sgn\ (< \alpha_i , \mathbf{x} > - \beta_i), \qquad i = 1, 2, \dots, p \tag{2.6}$$

then the $p \times 1$ vector $\mathbf{w}_k(\mathbf{x}) = [\ \omega_{k_1}(\mathbf{x}), \omega_{k_2}(\mathbf{x}), \dots\ \omega_{k_p}(\mathbf{x})\ ]^T$ is called the *sign-sequence vector* of R_k. We can actually drop the dependence of $\mathbf{x}$ on $\mathbf{w}_k$ since by (2.6) $\mathbf{w}_k$ remains the same for all $\mathbf{x}$ in the interior of R_k. Thus *a polyhedral region is uniquely identified by its sign-sequence vector*.

It is shown in [8] that any continuous piecewise-linear function $f : \mathbb{R}^n \to \mathbb{R}^1$ with a non-degenerate linear partition defined by (2.5) can be represented globally by the following canonical form :

$$\boxed{f(\mathbf{x}) = a + < \mathbf{b} , \mathbf{x} > + \sum_{i=1}^{p} c_i \mid < \alpha_i , \mathbf{x} > - \beta_i \mid} \tag{2.7}$$

where $\mathbf{x}$, $\mathbf{b}$, α_i, $i = 1, 2, \dots, p$ are in $\mathbb{R}^n$, and the coefficients can be calculated explicitly by

$$\mathbf{b} = \frac{1}{k} \sum_{j=1}^{k} \nabla f(\mathbf{x}) |_{R_{j\infty}} \tag{2.8}$$

$$c_i = \frac{1}{2} \frac{\alpha_i^T (\nabla f(\mathbf{x}) |_{R_{i+}} - \nabla f(\mathbf{x}) |_{R_{i-}})}{< \alpha_i , \alpha_i >} \tag{2.9}$$

$$a = f(0) - \sum_{i=1}^{p} c_i \mid \beta_i \mid \tag{2.10}$$

where $R_{j\infty}$, $j = 1, 2, \dots, k$, denotes the *essentially-unbounded regions*, and R_{i+} and R_{i-} denote the two adjacent regions (associated with the i-th hyperplane) where their *sign-sequence vectors* differ only at the i-th position. A simplified derivation in of these relationships are given in *Appendix B*.

Note that for any $t \in \mathbb{R}$, $t \neq 0$, $< t\alpha_i , \mathbf{x} > = t\beta_i$ and (2.5) represent the same hyperplane. Hence, the coefficients in (2.7) as computed by (2.8)–(2.10) are not unique but rather depend on the scale factor t used in representing the linear partition. However, if we normalize the representation (2.5) by defining the scale factor $t_i \overset{def}{=} \frac{sgn(\beta_i)}{\sqrt{< \alpha_i , \alpha_i >}}$, for $i = 1, 2, \dots, p$, $\beta_i \neq 0$,[1] then we will get a unique representation for the linear partition as well as a unique set of coefficients for the canonical form of $f(.)$.

Remark : Non-degeneracy is only a sufficient condition for the existence of a canonical form representation. Examples exist (see [8]) such that a piecewise-linear function $f(.)$ can be represented in canonical form (2.7) even though its associated linear partition is degenerate. We will show in *section 3* that in formulating equations for piecewise-linear circuits, this sufficient condition can be ignored.

2.3 n-dimensional vector function: $f : \mathbb{R}^n \to \mathbb{R}^n$

Definition 2.1

An n-dimensional function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is said to be continuous piecewise-linear if all its components $f_i : \mathbb{R}^n \to \mathbb{R}^1$, $i = 1, 2, \dots, n$ are continuous piecewise-linear.

Any n-dimensional function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ with its components $f_i : \mathbb{R}^n \to \mathbb{R}^1$ represented in the canonical form (2.7) can be represented globally by the following explicit analytical expression :

$$\boxed{\mathbf{f}(\mathbf{x}) = \mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{i=1}^{p} \mathbf{c}_i \mid < \alpha_i , \mathbf{x} > - \beta_i \mid} \tag{2.11}$$

where $\mathbf{x}, \mathbf{a}, \mathbf{c}_i \in \mathbb{R}^n$ and $\mathbf{B} \in \mathbb{R}^{n \times n}$. The coefficients are given explicitly by :

$$\mathbf{B} = \frac{1}{k} \sum_{j=1}^{k} \mathbf{J}(\mathbf{x}) \mid_{\mathbf{x} \in R_{j\infty}} \tag{2.12}$$

$$\mathbf{c}_i = \frac{1}{2} \frac{[\, \mathbf{J}(\mathbf{x}) \mid_{\mathbf{x} \in R_{i+}} - \mathbf{J}(\mathbf{x}) \mid_{\mathbf{x} \in R_{i-}}]\, \alpha_i}{< \alpha_i , \alpha_i >} \tag{2.13}$$

$$\mathbf{a} = \mathbf{f}(0) - \sum_{i=1}^{p} \mathbf{c}_i \mid \beta_i \mid \tag{2.14}$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of $\mathbf{f}$; $R_{j\infty}$, $j = 1, 2, \dots, k$, denotes the essentially-unbounded regions, and R_{i+}, R_{i-} denote the two adjacent regions where their sign-sequence vectors differs only at the i-th position.

The derivation of (2.12) - (2.14) is similar to that of (2.8) - (2.10). The coefficients in (2.11) clearly depends on the scalar factor used in representing the linear partition.

It is important to note that in generalizing from (2.7) to (2.11), we have assumed that each component f_i of $\mathbf{f}$ is associated with the same linear partition in the domain. This

[1]For the case $\beta_i = 0$ where $sgn(\beta_i)$ is undefined, we choose $t_i \overset{def}{=} \frac{1}{\sqrt{< \alpha_i , \alpha_i >}}$.

assumption, however, entails no loss of generality since we can always introduce additional hyperplanes for each f_i until their linear partitions are identical for all $i = 1, 2, \dots, p$.

Remark : We can write (2.11) in an even more compact matrix form. Define $\mathbf{C} \overset{def}{=} [\, \mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_p \,] \in \mathbb{R}^{n \times p}$, $\mathbf{D} \overset{def}{=} [\, \alpha_1, \alpha_2, \dots, \alpha_p \,] \in \mathbb{R}^{n \times p}$ and $\mathbf{e} \overset{def}{=} [\, \beta_1, \beta_2, \dots, \beta_p \,]^T \in \mathbb{R}^p$, then (2.11) can be written as[2]

$$\mathbf{f}(\mathbf{x}) = \mathbf{a} + \mathbf{B}\mathbf{x} + \mathbf{C}\, abs\, (\, \mathbf{D}^T \mathbf{x} - \mathbf{e} \,) \tag{2.15}$$

Example 1.

The linear partition shown in Fig. 3 is defined by $< [\, 1\ \ -1 \,]^T, \mathbf{x} > = 0$ and $< [\, 1\ \ 1 \,]^T, \mathbf{x} > = 0$. A continuous piecewise-linear function $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ is defined in each region as follow :

$$\mathbf{f}(x_1, x_2) = \begin{cases} \begin{bmatrix} 4x_1 \\ 3x_2 \end{bmatrix}, & x_1, x_2 \in R_1 \ ; \quad \begin{bmatrix} 2x_1 \\ x_2 \end{bmatrix}, \quad x_1, x_2 \in R_3 \\ \begin{bmatrix} 3x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}, & x_1, x_2 \in R_2 \ ; \quad \begin{bmatrix} 3x_1 - x_2 \\ -x_1 + 2x_2 \end{bmatrix}, \quad x_1, x_2 \in R_4 \end{cases}$$

the associated Jacobian matrices are :

$$\mathbf{J}_f(x_1, x_2) = \begin{cases} \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}, & x_1, x_2 \in R_1 \ ; \quad \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad x_1, x_2 \in R_3 \\ \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, & x_1, x_2 \in R_2 \ ; \quad \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix}, \quad x_1, x_2 \in R_4 \end{cases}$$

using (2.12) - (2.14) to compute the coefficients, we can express $\mathbf{f}$ in the form of either (2.11) or (2.15) :

$$\mathbf{f}(x_1, x_2) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} |\, x_1 + x_2 \,| + \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} |\, x_1 - x_2 \,|$$

$$= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} abs \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \qquad \square$$

3. CANONICAL EQUATION FORMULATION FOR PIECEWISE-LINEAR RESISTIVE CIRCUITS

Consider the simple circuit shown in Fig. 4. R1 is voltage controlled and R2 is current controlled. The v-i characteristics of R1 and R2 are approximated by continuous piecewise-linear segments and represented in the canonical form (2.1) as follow :

[2]For $\mathbf{x} = [\, x_1, x_2, \dots, x_n \,]^T \in \mathbb{R}^n$, $abs(\mathbf{x}) \overset{def}{=} [\, |x_1|, |x_2|, \dots, |x_n| \,]^T$.

$$\text{R1}: \; i_1 = g(v_1) = a_1 + b_1 v_1 + \sum_{i=1}^{p_1} c_{1i} \, | v_1 - V_{1i} |$$

$$\text{R2}: \; v_2 = r(i_2) = a_2 + b_2 i_2 + \sum_{j=1}^{p_2} c_{2i} \, | i_2 - I_{2j} |$$

KVL implies $v_1 + v_2 + 2i_1 = E$ and KCL implies $i_1 = i_2$. Therefore $v_2 = r(i_2) = r(i_1) = r(g(v_1))$ and the equilibrium equation of the circuit is $v_1 + r(g(v_1)) + 2g(v_1) = E$. We note that this equation is no longer in the canonical form (2.1) since the composition of $r(.)$ and $g(.)$ causes the absolute sign to be *nested*. On the other hand, the equilibrium equation can also be formulated as follow : $i_1 = i_2$ implies

$$v_1 + v_2 + 2\,i_1 = E \quad \text{and} \quad v_1 + v_2 + 2\,i_2 = E$$

Writing these equations in the vector form, we get

$$\begin{bmatrix} v_1 + r(i_2) + 2g(v_1) \\ v_1 + r(i_2) + 2\,i_2 \end{bmatrix} = \begin{bmatrix} E \\ E \end{bmatrix}$$

Substituting $r(.)$ and $g(.)$ into the above equation and rearranging terms, we obtain

$$\begin{bmatrix} 2a_1 + a_2 - E \\ a_2 - E \end{bmatrix} + \begin{bmatrix} 2b_1 + 1 & b_2 \\ 1 & b_2 + 2 \end{bmatrix} + \begin{bmatrix} v_1 \\ i_2 \end{bmatrix} + \sum_{i=1}^{p_1} \begin{bmatrix} 2c_{1i} \\ 0 \end{bmatrix} | v_1 - V_{1i} |$$

$$+ \sum_{j=1}^{p_2} \begin{bmatrix} c_{2j} \\ c_{2j} \end{bmatrix} | i_2 - I_{2j} | = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Indeed, this equation is in the canonical form (3.11) provided that we identify $\mathbf{x} = [\, v_1 \; i_2 \,]^T$, $\alpha_i = [\,1\;0\,]^T$, $\alpha_j = [\,0\;1\,]^T$, $\beta_i = V_{1i}$, $\beta_j = I_{2j}$ for $i = 1, 2, \ldots, p_1$, $j = 1, 2, \ldots, p_2$.

The above discussion shows that arbitrary elimination of variables will *not*, in general, give rise to a system of equations in canonical form, even though KVL and KCL are *linear* equations and the constitutive relation of all resistors are expressed in canonical form (2.1). However, by proper formulation, it is possible to reduce the equations in canonical form.

Consider the class of nonlinear circuits made of linear and nonlinear 2-terminal resistors, linear and nonlinear controlled sources (all 4 types), each depending on a single controlling variable, dc independent sources (battery and current sources), as well as any linear multi-terminal resistive elements, such as ideal transformers gyrators, etc. Through equivalent circuit transformation techniques, virtually any dc resistive circuit can be reduced to this class [9-10]. Our objective in this section is to show that if we approximate the characteristics defining each nonlinear resistor or nonlinear controlled source by a continuous piecewise-linear function and represent it in the canonical form (2.1), then, for *all conventional methods of circuit formulation*, the equilibrium equation of the resulting circuit will always assume the n-dimensional canonical form given in (2.11). We will prove this rather remarkable and unifying result for each of the following common equation formulation methods : *nodal analysis*, *mesh analysis*, *cut set analysis*, *loop analysis*, *hybrid analysis*, *modified nodal analysis*, and *tableau formulation*.

3.1 Nodal, mesh, cut-set and loop equations in canonical form

Theorem 3.1 (*Nodal and cut set analysis*)

Consider a connected resistive circuit N containing only linear 2-terminal resistors, dc independent sources, voltage-controlled piecewise-linear 2-terminal resistors and linear or piecewise-linear voltage-controlled current sources. If each piecewise-linear characteristic is represented in the canonical form (2.1), then the *nodal equation* and *cut set equation* of N always assume the canonical form (2.11).

Proof :

Observe that the assumptions assure the existence of the nodal equation. By applying source transformation, we can assume without loss of generality that each branch in N is in the composite form (see Fig. 5) where n_k can be a linear resistor, a voltage-controlled piecewise-linear resistor, a linear voltage-controlled current source or a piecewise-linear voltage-controlled current source. The $v_k - i_k$ relation of n_k takes the following forms :

(1) if n_k is a linear resistor, then

$$i_k = g_k v_k \tag{3.1}$$

(2) if n_k is a piecewise-linear voltage-controlled resistor, then

$$i_k = a_k + b_k v_k + \sum_{i=1}^{p_k} c_{ki} \mid v_k - V_{ki} \mid \tag{3.2}$$

(3) if n_k is a linear voltage-controlled current source, then

$$i_k = g_k v_j \tag{3.3}$$

where $v_j,\ j \neq k$ is the voltage of some other branch.

(4) if n_k is a piecewise-linear voltage-controlled current source, then

$$i_k = a_k + g_k v_j + \sum_{i=1}^{p_k} c_{ki} \mid v_j - V_{ki} \mid \tag{3.4}$$

Now let q be the total number of composite branches in N, $\mathbf{u}_k$, $k = 1, 2, \ldots, q$, be unit vectors in $\mathbb{R}^q$, and define $\hat{\mathbf{v}}_q = [\,\hat{v}_1\,\hat{v}_2 \ldots \hat{v}_q\,]^T$, $\mathbf{E}_q = [\,E_1\,E_2 \ldots E_q\,]^T$, then since $\hat{i}_k = J_k + i_k$, and $v_k = \hat{v}_k + E_k$ for $k = 1, 2, \ldots, q$, we can write (3.1) - (3.4) in the following general form :

$$\hat{i}_k = J_k + a_k + \sum_{j=1}^{q} < b_{k_j} \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q + \mathbf{E}_q > + \sum_{j=1}^{q} \sum_{i=1}^{p_j} c_{ji} \mid < \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q + \mathbf{E}_q > - V_{ji} \mid$$

$$= \left[J_k + a_k + \sum_{j=1}^{q} < b_{k_j} \mathbf{u}_j \,,\, \mathbf{E}_q > \right] + \sum_{j=1}^{q} < b_{k_j} \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q >$$

$$+ \sum_{j=1}^{q} \sum_{i=1}^{p_j} c_{ji} \mid < \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q > - (\, V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q >) \mid$$

Define $\hat{\mathbf{i}}_q = [\,\hat{i}_1\,\hat{i}_2 \ldots \hat{i}_q\,]^T$, then the above equation gives :

$$\hat{\mathbf{i}}_q = \begin{bmatrix} J_1 + a_1 + \sum_{j=1}^{q} < b_{1j}\, \mathbf{u}_j \,,\, \mathbf{E}_q > \\ J_2 + a_2 + \sum_{j=1}^{q} < b_{2j}\, \mathbf{u}_j \,,\, \mathbf{E}_q > \\ \vdots \\ J_q + a_q + \sum_{j=1}^{q} < b_{qj}\, \mathbf{u}_j \,,\, \mathbf{E}_q > \end{bmatrix} + \sum_{j=1}^{q} \begin{bmatrix} b_{1j}\, \mathbf{u}_j^T \\ b_{2j}\, \mathbf{u}_j^T \\ \vdots \\ b_{qj}\, \mathbf{u}_j^T \end{bmatrix} \hat{\mathbf{v}}_q$$

$$+ \sum_{j=1}^{q} \sum_{i=1}^{p_j} c_{ji}\, \mathbf{u}_j \,|\, < \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q > - (\, V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q >) \,|$$

$$\overset{def}{=} \hat{\mathbf{a}} + \hat{\mathbf{B}}\, \hat{\mathbf{v}}_q + \sum_{j=1}^{q} \sum_{i=1}^{p_j} \hat{\mathbf{c}}_{ji} \,|\, < \mathbf{u}_j \,,\, \hat{\mathbf{v}}_q > - (\, V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q >) \,| \tag{3.5}$$

Let $\mathbf{A}$ be the reduced incidence matrix associated with N relative to an arbitrary datum node, and $\mathbf{e}$ be the node-to-datum voltage vector, then KCL implies $\mathbf{A}\,\hat{\mathbf{i}}_q = \mathbf{0}$ and KVL implies $\hat{\mathbf{v}}_q = \mathbf{A}^T\, \mathbf{e}$. Applying these two equations to (3.5), we get :

$$\mathbf{A}\,\hat{\mathbf{a}} + \mathbf{A}\,\hat{\mathbf{B}}\,\mathbf{A}^T\, \mathbf{e} + \sum_{j=1}^{q} \sum_{i=1}^{p_j} \mathbf{A}\,\hat{\mathbf{c}}_{ji} \,|\, < \mathbf{u}_j \,,\, \mathbf{A}^T \mathbf{e} > - (\, V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q >) \,|$$

$$= \mathbf{a} + \mathbf{B}\, \mathbf{e} + \sum_{j=1}^{q} \sum_{i=1}^{p_j} \mathbf{c}_{ji} \,|\, < \mathbf{A}\mathbf{u}_j \,,\, \mathbf{e} > - \beta_{ji} \,| = 0 \tag{3.6}$$

where

$$\mathbf{a} \overset{def}{=} \mathbf{A}\,\hat{\mathbf{a}} \,,\ \mathbf{B} \overset{def}{=} \mathbf{A}\,\hat{\mathbf{B}}\,\mathbf{A}^T \,,\ \mathbf{c}_{ji} \overset{def}{=} \mathbf{A}\,\hat{\mathbf{c}}_{ji} \,,\ \beta_{ji} \overset{def}{=} V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q > \tag{3.7}$$

Note that the *nodal equation* (3.6) is precisely in the form of (2.11) provided we relabel the double indices in the last term.

To formulate the *cut set equation* of N, pick a tree and choose the tree branch voltages $\mathbf{v}_e$ as independent variables. Replacing $\mathbf{A}$ in (3.6) by the fundamental cut set matrix $\mathbf{Q}$, we obtain the following equation

$$\mathbf{Q}\,\hat{\mathbf{a}} + \mathbf{Q}\,\hat{\mathbf{B}}\,\mathbf{Q}^T\, \mathbf{v}_e + \sum_{j=1}^{q} \sum_{i=1}^{p_j} \mathbf{Q}\,\hat{\mathbf{c}}_{ji} \,|\, < \mathbf{u}_j \,,\, \mathbf{Q}^T \mathbf{v}_e > - (\, V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q >) \,|$$

$$= \mathbf{a} + \mathbf{B}\, \mathbf{v}_e + \sum_{j=1}^{q} \sum_{i=1}^{p_j} \mathbf{c}_{ji} \,|\, < \mathbf{Q}\mathbf{u}_j \,,\, \mathbf{v}_e > - \beta_{ji} \,| = 0 \tag{3.8}$$

where

$$\mathbf{a} = \mathbf{Q}\,\hat{\mathbf{a}} \,,\ \mathbf{B} = \mathbf{Q}\,\hat{\mathbf{B}}\,\mathbf{Q}^T \,,\ \mathbf{c}_{ji} = \mathbf{Q}\,\hat{\mathbf{c}}_{ji} \,,\ \beta_{ji} = V_{ji} - < \mathbf{u}_j \,,\, \mathbf{E}_q > \tag{3.9}$$

Equation (3.8) is precisely in the form of (2.11) provided we relabel the double indices in the last term. □

Example 2.

Consider the bridge circuit containing five voltage-controlled piecewise-linear resistors as shown in Fig. 6. Assume the $v-i$ characteristics of these resistors are expressed in the canonical form (2.1) :

$$i_k = a_k + b_k v_k + \sum_{i=1}^{p_k} c_{k_i} \mid v_k - V_{k_i} \mid , \quad k = 1, 2, 3, 4, 5$$

Shift the current source I_s in parallel with R1 and R3 so that the resulting circuit contains only 5 composite branches and 4 nodes. The reduced incidence matrix **A** with respect to datum node ④ is

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ -1 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 1 \end{bmatrix}$$

Computing the coefficients using (3.7), we obtain

$$\mathbf{a} = \mathbf{A} \begin{bmatrix} a_1 - I_s \\ a_2 \\ a_3 - I_s \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} a_1 + a_4 - I_s \\ -a_1 + a_2 + a_3 \\ -a_2 - a_4 + a_5 \end{bmatrix}$$

$$\mathbf{B} = \mathbf{A}\, diag\, [\, b_1\, b_2\, b_3\, b_4\, b_5\,]\, \mathbf{A}^T = \begin{bmatrix} b_1 + b_4 & -b_1 & -b_4 \\ -b_1 & b_1 + b_2 + b_3 & -b_2 \\ -b_4 & -b_2 & b_2 + b_4 + b_5 \end{bmatrix}$$

$$\mathbf{c}_{ji} = c_{ji}\, \mathbf{A}\, \mathbf{u}_j \,, \quad \beta_{ji} = V_{ji} \,, \quad j = 1, 2, 3, 4, 5$$

The nodal equation in the canonical form is :

$$\mathbf{f}\left(\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \right) = \begin{bmatrix} a_1 + a_4 - I_s \\ -a_1 + a_2 + a_3 \\ -a_3 - a_4 + a_5 \end{bmatrix} + \begin{bmatrix} b_1 + b_4 & -b_1 & -b_4 \\ -b_1 & b_1 + b_2 + b_3 & -b_2 \\ -b_4 & -b_2 & b_2 + b_4 + b_5 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}$$

$$+ \sum_{i=1}^{p_1} \begin{bmatrix} c_{1i} \\ -c_{1i} \\ 0 \end{bmatrix} \mid e_1 - e_2 - V_{1i} \mid + \sum_{i=1}^{p_2} \begin{bmatrix} 0 \\ c_{2i} \\ -c_{2i} \end{bmatrix} \mid e_2 - e_3 - V_{2i} \mid$$

$$+ \sum_{i=1}^{p_3} \begin{bmatrix} 0 \\ c_{3i} \\ 0 \end{bmatrix} \mid e_2 - V_{3i} \mid + \sum_{i=1}^{p_4} \begin{bmatrix} c_{4i} \\ 0 \\ -c_{4i} \end{bmatrix} \mid e_1 - e_3 - V_{4i} \mid$$

$$+ \sum_{i=1}^{p_5} \begin{bmatrix} 0 \\ 0 \\ c_{5i} \end{bmatrix} \mid e_3 - V_{5i} \mid = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \square$$

It is remarkable to observe that the resulting nodal equation is in closed analytic form. Changing the resistor characteristics only changes the parameters in this equation.

Theorem 3.2 (*Mesh and loop analysis*)

Consider a connected resistive circuit N containing linear 2-terminal resistors, dc independent sources, current-controlled piecewise-linear 2-terminal resistors and linear or piecewise-linear current-controlled voltage sources. If each piecewise-linear characteristic is represented in the canonical form (2.1), then the ***mesh equation*** and ***loop equation*** of N always assume the canonical form (2.11).

Proof : Dual of *Theorem 3.1.* □

3.2 Canonical equation for modified nodal analysis and tableau formulation

In the case when N contains both voltage and current controlled elements, we may resort to modified nodal analysis.

Theorem 3.3 (Modified nodal analysis)

Consider a connected resistive circuit N containing only linear 2-terminal resistors, dc independent sources, current-controlled and voltage-controlled piecewise-linear 2-terminal resistors, linear and piecewise-linear controlled sources (all 4 types) and any linear multi-terminal resistive elements. If each piecewise-linear function is represented in the canonical form (2.1), then the ***modified nodal analysis*** of N always assumes the canonical form (2.15).

