FINDING ALL SOLUTIONS OF PIECEWISE-LINEAR EQUATIONS

by

Leon O. Chua and Robin L. P. Ying

Memorandum No. UCB/ERL M81/54

23 July 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Finding All Solutions of Piecewise-Linear Equations[†]

Leon O. Chua and Robin L.P. Ying[§]

ABSTRACT

A new algorithm is given for solving piecewise-linear equations of nonlinear electronic circuits. Unlike other methods, this algorithm guarantees that all solutions will be found in a finite number of steps. The method depends crucially on a recent development which allows a multi-dimensional piecewise-linear function to be represented in a closed canonical form. This highly compact representation requires only a minimum amount of computer storage and is responsible for the efficiency of the algorithm.

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

[§]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

Nonlinear circuits exhibiting multiple equilibrium points (dc solutions) are indispensable building blocks (e.g., flip flops) of many modern electronic systems. Multi-valued circuits has received a great deal of attention recently in view of its potential applications to VLSI circuits [1-6] where significantly fewer wirings are required over conventional designs. The phenomenon of multiple equilibrium points is also encountered in many physical devices [7-10] and models [11-12].

Although many algorithms have been published over the past decade which are capable of finding multiple solutions of nonlinear resistive circuits [13-20], except for [13-14], none can guarantee that all solutions will be found. The other algorithms will usually find only those solutions which fall on a certain solution branch. Random searches will sometimes uncover additional solutions falling on other solution branches. However, these algorithms all share the serious shortcoming that they can not guarantee that all solutions will be found.

The algorithm described in [13-14] is an improved version of the brute-force piecewise-linear combinatorial algorithm described in [21].[1] Unfortunately, this algorithm is still quite inefficient and is difficult to implement in a computer.

One objective in this paper is to describe a new algorithm which is more efficient and more easily programmed. This algorithm takes advantage of a new canonical representation for single- and multi-dimensional [23,24] piecewise linear functions. It is applicable to any resistive circuit described by a piecewise-linear hybrid equation to be described in Section 2. The algorithm is derived in Section 3 with illustration example given in Section 4. The ill-conditioned cases are analyzed in Section 5 along with remedies. Finally, the computational efficiency of this algorithm is compared in Section 6 with the brute-force combinatorial method.

2. PIECEWISE-LINEAR EQUATION FORMULATION

Let N denote any circuit made of linear, possibly coupled, resistive elements (e.g., linear controlled sources, transformers, gyrators, etc.) and 2-terminal nonlinear resistors. We assume the nonlinear resistors are either voltage or current-controlled and are approximated by continuous piecewise-linear functions. Hence the class of circuits we allow can be depicted as in Fig. 1, where all nonlinear resistors have been extracted across a linear n-port $\tilde{N}$. Note that

[1] In spite of the tremendous advances in the development of "computer circuit analysis programs" over the last decade [14], MECA [22] remains the only existing resistive circuit analysis program capable of finding all solutions.

since $\overline{N}$ may contain any type of linear controlled sources, and since most device circuit models are made simply of 2-terminal nonlinear resistors and controlled sources [14], most practical resistive circuits are allowed. In fact, using the recent "decomposition theorem" in [25] which asserts that any multi-terminal nonlinear resistor can be modeled in terms of a circuit made of only 2-terminal nonlinear resistors and linear controlled sources, we can in principle allow all resistive circuits provided certain preliminary transformations are performed. In other words, there is little loss of generality in developing algorithms for the class of circuits shown in Fig. 1.

The only additional assumption we make is that