Proof :

Partition the elements in N into the following groups :

$$\mathbf{i}_G = \mathbf{G}(\mathbf{v}_G) + \mathbf{N}(\mathbf{i}_R) \tag{3.10}$$

$$\mathbf{v}_E = \mathbf{M}(\mathbf{v}_G) + \mathbf{S}(\mathbf{i}_R) \tag{3.11}$$

$$\mathbf{v}_R = \mathbf{R}(\mathbf{i}_R) \tag{3.12}$$

where $\mathbf{G}(.)$ in (3.10) includes all voltage-controlled resistors, voltage-controlled current sources and dc current sources; $\mathbf{N}(.)$ in (3.10) includes all current-controlled current sources; $\mathbf{M}(.)$ in (3.11) includes all voltage-controlled voltage sources; $\mathbf{S}(.)$ in (3.11) includes all current-controlled voltage sources and dc voltage sources; $\mathbf{R}(.)$ in (3.12) includes all current-controlled resistors.

Using the same procedure as in the proof of *Theorem 3.1*, we can express (3.10) - (3.12) into the canonical form (2.15) :

$$\mathbf{i}_G = [\,\mathbf{a}_G + \mathbf{B}_G \mathbf{v}_G + \mathbf{C}_G\, abs\,(\mathbf{D}_G^T \mathbf{v}_G - \mathbf{e}_G)] + [\,\mathbf{a}_N + \mathbf{B}_N \mathbf{i}_R + \mathbf{C}_N\, abs\,(\mathbf{D}_N^T \mathbf{i}_R - \mathbf{e}_N)] \tag{3.13}$$

$$\mathbf{v}_E = [\,\mathbf{a}_M + \mathbf{B}_M \mathbf{v}_G + \mathbf{C}_M\, abs\,(\mathbf{D}_M^T \mathbf{v}_G - \mathbf{e}_M)] + [\,\mathbf{a}_S + \mathbf{B}_S \mathbf{i}_R + \mathbf{C}_S\, abs\,(\mathbf{D}_S^T \mathbf{i}_R - \mathbf{e}_S)] \tag{3.14}$$

$$\mathbf{v}_R = \mathbf{a}_R + \mathbf{B}_R \mathbf{i}_R + \mathbf{C}_R\, abs\,(\mathbf{D}_R^T \mathbf{i}_R - \mathbf{e}_R) \tag{3.15}$$

Let $\mathbf{A}$ be the reduced incidence matrix of N relative to some datum node, and partitioned $\mathbf{A}$ as $\mathbf{A} = \left[\mathbf{A}_G \mid \mathbf{A}_E \mid \mathbf{A}_R\right]$. Let $\mathbf{J}$ denote the source current vector and $\mathbf{v}_n$ denote the node-to-datum voltage vector, then KCL and KVL give :[3]

$$\mathbf{A}_G \mathbf{i}_G + \mathbf{A}_E \mathbf{i}_E + \mathbf{A}_R \mathbf{i}_R = \mathbf{0} \tag{3.16}$$

$$\mathbf{A}_G^T \mathbf{v}_n = \mathbf{v}_G \;,\; \mathbf{A}_E^T \mathbf{v}_n = \mathbf{v}_E \;,\; \mathbf{A}_R^T \mathbf{v}_n = \mathbf{v}_R \tag{3.17}$$

[3]Note: Current sources have been absorbed into $\mathbf{i}_G$.

Substituting (3.13), (3.17) into (3.16) and (3.17) into (3.14), (3.15) respectively, we get

$$\mathbf{A}_G\,[\,\mathbf{a}_G + \mathbf{B}_G\,\mathbf{A}_G^T\,\mathbf{v}_n + \mathbf{C}_G\,abs\,(\,\mathbf{D}_G^T\,\mathbf{A}_G^T\,\mathbf{v}_n - \mathbf{e}_G\,)\,] + \mathbf{A}_E\,\mathbf{i}_E + \mathbf{A}_R\,\mathbf{i}_R = 0 \qquad (3.18)$$

$$\mathbf{A}_E^T\,\mathbf{v}_n = [\,\mathbf{a}_M + \mathbf{B}_M\,\mathbf{A}_G^T\,\mathbf{v}_n + \mathbf{C}_M\,abs\,(\,\mathbf{D}_M^T\,\mathbf{A}_G^T\,\mathbf{v}_n - \mathbf{e}_M\,)\,]$$

$$+\,[\,\mathbf{a}_S + \mathbf{B}_S\,\mathbf{i}_R + \mathbf{C}_S\,abs\,(\,\mathbf{D}_S^T\,\mathbf{i}_R - \mathbf{e}_S\,)\,] \qquad (3.19)$$

$$\mathbf{A}_R^T\,\mathbf{v}_n = \mathbf{a}_R + \mathbf{B}_R\,\mathbf{i}_R + \mathbf{C}_R\,abs\,(\,\mathbf{D}_R^T\,\mathbf{i}_R - \mathbf{e}_R\,) \qquad (3.20)$$

Let $\mathbf{x} = [\,\mathbf{v}_n^T \mid \mathbf{i}_E^T \mid \mathbf{i}_R^T\,]^T$ be the vector of independent variables and rewrite (3.18) - (3.20) into vector form, we get

$$\begin{bmatrix} \mathbf{A}_G\,\mathbf{a}_G \\ \mathbf{a}_M + \mathbf{a}_S \\ \mathbf{a}_R \end{bmatrix} + \begin{bmatrix} \mathbf{A}_G\,\mathbf{B}_G\,\mathbf{A}_G^T & \mathbf{A}_E & \mathbf{A}_R \\ -\mathbf{A}_E^T + \mathbf{B}_M\,\mathbf{A}_G^T & 0 & \mathbf{B}_S \\ -\mathbf{A}_R^T & 0 & \mathbf{B}_R \end{bmatrix} \mathbf{x}$$

$$+ \begin{bmatrix} \mathbf{A}_G\,\mathbf{C}_G & 0 & 0 & 0 \\ 0 & \mathbf{C}_M & \mathbf{C}_S & 0 \\ 0 & 0 & 0 & \mathbf{C}_R \end{bmatrix} abs \left(\begin{bmatrix} \mathbf{A}_G\,\mathbf{D}_G & \mathbf{A}_G\,\mathbf{D}_M & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_S & \mathbf{D}_R \end{bmatrix}^T \mathbf{x} - \begin{bmatrix} \mathbf{e}_G \\ \mathbf{e}_M \\ \mathbf{e}_S \\ \mathbf{e}_R \end{bmatrix} \right) = 0 \qquad (3.21)$$

Defining

$$\mathbf{a} = \begin{bmatrix} \mathbf{A}_G\,\mathbf{a}_G \\ \mathbf{a}_M + \mathbf{a}_S \\ \mathbf{a}_R \end{bmatrix},\quad \mathbf{B} = \begin{bmatrix} \mathbf{A}_G\,\mathbf{B}_G\,\mathbf{A}_G^T & \mathbf{A}_E & \mathbf{A}_R \\ -\mathbf{A}_E^T + \mathbf{B}_M\,\mathbf{A}_G^T & 0 & \mathbf{B}_S \\ -\mathbf{A}_R^T & 0 & \mathbf{B}_R \end{bmatrix},\quad \mathbf{C} = \begin{bmatrix} \mathbf{A}_G\,\mathbf{C}_G & 0 & 0 & 0 \\ 0 & \mathbf{C}_M & \mathbf{C}_S & 0 \\ 0 & 0 & 0 & \mathbf{C}_R \end{bmatrix},$$

$$\mathbf{D} = \begin{bmatrix} \mathbf{A}_G\,\mathbf{D}_G & \mathbf{A}_G\,\mathbf{D}_M & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_S & \mathbf{D}_R \end{bmatrix},\quad \mathbf{e} = \left[\mathbf{e}_G^T\,\mathbf{e}_M^T\,\mathbf{e}_S^T\,\mathbf{e}_R^T\right]^T$$

we can recast (3.21) in the following canonical form :

$$\mathbf{a} + \mathbf{B}\,\mathbf{x} + \mathbf{C}\,abs\,(\,\mathbf{D}^T\,\mathbf{x} - \mathbf{e}\,) = 0 \qquad \square$$

Theorem 3.4 (*Tableau analysis*)

Consider a connected resistive circuit N containing only linear 2-terminal resistors, dc independent sources, current-controlled and voltage-controlled piecewise-linear 2-terminal resistors, linear and piecewise-linear controlled sources (all 4 types) and any linear multi-terminal resistive elements. If each piecewise-linear function is represented in the canonical form (2.1), then the ***tableau formulation*** always assumes the form of (2.15).

Proof :

Let $\mathbf{A}$ be the reduced incidence matrix of N relative to some datum node, and $\mathbf{v}_n$ be the node-to-datum voltage vector, then KCL, KVL and element constitutive relations give :

$$\mathbf{A}\,\mathbf{i} = \mathbf{A}\,\mathbf{J} \qquad (3.22)$$

$$\mathbf{v} = \mathbf{A}^T\,\mathbf{v}_n + \mathbf{E} \qquad (3.23)$$

$$\mathbf{f}_I(\mathbf{i}) + \mathbf{f}_V(\mathbf{v}) = \mathbf{S} \tag{3.24}$$

using the same procedure as in the proof of *Theorem 3.1*, we can express $\mathbf{f}_I(.)$ and $\mathbf{f}_V(.)$ in the canonical form (2.15)

$$\mathbf{f}_I(\mathbf{i}) = \mathbf{a}_I + \mathbf{B}_I\,\mathbf{i} + \mathbf{C}_I\, abs\,(\mathbf{D}_I^T\mathbf{i} - \mathbf{e}_I) \tag{3.25}$$

$$\mathbf{f}_V(\mathbf{v}) = \mathbf{a}_V + \mathbf{B}_V\,\mathbf{v} + \mathbf{C}_V\, abs\,(\mathbf{D}_V^T\mathbf{v} - \mathbf{e}_V) \tag{3.26}$$

Substituting (3.25) - (3.26) into (3.24) and writing (3.22) - (3.24) into vector form, we get :

$$\begin{bmatrix} -\mathbf{A}\,\mathbf{J} \\ -\mathbf{E} \\ \mathbf{a}_I + \mathbf{a}_V - \mathbf{S} \end{bmatrix} + \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{A}^T \\ \mathbf{B}_I & \mathbf{B}_V & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \mathbf{v} \\ \mathbf{v}_n \end{bmatrix}$$

$$+ \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{C}_I & \mathbf{C}_V & \mathbf{0} \end{bmatrix} abs \left(\begin{bmatrix} \mathbf{D}_I & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_V & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}^T \begin{bmatrix} \mathbf{i} \\ \mathbf{v} \\ \mathbf{v}_n \end{bmatrix} - \begin{bmatrix} \mathbf{e}_I \\ \mathbf{e}_V \\ \mathbf{0} \end{bmatrix} \right) = \mathbf{0} \tag{3.27}$$

Clearly (3.27) is in the canonical form (2.15). $\square$

3.3 Canonical equation for hybrid analysis

Theorem 3.4 (*Hybrid analysis*)

Consider a resistive circuit N containing linear 2-terminal resistors, dc independent sources, voltage and current controlled piecewise-linear resistors, linear controlled sources (all 4 types) and linear multi-terminal resistive elements. Let $\hat{N}$ denote the m-port obtained by extracting all voltage-controlled piecewise-linear resistors in N across "voltage" ports, and all current-controlled piecewise-linear resistors across "current" ports (Fig. 7). If the characteristic of each piecewise-linear resistor is represented in the canonical form (2.1), and if $\hat{N}$ admits *hybrid representation* (3.28), then

(a) the hybrid equation of N always assumes the canonical form (2.11):

(b) the linear partition in the domain always forms a *lattice structure* in the sense that the partition hyperplanes are parallel to the coordinate axes defined by the independent variables.[4]

Proof :

Consider the linear m-port $\hat{N}$ shown in Fig. 7. Let $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_l$ be the voltages of the voltage ports and $\hat{i}_{l+1}, \hat{i}_{l+2}, \ldots, \hat{i}_m$ be the currents of the current ports. The assumption implies that $\hat{N}$ has a hybrid representation :

$$\begin{bmatrix} \hat{\mathbf{i}}_a \\ \hat{\mathbf{v}}_b \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{H}}_{aa} & \hat{\mathbf{H}}_{ab} \\ \hat{\mathbf{H}}_{ba} & \hat{\mathbf{H}}_{bb} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{v}}_a \\ \hat{\mathbf{i}}_b \end{bmatrix} + \begin{bmatrix} \hat{\mathbf{s}}_a \\ \hat{\mathbf{s}}_b \end{bmatrix} \tag{3.28}$$

where $\hat{\mathbf{v}}_a = [\,\hat{v}_1\,\hat{v}_2 \ldots \hat{v}_l\,]^T$, $\hat{\mathbf{v}}_b = [\,\hat{v}_{l+1}\,\hat{v}_{l+2} \ldots \hat{v}_m\,]^T$, $\hat{\mathbf{i}}_a = [\,\hat{i}_1\,\hat{i}_2 \ldots \hat{i}_l\,]^T$,

[4]Any piecewise-linear equation having this property will henceforth be called an "equation with a lattice structure".

$\hat{\mathbf{i}}_b = [\ \hat{i}_{l+1}\ \hat{i}_{l+2}\ \ldots .\ \hat{i}_m\ ]^T$, and $[\ \hat{\mathbf{s}}_a\ \hat{\mathbf{s}}_b\ ]^T$ is the source vector.

By assumption, each voltage-controlled piecewise-linear resistor is represented by

$$i_k = a_k + b_k\, v_k + \sum_{i=1}^{p_k} c_{ki}\ |\ v_k - V_{ki}\ |\ , \quad k = 1, 2, \ldots . , l \tag{3.29}$$

Similarly, each current-controlled piecewise-linear resistor is represented by

$$v_k = a_k + b_k\, i_k + \sum_{i=1}^{p_k} c_{ki}\ |\ i_k - I_{ki}\ |\ , \quad k = l+1, l+2, \ldots . , m \tag{3.30}$$

Defining $\mathbf{v}_a = [\ v_1\ v_2\ \ldots .\ v_l\ ]^T$, $\mathbf{v}_b = [\ v_{l+1}\ v_{l+2}\ \ldots .\ v_m\ ]^T$, $\mathbf{i}_a = [\ i_1\ i_2\ \ldots .\ i_l\ ]^T$,
$\mathbf{i}_b = [\ i_{l+1}\ i_{l+2}\ \ldots .\ i_m\ ]^T$, and recasting (3.29) - (3.30) into vector form, we get

$$\begin{bmatrix} \mathbf{i}_a \\ \mathbf{v}_b \end{bmatrix} = \tilde{\mathbf{a}} + \tilde{\mathbf{B}} \begin{bmatrix} \mathbf{v}_a \\ \mathbf{i}_b \end{bmatrix} + \sum_{j=1}^{m} \sum_{i=1}^{p_j} c_{ji}\, \mathbf{u}_j\ |\ < \mathbf{u}_j\ , \begin{bmatrix} \mathbf{v}_a \\ \mathbf{i}_b \end{bmatrix} > - \beta_{ji}\ | \tag{3.31}$$

where

$$\tilde{\mathbf{a}} = [\ a_1\ a_2\ \ldots .\ a_l\ a_{l+1}\ \ldots .\ a_m\ ]^T\ ,\ \tilde{\mathbf{B}} = diag\ [\ b_1\ b_2\ \ldots .\ b_l\ b_{l+1}\ \ldots .\ b_m\ ]$$

$$\beta_{ji} = \begin{cases} V_{ji} & j = 1, 2, \ldots . , l \\ I_{ji} & j = l+1, l+2, \ldots . , m \end{cases}$$

and $\mathbf{u}_j$ is the unit vector in $\mathbb{R}^m$, $j = 1, 2, \ldots . , m$.

From Fig. 7 we have $\hat{v}_k = v_k\ ,\ \hat{i}_k = i_k\ ,\ k = 1, 2, \ldots . , m$. Hence, $\hat{\mathbf{v}}_a = \mathbf{v}_a$, $\hat{\mathbf{i}}_a = \mathbf{i}_a$, $\hat{\mathbf{v}}_b = \mathbf{v}_b$, $\hat{\mathbf{i}}_b = \mathbf{i}_b$. Equating the right hand side of (3.28) and (3.31) we get

$$\begin{bmatrix} \hat{\mathbf{H}}_{aa} & \hat{\mathbf{H}}_{ab} \\ \hat{\mathbf{H}}_{ba} & \hat{\mathbf{H}}_{bb} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \hat{\mathbf{s}}_a \\ \hat{\mathbf{s}}_b \end{bmatrix} = \tilde{\mathbf{a}} + \tilde{\mathbf{B}} + \sum_{j=1}^{m} \sum_{i=1}^{p_j} c_{ji}\, \mathbf{u}_j\ |\ < \mathbf{u}_j\ , \mathbf{x} > - \beta_{ji}\ |$$

After rearranging terms, we get

$$\mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{j=1}^{m} \sum_{i=1}^{p_j} \mathbf{c}_{ji}\ |\ < \mathbf{u}_j\ , \mathbf{x} > - \beta_{ji}\ | = 0 \tag{3.32}$$

where $\mathbf{x} = \begin{bmatrix} \mathbf{v}_a \\ \mathbf{i}_b \end{bmatrix}$, $\mathbf{a} = \tilde{\mathbf{a}} - \begin{bmatrix} \hat{\mathbf{s}}_a \\ \hat{\mathbf{s}}_b \end{bmatrix}$, $\mathbf{B} = \tilde{\mathbf{B}} - \begin{bmatrix} \hat{\mathbf{H}}_{aa} & \hat{\mathbf{H}}_{ab} \\ \hat{\mathbf{H}}_{ba} & \hat{\mathbf{H}}_{bb} \end{bmatrix}$, $\mathbf{c}_{ji} = c_{ji}\, \mathbf{u}_j$

Equation (3.32) is precisely in the form of (2.11) provided we relabel the double indices in the last term. This completes the proof of (a).

The partition hyperplanes associated with (3.32) are defined by $< \mathbf{u}_j\ , \mathbf{x} > = \beta_{ji}$, $i = 1, 2, \ldots . , p_j$. Since $\mathbf{u}_j$ defines the normal direction of the j-th hyperplane in the linear partition and since $\mathbf{u}_j$ is a unit vector in $\mathbb{R}^m$, (b) follows immediately. □

Remark : Theorem 3.4 can be easily generalized such that the piecewise-linear resistors extracted across the voltage and current ports of N can be mutually coupled; i.e. $\mathbf{i}_a = \mathbf{f}\ (\ \mathbf{v}_a\ , \mathbf{i}_b\ )$ and $\mathbf{v}_b = \mathbf{g}\ (\ \mathbf{v}_a\ , \mathbf{i}_b\ )$, where $\mathbf{f}\ (.)$ and $\mathbf{g}\ (.)$ are represented in the canonical form (2.11).

Example 3.

Consider the resistor circuit shown in Fig. 8(a) where R1 is voltage controlled with its v-i characteristic shown in Fig. 8(b) and R2 is current-controlled with its v-i characteristic shown in Fig. 8(c). The v-i characteristics of R1 and R2 are represented as follow :

$$\text{R1}: \ i_1 = \frac{1}{4} + \frac{3}{4} v_1 - | v_1 | + \frac{3}{4} | v_1 - 1 | \tag{3.33}$$

$$\text{R2}: \ v_2 = -\frac{1}{4} + \frac{3}{4} i_2 - \frac{1}{4} | i_2 + 1 | \tag{3.34}$$

Extracting R1 across the voltage port and R2 across the current port (see Fig. 8(d)), we obtain the following hybrid representation for the remaining linear 2-port $\hat{N}$:

$$\begin{bmatrix} i_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ -2 & \frac{2}{3} \end{bmatrix} \begin{bmatrix} v_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.35}$$

Substituting (3.33) and (3.34) into (3.35) and rearranging terms, we obtain the following hybrid equation in the canonical form (2.15) :

$$\begin{bmatrix} \frac{1}{4} \\ -\frac{1}{4} \end{bmatrix} + \begin{bmatrix} -\frac{5}{4} & 2 \\ 2 & \frac{1}{12} \end{bmatrix} \begin{bmatrix} v_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} -1 & \frac{3}{4} & 0 \\ 0 & 0 & -\frac{1}{4} \end{bmatrix} abs \left(\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ i_2 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The lattice structure of the linear partition is shown in Fig. 8(e). □

4. CANONICAL KATZENELSON ALGORITHM

In this section, we adapt the canonical form representation to the *Katzenelson algorithm.* We state the conditions which guarantee the convergence of this algorithm in terms of the coefficients of the n-dimensional canonical representation. An alternate way of handling the familiar "corner" problem is also discussed with an illustrative example. Finally, we compare the bookkeeping complexity and computational efficiency involved in equations with and without lattice structures.

Definition 4.1

A continuous mapping $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ is said to be *norm-coercive* if $\| \mathbf{f}(\mathbf{x}) \| \to \infty$ as $\| \mathbf{x} \| \to \infty$.

The following theorem gives sufficient conditions for the existence of solution in terms of the coefficients of $\mathbf{f}(.)$, where $\mathbf{f}(.)$ is expressed in (2.15). The proof follows directly from a corresponding theorem in [6].

Theorem 4.1

Let $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ be a norm-coercive continuous piecewise-linear function expressed in the form (2.15). If there exists a point $\mathbf{x}_0 \in \mathbb{R}^n$ which satisfies[5]

(1) $\mathbf{D}^T \mathbf{x}_0 - \mathbf{e} \neq 0$

(2) $\mathbf{B}(\mathbf{x} - \mathbf{x}_0) \neq -\mathbf{C}[\,abs(\mathbf{D}^T\mathbf{x} - \mathbf{e}) - abs(\mathbf{D}^T\mathbf{x}_0 - \mathbf{e})\,] \quad \mathbf{x} \in \mathbb{R}^n, \ \mathbf{x} \neq \mathbf{x}_0$

(3) $\det\{\,\mathbf{B} + \mathbf{C}[\,\underset{\sim}{sgn}(\mathbf{D}^T\mathbf{x}_0 - \mathbf{e})\,]\,\mathbf{D}^T\} > 0$

then for any given $\mathbf{y}_0 \in \mathbb{R}^n$, there exists an $\mathbf{x} \in \mathbb{R}^n$, such that $\mathbf{f}(\mathbf{x}) = \mathbf{y}_0$. Furthermore, the *Katzenelson algorithm* starting from $\mathbf{x}_0$ *converges to a solution in a finite number of steps.*

For complete generality, we will present the *canonical Katzenelson algorithm* without assuming the equilibrium equation has a lattice structure. However, remarks will be given at each step where a lattice structure can bring significant savings in bookkeeping and/or computation.

Canonical Katzenelson Algorithm

Consider the equation

$$\mathbf{f}(\mathbf{x}) \overset{def}{=} \mathbf{a} + \mathbf{B}\mathbf{x} + \mathbf{C}\,abs\,(\mathbf{D}^T\mathbf{x} - \mathbf{e}) = \mathbf{y}_0 \tag{4.1}$$

Step 1 : Choose an initial point $\mathbf{x}_0$ satisfying conditions (1), (2) and (3) in *Theorem 4.1.* Let $k = 1$, $h = 0$, $\mathbf{x}^{(k)} = \mathbf{x}_0$. Let R_k denote the region containing $\mathbf{x}^{(k)}$. R_k can be identified by its sign-sequence vector (2.6). Go to *step 2.*

Step 2: Compute $\mathbf{n}^{(k)}$ by

$$\mathbf{n}^{(k)} = [\,\mathbf{J}_{R_k}\,]^{-1}\,[\,\mathbf{y}_0 - \mathbf{f}(\mathbf{x}^{(k)})\,] \tag{4.2}$$

where $\mathbf{J}_{R_k}$ is the Jacobian matrix of $\mathbf{f}$ in region R_k. Go to *step 3.*

Step 3 : If $\mathbf{n}^{(k)}$ is a zero vector, then $\mathbf{x}^{(k)}$ is the solution, and the algorithm terminates here. Otherwise let $A^{(k)}$ be a subset of $\{1, 2, \ldots, p\}$ such that for $i \in A^{(k)}$, $<\alpha_i\,,\mathbf{x}> = \beta_i$ is a boundary hyperplane of region R_k. Then for each $i \in A^{(k)}$ such that $<\alpha_i\,,\mathbf{n}^{(k)}> \neq 0$, compute

$$t_i^{(k)} = \frac{\beta_i - <\alpha_i\,,\mathbf{x}^{(k)}>}{<\alpha_i\,,\mathbf{n}^{(k)}>} \tag{4.3}$$

and go to *step 4.*

Remark : Since it is generally impossible to identify the boundaries associated with each region of (4.1), the number of elements in set $A^{(k)}$ is always equal to $p-1$ (i.e. all hyperplanes except the one which contains $\mathbf{x}^{(k)}$). However, if the equilibrium equation has a lattice structure, then each region has at most $2n$ boundaries (therefore the size of $A^{(k)}$ is always $2n-1$ after the first iteration) and can be easily identified by simply comparing the coordinates of $\mathbf{x}^{(k)}$ with the β_i's. Substantial computing time can be saved at this point since not only the total number of $t_i^{(k)}$ to be calculated is reduced, but also no multiplication is required for the dot products in (4.3) because α_i is now a unit vector. Note that in general p is much greater than $2n$, and $p = 2n$ occurs only when each of the piecewise-linear characteristic has exactly

[5]For $\mathbf{x} \in \mathbb{R}^n$, $\underset{\sim}{sgn}(\mathbf{x}) \overset{def}{=} diag\,(sgn\,(x_1),\, sgn\,(x_2),\, \ldots,\, sgn\,(x_n))$, where $sgn\,(x_i) = 1$ when $x_i > 0$ and

one breakpoint.

Step 4: (a) If det $\mathbf{J}_{R_k} \neq 0$, let $\hat{A}^{(k)}$ be the subset of $A^{(k)}$ such that for all $i \in \hat{A}^{(k)}$

$$sgn\ (t_i^{(k)})\ sgn\ (\det \mathbf{J}_{R_k}) = 1 \tag{4.4}$$

If $\hat{A}^{(k)}$ is an empty set, then $\mathbf{x}^{(k)} + \mathbf{n}^{(k)}$ is the solution and the algorithm terminates. Otherwise, find $\hat{t}_i^{(k)} \overset{def}{=} \min_{i \in \hat{A}^{(k)}} \mid t_i^{(k)} \mid$ and go to *step 5*.

(b) If det $\mathbf{J}_{R_k} = 0$, then find

$$\hat{t}_i^{(k)} = \min_{i \in A^{(k)}} \mid t_i^{(k)} \mid \tag{4.5}$$

and go to *step 5*.

Remark : Again, observe that if equation (4.1) has a lattice structure, then the size of $\hat{A}^{(k)}$ will be smaller and searching for $\hat{t}_i^{(k)}$ will also be easier.

Step 5 : If $\hat{t}_i^{(k)} \geq 1$, then $\mathbf{x}^{(k)} + \mathbf{n}^{(k)}$ is the solution and the algorithm terminates here. If $\hat{t}_i^{(k)} < 1$ and it is not unique[6], go to *step 8*. If $\hat{t}_i^{(k)} < 1$ and it is unique, let $h = i$, $t_{\min}^{(k)} = t_i^{(k)}$ and go to *step 6*.

Step 6: Compute

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + t_{\min}^{(k)}\ \mathbf{n}^{(k)} \tag{4.6}$$

$$\mathbf{J}_{R_{k+1}} = \mathbf{J}_{R_k} + 2\ \mathbf{c}_h\ \alpha_h^T \tag{4.7}$$

where $\mathbf{c}_h$ is the h-th column in matrix **C** and α_h is the h-th column in matrix **D**. Go to *step 7*.

Remark : If equation (4.1) has a lattice structure, then α_h in (4.7) will be a unit vector, and hence the dyad product of $\mathbf{c}_h$ and α_h^T requires no multiplication at all.

Step 7: Increment k by 1. If det $\mathbf{J}_{R_k} \neq 0$, then go to *step 2*. Otherwise, find a vector $\hat{\mathbf{n}} \neq 0$ in the null space[7] of $\mathbf{J}_{R_k}$. Let $\mathbf{n}^{(k)} = \hat{\mathbf{n}}$ and go to *step 3*.

Step 8: (the corner problem)[8]. Since the $t_i^{(k)}$'s are not unique, let $I^{(k)}$ be a subset of $\hat{A}^{(k)}$ containing the indices i where the $t_i^{(k)}$'s are equal. Choose any i from $I^{(k)}$, let $h = i$, $t_{\min}^{(k)} = \hat{t}_i^{(k)}$ and compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + t_{\min}^{(k)}\ \mathbf{n}^{(k)}$. Go to *step 9*.

Step 9: Check $\mathbf{x}^{(k+1)}$ against the existing corner points stored in *step 10*. If $\mathbf{x}^{(k+1)}$ is a new corner point then go to *step 10*. Otherwise go to *step 11*.

Step 10: Store the corner point $\mathbf{x}^{(k+1)}$ Go to *step 12*.

Step 11 : Let $N_I^{(k)}$ be the total number of elements in set $I^{(k)}$. Construct a region list $L_I^{(k)}$ which consists of all neighborhood regions of the corner point except region R_k.

$sgn\,(x_i) = -1$ when $x_i < 0$.

[6]We say $\hat{t}_i^{(k)}$ is *not unique* if there exist at least 2 indices "m" and "n" such that $\hat{t}_i^{(k)} = \mid t_m^{(k)} \mid = \mid t_n^{(k)} \mid$.

[7]The vector $\hat{\mathbf{n}}$ in the null space of $\mathbf{J}_{R_{k+1}}$ can be found using standard linear system techniques.

[8]Note that $\mathbf{x}^{(k+1)}$ is now a corner point.

Remark : Since each region is uniquely identified by its sign-sequence vector, *step 11* consists merely of generating those sign-sequence vectors which represents the neighborhood regions. Let $\mathbf{w}^{(k)}$ denote the sign-sequence vector of region R_k. We generate the sign-sequence vectors from $\mathbf{w}^{(k)}$ by varying the sign at the j-th position of $\mathbf{w}^{(k)}$ for $j \in I^{(k)}$, while keeping the remaining entries identical. Clearly this procedure will generate $2^{N_I^{(k)}} - 1$ sign-sequence vectors.

Step 12 : Select a new region R_{k+1} from the list $L_I^{(k)}$ associated with the corner point $\mathbf{x}^{(k+1)}$ in accordance with the following precedence : (1) the sign sequence of R_{k+1} matches $\mathbf{w}(\hat{\mathbf{x}})$, (2) det $\mathbf{J}_{R_{k+1}} > 0$, (3) det $\mathbf{J}_{R_{k+1}} = 0$, (4) det $\mathbf{J}_{R_{k+1}} < 0$. Remove R_{k+1} from the list $L_I^{(k)}$ and go to *step 13*.

Remark : *Step 12* and *step 13* form a loop until we identify the correct region to proceed. Here, $\mathbf{w}(\hat{\mathbf{x}})$ is the sign-sequence vector computed in *step 13*.

Step 13 : Repeat *step 2* and *step 3* (i.e. repeat (4.2) and (4.3) with k replaced by $k+1$) to compute $\mathbf{n}^{(k+1)}$ and $t_i^{(k+1)}$ for $i \notin I^{(k+1)}$ (go through *step 7* first if det $\mathbf{J}_{R_{k+1}} \neq 0$). Repeat *step 4* to find $\hat{t}_i^{(k+1)}$, then compute $\hat{\mathbf{x}} = \mathbf{x}^{(k+1)} + \frac{1}{2}\hat{t}_i^{(k+1)}\mathbf{n}^{(k+1)}$ and the sign-sequence vector $\mathbf{w}(\hat{\mathbf{x}})$. To determine if region R_{k+1} is the correct region to proceed, we compare $\mathbf{w}(\hat{\mathbf{x}})$ with $\mathbf{w}^{(k+1)}$ (the sign-sequence vector of region R_{k+1}). If they are equal, (then R_{k+1} is the correct region to continue), set $\mathbf{x}^{(k+2)} = \hat{\mathbf{x}}$, $\mathbf{J}_{R_{k+2}} = \mathbf{J}_{\mathbf{w}(\hat{\mathbf{x}})}$ and increment k by 2. Go to *step 2*. If $\mathbf{w}(\hat{\mathbf{x}})$ and $\mathbf{w}^{(k+1)}$ are not equal, then go to *step 12*.

Remark : Since most of *step 13* consists of repeating *step 2*, *step 3* and *step 4*, substantial savings in computing time will result if equation (4.1) has a lattice structure (recall the *Remarks* following *step 3* and *step 4*.)

Justification :

(1) To show the Jacobian matrix can be computed from the one in the adjacent region by (4.7), let R_1, R_2 be two adjacent regions and $\mathbf{J}_1$, $\mathbf{J}_2$ be the Jacobian matrices in R_1 and R_2 respectively. Then :

$$\mathbf{J}_1 = \mathbf{B} + \mathbf{C}\,[\,\underset{\sim}{sgn}\,(\,\mathbf{D}^T\mathbf{x}^{(1)} - \mathbf{e}\,)\,]\,\mathbf{D}^T \tag{4.8}$$

$$\mathbf{J}_2 = \mathbf{B} + \mathbf{C}\,[\,\underset{\sim}{sgn}\,(\,\mathbf{D}^T\mathbf{x}^{(2)} - \mathbf{e}\,)\,]\,\mathbf{D}^T \tag{4.9}$$

where $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ are arbitrary interior points in R_1 and R_2 respectively. For *continuous* piecewise-linear functions,

$$\mathbf{J}_1 = \mathbf{J}_2 + \underset{\sim}{\gamma}\,\alpha_i^{\,T} \tag{4.10}$$

where $\underset{\sim}{\gamma}$ is a constant vector to be determined [1]. Substituting (4.8) and (4.9) into (4.10), we get :

$$\mathbf{J}_1 - \mathbf{J}_2 = \mathbf{C}\,[\,\underset{\sim}{sgn}\,(\,\mathbf{D}^T\mathbf{x}^{(1)} - \mathbf{e}\,) - \underset{\sim}{sgn}\,(\,\mathbf{D}^T\mathbf{x}^{(2)} - \mathbf{e}\,)\,]\,\mathbf{D}^T \tag{4.11}$$

since the sign-sequence vectors differ only at the i-th position for R_1 and R_2, we have

$$\underset{\sim}{sgn}\,(\mathbf{D}^T\mathbf{x}^{(1)} - \mathbf{e}) - \underset{\sim}{sgn}\,(\mathbf{D}^T\mathbf{x}^{(2)} - \mathbf{e}) = diag\,(0, 0, \ldots, 0, 2, 0, \ldots, 0)$$

where the number "2" is at the i-th diagonal position. Therefore (4.11) reduces to

$$\mathbf{J}_1 - \mathbf{J}_2 = 2\,\mathbf{c}_i\,\boldsymbol{\alpha}_i^T \tag{4.12}$$

Comparing (4.10) with (4.12), we identify $\underset{\sim}{\gamma} = 2\,\mathbf{c}_i$ and hence (4.7) is proved.

(2) The method we used to solve the corner problem (*step 8 - 13*) is fully justified in [6] (Theorem 3 through Theorem 5), and is not repeated here. However, we will illustrate the method in *Example 4*.

Example 4. (Step-by-step illustration of the canonical Katzenelson algorithm)

Consider the circuit shown in Fig. 9(a) where both resistors R1 and R2 are voltage controlled. Their v-i characteristics in Figs. 9(b) and 9(c) are represented in canonical form :

R1 : $i_1 = \frac{1}{4} + \frac{3}{4}v_1 - |\,v_1\,| + \frac{3}{4}\,|\,v_1 - 1\,|$

R2 : $i_2 = -\frac{1}{4} + \frac{3}{4}v_2 - \frac{1}{4}\,|\,v_2 + 1\,|$

The canonical form equation for this circuit is found to be

$$\begin{bmatrix} \frac{5}{8} & -\frac{3}{4} \\ \frac{3}{4} & 2 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & -\frac{3}{8} & \frac{1}{4} \\ -1 & \frac{3}{4} & -\frac{1}{3} \end{bmatrix} abs\left(\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}\right)$$

$$= \begin{bmatrix} -\frac{1}{8} \\ E_0 + \frac{1}{12} \end{bmatrix} \tag{4.13}$$

let $E_0 = 1$ so that $\mathbf{y}_0 = [\,-\frac{1}{8}\ \frac{13}{12}\,]^T$. The linear partition in this case consists of 3 straight lines $v_1 = 0$, $v_2 = 1$, and $v_2 = -1$, as shown in Fig. 9(d). Define $\mathbf{x} = [\,v_1\,v_2\,]^T$.

Step 1 : Choose an ***arbitrary*** point $\mathbf{x}_0 = [\,-1\ -\frac{3}{2}\,]^T$ as shown in Fig. 9(d).

Step 2 & 3: (k=1).

$$\mathbf{n}^{(1)} = \begin{bmatrix} \frac{1}{2} & -1 \\ 1 & \frac{7}{3} \end{bmatrix}^{-1}\left(\begin{bmatrix} -\frac{1}{8} \\ \frac{13}{12} \end{bmatrix} - \begin{bmatrix} \frac{3}{8} \\ -\frac{41}{12} \end{bmatrix}\right) = \begin{bmatrix} \frac{20}{13} \\ \frac{33}{26} \end{bmatrix}$$

Since the region R_1 containing $\mathbf{x}_0$ is bounded only by the first ($v_1 = 0$) and third ($v_2 = -1$) hyperplane, we set $A^{(1)} = \{\,1, 3\,\}$ and compute $t_1^{(1)}$ and $t_3^{(1)}$:

$$t_1^{(1)} = \frac{0 - <[\,1\,0\,]^T, [\,-1\ -\frac{3}{2}\,]^T>}{<[\,1\,0\,]^T, [\,\frac{20}{13}\ \frac{33}{26}\,]^T>} = \frac{13}{20},\quad t_3^{(1)} = \frac{-1 - <[\,0\,1\,]^T, [\,-1\ -\frac{3}{2}\,]^T>}{<[\,0\,1\,]^T, [\,\frac{20}{13}\ \frac{33}{26}\,]^T>} = \frac{13}{33}$$

Step 4 & 5: (k=1). Since $\det \mathbf{J}_{R_1} = \frac{13}{6} > 0$, we have $\hat{A}^{(1)} = \{\,1, 3\,\}$, $t_{\min}^{(1)} = \frac{13}{33}$, and $h = 3$.

Step 6: (k=1). (4.6) implies

$$\mathbf{x}^{(2)} = \begin{bmatrix} -1 \\ -\frac{3}{2} \end{bmatrix} + \frac{13}{33}\begin{bmatrix} \frac{20}{13} \\ \frac{33}{26} \end{bmatrix} = \begin{bmatrix} -\frac{13}{33} \\ -1 \end{bmatrix}$$

(4.7) implies

$$\mathbf{J}_{R_2} = \begin{bmatrix} \frac{1}{2} & -1 \\ 1 & \frac{7}{3} \end{bmatrix} + 2\begin{bmatrix} \frac{1}{4} \\ -\frac{1}{3} \end{bmatrix}[\,0\;1\,] = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} \\ 1 & \frac{5}{3} \end{bmatrix}$$

Step 7: (k=1). $\det \mathbf{J}_{R_2} = \frac{4}{3} > 0$. Set k = 2 and go to *step 2*.

Step 2 & 3: (k=2). (4.2) implies $\mathbf{n}^{(2)} = [\,\frac{85}{132}\;\frac{5}{4}\,]^T$. Set $A^{(2)} = \{1, 2\}$ and compute $t_1^{(2)}$ and $t_2^{(2)}$:

$$t_1^{(2)} = \frac{0 - <[\,1\;0\,]^T, [\,-\frac{13}{33} - 1\,]^T>}{<[\,1\;0\,]^T, [\,\frac{85}{132}\;\frac{5}{4}\,]^T>} = \frac{52}{85}, \quad t_2^{(2)} = \frac{1 - <[\,1\;0\,]^T, [\,-\frac{13}{33} - 1\,]^T>}{<[\,1\;0\,]^T, [\,\frac{85}{132}\;\frac{5}{4}\,]^T>} = \frac{184}{85}$$

Step 4 & 5: (k=2). Since $\det \mathbf{J}_{R_2} = \frac{4}{3} > 0$, we have $\hat{A}^{(2)} = \{1, 2\}$, $t_{\min}^{(2)} = 1$, and h = 1.

Step 6: (k=2). (4.6) implies $\mathbf{x}^{(3)} = [\,0\; -\frac{4}{17}\,]^T$, and (4.7) implies $\mathbf{J}_{R_3} = \begin{bmatrix} \frac{3}{2} & -\frac{1}{2} \\ -1 & \frac{5}{3} \end{bmatrix}$

Step 7: (k=2). $\det \mathbf{J}_{R_3} = 2 > 0$. Set k = 3 and go to *step 2*.

Step 2 & 3: (k=3). (4.2) implies $\mathbf{n}^{(3)} = [\,\frac{1}{6}\;\frac{25}{34}\,]^T$. Set $A^{(3)} = \{2, 3\}$ and compute $t_2^{(3)}$ and $t_3^{(3)}$:

$$t_2^{(3)} = \frac{1 - <[\,1\;0\,]^T, [\,0\; -\frac{4}{17}\,]^T>}{<[\,1\;0\,]^T, [\,\frac{1}{6}\;\frac{25}{34}\,]^T>} = 6\,, \quad t_3^{(2)} = \frac{-1 - <[\,0\;1\,]^T, [\,0\; -\frac{4}{17}\,]^T>}{<[\,0\;1\,]^T, [\,\frac{1}{6}\;\frac{25}{34}\,]^T>} = -\frac{26}{25}$$

Step 4 & 5: (k=3). $\det \mathbf{J}_{R_3} = 2 > 0$ implies $\hat{A}^{(3)} = \{2\}$. Since $\hat{t}_2^{(3)} = t_2^{(3)} = 6 > 1$ solution of (4.13) is given by :

$$\mathbf{x}^{(3)} + \mathbf{n}^{(3)} = \begin{bmatrix} 0 \\ -\frac{4}{17} \end{bmatrix} + \begin{bmatrix} \frac{1}{6} \\ \frac{25}{34} \end{bmatrix} = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{2} \end{bmatrix}$$

The iteration procedure is now terminated. The solution path is shown in Fig. 9(d). □

Example 5. (Step-by-step illustration of *step 8 - 13*: corner problem)

Consider the same example as in *Example 4*. However, instead of $\mathbf{x}_0 = [\,-1\; -\frac{3}{2}\,]^T$, let us choose $\mathbf{x}_0 = [\,-1\; -\frac{17}{7}\,]^T$.

Step 2 & 3: $(k=1)$.

$$\mathbf{n}^{(1)} = \begin{bmatrix} \frac{1}{2} & -1 \\ 1 & \frac{7}{3} \end{bmatrix}^{-1} \left(\begin{bmatrix} -\frac{1}{8} \\ \frac{13}{12} \end{bmatrix} - \begin{bmatrix} \frac{73}{56} \\ -\frac{67}{12} \end{bmatrix} \right) = \begin{bmatrix} \frac{20}{13} \\ \frac{200}{91} \end{bmatrix}; \quad A^{(1)} = \{ 1, 3 \}$$

$$t_1^{(1)} = \frac{0 - < [\,1\ 0\,]^T, [\,-1\ -\frac{17}{7}\,]^T >}{< [\,1\ 0\,]^T, [\,\frac{20}{13}\ \frac{200}{91}\,]^T >} = \frac{13}{20}, \quad t_3^{(1)} = \frac{-1 - < [\,0\ 1\,]^T, [\,-1\ -\frac{17}{7}\,]^T >}{< [\,0\ 1\,]^T, [\,\frac{20}{13}\ \frac{200}{91}\,]^T >} = \frac{13}{20}$$

Step 5: $(k=1)$. Since $\hat{t}_1^{(1)} = \hat{t}_3^{(1)} = \frac{13}{20}$, they are not unique. Go to *step 8.*

Step 8: $(k=1)$. $I^{(1)} = \{ 1, 3 \}$. Set $h = 1$ and compute

$$\mathbf{x}^{(2)} = \begin{bmatrix} -1 \\ -\frac{17}{7} \end{bmatrix} + \frac{13}{20} \begin{bmatrix} \frac{20}{13} \\ \frac{200}{91} \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \text{go to } step\,9.$$

Step 9: $(k=1)$. This is the first time we hit the corner $[\,0\ -1\,]^T$. Go to *Step 10.*

Step 10 & 11 : $(k=1)$. $N_I^{(2)} = 2$. Sign-sequence vector for region R_1 is $\mathbf{w}_0^{(1)} = [\,-1\ -1\ -1\,]^T$; construct $L_I^{(1)} = \{ \mathbf{w}_1^{(1)}, \mathbf{w}_2^{(1)}, \mathbf{w}_3^{(1)} \}$, where $\mathbf{w}_1^{(1)} = [\,1\ -1\ -1\,]^T$, $\mathbf{w}_2^{(1)} = [\,-1\ -1\ 1\,]^T$, $\mathbf{w}_3^{(1)} = [\,1\ -1\ 1\,]^T$.

Step 12 : $(k=1)$. Since $\det \mathbf{J}_{\mathbf{w}_1^{(1)}} = \frac{5}{2}$, $\det \mathbf{J}_{\mathbf{w}_2^{(1)}} = \frac{4}{3}$, and $\det \mathbf{J}_{\mathbf{w}_3^{(1)}} = 2$ are all positive, and since $\mathbf{w}(\hat{\mathbf{x}})$ is still undefined at this moment, pick an arbitrary region (say $\mathbf{w}_1^{(1)}$) to continue. Go to *Step 13.*

Step 13 : $(k=1)$. Repeat *step 2.* $\mathbf{n}^{(2)} = [\,\frac{7}{15}\ \frac{6}{5}\,]^T$. Since $I^{(1)} = \{ 1, 3 \}$, we only need to compute $t_2^{(2)}$ in *step 3.*

$$t_2^{(2)} = \frac{1 - < [\,1\ 0\,]^T, [\,0\ -1\,]^T >}{< [\,1\ 0\,]^T, [\,\frac{7}{15}\ \frac{6}{5}\,]^T >} = \frac{15}{7}$$

compute

$$\hat{\mathbf{x}} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \frac{1}{2}\,\frac{15}{7} \begin{bmatrix} \frac{7}{15} \\ \frac{6}{5} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{2}{7} \end{bmatrix} \quad \text{and } \mathbf{w}(\hat{\mathbf{x}}) = [\,1\ -1\ 1\,]^T$$

Since $\mathbf{w}_1^{(1)} = [\,-1\ -1\ -1\,]^T$ is *not* equal to $\mathbf{w}(\hat{\mathbf{x}})$, go back to *step 12* to select another region.

Step 12 : $(k=1)$. Since $\mathbf{w}_3^{(1)} = \mathbf{w}(\hat{\mathbf{x}})$, choose the region represented by $\mathbf{w}_3^{(1)}$ and go to *step 13.*

Step 13 : $(k=1)$. Repeat *step 2* and *step 3*: $\mathbf{n}^{(2)} = [\,\frac{1}{6}\ \frac{3}{2}\,]^T$, $t_2^{(2)} = 6$, and $\hat{\mathbf{x}} = [\,\frac{1}{2}\ \frac{7}{2}\,]^T$. Since $\mathbf{w}(\hat{\mathbf{x}}) = [\,1\ -1\ 1\,]^T$ is equal to $\mathbf{w}_3^{(1)}$, set $\mathbf{x}^{(3)} = [\,\frac{1}{2}\ \frac{7}{2}\,]^T$, $\mathbf{J}_{R_3} = \mathbf{J}_{\mathbf{w}_3^{(1)}}$, increment k by 2 and go to *step 2.*

Step 2 & 3: (k=3). (4.2) implies $\mathbf{n}^{(3)} = [\,-\frac{1}{3}\ -3\,]^T$. Set $A^{(3)} = \{\,1, 2, 3\,\}$ and (4.3) implies $t_1^{(3)} = \frac{3}{2}$, $t_2^{(3)} = -\frac{3}{2}$, $t_3^{(3)} = \frac{3}{2}$.

Step 4 & 5 : (k=3). det $J_{R_3} = 2 > 0$ implies $\hat{A}^{(3)} = \{\,1, 3\,\}$. Since $\hat{t}_1^{(3)} = \frac{3}{2} > 1$, solution of (4.13) is given by :

$$\mathbf{x}^{(3)} + \mathbf{n}^{(3)} = \begin{bmatrix} \frac{1}{2} \\ \frac{7}{2} \end{bmatrix} + \begin{bmatrix} -\frac{1}{3} \\ -3 \end{bmatrix} = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{2} \end{bmatrix} \qquad \square$$

Remark :

Two programs have been written to implement the *canonical Katzenelson algorithm* on a PDP-11/780 VAX computer running a UNIX time-sharing operating system[9].

The first program exploits the special properties possessed by equations having a *lattice structure*. The second program does not exploit this structure and is therefore more general, but less efficient. *Examples 4, 6, 7* and *8*[10] are used to compare these two programs and the result is shown in *Table 1*. Note that every example in *Table 1* has lattice structure. By exploiting this structure, one can expect the saving in cpu time to increase with the size of the circuit.

Table 1. Summary of computation for Example 4, 6, 7 and 8

Examples	Total number of regions traversed	Total CPU time (in seconds)[11]	
		general case	optimized for lattice structure
4	3	0.20 - 0.28	0.18 - 0.20
6	5	0.32 - 0.52	0.25 - 0.40
7	18	0.82 - 0.88	0.33 - 0.47
8	11	0.95 - 1.32	0.58 - 0.80

5. CONCLUDING REMARKS

Since the dc circuit model of most electronic devices are made of 2-terminal nonlinear resistors and nonlinear controlled sources depending on a single controlling variable, the equilibrium equation of most dc electronic circuits are automatically in the canonical form, upon approximating each nonlinear function by a 1-dimensional piecewise-linear function.

[9]PDP and VAX are Trademarks of the Digital Equipment Co., UNIX is a Trademark of Bell Laboratories.

[10]*Examples 6, 7* and *8* are listed in *Appendix C*.

[11] Since UNIX is a time-sharing system, the actual time depends on the current load on the system at that time. Hence we give only a range of the total CPU time.

In the more general situation when the nonlinearity consists of coupled equations, they can often be approximated by combinations of a few functions of one variable. In fact, from a theoretical point of view, we can invoke *Kolmogorov's theorem* which asserts that any *continuous* multi-dimensional function $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^n$ can be approximated to *any desired accuracy* over any compact region in $\mathbb{R}^n$ by a composition of *functions of only one variable*. This in turn will allow us to construct a circuit model whose *only nonlinear elements* are 2-terminal resistors [10]. It follows therefore that *any resistive circuit can be modeled, to within any desired accuracy, by a system of piecewise-linear equations in canonical form.*

This result is particularly appealing from a circuit-theoretic point of view because all methods of circuit analysis give rise to an equilibrium equation of the same form. In fact, for small circuits, such as the bridge circuit presented in *Example 2*, it is now possible to derive highly efficient "canned computer programs" where the user need only specify the breakpoints and slopes of the nonlinear resistors in the circuit. Such a program are ideally suited for implementation in personal *microcomputers*. For example, one can easily store in a single 5¼" floppy disk, the canned program for analyzing piecewise-linear resistive circuits in all standard configurations, e.g., ladder, lattice, bridge-T, twin-T, etc.

In the case where the circuit has a hybrid equation, the canonical equation always possesses a *lattice structure*. This remarkable property allows a substantial saving in computing time in the canonical Katzenelson algorithm. In the case where the circuit has *multiple solutions*, this property allows the development of a special algorithm which guarantees that all solutions are found.

From an *analytical* point of view, the canonical form allows us to carry out *algebraic manipulations* on piecewise-linear circuits. This make it possible to derive *closed form solutions* and *sensitivity formulas* for many prototype circuits.

Finally, it would be desirable to compare the computational efficiency between the canonical Katzenelson algorithm with the recent piecewise-linear approach described in [12] and [13].

REFERENCES

[1] J. Katzenelson, "An algorithm for solving nonlinear resistive networks," *Bell System Tech. J.*, vol. 44, pp. 1605-1620, Oct. 1965.

[2] T. Fujisawa and E.S. Kuh, "Piecewise-linear theory of nonlinear networks," *SIAM J. Appl. Math.*, vol. 22, no. 2, pp. 307-328, Mar. 1972.

[3] T. Fujisawa, E.S. Kuh and T. Ohtsuki, "A sparse matrix method for analysis of piecewise-linear resistive networks," *IEEE Trans. Circuit Theory*, vol. CT-19, no. 6, pp. 571-584, Nov. 1972.

[4] M.J. Chien and E.S. Kuh, "Solving piecewise-linear equation for resistive networks," *Circuit Theory and Applications*, vol. 4, pp. 3-24, 1976.

[5] M.J. Chien and E.S. Kuh, "Solving nonlinear resistive networks using piecewise-linear analysis and simplicial subdivision," *IEEE Trans. Circuits and Systems*, vol. CAS-24, no. 6, pp. 305-317, Jan. 1977.

[6] T. Ohtsuki, T. Fujisawa, and S. Kumagai, "Existence theorems and a solution algorithm for piecewise-linear resistor networks," *SIAM J. Math. Anal.*, vol. 8 no. 1, pp. 69-99, Feb. 1977.

[7] L.O. Chua and S.M. Kang, "Section-wise piecewise-linear functions: canonical representation, properties, and applications," *Proceedings of the IEEE*, vol. 65, no. 6, pp. 915-929, June 1977.

[8] S.M. Kang and L.O. Chua, "A global representation of multidimensional piecewise-linear functions with linear partitions," *IEEE Trans. Circuits and Systems*, vol. CAS-25, no. 11, pp. 938-940, Nov. 1978.

[9] L.O. Chua, "Device modeling via basic nonlinear circuit elements," *IEEE Trans. Circuits and Systems*, vol. CAS-27, pp. 1014-1044, Nov. 1980.

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

[11] L.O. Chua and R. Ying, "Finding all solutions of piecewise-linear equations", U.C. Berkeley ERL Memorandom No. UCB/ERL M81/54, July 23, 1981.

[12] S.N. Stevens and P.M. Lin, "Analysis of piecewise-linear resistive networks using complementary pivot theory," *IEEE Trans. Circuits and Systems*, vol. CAS-28, pp. 429-441, May 1981.

[13] W.M.G. van Bokhoven, "Macromodelling and simulation of mixed analog-digital networks by a piecewise-linear system approach," *Proceedings of the 1980 IEEE International Conference on Circuits and Computers*.

APPENDIX

A. Justification of coefficients for canonical equation (2.1)

For $x \neq x_i$, $i = 1, 2, \ldots, p$, (2.1) implies that

$$\frac{d}{dx} f(x) = b + \sum_{i=1}^{p} c_i \, sgn \, (x - x_i) \tag{A.1}$$

since $m_j = \frac{d}{dx} f(x) \mid_{x_j < x < x_{j+1}}$, (A.1) implies that

$$m_0 = b - \sum_{i=1}^{p} c_i \tag{A.2}$$

$$m_j = b + \sum_{i=1}^{j} c_i - \sum_{i=j+1}^{p} c_i \quad j = 1, 2, \ldots, n-1 \tag{A.3}$$

$$m_p = b + \sum_{i=1}^{p} c_i \tag{A.4}$$

It is clear that (A.2) and (A.4) imply (2.2), (A.3) implies (2.3), and finally (2.4) follows from (2.1) upon substituting $x = 0$.

B. Justification of coefficients for equation (2.7)

We derive (2.8) first. For $\mathbf{x} \in \mathbb{R}^n$ and $< \alpha_i , \mathbf{x} > \neq \beta_i, i = 1, 2, \ldots, p$, (2.7) implies

$$\nabla f(\mathbf{x}) = \mathbf{b} + \sum_{i=1}^{p} c_i [\, sgn \, (< \alpha_i , \mathbf{x} > - \beta_i) \,] \, \alpha_i \tag{A.5}$$

(A.5) implies

$$\sum_{j=1}^{k} \left\{ \nabla f(\mathbf{x}) \mid_{\mathbf{x} \in R_{j\infty}} \right\} = \sum_{j=1}^{k} \mathbf{b} + \sum_{j=1}^{k} \left\{ \sum_{i=1}^{p} c_i [\, sgn \, (< \alpha_i , \mathbf{x} > - \beta_i) \,] \, \alpha_i \mid_{\mathbf{x} \in R_{j\infty}} \right\}$$

which can be simplified to

$$\mathbf{b} = \frac{1}{k} \sum_{j=1}^{k} \left\{ \nabla f(\mathbf{x}) \mid_{\mathbf{x} \in R_{j\infty}} \right\} + \frac{1}{k} \sum_{j=1}^{k} \left\{ \sum_{i=1}^{p} c_i [\, sgn \, (< \alpha_i , \mathbf{x} > - \beta_i) \,] \, \alpha_i \mid_{\mathbf{x} \in R_{j\infty}} \right\}$$

Therefore to derive (2.8), it suffices to show

$$\sum_{j=1}^{k} \left\{ \sum_{i=1}^{p} c_i [\, sgn \, (< \alpha_i , \mathbf{x} > - \beta_i) \,] \, \alpha_i \mid_{\mathbf{x} \in R_{j\infty}} \right\} = 0 \tag{A.6}$$

Since we are only interested in $\mathbf{x} \in R_{j\infty}$ for $j = 1, 2, \ldots, k$, the procedure described below will eliminate all bounded regions as well as pseudo-unbounded regions.

Consider a very general case such that there are g groups of hyperplanes in the linear partition, each containing n_g parallel hyperplanes. For each group, we merge the n_g hyperplanes into one. (Note: we have $k_1 \overset{def}{=} p - \sum_{j=1}^{g} n_j + g$ hyperplanes left.) Then move each of the k_1 hyperplanes in parallel to itself until all pseudo-unbounded and bounded regions vanish. This is always possible since no two of the k_1 hyperplanes are parallel to each other. Clearly there are $k \overset{def}{=} 2k_1$ regions left and all of them are essentially-unbounded. Since each hyperplane $< \alpha_i , \mathbf{x} > = \beta_i$, $i = 1, 2, \ldots , p$, divides $\mathbb{R}^n$ into two regions whose sign-sequence vectors differ only at the i-th position, we have

$$\sum_{j=1}^{k} sgn\ (< \alpha_i , \mathbf{x} >) = 0 \qquad \text{for } i = 1, 2, \ldots , p$$

where $\mathbf{x}_j \in R_{j_\infty}$ for $j = 1, 2, \ldots , k$. Equation (4.6) follows upon interchanging the summation sign.

$$\sum_{j=1}^{k} sgn\ (< \alpha_i , \mathbf{x}_j > - \beta_i)] = 0 \qquad \text{for } i = 1, 2, \ldots , p$$

To derive (2.9), let $\mathbf{x} \in R_{j+}$ in (A.5), we get

$$\nabla f(\mathbf{x}) \,|_{\mathbf{x} \in R_{j+}} = \mathbf{b} + \sum_{\substack{i=1 \\ i \neq j}}^{p} c_i \,[\, sgn\ (< \alpha_i , \mathbf{x} > - \beta_i) \,]\, \alpha_i + c_j\ \alpha_i \tag{A.7}$$

Similarly, let $\mathbf{x} \in R_{j-}$ in (A.5), we get

$$\nabla f(\mathbf{x}) \,|_{\mathbf{x} \in R_{j-}} = \mathbf{b} + \sum_{\substack{i=1 \\ i \neq j}}^{p} c_i \,[\, sgn\ (< \alpha_i , \mathbf{x} > - \beta_i) \,]\, \alpha_i - c_j\ \alpha_i \tag{A.8}$$

Multiplying both sides of (A.7) and (A.8) by $\alpha_i{}_j^T$ and subtracting, we obtain (2.9).

Finally, substituting $\mathbf{x} = \mathbf{0}$ in (2.7), we get (2.10). $\square$

C. Examples

Example 6.

Consider the circuit shown in Fig. A1. Piecewise-linear resistors R1 and R3 are voltage controlled, piecewise-linear resistor R2 is current controlled. Their canonical form representation are given by

$$\text{R1}: \ i_1 = \frac{5}{6} |\, v_1 + 6 \,| - \frac{5}{6} |\, v_1 - 6 \,|$$

$$\text{R2}: \ v_2 = \frac{1}{6} |\, i_2 + 1 \,| - \frac{1}{6} |\, i_2 - 5 \,|$$

$$\text{R3}: \ i_3 = v_3 - \frac{5}{4} |\, v_3 - 1 \,| - 2 \,|\, v_3 - 2 \,| - |\, v_3 - 3 \,|$$

The associated circuit equation is represented in the canonical form as follow :

$$\mathbf{B} \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} + abs\ (\mathbf{D}^T \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} - \mathbf{e}) = \begin{bmatrix} 5 \\ 5 \\ -5 \end{bmatrix}$$

where

$$\mathbf{B}=\begin{bmatrix}1&1&1\\0&1&1\\0&0&-2\end{bmatrix},\quad \mathbf{C}=\begin{bmatrix}-\frac{5}{6}&\frac{5}{6}&0&0&0&0&0\\0&0&-\frac{1}{6}&\frac{1}{6}&0&0&0\\0&0&0&0&\frac{5}{4}&-2&1\end{bmatrix},\quad \mathbf{D}=\begin{bmatrix}1&1&0&0&0&0&0\\0&0&1&1&0&0&0\\0&0&0&0&1&1&1\end{bmatrix}$$

$$\mathbf{e}=[\,-6\;\;6\;\;-1\;\;5\;\;1\;\;2\;\;3\,]^T$$

The initial point is $\mathbf{x}_0=[\,7\;\;6\;\;-1\,]^T$ and the solution is $[\,0.1\;\;2.2\;\;2.87\,]^T$. □

Example 7.

Consider the circuit shown in Fig. A2. R1 and R2 are voltage controlled. Their characteristics are given in the canonical form as follow :

$$\text{R1}:\; i_1=-\frac{125}{8}+\frac{9}{8}v_1+\frac{7}{8}|\,v_1+1\,|-\frac{3}{2}|\,v_1-2\,|+\frac{3}{4}|\,v_1-5\,|$$

$$-\frac{1}{8}|\,v_1\,11\,|-\frac{9}{8}|\,v_1-13\,|+2\,|v_1-15\,|$$

$$\text{R2}:\; i_2=-\frac{29}{4}+\frac{3}{2}v_2-\frac{3}{2}|\,v_2+8\,|+\frac{3}{2}|\,v_2+5\,|-\frac{3}{2}|\,v_2+1\,|$$

$$+\frac{3}{2}|\,v_2+1\,|-\frac{3}{4}|\,v_2-3\,|-\frac{5}{4}|\,v_2-8\,|+\frac{3}{2}|\,v_2-10\,|$$

$$+|\,v_2-13\,|-\frac{5}{4}|\,v_2-16\,|+\frac{1}{4}|\,v_2-18\,|$$

The associated circuit equation is represented in the canonical form as follow :

$$\begin{bmatrix}\frac{13}{8}&\frac{1}{2}\\\frac{1}{2}&2\end{bmatrix}\begin{bmatrix}v_1\\v_2\end{bmatrix}+\sum_{i=1}^{16}\mathbf{c}_i\,|<\boldsymbol{\alpha}_i\,,\begin{bmatrix}v_1\\v_2\end{bmatrix}>-\beta_i\,|=\begin{bmatrix}\frac{161}{8}\\-\frac{11}{4}\end{bmatrix}$$

where $\boldsymbol{\alpha}_i=[\,1\;0\,]^T$ for $i=1,2,\ldots,6$ and $\boldsymbol{\alpha}_i=[\,0\;1\,]^T$ for $i=7,8,\ldots,16$.

$$\mathbf{c}_1=\begin{bmatrix}\frac{7}{8}\\0\end{bmatrix},\;\mathbf{c}_2=\begin{bmatrix}-\frac{3}{2}\\0\end{bmatrix},\;\mathbf{c}_3=\begin{bmatrix}\frac{3}{4}\\0\end{bmatrix},\;\mathbf{c}_4=\begin{bmatrix}-\frac{1}{8}\\0\end{bmatrix},\;\mathbf{c}_5=\begin{bmatrix}-\frac{9}{8}\\2\end{bmatrix},\;\mathbf{c}_6=\begin{bmatrix}2\\0\end{bmatrix},$$

$$\mathbf{c}_7=\begin{bmatrix}0\\-\frac{3}{2}\end{bmatrix},\;\mathbf{c}_8=\begin{bmatrix}0\\\frac{3}{2}\end{bmatrix},\;\mathbf{c}_9=\begin{bmatrix}0\\-\frac{3}{2}\end{bmatrix},\;\mathbf{c}_{10}=\begin{bmatrix}0\\\frac{3}{2}\end{bmatrix},\;\mathbf{c}_{11}=\begin{bmatrix}0\\-\frac{3}{4}\end{bmatrix},\;\mathbf{c}_{12}=\begin{bmatrix}0\\-\frac{5}{4}\end{bmatrix},$$

$$\mathbf{c}_{13}=\begin{bmatrix}0\\\frac{3}{2}\end{bmatrix},\;\mathbf{c}_{14}=\begin{bmatrix}0\\1\end{bmatrix},\;\mathbf{c}_{15}=\begin{bmatrix}0\\-\frac{5}{4}\end{bmatrix},\;\mathbf{c}_{16}=\begin{bmatrix}0\\\frac{1}{4}\end{bmatrix}$$

$$\beta_1=-1,\;\beta_2=2,\;\beta_3=5,\;\beta_4=11,\;\beta_5=13,\;\beta_6=15,\;\beta_7=-8,\;\beta_8=-5,$$

$$\beta_9=-3,\;\beta_{10}=-1,\;\beta_{11}=3,\;\beta_{12}=8,\;\beta_{13}=10,\;\beta_{14}=13,\;\beta_{15}=16,\;\beta_{16}=18$$

The initial point is $\mathbf{x}_0=[\,16\;\;20\,]^T$ and the solution is $[\,1.5\;\;1.5\,]^T$. □

Example 8.

Consider the four transistor circuit shown in Fig. A3(a). Each transistor is modeled by a controlled source in series with a p-n junction diode as shown in Fig A3(b). The diode $I_D - V_D$ characteristic is represented by a continuous piecewise-linear function with 6 segments as shown in Fig. A3(c). The reader is referred to [11] for the canonical representation of the equilibrium equation. The initial point is $\mathbf{x}_0 = [\ 1\ 0\ 1\ 1\ ]^T$ and the solution is $[\ 3.290404\times10^{-1}\ 3.652363\times10^{-1}\ 3.376266\times10^{-1}\ 3.602442\times10^{-1}\ ]^T$. $\square$

FIGURE CAPTIONS

Fig. 1. A continuous 1-dimensional piecewise-linear function.

Fig. 2. Examples illustrating pseudo-unbounded regions and essentially-unbounded regions :

(a) R_1, R_5 and R_{10} are the only pseudo-unbounded regions.

(b) R_7 is the only pseudo-unbounded region.

Fig. 3. In *Example 1*, $\mathbb{R}^2$ is partitioned by two 1-dimensional hyperplanes (straight lines) $x_1 = x_2$ and $x_1 = -x_2$ into 4 regions R_1, R_2, R_3 and R_4.

Fig. 4. Circuit containing 2 piecewise-linear resistors.

Fig. 5. A composite branch.

Fig. 6. Bridge circuit containing 5 voltage-controlled piecewise-linear resistors.

Fig. 7. Extracting all nonlinear resistors we obtain a resistive m-port $\hat{N}$ containing only *linear* resistive elements and dc independent sources. The ports attached to voltage-controlled {resp. current-controlled} resistors are called voltage {resp. current} ports.

Fig. 8. Figures for *Example 3*.

(a) Circuit containing 2 piecewise-linear resistors and a controlled source.

(b) v-i characteristics for piecewise-linear resistor R1.

(c) v-i characteristics for piecewise-linear resistor R2.

(d) $\hat{N}$ is obtained by extracting R1 across the voltage port and R2 across the current port.

(e) Lattice structure defined by 3 1-dimensional hyperplanes (straight lines) $v_1 = 0$, $v_2 = 1$ and $i_2 = -1$, each one parallel to either coordinate axis v_1 or i_2. Note that all regions have the same *regular* pattern – bounded or unbounded rectangles – typical in a lattice structure.

Fig. 9. Figures for *Example 4*.

(a) Circuit containing 2 piecewise-linear resistors and a controlled source.

(b) v-i characteristics for piecewise-linear resistor R1.

(c) v-i characteristics for piecewise-linear resistor R2.

(d) Lattice structure in the v_1–v_2 plane. The dotted line path indicates the iteration goes from $\mathbf{x}^{(1)} = \mathbf{x}_0$ to $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, and finally converges to the true solution at $(\frac{1}{6}, \frac{1}{2})$.

Fig. A1. Figure for *Example 6*. Circuit containing 3 piecewise-linear resistors.

Fig. A2. Figure for *Example 7*. Circuit containing 2 piecewise-linear resistors.

Fig. A3. Figures for *Example 8*.

(a) A 4-transistor circuit.

(b) Simplified Ebers-Moll model of an NPN transistor.

(c) Piecewise-linear approximation of diode v-i characteristic: $m_0 = 0$, $m_1 = 2.153\times10^{-2}$, $m_2 = 2.666\times10^{-2}$, $m_3 = 3.765\times10^{-2}$, $m_4 = 8.603\times10^{-2}$, $m_5 = 1.865\times10^{-1}$;

$V_1 = 0.306$, $V_2 = 0.321$, $V_3 = 0.336$, $V_4 = 0.351$, $V_5 = 0.376$.

$$I_D = -3.33570322\times10^{-2} + 9.32400146\times10^{-2}\, v_D + 1.25666608\times10^{-2} \mid v_D - V_1 \mid$$
$$+ 2.23537270\times10^{-3} \mid v_D - V_2 \mid + 8.49354618\times10^{-3} \mid v_D - V_3 \mid$$
$$+ 2.41900658\times10^{-2} \mid v_D - V_4 \mid + 5.02251145\times10^{-2} \mid v_D - V_5 \mid .$$

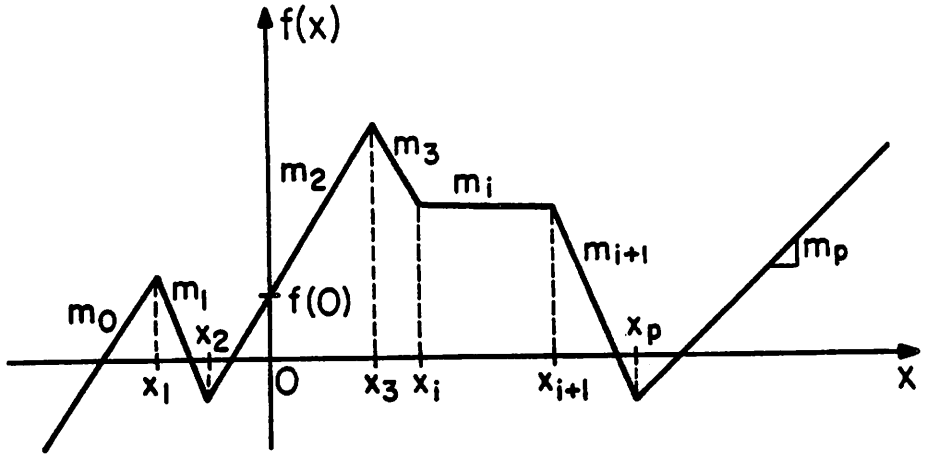

f(x)
m_0
m_1
m_2
m_3
m_i
m_{i+1}
m_p
f(0)
x_1
x_2
O
x_3
x_i
x_{i+1}
x_p
x

Fig. 1

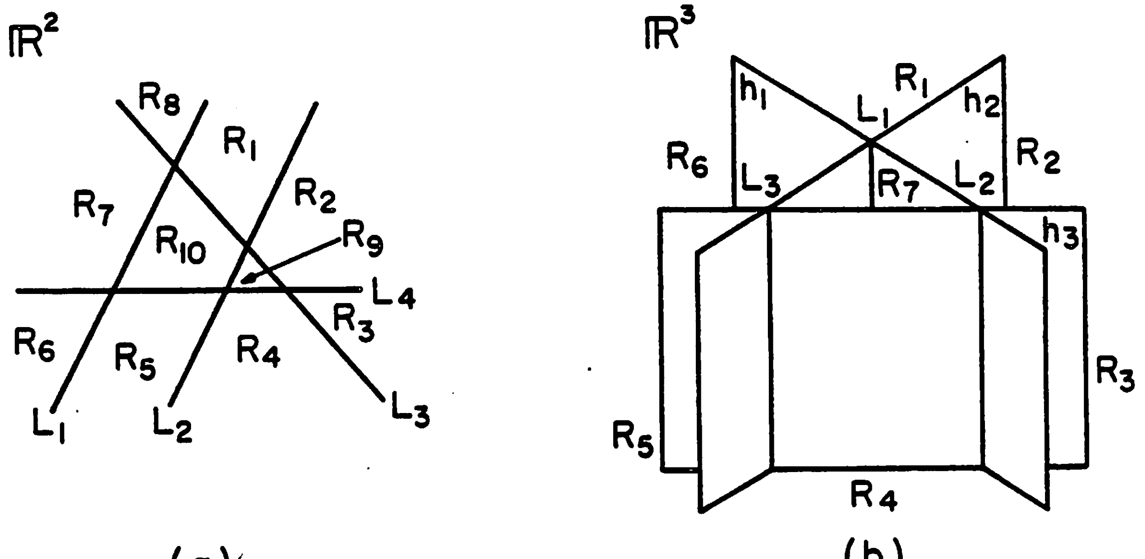
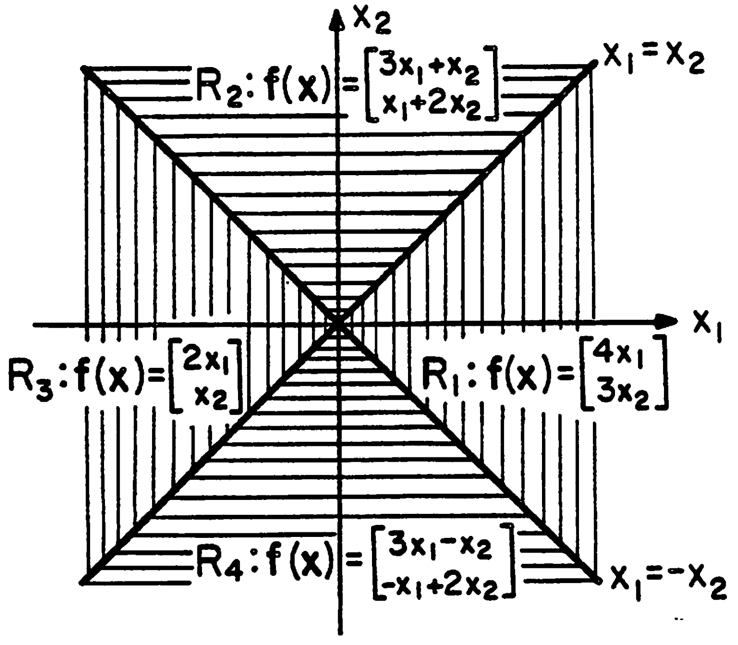

$\mathbb{R}^2$
R_8
R_1
R_7
R_2
R_{10}
R_9
L_4
R_3
R_6
R_5
R_4
L_1
L_2
L_3
(a)
$\mathbb{R}^3$
h_1
R_1
h_2
L_1
R_6
R_2
L_3
R_7
L_2
h_3
R_3
R_5
R_4
(b)

Fig. 2



x_2
x_1
$x_1 = x_2$
$x_1 = -x_2$
$R_2 : f(x) = \begin{bmatrix} 3x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}$
$R_3 : f(x) = \begin{bmatrix} 2x_1 \\ x_2 \end{bmatrix}$
$R_1 : f(x) = \begin{bmatrix} 4x_1 \\ 3x_2 \end{bmatrix}$
$R_4 : f(x) = \begin{bmatrix} 3x_1 - x_2 \\ -x_1 + 2x_2 \end{bmatrix}$

Fig. 3

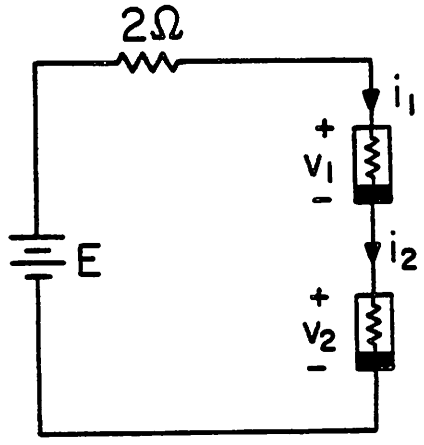


Fig. 4

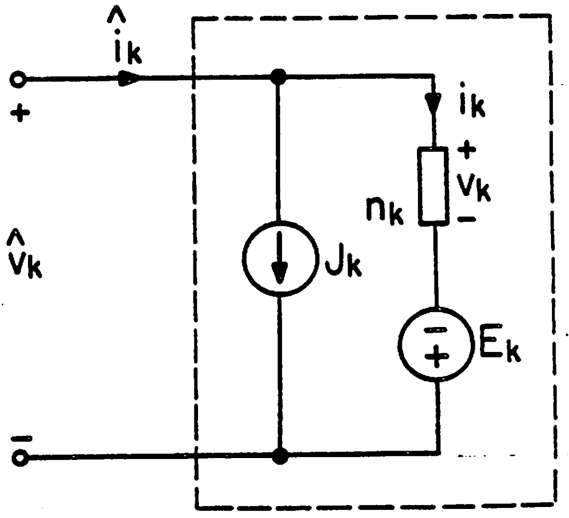
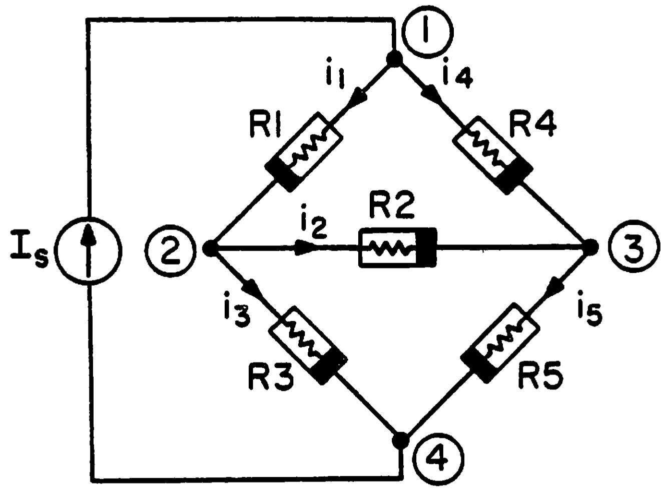
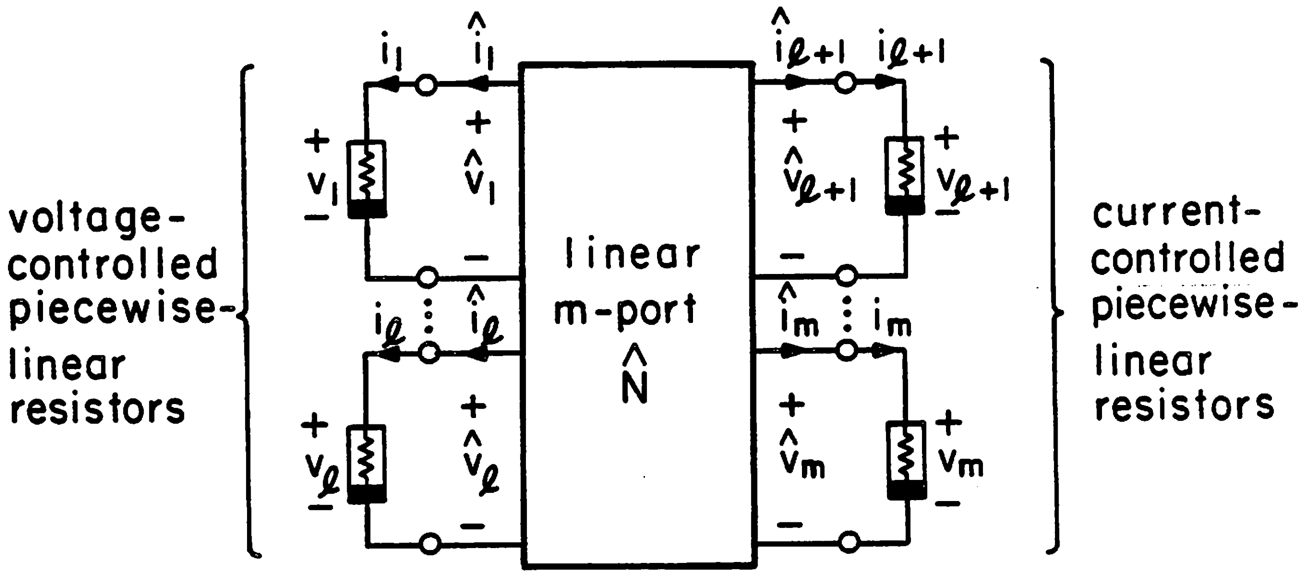


Fig. 5



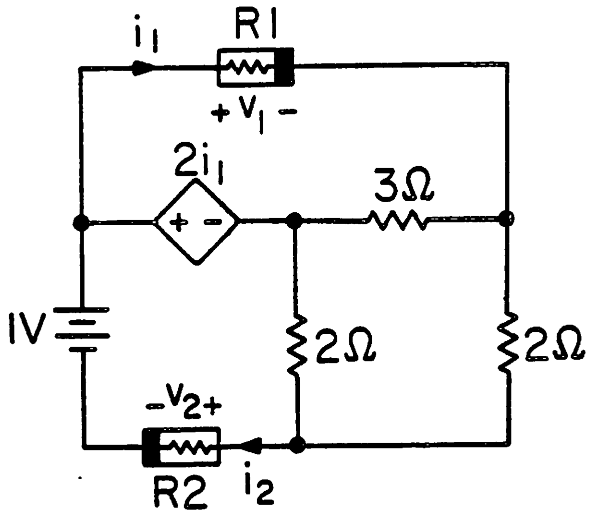
1
i_1
i_4
R1
R4
R2
I_s
2
i_2
3
i_3
i_5
R3
R5
4

Fig. 6



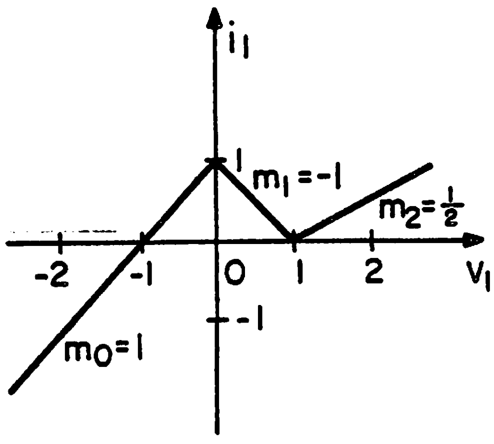
i_1
$\hat{i}_1$
$\hat{i}_{\ell+1}$
$i_{\ell+1}$
+
v_1
−
+
$\hat{v}_1$
−
+
$\hat{v}_{\ell+1}$
−
+
$v_{\ell+1}$
−
voltage-controlled piecewise-linear resistors
linear m-port $\hat{N}$
current-controlled piecewise-linear resistors
i_ℓ
$\hat{i}_\ell$
$\hat{i}_m$
i_m
+
v_ℓ
−
+
$\hat{v}_\ell$
−
+
$\hat{v}_m$
−
+
v_m
−

Fig. 7



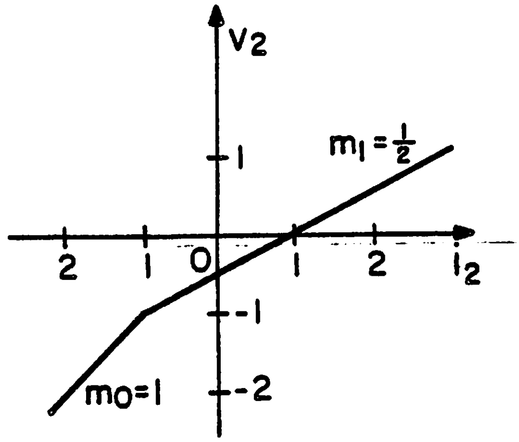
i1
R1
+ v1 -
2i1
3Ω
1V
2Ω
2Ω
-v2+
R2
i2

(a)



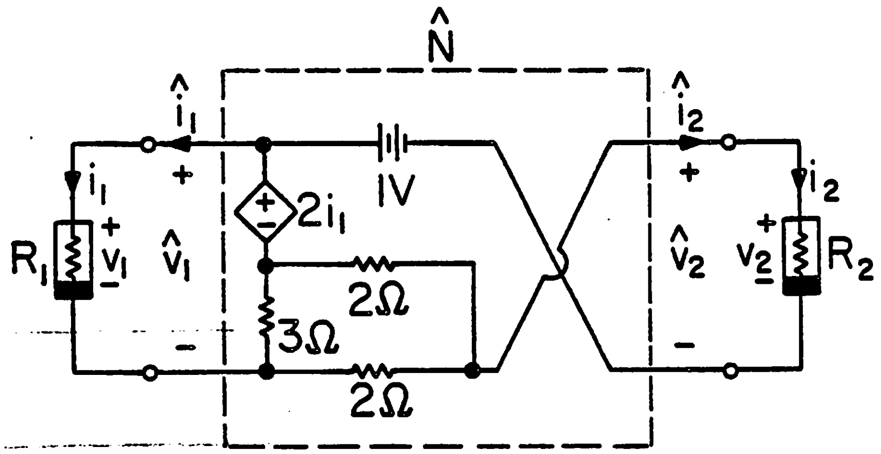
i1
1
m1 = -1
m2 = 1/2
-2
-1
0
1
2
v1
-1
m0=1

(b)



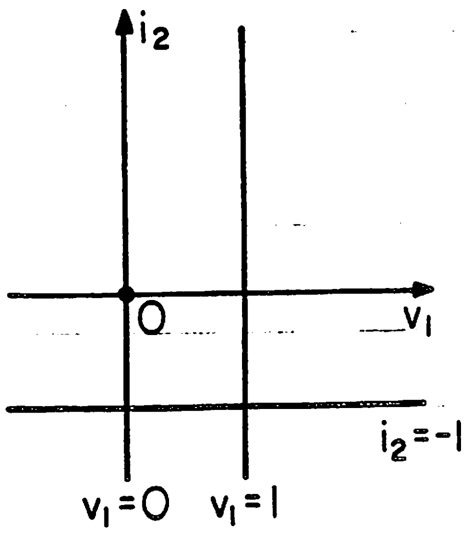
v2
1
m1 = 1/2
2
1
0
1
2
i2
-1
-2
m0=1

(c)



N̂
î1
î2
i1
+
2i1
1V
+
i2
R1
+
v1
-
v̂1
2Ω
v̂2
+
v2
-
R2
3Ω
-
-
2Ω

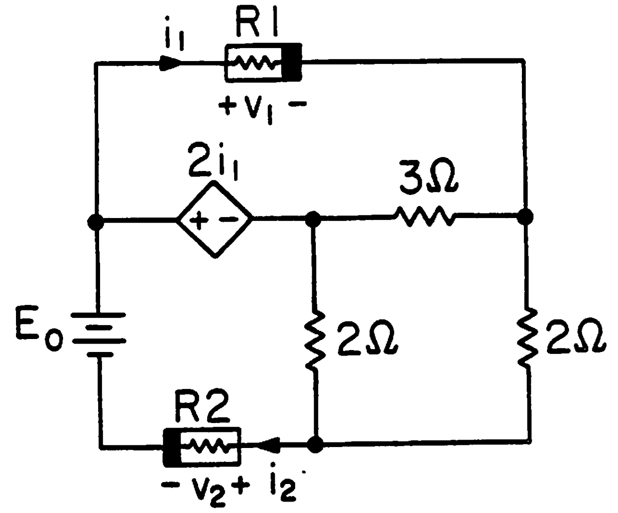
(d)



i2
O
v1
i2=-1
v1=0
v1=1

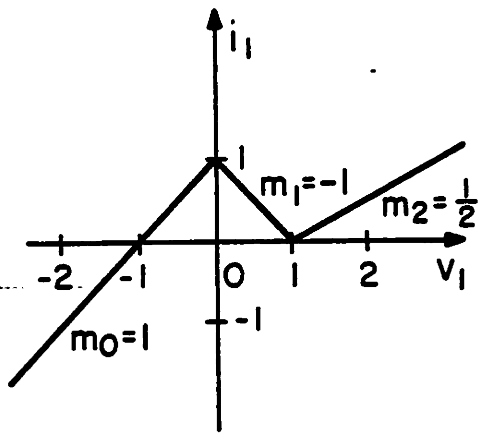
(e)

Fig. 8



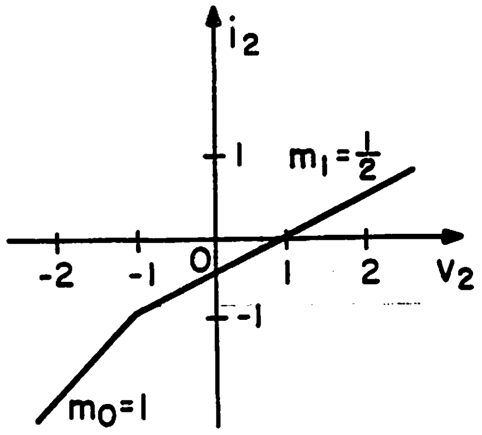
i_1
R1
$+v_1-$
$2i_1$
3Ω
E_0
2Ω
2Ω
R2
$-v_2+$
i_2

(a)



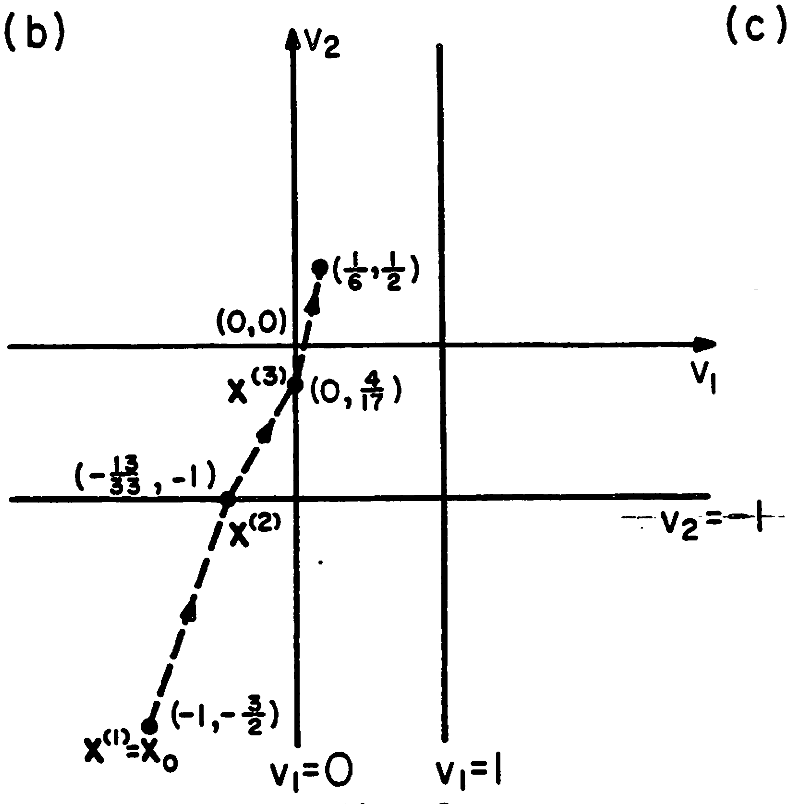
i_1
1
$m_1=-1$
$m_2=\frac{1}{2}$
-2
-1
0
1
2
v_1
-1
$m_0=1$

(b)



i_2
1
$m_1=\frac{1}{2}$
-2
-1
0
1
2
v_2
-1
$m_0=1$

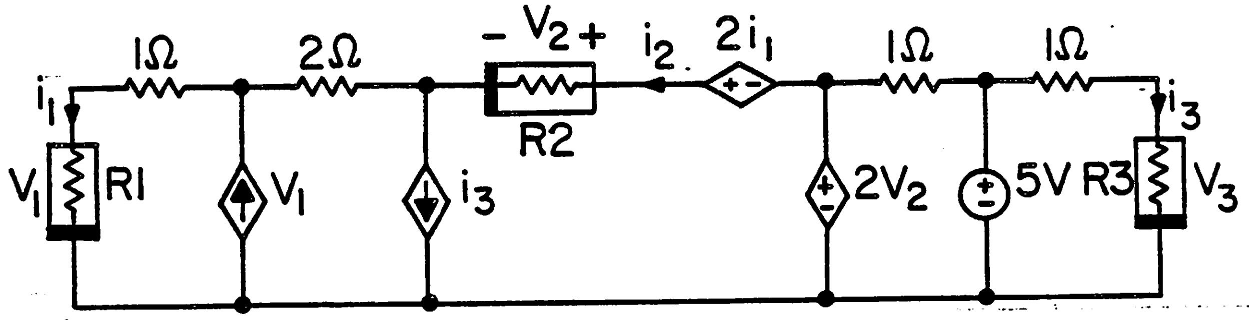
(c)



v_2
$(\frac{1}{6},\frac{1}{2})$
(0,0)
v_1
$X^{(3)}$
$(0,\frac{4}{17})$
$(-\frac{13}{33},-1)$
$X^{(2)}$
$v_2=-1$
$(-1,-\frac{3}{2})$
$X^{(1)}=X_0$
$v_1=0$
$v_1=1$

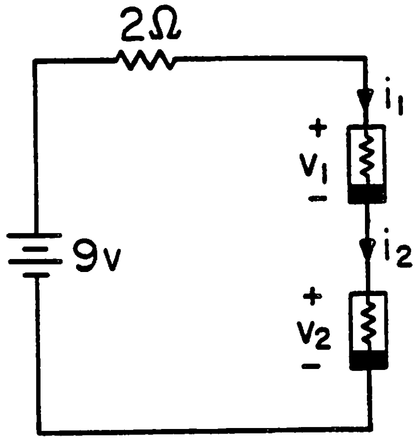
(d)

Fig. 9



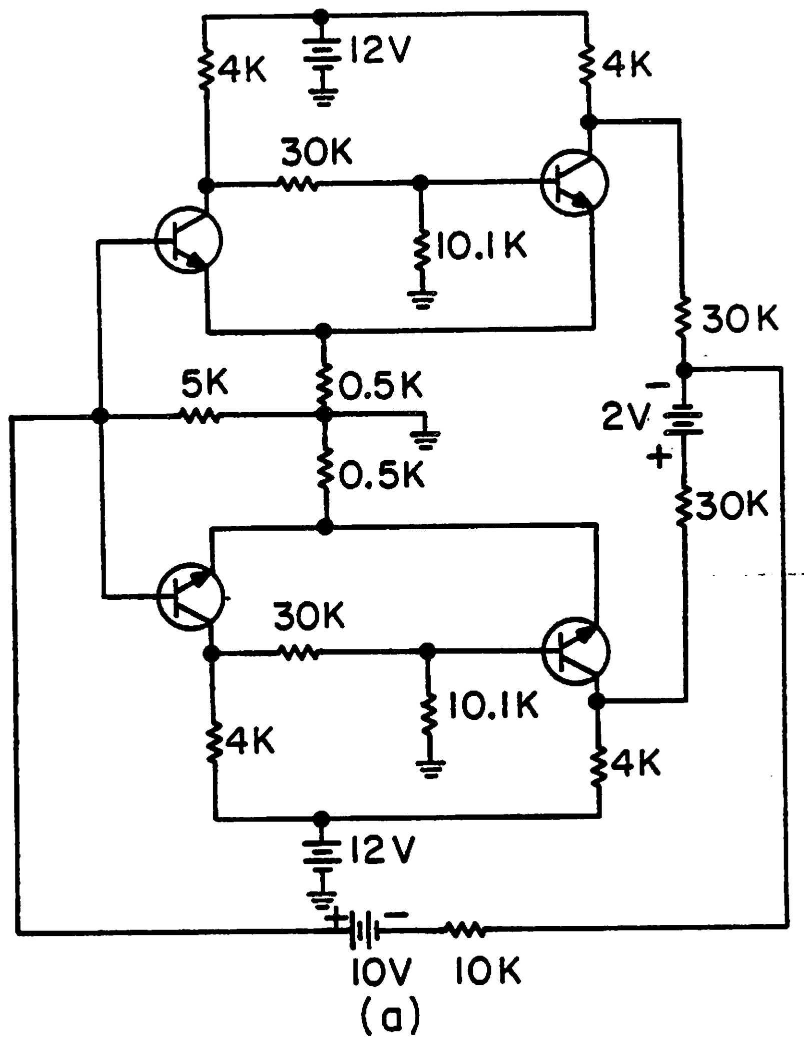
1Ω
2Ω
− V2 +
R2
i_2
$2i_1$
1Ω
1Ω
i_1
V_1
R1
V_1
i_3
$2V_2$
5V
R3
i_3
V_3

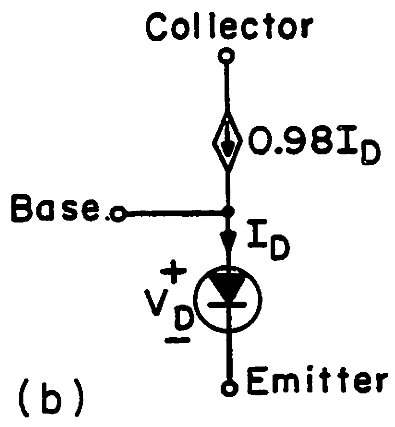
Fig. A1



2Ω
i_1
+
V_1
−
i_2
9v
+
V_2
−

Fig. A2


12V
4K
4K
30K
10.1K
30K
5K
0.5K
2V
0.5K
30K
30K
10.1K
4K
4K
12V
10V
10K
(a)


Collector
0.98I_D
Base
I_D
V_D
Emitter
(b)

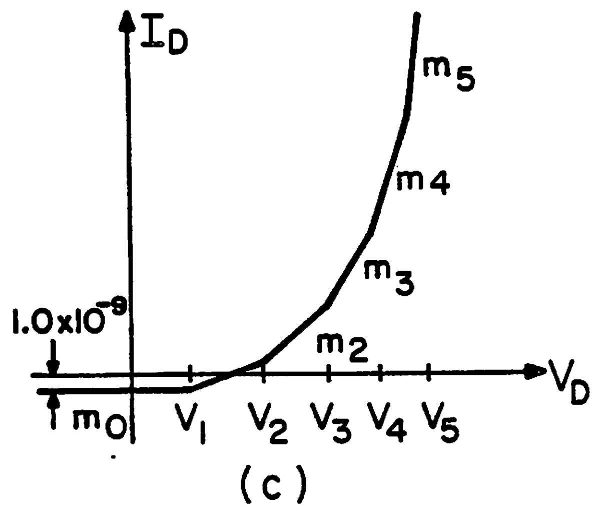

I_D
m_5
m4
m_3
m_2
1.0×10^{-9}
m_0
V_1
V_2
V_3
V_4
V_5
V_D
(c)

Fig. A3

PROGRAM LISTING

```
/*                  --  Canonical  Katzenelson  Algorithm  --
**
**
**                        Copyright (c) 1982  Robin L.P. Ying
**
**
** This package uses the "Canonical Katzenelson Algorithm" to
** solve the piece-wise linear algebraic system
**                              f(x) = y
** where f(.) is in the piecewise-linear canonical form.  If
** f(.) has a lattice structure, then a specially optimized
** version of the algorithm is used.
**
** It can be run on a PDP-11/780 VAX-UNIX system which supports
** the double-precision IMSL library.  It contains the following
** separated modules:
**
**        cka.h:  contains definitions of data structures and global
**                variables.
**
**       main.c:  handles command line options.
**
**      input.c: reads input from user terminal or a data file.
**
**       katz.c:  contains the main routines for solving the given
**                piecewise-linear system.
**
**    lattice.c: same as katz.c but optimized for lattice structure.
**
**     corner.c: handles the corner problem.
**
**      init.c : does initializations for lattice structure.
**
**     region.c:  checks and computes items in structure REGION.
**
**      queue.c:  contains REGION_QUEUE manipulating routines.
**
**      print.c:  contains printing routines.
**
**      error.c:  prints run-time error messages.
**
**    support.c:  containing various supporting routines.
**
**     Makefile:  file maintenance program.
**
**
** The folowing routines are needed from the double-precision IMSL
** library:
**       ludatf(), luelmf(), lureff(), uertst(), ugetio(), vxadd(),
**       vxmul(), vxsto().
*/
```

```
/*          - katz.h -
**
** This module is the header file for all KATZ routines.
**
*/


#define pf          printf            /* abbreviations */
#define  fpf        fprintf
#define pm          prdmtrx
#define pv          prdvctr
#define piv         privctr

#define B_SIZE      256               /* input string buffer size */
char        buf[B_SIZE];              /* input string buffer */

#define   FORMAT1"%6.3f "             /* printing format */
#define   FORMAT2"%13.6e "

#define RNIL        (REGION  *)       0177777
#define CNIL        (CORNER  *)       0177777

typedef struct      region {
          struct    region *link;     /* region queue link */
          int       id;               /* region id */
          int       *sgnsq;           /* sign sequence vector [hyp] */
          double    *px;              /* point in region [dim] */
          double    *py;              /* f(px) [dim] */
          double    *dx;              /* <alpha,x> [hyp] */
          double    *nk;              /* solution curve direction [dim] */
          double    *jcbn;            /* Jacobian matrix [dim][dim] */
          int       detsgn;           /* sign of determinant of Jacobian */
          int       *bdry;            /* boundaries [hyp] (lattice) */
} REGION;

typedef struct      {
          REGION    *head;            /* head of queue */
          REGION    *tail;            /* tail of queue */
          int       n;                /* number of elements in queue */
} REGION_QUEUE;

/* intersection of solution curve and boundary */
typedef   struct    {
          double    value;
          int       index;
} TI;

TI        *pti;                       /* array of TI */
TI        **qti;                      /* array of pti */

typedef struct      {                 /* indices of e[] */
          int       *beta;
} BETA;

typedef struct      corner {
          struct    corner *link;     /* corner queue link */
          double    *cx;              /* corner point [dim] */
          int       *index;           /* indices [n_uniq] */
          int       *sgnsq;           /* sign sequence vector [hyp] */
          int       sgnflg;           /* =1 if sgnsq[] is set */
```

```
        REGION_QUEUE *CNRRGN; /* neighborhood regions of the corner */
} CORNER;

typedef struct {
        CORNER *head;           /* head of queue */
        CORNER *tail;           /* tail of queue */
        int    n;               /* number of elements in queue */
} CORNER_QUEUE;

/* variables defined in main.c */
extern  int     lattice;        /* =1 : lattice structure */
extern  int     dim;            /* dimension of system */
extern  int     hyp;            /* total # of hyperplanes */
extern  int     aflg;           /* =1 : debug flag */
extern  int     pflg;           /* =1 : print details of iteration */
extern  int     tflg;           /* =1 : test solution */
extern  int     xflg;           /* =1 : x[] is defined in input file */
extern  int     imsl;           /* =1 : using IMSL routines */
extern  int     sigdgt;         /* significant digit */
extern  double  epsilon;        /* if |x| < epsilon then x:=0 */
extern  double  *B;             /* B[dim][dim] */
extern  double  *C;             /* C[dim][hyp] */
extern  double  *D;             /* D[dim][hyp] */
extern  double  *e;             /* e[hyp] */
extern  double  *x;             /* x[dim] */
extern  double  *y;             /* y[dim] */

/* variables defined in input.c */
extern  int     line;           /* line # in the input file */

/* variables defined in region.c */
extern  int     rgn_count;
extern  int     ier;

/* variables defined in init.c */
extern  REGION_QUEUE  *RGNQ;
extern  CORNER_QUEUE  *CNRQ;
extern  double  *wk;            /* wk[dim*(dim+3)] */
extern  double  *jcbntmp;       /* jcbntmp[dim][dim] */
extern  BETA    *axis;          /* *axis[dim] */
extern  int     *beta_count;    /* beta_count[dim] */
extern  int     *dcolumn;       /* dcolumn[hyp] */
```

```
/*          - main.c -
**
** main() -- the main interactive routine.
**
**          Command line flags:
**
**          '-a':    this option sets the 'aflg' as well as 'pflg' &
**                   'tflg' so that all details of iteration will be
**                   printed.
**
**          '-p':    this option sets the 'pflg' so that details of
**                   each region will be printed.
**
**          '-t':    this option sets the 'tflg' so that the solution
**                   will be tested by substituting it back to f().
**
**          '-i':    this option sets the 'imsl' flag so that the IMSL
**                   routines are used.
**
**          '-s n':  this option sets the significant digit to 'n'
**                   decimal digits (default: n=9), n < 0 is ignored;
**                   if n = 0 then the accuracy test in IMSL routine
**                   is disabled; this option automatically sets the
**                   '-i' option.
**
**          '-f input-file':
**                   the file name after '-f' is taken to be the input
**                   file, otherwise the program will prompt for input
**                   option.
**
**          '-o output-file':
**                   the file name after '-o' is taken to be the output
**                   file which contains the coefficients of the pwl
**                   function key-inned from the user terminal, so that
**                   the output file can be used as an input file next
**                   time; if the user has selected the terminal input
**                   option but did not specify the output file name,
**                   the default output file name is "Pwlf.def".
*/

#include <stdio.h>
#include "cka.h"

FILE      *fpin=stdin, *fpout=stdout;

int       lattice=0;
int       dim, hyp;
int       aflg=0, pflg=0, tflg=0, xflg=0, imsl=0, sigdgt=9;
double    epsilon, *B, *C, *D, *e, *x, *y;

main (argc, argv)                                                  main

register int      argc;
register char     **argv;
{
          register int      i, fflg=0, oflg=0, sflg=0;
          register char     *str;
          double            *pd;

          FILE     *fopen();
          char     *s_get();

          pf("\tC K A\t2.0\n\n");
```

```
/* set lattice structure flag */
if ( strcmpr("Lx",argv[0],2) ) {
     lattice++;
     pf("** [ optimized for lattice structure ]\n\n");
}

while ( --argc > 0 ) if ( (*++argv)[0] == '-' )
while (*++*argv) switch (**argv) {
     case 'a':                         /* turn on aflg */
          aflg++;
          pflg++;
          tflg++;
          continue;
     case 'p':                         /* turn on pflg */
          pflg++;
          continue;
     case 't':                         /* turn on tflg */
          tflg++;
          continue;
     case 'i':                         /* turn on imsl */
          imsl++;
          continue;
     case 's':                         /* reset sigdgt */
          imsl++;
          i = atoi(argv[1]);
          if ( i > 0 ) {
               sigdgt = i;
               sflg++;
          }
          continue;
     case 'f':                         /* read input file */
          if ( oflg ) error(1);
          if ( (fpin=fopen(argv[1],"r")) == NULL )
               error(2,argv[1]);
          fflg++;
          continue;
     case 'o':                         /* write output file */
          if ( fflg ) error(1);
          if ( (fpout=fopen(argv[1],"w")) == NULL )
               error(2,argv[1]);
          oflg++;
          continue;
     default:
          fpf(stderr,"Specify only '-[aptisfo]'.\n");
          exit(1);
}

if ( !fflg ) {
     loop:;
     fpf(stderr,"Input options (f: file, t: terminal): ");
     switch ( *(s_get(buf,stdin)) ) {
     case 'f':
          fpf(stderr,"Enter input file name: ");
          str = s_get(buf,stdin);
          fflg++;
          if ( (fpin=fopen(str,"r")) == NULL ) error(2,str);
          break;
     case 't':
          fpf(stderr,"Take input from terminal.\n\n");
          fflg = 0;
          break;
     default:
          goto loop;
```

```
        }
    }

    if ( !fflg && !oflg ) {
        fpf(stderr,"Enter output file name: ");
        str = s_get(buf,stdin);
        oflg++;
        if ( (*str == '\n') ||
            (fpout=fopen(str,"w")) == NULL ) {
            fpf(stderr,"Using default file: 'Pwlf.def'\n");
            if ( (fpout=fopen("Pwlf.def","w")) == NULL )
                error(2,"Pwlf.def");
        }
    }

    if ( imsl )
        pf("** [ using IMSL routine ]\n");

    epsilon = 5.0;
    for (i=0; i < sigdgt+2; i++)
        epsilon *= 0.1;

    if ( sflg || pflg ) {
        pf("** [ significant digit is set to %d ]\n",sigdgt);
        pf("** [ epsilon = %8.1e ]\n\n",epsilon);
    }

    if ( fflg )                 /* read title, dim, hyp */
        filread(1);
    else
        ttyread(1);

    /* allocate spaces */
    B = (double *) palloc(dim*dim*sizeof(double));
    C = (double *) palloc(dim*hyp*sizeof(double));
    D = (double *) palloc(dim*hyp*sizeof(double));
    e = (double *) palloc(hyp*1*sizeof(double));
    x = (double *) palloc(dim*1*sizeof(double));
    y = (double *) palloc(dim*1*sizeof(double));

    /* initialize them to 0 */
    i = dim*dim;
    pd = B; while ( i-- > 0 ) *pd++ = 0;
    i = dim*hyp;
    pd = C; while ( i-- > 0 ) *pd++ = 0;
    i = dim*hyp;
    pd = D; while ( i-- > 0 ) *pd++ = 0;
    i = hyp;
    pd = e; while ( i-- > 0 ) *pd++ = 0;
    i = dim;
    pd = x; while ( i-- > 0 ) *pd++ = 0;
    i = dim;
    pd = y; while ( i-- > 0 ) *pd++ = 0;

    if ( fflg )                 /* read B[,],C[,],D[,],e[],y[] */
        filread(2);
    else
        ttyread(2);

    print_eqn();                /* print coefficients */
```

```
        if ( lattice )              /* switch iteration routines */
            latt();
        else
            katz();
}
```

```
/*          - input.c -
**
** input.c :
**          ttyread() -- read input from user tty and write to output file.
**          filread() -- read data from input file.
**          b_pack()  -- pack blanks and tabs in a string.
**          strcmpr() -- compare strings up to n bytes.
**          read_rc() -- read values of 'row' and 'col'.
**          read_B()  -- read entries of B[][].
**          read_C()  -- read entries of C[][].
**          read_D()  -- read entries of D[][].
**          read_e()  -- read entries of e[].
**          read_x()  -- read entries of x[].
**          read_y()  -- read entries of y[].
*/


#include <stdio.h>
#include "cka.h"


/*          - ttyread() -                              - [input.c] -
**
** ttyread() -- read input from user tty and write to output file.
*/


extern  FILE    *fpin, *fpout;          /* defined in main() */
int     line;                           /* input line # */

ttyread (flag)

register int    flag;
{
        register int    i, j;
        register char   *str;

        int     i_get(), tim[2];
        char    *ctime(), *s_get();
        double  d_get(), tmp;

        switch ( flag ) {
        case 1:                 /* read title, dim, hyp */
             time(tim);
             fpf(fpout,"[ %24.24s ]\n\n",ctime(tim));
             pf("\nEnter title: ");
             str = s_get(buf,stdin);
             fpf(fpout,"Title: %s\n",str);
             pf("Enter dimension of matrix B[,]: ");
             dim = i_get(buf,stdin);
             pf("Enter column dimension of matrix D[,]: ");
             hyp = i_get(buf,stdin);
             fpf(fpout,"\ndim\t= %d\nhyp\t= %d\n",dim,hyp);
             break;

        case 2:
             /* read B[,] */
             pf("\nEnter elements of matrix B[,]:");
             for (i=0; i < dim; i++) {
                  pf("\n\trow %d:\n",i+1);
                  for (j=0; j < dim; j++) {
                       pf("\t\tB[%d,%d] = ",i+1,j+1);
                       tmp = B[i*dim+j] = d_get(buf,stdin);
                       fpf(fpout,"B[%d,%d]\t= %16.9e\n",i+1,j+1,tmp);
```

ttyread

```
                }
            }

            /* read C[,] */
            pf("\nEnter elements of matrix C[,]:");
            for (i=0; i < dim; i++) {
                pf("\n\trow %d:\n",i+1);
                for (j=0; j < hyp; j++) {
                    pf("\t\tC[%d,%d] = ",i+1,j+1);
                    tmp = C[i*hyp+j] = d_get(buf,stdin);
                    fpf(fpout,"C[%d,%d]\t= %16.9e\n",i+1,j+1,tmp);
                }
            }

            /* read D[,] */
            pf("\nEnter elements of matrix D[,]:");
            for (i=0; i < dim; i++) {
                pf("\n\trow %d:\n",i+1);
                for (j=0; j < hyp; j++) {
                    pf("\t\tD[%d,%d] = ",i+1,j+1);
                    tmp = D[i*hyp+j] = d_get(buf,stdin);
                    fpf(fpout,"D[%d,%d]\t= %16.9e\n",i+1,j+1,tmp);
                }
            }

            /* read e[,] */
            pf("\nEnter elements of vector e[]:\n");
            for (j=0; j < hyp; j++) {
                pf("\t\te[%d] = ",j+1);
                tmp = e[j] = d_get(buf,stdin);
                fpf(fpout,"e[%d]\t= %16.9e\n",j+1,tmp);
            }

            /* read y[,] */
            pf("\nEnter elements of vector y[]:\n");
            for (i=0; i < dim; i++) {
                pf("\t\ty[%d] = ",i+1);
                tmp = y[i] = d_get(buf,stdin);
                fpf(fpout,"y[%d]\t= %16.9e\n",i+1,tmp);
            }

            /* read x[,] */
            pf("\nEnter initial point x[]:\n");
            for (i=0; i < dim; i++) {
                pf("\t\tx[%d] = ",i+1);
                tmp = x[i] = d_get(buf,stdin);
                fpf(fpout,"x[%d]\t= %16.9e\n",i+1,tmp);
            }
            xflg++;
            break;
        }
}


/*          - filread() -                               - [input.c] -
**
** filread() -- read data from input file.
*/
```

filread

```
filread (flag)

register int        flag;
```

```
{
        register int    dflg=0, hflg=0;
        char            *s_get(), btmp[B_SIZE];
        int             atoi();

        if ( flag == 1 ) {
             while ( s_get(buf,fpin) != NULL ) {
                  line++;
                  b_pack(buf,btmp);
                  if ( strcmpr("dim=",btmp,4) ) {
                       dim = atoi(&btmp[4]);
                       if ( dim <= 0 ) error(4,"filread()");
                       dflg++;
                  }
                  else if ( strcmpr("hyp=",btmp,4) ) {
                       hyp = atoi(&btmp[4]);
                       if ( hyp <= 0 ) error(5,"filread()");
                       hflg++;
                  }
                  else {
                       pf("%s\n",buf);
                  }
                  if ( dflg && hflg ) return;
             }
             error(6,"filread()");
        }

        else {
             while ( s_get(buf,fpin) != NULL ) {
                  line++;
                  b_pack(buf,btmp);
                  switch ( btmp[0] ) {
                       case 'B':
                            read_B(&btmp[1]);
                            break;
                       case 'C':
                            read_C(&btmp[1]);
                            break;
                       case 'D':
                            read_D(&btmp[1]);
                            break;
                       case 'e':
                            read_e(&btmp[1]);
                            break;
                       case 'y':
                            read_y(&btmp[1]);
                            break;
                       case 'x':
                            read_x(&btmp[1]);
                            xflg++;
                            break;
                  }
             }
        }
}


/*           - b_pack() -                        - [input.c] -
**
** b_pack() --  pack blanks and tabs in a string.
*/
```

```
b_pack (buf1, buf2)

register char      buf1[], buf2[];
{
        register char    *p1, *p2;

        p1 = buf1;                        /* start */
        p2 = buf2;

        while ( *p1 != '\0' )             /* packing */
        switch ( *p1 ) {
            case ' ':
            case '\t':
                p1++;
                break;
            default:
                *p2++ = *p1++;
                break;
        }
        *p2 = '\0';
}


/*          - strcmpr() -                          - [input.c] -
**
** strcmpr() -- compare strings up to n bytes.
*/
```

```
strcmpr (str1, str2, n)

register char      *str1, *str2;
register int       n;
{
        while ( n-- > 0 ) {
            if ( *str1++ == *str2++ )
                continue;
            else
                return(0);
        }
        return(1);
}


/*          - read_rc() -                          - [input.c] -
**
** read_rc() -- read values of 'row' and 'col'.
*/
```

```
read_rc (flag, str, row, col)

register int       flag, *row, *col;
register char      *str;
{
        register int      count=0;
        int       atoi();

        if ( *str != '[' && *str != '(' ) error(6,"read_rc()");

        *row = atoi(++str);
        if ( *row <= 0 ) error(6,"read_rc(): row < 0");
        count++;

        if ( flag ) {                   /* search for ',' */
            while ( *str != ',' ) {
                if ( *str == '\0' ) error(6,"read_rc()");
                str++;
                count++;
            }

            *col = atoi(++str);
            if ( *col <= 0 ) error(6,"read_rc(): col < 0");
            count++;
        }

        while ( *str != ']' && *str != ')' ) {
            if ( *str == '\0' ) error(6,"read_rc()");
            str++;
            count++;
        }

        if ( *++str != '=' ) error(6,"read_rc()");
        count++;
        return(++count);
}


/*                                                  - [input.c] -
**
** read_B(), read_C(), read_D(), read_e(), read_x(), read_y()
** -- read values of B[][], C[][], D[][], e[], x[], y[].
**
*/
```

read_B

```
read_B (str)

register char     *str;
{
        int       read_rc(), index, row, col;
        double    atof();

        index = read_rc(1,str,&row,&col);
        if ( row > dim || col > dim ) error(7,"read_B()");

        B[(--row)*dim+(--col)] = atof(str+index);
}
```

read_C

```
read_C (str)

register char     *str;
{
        int       read_rc(), index, row, col;
        double    atof();

        index = read_rc(1,str,&row,&col);
        if ( row > dim || col > hyp ) error(7,"read_C()");

        C[(--row)*hyp+(--col)] = atof(str+index);
}
```

read_D

```
read_D (str)
```

```
register char       *str;
{
            int       read_rc(), index, row, col;
            double    atof();

            index = read_rc(1,str,&row,&col);
            if ( row > dim || col > hyp ) error(7,"read_D()");

            D[(--row)*hyp+(--col)] = atof(str+index);
}
```

read_e

```
read_e (str)

register char       *str;
{
            int       read_rc(), index, col;
            double    atof();

            index = read_rc(0,str,&col);
            if ( col > hyp ) error(7,"read_e()");

            e[--col] = atof(str+index);
}
```

read_x

```
read_x (str)

register char       *str;
{
            int       read_rc(), index, row;
            double    atof();

            index = read_rc(0,str,&row);
            if ( row > dim ) error(7,"read_x()");

            x[--row] = atof(str+index);
}
```

read_y

```
read_y (str)

register char       *str;
{
            int       read_rc(), index, row;
            double    atof();

            index = read_rc(0,str,&row);
            if ( row > dim ) error(7,"read_y()");

            y[--row] = atof(str+index);
}
```

```
/*          - katz.c -
**
** katz() -- this is the main iteration routine, each action
**         falls in clearly defined steps; called by main().
*/


#include <stdio.h>
#include "cka.h"

katz ()                                                        katz
{
        register int      i, j, k, flag, itmp, isave;
        int      Sgn(), nti, n_uniq, *sgntmp, detflg;
        double   d_get(), Abs(), dtmp1, dtmp2;
        REGION   *rgn_init(), *corn(), *rgn, *rgntmp;


  /* STEP 0: initialization */

        init();
        sgntmp = (int *) palloc(hyp*sizeof(int));
        isave = -1;
        rgn = rgn_init(0);

  start:;
        /* get initial point from the user terminal */
        if ( !xflg ) {
            fpf(stderr,"\nEnter initial point:\n");
            flag = 1;                    /* it is an initial guess */
            for (i=0; i < dim; i++) {
                fpf(stderr,"\tx[%d]= ",i+1);
                rgn->px[i] = d_get(buf,stdin);
            }
            pf("\n");
        }
        else {
            for (i=0; i < dim; i++)
                rgn->px[i] = x[i];
            pf("\n** Initial point x0[] = ");
            pv(x,dim,FORMAT1);
            pf("\n");
        }


  /* STEP 1 & 2: check region and compute rgn->nk[] */

  step1:;
        detflg = 0;
        switch ( rgn_check(rgn,flag) ) {
        case 1:                          /* x[] on boundary */
            if ( flag ) {
                pf("** x[] is on a boundary.\n");
                xflg = 0;
                goto start;
            }
            break;

        case 2:                          /* detJ = 0 */
            if ( flag || pflg )
                pf("** x[] is in a region where detJ=0.\n");
            if ( flag )
                goto start;
            else {
                detflg++;                   /* used in step 4 */
                goto step7;
            }
            break;

        case 3:                             /* just hit the solution */
            print_sol(rgn);
            return;
            break;

        default:
            if ( flag ) flag = 0;           /* no longer initial guess */
            break;
        }

/* STEP 3: find intersections */

step3:;
        flag = 0;                           /* intersection flag */
        /* find all intersections */
        for (j=0; j < hyp; j++) {
            pti[j].index = -1;              /* clear */
            if ( j != isave ) {
                dtmp1 = 0;
                for (i=0; i < dim; i++)
                    dtmp1 += D[i*hyp+j]*(rgn->nk[i]);
                if ( Abs(dtmp1) > epsilon ) {
                    pti[j].value = (e[j] - rgn->dx[j])/dtmp1;
                    pti[j].index = j;
                    flag++;                 /* there is an intersection */
                    if ( aflg ) {
                        pf("pti[%d].value=%6.3f\n",j,pti[j].value);
                        pf("pti[%d].index=%d\n",j,pti[j].index);
                    }
                }
            }
        }

/* STEP 4: get valid pti[]'s, put them in qti[] */

        if ( flag ) {
            i = 0;                          /* counter */
            for (j=0; j < hyp; j++)
            if ( (pti[j].index != -1) && (detflg ||
                    (Sgn(pti[j].value) == rgn->detsgn)) )
                qti[i++] = &pti[j];
            nti = i;                        /* total # of valid pti[] */

            if ( aflg ) {
                pf("nti=%d\n",nti);
                for (i=0; i < nti; i++) {
                    pf("qti[%d]->value=%6.3f\n",i,qti[i]->value);
                    pf("qti[%d]->index=%d\n",i,qti[i]->index);
                }
            }
        }

        /* solution is in unbounded region */
        if ( !flag || !nti ) {
            for (i=0; i < dim; i++)
                rgn->px[i] += rgn->nk[i];
            print_sol(rgn);
            return;
        }

    /* STEP 5: find the minimum and determine if it is unique */

        /* sorting */
        n_uniq = 1;                 /* uniqueness flag */
        if ( nti > 1 ) {
            for (itmp=nti/2; itmp > 0; itmp/=2)
            for (i=itmp; i < nti; i++)
            for (j=i-itmp; j >= 0 &&
                (dtmp1=Abs(qti[j]->value))
                >= (dtmp2=Abs(qti[j+itmp]->value));
                j-=itmp) {
                    k = (int)qti[j];
                    qti[j] = qti[j+itmp];
                    qti[j+itmp] = (TI *)k;
            }
            for (i=1; i < nti; i++)
            if (qti[0]->value == qti[i]->value)
                n_uniq++;               /* size of set\Ik */
        }

        if ( aflg ) {
            pf("min: qti[0]->value=%6.3f\n",qti[0]->value);
            pf("min: qti[0]->index=%d\n",qti[0]->index);
        }

        if ( qti[0]->value >= 1.0 ) {       /* solution is reached */
            for (i=0; i < dim; i++)
                rgn->px[i] += rgn->nk[i];
            print_sol(rgn);
            return;
        }

    /* STEP 6: as minimum is unique, find next iteration point x[] */

        for (i=0; i < dim; i++)
            x[i] = rgn->px[i] + qti[0]->value * rgn->nk[i];

        if ( n_uniq > 1 ) goto step8;       /* corner problem */

        /* setup sign sequence vector for next region */
        for (j=0; j < hyp; j++)
            sgntmp[j] = rgn->sgnsq[j];

        isave = qti[0]->index;
        sgntmp[isave] = -sgntmp[isave];

        /* this region is over, get next region to iterate */
        putrgn(RGNQ,rgn);                   /* put region to RGNQ */
        flag = 0;
        rgn  = rgn_init(0);
        for (i=0; i < dim; i++)             /* setup rgn->px[] */
            rgn->px[i] = x[i];
        for (j=0; j < hyp; j++)             /* setup rgn->sgnsq[] */
            rgn->sgnsq[j] = sgntmp[j];
        goto step1;
```

...katz

```
  /* STEP 7: Jacobian matrix is singular */
  step7:;
          zerodet(rgn);
          goto step3;


  /* STEP 8: corner problem */
  step8:;
          rgntmp = corn(x,n_uniq,rgn);
          putrgn(RGNQ,rgn);
          rgn = rgntmp;
          goto step3;

    end:;
          pf("\n??-> Program ends abnormally <-??\n");
}
```

```
/*          - lattice.c -
**
** latt() -- this is the main iteration routine which has
**          been optimized for lattice structure, each action
**          falls in clearly defined steps; called by main().
*/


#include <stdio.h>
#include "cka.h"

latt ()
{
        register int    i, j, k, flag, itmp, isave;
        int     Sgn(), dim2=2*dim, nti, n_uniq, *sgntmp, *bdtmp, detflg;
        double  d_get(), Abs(), dtmp1, dtmp2, *xtmp;
        REGION  *rgn_init(), *corn(), *rgn, *rgntmp;


   /* STEP 0: initialization */

        init();
        sgntmp = (int *)  palloc(hyp*sizeof(int));
        bdtmp  = (int *)  palloc(hyp*sizeof(int));
        xtmp = (double *) palloc(dim*sizeof(double));
        isave = -1;
        rgn = rgn_init(0);

   start:;
        /* get initial point from the user terminal */
        if ( !xflg ) {
            fpf(stderr,"\nEnter initial point:\n");
            flag = 1;                    /* it is an initial guess */
            for (i=0; i < dim; i++) {
                fpf(stderr,"\tx[%d]= ",i+1);
                rgn->px[i] = d_get(buf,stdin);
            }
            pf("\n");
        }
        else {
            for (i=0; i < dim; i++)
                rgn->px[i] = x[i];
            pf("\n** Initial point x0[] = ");
            pv(x,dim,FORMAT1);
            pf("\n");
        }


   /* STEP 1 & 2: check region and compute rgn->nk[] */

   step1:;
        detflg = 0;
        switch ( rgn_check(rgn,flag) ) {
        case 1:                          /* x[] on boundary */
            if ( flag ) {
                pf("** x[] is on a boundary.\n");
                xflg = 0;
                goto start;
            }
            break;
        case 2:                         /* det=0 */
            if ( flag || pflg )
                pf("** x[] is in a region where detJ=0.\n");
            if ( flag )
                goto start;
            else {
                detflg++;               /* used in step 4 */
                goto step7;
            }
            break;

        case 3:                         /* just hit the solution */
            print_sol(rgn);
            return;
            break;

        default:
            if ( flag ) flag = 0;       /* no longer initial guess */
            break;
        }

/* STEP 3: find intersections */

step3:;
        i = 0;                          /* counter */
        for (j=0; j < hyp; j++) {
            pti[i].index = -1;          /* clear */
            if ( j != isave ) {
                k = dcolumn[j];
                if ( (rgn->bdry[j]) && (Abs(rgn->nk[k])>epsilon) ) {
                    pti[i].value = (e[j] - rgn->px[k])/rgn->nk[k];
                    pti[i++].index = j;
                    if ( aflg ) {
                        pf ("pti[%d].value=%6.3f\n",i-1,pti[i-1].value);
                        pf ("pti[%d].index=%d\n",i-1,pti[i-1].index);
                    }
                }
            }
        }
        nti = i;                        /* total # of intersections */

/* STEP 4: get valid pti[]'s, put them in qti[] */

        if ( nti ) {                    /* if # of intersections > 0 */
            j = 0;                      /* counter */
            for (i=0; i < nti; i++)
            if ( (pti[i].index != -1) && (detflg ||
                    (Sgn(pti[i].value) == rgn->detsgn)) )
                qti[j++] = &pti[i];
            nti = j;                    /* total # of valid pti[] */

            if ( aflg ) {
                pf("nti=%d\n",nti);
                for (i=0; i < nti; i++) {
                    pf("qti[%d]->value=%6.3f\n",i,qti[i]->value);
                    pf("qti[%d]->index=%d\n",i,qti[i]->index);
                }
            }
        }
        /* solution is in unbounded region */
        if ( !nti ) {
            for (i=0; i < dim; i++)
                rgn->px[i] += rgn->nk[i];
            print_sol(rgn);
            return;
        }

/* STEP 5: find the minimum and determine if it is unique */

        /* sorting */
        n_uniq = 1;                /* uniqueness flag */
        if ( nti > 1 ) {
            for (itmp=nti/2; itmp > 0; itmp/=2)
            for (i=itmp; i < nti; i++)
            for (j=i-itmp; j >= 0 &&
                (dtmp1=Abs(qti[j]->value))
                >= (dtmp2=Abs(qti[j+itmp]->value));
                j-=itmp) {
                    k = (int)qti[j];
                    qti[j] = qti[j+itmp];
                    qti[j+itmp] = (TI *)k;
            }
            for (i=1; i < nti; i++)
            if (qti[0]->value == qti[i]->value)
                n_uniq++;         /* size of set Ik */
        }

        if ( aflg ) {
            pf("min: qti[0]->value=%6.3f\n",qti[0]->value);
            pf("min: qti[0]->index=%d\n",qti[0]->index);
        }

        if ( qti[0]->value >= 1.0 ) {        /* solution is reached */
            for (i=0; i < dim; i++)
                rgn->px[i] += rgn->nk[i];
            print_sol(rgn);
            return;
        }

/* STEP 6: as minimum is unique, find next iteration point x[] */

        /* must save x[] since rgn_check() uses it */
        for (i=0; i < dim; i++) {
            x[i] = rgn->px[i];
            xtmp[i] = x[i] + qti[0]->value * rgn->nk[i];
        }

        if ( n_uniq > 1 ) goto step8;       /* corner problem */

        /* save rgn->sgnsq[] and rgn->bdry[] for next region */
        for (j=0; j < hyp; j++) {
            sgntmp[j] = rgn->sgnsq[j];
            bdtmp[j] = rgn->bdry[j];
        }

        /* setup sign sequence vector for next region */
        isave = qti[0]->index;
        sgntmp[isave] = -sgntmp[isave];
        /* this region is over, get next region to iterate */
        putrgn(RGNQ,rgn);                     /* put region to RGNQ */
        flag = 0;                             /* not initial guess */
        rgn  = rgn_init(0);

        /* setup rgn->px[] */
        for (i=0; i < dim; i++)
             rgn->px[i] = xtmp[i];

        /* setup rgn->sgnsq[] and rgn->bdry[] */
        for (j=0; j < hyp; j++) {
             rgn->sgnsq[j] = sgntmp[j];
             rgn->bdry[j] = bdtmp[j];
        }
        goto step1;


  /* STEP 7: Jacobian matrix is singular */
  step7:;
        zerodet(rgn);
        goto step3;


  /* STEP 8: corner problem */
  step8:;
        rgntmp = corn(xtmp,n_uniq,rgn);
        putrgn(RGNQ,rgn);
        rgn = rgntmp;
        goto step3;

    end:;
        pf("\n??-> Program ends abnormally <-??\n");
}
```

```
/*          - corner.c -
**
** corner.c :
**       corn()     -- deals with corner problem;
**       zerodet() -- deals with det J = 0 case.
*/


#include <stdio.h>
#include "cka.h"


/*          - corn() -                              - [corner.c] -
**
** corn() -- this routine handles the corner problem, called
**        by katz() and latt().
*/


REGION   *corn (cx, n_uniq, rgn_in)

double    *cx;
register int        n_uniq;
register REGION   *rgn_in;
{
        register int       i, j, k, n;
        int       nomatch, mask, nti, found, sgnmatch, flag, detflg;
        double    Abs(), dtmp1, dtmp2, *xtmp;
        CORNER  *cnrinit(), *getcnr(), *cnr;
        REGION   *rgn_init(), *getrgn(), *rgntmp, *rgntmp2;

        if ( aflg ) pf("\n** hit corner **\n");


   /* STEP 8 & 9 */

        n = CNRQ->n;

        /* check if the corner point is new */
        if ( n != 0 )   for (j=0; j < n; j++) {
             cnr = getcnr(CNRQ);
             nomatch = 0;
             for (i=0; i < dim; i++) if ( cx[i] != cnr->cx[i] ) {
                  nomatch++;
                  break;
             }
             if ( nomatch )
                  putcnr(CNRQ,cnr);
             else
                  break;
        }


   /* STEP 10 & 11 */

        if ( !n || nomatch ) {
             cnr = cnrinit();                  /* new corner */

             /* store the corner point x[] */
             for (i=0; i < dim; i++)
                  cnr->cx[i] = cx[i];
```

```
                /* save indices */
                cnr->index = (int *) palloc(n_uniq*sizeof(int));
                for (k=0; k < n_uniq; k++)
                    cnr->index[k] = qti[k]->index;

                /* generate (2**n_uniq-1) neighborhood regions */
                nti = 1 << n_uniq;              /* nti = 2**n_uniq */
                for (i=1; i < nti; i++) {
                    rgntmp = rgn_init(-i);      /* rgntmp->id = -i */
                    for (j=0; j < hyp; j++)     /* copy */
                        rgntmp->sgnsq[j] = rgn_in->sgnsq[j];

                    for (mask=1,k=0; k < n_uniq; mask<<=1,k++)
                    if ( mask & i )
                        rgntmp->sgnsq[cnr->index[k]]
                        = -rgn_in->sgnsq[cnr->index[k]];

                    for (j=0; j < dim; j++)
                        rgntmp->px[j] = cx[j];

                    /* compute jcbn[,] and nk[] but don't compute sgnsq[] */
                    if ( rgn_check(rgntmp,0) == 2) zerodet(rgntmp);

                    putrgn(cnr->CNRRGN,rgntmp);
                }
            }

    /* STEP 12 */

            sgnmatch = 0;
    step12:;
            detflg = 0;
            n = cnr->CNRRGN->n;
            if ( !cnr->sgnflg ) {           /* cnr->sgnsq[] is not set */
                found = 0;
                for (k=0; k < n; k++) {     /* det > 0 */
                    rgntmp = getrgn(cnr->CNRRGN);
                    if ( rgntmp->detsgn > 0 ) {
                        found++;
                        break;
                    }
                    else
                        putrgn(cnr->CNRRGN,rgntmp);
                }
                if ( !found )               /* det = 0 */
                for (k=0; k < n; k++) {
                    rgntmp = getrgn(cnr->CNRRGN);
                    if ( rgntmp->detsgn = 0 ) {
                        found++;
                        detflg++;
                        break;
                    }
                    else
                        putrgn(cnr->CNRRGN,rgntmp);
                }
                if ( !found )               /* det < 0 */
                for (k=0; k < n; k++) {
                    rgntmp = getrgn(cnr->CNRRGN);
                    if ( rgntmp->detsgn < 0 ) {
                        found++;
                        break;
                    }
                    else
                putrgn(cnr->CNRRGN,rgntmp);
            }
        }
        else {                              /* match the sgnsq[] */
            for (k=0; k < n; k++) {
                rgntmp = getrgn(cnr->CNRRGN);
                nomatch = 0;
                for (j=0; j < hyp; j++)
                if ( rgntmp->sgnsq[j] != cnr->sgnsq[j]) {
                    nomatch++;
                    break;
                }
                if ( nomatch )
                    putrgn(cnr->CNRRGN,rgntmp);
                else
                    break;
            }
            sgnmatch++;
        }

    /* STEP 13: repeat STEP 2 & 3 to find intersections */

        flag = 0;                           /* intersection flag */
        nomatch = 1;                        /* assume j != cnr->index[k] */
        for (j=0; j < hyp; j++) {           /* find all intersections */
            pti[j].index = -1;              /* clear */
            for (k=0; k < n_uniq; k++)
                if ( j == cnr->index[k] ) nomatch = 0;
            if ( nomatch ) {
                dtmp1 = 0;
                for (i=0; i < dim; i++)
                    dtmp1 += D[i*hyp+j]*(rgntmp->nk[i]);
                if ( Abs(dtmp1) > epsilon ) {
                    pti[j].value = (e[j] - rgntmp->dx[j])/dtmp1;
                    pti[j].index = j;
                    flag++;                 /* there is an intersection */
                    if ( aflg ) {
                        pf("pti[%d].value=%6.3f\n",j,pti[j].value);
                        pf("pti[%d].index=%d\n",j,pti[j].index);
                    }
                }
            }
        }

        /* get valid pti[]'s, put them in qti[] */
        if ( flag ) {
            i = 0;                          /* counter */
            for (j=0; j < hyp; j++)
            if ( (pti[j].index != -1) && (detflg ||
                    (Sgn(pti[j].value) == rgntmp->detsgn)) )
                qti[i++] = &pti[j];
            nti = i;                        /* total # of valid pti[] */

            if ( aflg ) {
                pf("nti=%d\n",nti);
                for (i=0; i < nti; i++) {
                    pf("qti[%d]->value=%6.3f\n",i,qti[i]->value);
                    pf("qti[%d]->index=%d\n",i,qti[i]->index);
                }
            }
        }
        if ( nti > 1 )                          /* find the minimum */
        /* use mask as an arbitrary integer buffer */
        for (mask=nti/2; mask > 0; mask/=2)
        for (i=mask; i < nti; i++)
        for (j=i-mask; j >= 0 &&
          (dtmp1=Abs(qti[j]->value)) >= (dtmp2=Abs(qti[j+mask]->value));
          j-=mask) {
            k = (int)qti[j];
            qti[j] = qti[j+mask];
            qti[j+mask] = (TI *)k;
        }

        if ( aflg ) {
            pf("min: qti[0]->value=%6.3f\n",qti[0]->value);
            pf("min: qti[0]->index=%d\n",qti[0]->index);
        }

        /* find point x[] */
        xtmp = (double *) palloc(dim*sizeof(double));
        for (i=0; i < dim; i++)
            xtmp[i] = rgntmp->px[i] + 0.5 * qti[0]->value * rgntmp->nk[i];

        /* determine sgnsq[] and put it in cnr->sgnsq[] */
        if ( !cnr->sgnflg ) {
            rgntmp2 = rgn_init(-999);
            for (i=0; i < dim; i++)
                rgntmp2->px[i] = xtmp[i];
            compute_jy(rgntmp2,-1);         /* compute sgnsq[] */
            for (j=0; j < hyp; j++)
                cnr->sgnsq[j] = rgntmp2->sgnsq[j];
            cnr->sgnflg++;                  /* set flag */
        }

        if ( !sgnmatch )
            goto step12;
        else {
            for (i=0; i < dim; i++)
                rgntmp->px[i] = xtmp[i];/* setup rgntmp->px[] */
            rgntmp->id = rgn_count++;   /* reset rgntmp->id */
            return(rgntmp);
        }
}


/*          - zerodet() -                           - [corner.c] -
**
** zerodet() -- deals with det J[,] = 0 case, called by katz(),
**          latt() and corn().
*/


zerodet (rgn)                                                   zerodet

register REGION    *rgn;
{
        register int        i;
        double  d_get();

        pf("\n** Jacobian matrix is singular:");
        pm(rgn->jcbn,dim,dim,FORMAT1);
        fpf(stderr,"\nEnter a non-zero vector nk[]");
        fpf(stderr," so that J[,]nk[] = 0 :\n");
        for (i=0; i < dim; i++) {
                fpf(stderr,"\tnk[%d] = ",i+1);
                rgn->nk[i] = d_get(buf,stdin);
            }
            pf("\nnk[]:\t");
            pv(rgn->nk,dim,FORMAT1);
            pf("\n");
}
```

```
/*          - init.c -
**
** init.c :
**          init()            -- call rest routines to initialize;
**          normalize_D()     -- normalize D[,] & e[];
**          parallel_groups() -- find parallel hyperplane groups;
**          cnrinit()         -- initialize CORNER and CORNER_QUEUE.
*/

#include "cka.h"

REGION_QUEUE    *RGNQ;          /* a queue of regions in the
                                   iteration */

CORNER_QUEUE    *CNRQ;          /* a queue of corners in the
                                   iteration */

BETA            *axis;          /* axis[i] is an array of the
                                   structure BETA */

int             *beta_count;    /* beta_count[j] is the # of parallel
                                   hyperplanes with normal direction
                                   being the j-th axis */

int             *dcolumn;       /* dcolumn[j] contains the index of
                                   the nonzero entry in the j-th column
                                   of D[][] */

double          *wk;            /* wk[] is the working area for IMSL
                                   routines */

double          *jcbntmp;       /* jcbntmp[][] is the working area for
                                   computing inverse of Jacobian when
                                   using IMSL routines */


/*          - init() -                          - [init.c] -
**
** init() -- takes care of all necesary initializations;
**           called by lattice().
*/

init ()                                                          init
{
        register int      i, j;

        /* allocate spaces */
        RGNQ = (REGION_QUEUE *) palloc(sizeof(REGION_QUEUE));
        RGNQ->head = RGNQ->tail = RNIL;
        RGNQ->n = 0;

        CNRQ = (CORNER_QUEUE *) palloc(sizeof(CORNER_QUEUE));
        CNRQ->head = CNRQ->tail = CNIL;
        CNRQ->n = 0;

        if ( imsl ) {
            wk      = (double *) palloc(dim*(dim+3)*sizeof(double));
            jcbntmp = (double *) palloc(dim*dim*sizeof(double));
        }

        if ( lattice ) {
```

```
        /*
        ** each region has maximum 2 boundary hyperplanes
        ** in each axis direction.
        */
        pti = (TI *)  palloc(2*dim*sizeof(TI));
        qti = (TI **) palloc(2*dim*sizeof(TI *));

        axis = (BETA *) palloc(dim*sizeof(BETA));
        dcolumn = (int *) palloc(hyp*sizeof(int));
        beta_count = (int *) palloc(dim*sizeof(int));

        for (i=0; i < dim; i++)
             beta_count[i] = 0;

        /* normalize D[,] and e[] */
        normalize_D();

        for (i=0; i < dim; i++) if ( beta_count[i] > 0 )
             axis[i].beta = (int *) palloc(beta_count[i]*sizeof(int));

        /* find parallel hyperplane groups */
        parallel_groups();

        if ( aflg ) {
             pf("\n");
             for (i=0; i < dim; i++) {
                  pf("beta_count[%d] = %d\n",i,beta_count[i]);
                  pf("axis[%d].beta[] = ",i);
                  for (j=0; j < beta_count[i]; j++)
                       pf("%d ",axis[i].beta[j]);
                  pf("\n");
             }
        }
    }

    else {
        /* maximum number of boundaries for each region = hyp */
        pti = (TI *)  palloc(hyp*sizeof(TI));
        qti = (TI **) palloc(hyp*sizeof(TI *));
    }

    /* initialize region counting */
    rgn_count = 0;
}


/*         - normalize_D() -                    - [init.c] -
**
** normalize_D() -- for hybrid representation, each column of D[,]
**        should contain one and only one nonzero entry; this routine
**        checks D[,] and normalizes D[,] and e[] by deviding e[] the
**        corresponding nonzero entry in the columns of D[,] and set
**        that entry to 1; called by init().
*/
```

normalize_D

```
normalize_D ()
{
        register int       i, j, flag;
        register double    *dtmp;
        for (j=0; j < hyp; j++) {           /* scan D by column */
            flag = 0;

            for (i=0; i < dim; i++) {       /* for each row in a column */
                dtmp = &D[i*hyp+j];

                if ( *dtmp != 0 ) {         /* hit nonzero entry */
                    if ( !flag ) {          /* the only nonzero */
                        flag++;

                        /* normalizing */
                        if ( *dtmp != 1.0 ) {
                            e[j] /= *dtmp;
                            *dtmp = 1.0;
                        }

                        dcolumn[j] = i;  /* the i-th row in the j-th
                                            column is nonzero */
                        beta_count[i]++;/* # of hyperplanes in the
                                            i-th parallel group */
                    }
                    else                    /* >= 2 nonzero entries */
                        error(8);
                }
            }

            /* all entries in column j are 0 */
            if ( !flag ) error(8);
        }
}


/*          - parallel_groups() -                       - [init.c] -
**
** parallel_groups() -- identical columns in D[,] represents
**          parallel hyperplanes; this routine groups those columns
**          in sets and sort (using SHELL sort) the corresponding
**          'beta (i.e. e[j])' in ascending order; called by init().
*/
```

```
parallel_groups ()
{
        register int        i, j, k, index, flag, gap;
        int         count, *tested;

        tested = (int *) palloc(hyp*sizeof(int));
        for (i=0; i < hyp; i++) tested[i] = 0;

        i = 0;
        count = 0;
        while ( count++ < hyp ) {
            index = 0;
            flag  = 0;
            k = -1;
            axis[dcolumn[i]].beta[index++] = i;
            tested[i]++;

            /*
            ** find parallel columns by searching for the same
            ** dcolumn[j].
            */
            for (j=i+1; j < hyp; j++)
            if ( !tested[j] ) {              /* if not tested */
                /* if the j-th column is parallel to the i-th */
                if ( dcolumn[j] == dcolumn[i] ) {
                    axis[dcolumn[j]].beta[index++] = j;
                    tested[j]++;
                    count++;
                }
                /* get the 1st nonparallel untested column */
                else if ( !flag ) {
                    flag++;
                    k = j;
                }
            }

            if ( k == -1 )
                break;                       /* all are parallel */
            else
                i = k;                       /* k = 1st nonparallel column */
        }

        /* SHELL sorting so that beta is in increasing order */
        for (k=0; k < dim; k++) if ( (count=beta_count[k]) > 0 )
        for (gap = count/2; gap > 0; gap /= 2) {
            for (i=gap; i < count; i++)
            for (j = i-gap; j >= 0 &&
                        e[axis[k].beta[j]] >= e[axis[k].beta[j+gap]];
                 j -= gap) {
                if ( e[axis[k].beta[j]] == e[axis[k].beta[j+gap]] )
                    error(9);
                else {
                    index = axis[k].beta[j];
                    axis[k].beta[j] = axis[k].beta[j+gap];
                    axis[k].beta[j+gap] = index;
                }
            }
        }
}


/*          - cnrinit() -                           - [init.c] -
**
** cnrinit() -- allocate spaces for structure CORNER and
**          CORNER_QUEUE.
*/


register CORNER   *cnrinit ()
{
        register CORNER *cnr;

        cnr = (CORNER *) palloc(sizeof(CORNER));

        cnr->cx = (double *) palloc(dim*sizeof(double));
        cnr->sgnsq = (int *) palloc(hyp*sizeof(int));
        cnr->sgnflg = 0;

        cnr->CNRRGN = (REGION_QUEUE *) palloc(sizeof(REGION_QUEUE));
        cnr->CNRRGN->head = cnr->CNRRGN->tail = RNIL;
        cnr->CNRRGN->n = 0;

        return(cnr);
}
```

```
/*          - region.c -
**
** region.c :
**         rgn_init()    -- allocates space for REGION,
**         rgn_check()   -- checks valid region,
**         compute_jy()  -- computes Jacobian & y[]=pwlf(x[]).
*/


#include "cka.h"

int      rgn_count;                /* count total number of regions
                                      encountered during the iteration */

int      ier;                      /* error code for IMSL routines */


/*          - rgn_init() -                            - [region.c] -
**
** rgn_init() -- allocate space for structure REGION.
*/


register REGION   *rgn_init (flag)
register int      flag;
{
        register int      j;
        register REGION *rgn;

        rgn        = (REGION *) palloc(sizeof(REGION));

        if ( !flag )                /* not associated with corner */
            rgn->id = rgn_count++;
        else
            rgn->id = flag;

        rgn->sgnsq = (int *)     palloc(hyp*sizeof(int));
        rgn->px    = (double *)  palloc(dim*sizeof(double));
        rgn->py    = (double *)  palloc(dim*sizeof(double));
        rgn->nk    = (double *)  palloc(dim*sizeof(double));
        rgn->jcbn  = (double *) palloc(dim*dim*sizeof(double));

        if ( lattice ) {
            rgn->bdry = (int *)  palloc(hyp*sizeof(int));
            for (j=0; j < hyp; j++) rgn->bdry[j] = 0;
        }
        else
            rgn->dx = (double *) palloc(hyp*sizeof(double));

        return(rgn);
}


/*          - rgn_check() -                           - [region.c] -
**
** rgn_check() -- check valid region:
**          1. is rgn->px[] on any boundary ?
**          2. is the Jacobian singular ?
**          3. is the determinant positive ?
**          4. compute the following quantities:
**                 rgn->sgnsq[],
**                 rgn->nk[],
**              rgn->dx[],
**              rgn->py[],
**              rgn->jcbn[],
**              rgn->detsgn.
*/
```

```
rgn_check (rgn, flag)

register REGION     *rgn;
register int        flag;                   /* flag==1 : initial point */
{
        register int        i, j, k, m, n;
        int         Sgn();
        double      Abs(), tmp1, tmp2, tmp3;

        /* compute rgn->sgnsq[], rgn->jcbn[][] and rgn->py[] */
        compute_jy(rgn,flag);

        /* check if the point rgn->px[] is on a boundary */
        if ( flag ) for (j=0; j < hyp; j++)
            if ( rgn->sgnsq[j] == 0 ) return(1);

        /* compute y[] - f(x[]) and store the result in rgn->py[] */
        k = 0;
        for (i=0; i < dim; i++) {
            rgn->py[i] = y[i] - rgn->py[i];
            if ( Abs(rgn->py[i]) > epsilon ) k++;
        }
        if ( !k ) return(3);                /* solution is found ! */

        /* determine boundary hyperplanes */
        if ( lattice ) {
            for (i=0; i < dim; i++) if ( beta_count[i] > 0 ) {

                m = beta_count[i]-1;
                j = axis[i].beta[0];
                k = axis[i].beta[m];

                if ( flag ) {               /* initial point */
                    if ( rgn->px[i] < e[j] )        /* leftmost */
                        rgn->bdry[j]++;
                    else if ( rgn->px[i] > e[k] )/* rightmost */
                        rgn->bdry[k]++;
                    else for (n=0; n < m; n++) {/* scan */
                        j = axis[i].beta[n];
                        k = axis[i].beta[n+1];
                        if ( e[j] < rgn->px[i] && rgn->px[i] < e[k] ) {
                            rgn->bdry[j]++;
                            rgn->bdry[k]++;
                        }
                    }
                }

                else for (n=0; n < beta_count[i]; n++) {
                    m = axis[i].beta[n];
                    if ( m == qti[0]->index ) {
                        rgn->bdry[m]++;
                        if ( x[i] < e[m] ) {
                            if ( m < k ) rgn->bdry[m+1]++;
                            if ( m > j ) rgn->bdry[m-1] = 0;
                        }
                        else if ( x[i] > e[m] ) {
                            if ( m < k ) rgn->bdry[m+1] = 0;
```

```
                                    if ( m > j ) rgn->bdry[m-1]++;
                                 }
                              }
                           }
                        }
                     }

                     /* compute rgn->nk[] */
                     if ( !imsl ) {
                         if ( lineqn(rgn->jcbn,rgn->nk,rgn->py,dim,0,&tmp1) != 0) {
                             if ( pflg ) error(10,"rgn_check()",rgn);
                             return(2);                  /* detJ = 0 */
                         }
                     }
                     else {
                         /*
                         ** using IMSL routines: see leqt2f.f in IMSL for details
                         ** on the calling sequence of ludatf(), luelmf() & lureff().
                         */
                         transp(rgn->jcbn,jcbntmp,dim,dim);
                         i = dim*dim;
                         for (j=0; j < i; j++) wk[j] = jcbntmp[j];
                         j = i + dim;
                         k = j + dim;
                         ludatf_(&wk[0],jcbntmp,&dim,&dim,&sigdgt,
                             &tmp1,&tmp2,&wk[i],&wk[j],&tmp3,&ier);
                         if ( ier > 128 ) {              /* jcbn singular */
                             if ( pflg ) error(10,"rgn_check()",rgn);
                             return(2);
                         }
                         else {
                             luelmf_(jcbntmp,rgn->py,&wk[i],&dim,&dim,&wk[k]);
                             lureff_(&wk[0],rgn->py,jcbntmp,&wk[i],&dim,&dim,
                                 &wk[k],&sigdgt,&wk[j],&wk[j],&ier);
                             for (i=0; i < dim; i++)
                                 rgn->nk[i] = wk[i+k];
                         }
                     }

                     /* find sign of det J */
                     rgn->detsgn = Sgn(tmp1);

                     if ( aflg ) print_rgn(rgn);

                     return(0);
}


/*              - compute_jy() -                              - [region.c] -
**
** compute_jy() — compute
**                          (1) rgn->sgnsq[],
**                          (2) rgn->jcbn[][] and
**                          (3) rgn->py[].
*/
```

compute_jy

```
compute_jy (rgn, flag)

register REGION    *rgn;
register int        flag;                       /* =1 : initial point,
                                                      compute (1), (2) & (3);
                                                   =0 : compute (2) & (3);
                                                   =-1: compute (1) only. */

{
          register int      i, j, k, m, n, flg1;
          double   tmp;

          /* compute rgn->dx[] and rgn->sgnsq[] */
          if ( lattice ) {
               /*
               ** note that since columns of D[][] are unit vectors,
               ** we only need one component of rgn->px[] at a time.
               */
               if ( flag ) for (j=0; j < hyp; j++)
                    rgn->sgnsq[j] = Sgn(rgn->px[dcolumn[j]]-e[j]);
          }
          else {
               for (j=0; j < hyp; j++) {
                    rgn->dx[j] = 0;
                    for (i=0; i < dim; i++)
                         rgn->dx[j] += D[i*hyp+j]*(rgn->px[i]);
                    if ( flag )
                         rgn->sgnsq[j] = Sgn(rgn->dx[j]-e[j]);
               }
          }

          /* compute rgn->jcbn[] and rgn->py[] */
          if ( flag >= 0 ) {
               for (i=0; i < dim; i++) {
                    m = i*dim;
                    n = i*hyp;
                    rgn->py[i] = 0;
                    flg1 = 0;

                    for (j=0; j < dim; j++) {
                         rgn->jcbn[m+j] = B[m+j];
                         rgn->py[i] += B[m+j] * rgn->px[j];

                         for (k=0; k < hyp; k++) {
                              tmp = C[n+k] * rgn->sgnsq[k];
                              rgn->jcbn[m+j] += tmp * D[j*hyp+k];
                              /*
                              ** the following line is excuted only
                              ** once for each i.
                              */
                              if ( !flg1 ) {
                                   if ( lattice )
                                        rgn->py[i] += tmp
                                                  * (rgn->px[dcolumn[k]]-e[k]);
                                   else
                                        rgn->py[i] += tmp * (rgn->dx[k]-e[k]);
                              }
                         }
                         flg1++;
                    }
               }
          }
}
```

```
/*          - print.c -
**
** This module contains 4 routines:
**        print_eqn() -- print the coefficients of the pwl function;
**        print_sol() -- print solution;
**        print_rgn() -- print all information in the structure REGION.
*/


#include "cka.h"


/*          - print_eqn() -                    - [print.c] -
**
** print_eqn() -- print coefficients of the pwlf(.).
*/

print_eqn ()                                                    print_eqn
{
          pf("\nCoefficients of the canonical equation:");

          pf("\nB[,]:");
          pm(B,dim,dim,FORMAT1);

          pf("C[,]:");
          pm(C,dim,hyp,FORMAT1);

          pf("D[,]:");
          pm(D,dim,hyp,"%2.0f ");

          pf("\ne[]:\t");
          pv(e,hyp,FORMAT1);

          pf("\n\ny[]:\t");
          pv(y,dim,FORMAT1);

          pf("\n");
}


/*          - print_sol() -                              - [print.c] -
**
**  print_sol() -- print solution.
*/

print_sol (rgn)                                                 print_sol

register REGION    *rgn;
{
          register int        i;
          register REGION     *rgntmp;
          REGION    *getrgn();

          pf("\n** Solution x[] = ");           /* print solution */
          pv(rgn->px,dim,FORMAT2);

          if ( tflg ) {                          /* test solution */
              compute_jy(rgn,1);
              for (i=0; i < dim; i++)
                  rgn->py[i] = y[i] - rgn->py[i];
```

```
        pf("\n=> f(x[]) - y[] = ");
        pv(rgn->py,dim,FORMAT2);
    }

    i = RGNQ->n;

    /* print out all regions where the solution curve passes by */
    if ( pflg && i > 0 ) {
        pf("\n\n** Regions in iteration :\n");
        while ( RGNQ->n != 0 ) {
            rgntmp = getrgn(RGNQ);
            print_rgn(rgntmp);
        }
    }

    if ( pflg ) {
        pf("\n\n** Last region in the iteration :\n");
        print_rgn(rgn);
    }

    pf("\n\n** Number of regions traveled: ");
    if ( i > 0)
        pf("%d.\n",i+1);
    else
        pf("1.\n");
}


/*          - print_rgn() -                              - [print.c] -
**
** print_rgn() -- print all information in the structure REGION.
*/
```

```
print_rgn (rgn)

register REGION   *rgn;
{
        register int      k;

        pf("\nrgn->id: %d",rgn->id);

        pf("\nsign sequence:   ");
        for (k=0; k < hyp; k++)
            pf("%2d ",rgn->sgnsq[k]);

        pf("\npoint x[]: ");
        pv(rgn->px,dim,FORMAT1);

        pf("\npoint y[]: ");
        pv(rgn->py,dim,FORMAT1);

        pf("\nvector nk[]: ");
        pv(rgn->nk,dim,FORMAT1);

        pf("\nJacobian matrix:");
            pm(rgn->jcbn,dim,dim,FORMAT1);

        pf("\nSign of determinant: %2d",rgn->detsgn);

        if ( lattice ) {
            pf("\nboundaries : ");
            for (k=0; k < hyp; k++)
```

```
                        pf("%2d ",rgn->bdry[k]);
            }
            pf("\n");
}
```

```
/*          - queue.c -
**
** This module contains routines: putrgn(), getrgn().
*/


#include "cka.h"


/*          - putrgn() -                              - [queue.c] -
**
** putrgn() -- places REGION at the end of REGION_QUEUE, it always
**          assumes queue != NIL.
*/


putrgn (rgnq, rgn)

register REGION_QUEUE    *rgnq;
register REGION          *rgn;
{
          rgn->link = RNIL;

          /* queue was initially empty */
          if (rgnq->head == RNIL) {
               rgnq->head = rgn;
               rgnq->tail = rgn;
          }

          /* queue was not empty, append at the end */
          else {
               rgnq->tail->link = rgn;
               rgnq->tail = rgn;
          }

          rgnq->n++;
          return;
}


/*          - getrgn() -                              - [queue.c] -
**
** getrgn() -- gets one REGION from the front of REGION_QUEUE and
**          returns a pointer to that REGION, it returns NIL if
**          the REGION_QUEUE is empty.
*/


REGION   *getrgn (rgnq)

register REGION_QUEUE    *rgnq;
{
          REGION  *rgn;

          /* if queue is empty, return NIL */
          rgn = RNIL;

          /* if queue is not empty, get one from the front */
          if (rgnq->head != RNIL) {
               rgn = rgnq->head;
               rgnq->head = rgnq->head->link;
               rgnq->n--;
          }
```

```
        return(rgn);
}


/*          - putcnr() -                              - [queue.c] -
**
** putcnr() -- places CORNER at the end of CORNER_QUEUE, it always
**          assumes queue != NIL.
*/
```

putcnr

```
putcnr (cnrq, cnr)

register CORNER_QUEUE    *cnrq;
register CORNER          *cnr;
{
        cnr->link = CNIL;

        /* queue was initially empty */
        if (cnrq->head == CNIL) {
            cnrq->head = cnr;
            cnrq->tail = cnr;
        }

        /* queue was not empty, append at the end */
        else {
            cnrq->tail->link = cnr;
            cnrq->tail = cnr;
        }

        cnrq->n++;
        return;
}


/*          - getcnr() -                              - [queue.c] -
**
** getcnr() -- gets one CORNER from the front of CORNER_QUEUE and
**          returns a pointer to that CORNER, it returns NIL if
**          the CORNER_QUEUE is empty.
*/


CORNER  *getcnr (cnrq)

register CORNER_QUEUE    *cnrq;
{
        CORNER *cnr;

        /* if queue is empty, return NIL */
        cnr = CNIL;

        /* if queue is not empty, get one from the front */
        if (cnrq->head != CNIL) {
            cnr = cnrq->head;
            cnrq->head = cnrq->head->link;
            cnrq->n--;
        }

        return(cnr);
}
```

```
/*         - error.c -
**
** error() -- print error messages.
*/


#include <stdio.h>
#include "cka.h"

error (flag, str, rgn)                                                  error

register int        flag;
register char       *str;
register REGION     *rgn;
{
        switch ( flag ) {
            case 1:                         /* main() */
                fpf(stderr,"Specify only one of '-f' and '-o'.\n");
                break;
            case 2:                         /* main() */
                fpf(stderr,"Can not open file: %s\n",str);
                break;
            case 3:
                fpf(stderr,
                    "Error from %s:\t'dim' or 'hyp' undefined.\n",str);
                break;
            case 4:
                fpf(stderr,"Error from %s:\tdim <= 0 ",str);
                fpf(stderr,"at line %d in the input file.\n",line);
                break;
            case 5:
                fpf(stderr,"Error from %s:\thyp <= 0 ",str);
                fpf(stderr,"at line %d in the input file.\n",line);
                break;
            case 6:                         /* filread() */
                pf("%s: input syntax error at line %d.\n",str,line);
                break;
            case 7:                         /* filread() */
                pf("%s: array index out of range at line %d.\n",str,line);
                break;
            case 8:                         /* normalize_D() */
                pf("Error: matrix D[,]");
                pf(" is not compatible with hybrid representation.");
                break;
            case 9:                         /* parallel_groups() */
                pf("Error: vector e[]");
                pf(" is not compatible with hybrid representation.");
                break;
            case 10:                        /* rgn_check() */
                pf("Jacobian matrix is ");
                if ( imsl ) {
                    if ( ier == 129 )
                        pf("is algorithmically singular.");
                    else if ( ier == 131 ) {
                        pf("is too ill-conditioned for iterative\n");
                        pf("\t\timprovement to be effective.");
                    }
                    pf(" [IMSL]\n");
                }
                else
                    pf("singular.\n");
                pf("** occured at:\n");
                print_rgn(rgn);
                break;
```

```
          }

          if ( flag < 10 ) {
               pf("\n**-> program aborted <-**\n");
               exit(1);                         /* fatal error */
          }
          else
               pf("\n**-> program continued <-**\n");
}
```

```
/*          - support.c -
**
** support.c -- contains the following supporting routines:
**         i_get(), d_get(), l_get(), s_get(),
**         Abs(), Sgn(), inprdct(), transp(), prdmtrx(),
**         privctr(), prdvctr(), lineqn(), rowech(), palloc().
*/

#include          <stdio.h>


/*
** i_get() -- get an integer from input.
*/

int       i_get (str, fp)

register char     *str;
FILE              *fp;
{
          char    *s_get();
          int     atoi();

          s_get(str,fp);
          return(atoi(str));
}


/*
** d_get() -- get a double number from input.
*/

double    d_get (str, fp)

register char     *str;
FILE              *fp;
{
          char    *s_get();
          double  atof();

          s_get(str,fp);
          return(atof(str));
}


/*
** l_get() -- get a line (with NL) from input.
*/

char      *l_get (str, fp)

register char     *str;
FILE              *fp;
{
          register int      c;
          register char     *cs;

          cs = str;
          while ((c=getc(fp)) != '\n' && c >= 0) *cs++ = c;

          if (c<0 && cs==str)
              return(NULL);
          else {
              *cs++ = '\n';
```

```
		*cs = '\0';
		return(str);
	}
}


/*
** s_get() -- get a string (without NL) from input.
*/

char	*s_get (str, fp)

register char	*str;
FILE		*fp;
{
	register int	c;
	register char	*cs;

	cs = str;
	while ((c=getc(fp)) != '\n' && c >= 0) *cs++ = c;

	if (c<0 && cs==str)
	    return(NULL);
	else {
	    *cs = '\0';
	    return(str);
	}
}


/*
** Abs() -- find absolute value with type double argument.
*/

double	Abs (x)

double	x;
{
	if (x >= 0.0)
	    return(x);
	else
	    return(-x);
}


/*
** Sgn() -- determine sign of a type double argument.
*/

int	Sgn (x)

double	x;
{
	if ( x > 0.0 )
	    return (1);
	else if ( x < 0.0 )
	    return (-1);
	else
	    return (0);
}
```

```
/*
** inprdct() -- inner product of 2 vectors:  c = <x,y>
*/

double   inprdct (px, py, dim)

register double   *px, *py;
register int      dim;
{
          register int      i;
          double   sum=0;

          for (i=0; i < dim; i++)
               sum += px[i] * py[i];
          return(sum);
}


/*
** transp() -- find trasnpose of a given matrix.
*/
```

transp

```
transp (pa, pat, row, col)

register double   *pa, *pat;
register int      row, col;
{
          register int      i, j;

          for (i=0; i < row; i++)
          for (j=0; j < col; j++)
               pat[j*row+i] = pa[i*col+j];
}


/*
** prdmtrx() -- print a double precision matrix.
*/
```

prdmtrx

```
prdmtrx (pm, row, col, format)

register double   *pm;
register int      row, col;
register char     *format;
{
          register int      i, j;

          for (i=0; i < row; i++) {
               printf("\n\t");
               for (j=0; j < col; j++)
                    printf(format,pm[i*col+j]);
          }
          printf("\n");
}


/*
** privctr() -- print an integer vector.
*/
```

privctr

```
privctr (pv, dim, format)

register int      *pv, dim;
register char     *format;
```

```
{
          register int        i;

          for (i=0; i < dim; i++)
               printf(format,pv[i]);
}


/*
** prdvctr() -- print a double precision vector.
*/
```

prdvctr

```
prdvctr (pv, dim, format)

register double     *pv;
register int        dim;
register char       *format;
{
          register int        i;

          for (i=0; i < dim; i++)
               printf(format,pv[i]);
}


/*
** lineqn() -- solve linear system Ax = b.
*/
```

lineqn

```
lineqn (pa, px, pb, dim, flag, deta)

register double     *pa;
double     *px, *pb, *deta;
int        dim, flag;
{
          int        axcol, err;
          register double    *pax;
          register int       i, j, m, n;

          axcol = dim+1;                        /* # of cols in AX[][] */
          pax = (double *) malloc(dim*axcol*sizeof(double));

          /* append x[] to the last column of A[][] => AX[][] */
          for (i=0; i < dim; i++) {
               m = i*axcol;
               n = i*dim;
               for (j=0; j < dim; j++)
                    pax[m+j] = pa[n+j];
               pax[m+dim] = pb[i];
          }

          /* compute row-echelon form of AX[][] */
          rowech (pax,pax,dim,axcol,deta,&err);

          /* if non-singular, start back substitution */
          if (!err) for (i=dim-1; i >= 0; i--) {
               m = i*axcol;
               px[i] = pax[m+dim];
               for (j=dim-1; j > i; j--)
                    px[i] -= px[j] * pax[m+j];
          }
```

```
        /* if flag != 0 then return A[][] in its row-echelon form */
        if (flag != 0) for (i=0; i < dim; i++) {
             m = i*axcol;
             n = i*dim;
             for (j=0; j < dim; j++)
                  pa[n+j] = pax[m+j];
        }

        free(pax);                          /* free spaces */
        return(err);
}


/*
** rowech() --   Reduce matrix A to the row echlon form.
**         The pivot element is chosen to be the maximum in that
**         column.
*/
```

```
rowech (pa, pr, arow, acol, deta, dep)

register double    *pa, *pr;
double    *deta;
int       arow, acol, *dep;
{
          double  Abs(), max, tmp;
          int     row, col, maxrow, stop;
          register int     i, j, m, n;

          for (i=0; i < arow; i++) {          /* copy A to R */
               m = i*acol;
               for (j=0; j < acol; j++)
                    pr[m+j] = pa[m+j];
          }

          stop=0; row=0; *deta=1.0;           /* initialize */

          while (!stop) {
          for (col=0; col < acol; col++) {
               /*
               ** find the maximum element in the column as the
               ** pivot element.
               */
               max = 0.0;
               for (i=row; i < arow; i++) {
                    tmp = pr[i*acol+col];
                    if (tmp != 0.0 && Abs(tmp) > Abs(max)) {
                         maxrow = i;
                         max = tmp;
                    }
               }
               if ( max != 0.0 ) {
                    m = maxrow*acol;
                    n = row*acol;
                    if ( maxrow != row ) {
                         /* interchange "maxrow" and "row" */
                         for (j=col; j < acol; j++) {
                              tmp = pr[m+j];
                              pr[m+j] = pr[n+j];
                              pr[n+j] = tmp;
                         }
                         (*deta) *= (-1.0);
                    }
                    /* normalize pivot element */
                    (*deta) *= max;
                    pr[n+col] = 1.0;
                    for (j=col+1; j < acol; j++)
                         pr[n+j] /= max;

                    row++;                        /* increment row */
                    if ( row < arow ) {
                         /*
                         ** reduce entries in "col" below "row" to 0.
                         */
                         for (i=row; i < arow; i++) {
                              tmp = pr[i*acol+col];
                              if (tmp != 0.0)
                                   for (j=col; j < acol; j++)
                                        pr[i*acol+j] += pr[(row-1)*acol+j]
                                                            * (-tmp);
                         }
                    }
               }
          }
          stop = 1;                               /* terminate iteration */
          }

          /* find first linear dependent column */
          *dep = 0;
          j = (arow < acol) ? arow : acol;/* j = min(arow,acol) */
          for (i=0; i < j; i++) {
               if ( pr[i*acol+i] != 1.0) {
                    *dep = i+1;
                    break;
               }
          }
}


/*
** palloc() -- C storage allocator, it calls "malloc()" to get 4096
**        bytes (2K words) at a time and re-distributes them to its
**        caller.  The purpose is to reduce the number of calls to
**        "malloc()".  If the number of bytes left is less than needed,
**        those spaces are waisted.
*/

#define   PAGESIZ 4096

char      *palloc (nbytes)
unsigned        nbytes;
{
          static    char      *pgtop;        /* page top */
          static    char      *cptr;         /* current pointer position */
          static    char      *nptr;         /* next pointer position */
          static    unsigned  tlngth;        /* total length used */
          static    int       flag;

          if ( nbytes > PAGESIZ )
               return ( (char *) malloc(nbytes) );

          if ( !flag ) {
               pgtop = (char *) malloc(PAGESIZ);
               nptr = pgtop;
               tlngth = 0;
               flag++;
          }
```

```
        if ( nbytes <= (PAGESIZ-tlngth) ) {
             cptr = nptr;
             tlngth += nbytes;            /* update used length */
             nptr += nbytes;              /* advance nptr */
             return(cptr);
        }
        else {                            /* not enough space left */
             flag = 0;
             return(palloc(nbytes));
        }
}
```

```
#          - Makefile -
#
# Maintain CKA program groups.

CFLAGS = -O

FILE =    Makefile cka.h main.c input.c katz.c lattice.c corner.c\
          init.c region.c print.c queue.c error.c support.c

OBJS =    main.o input.o katz.o lattice.o corner.o init.o region.o\
          print.o queue.o error.o support.o

a.out:    $(OBJS)
          cc $(CFLAGS) $(OBJS) -limsld -lF77 -lI77 -lm
          rm -f Kx Lx; mv a.out Kx; ln Kx Lx

main.o:              cka.h main.c
input.o:             cka.h input.c
katz.o:              cka.h katz.c
lattice.o:           cka.h lattice.c
corner.o:            cka.h corner.c
init.o:              cka.h init.c
region.o:            cka.h region.c
print.o:             cka.h print.o
queue.o:             cka.h queue.c
error.o:             cka.h error.c
support.o:           support.c
```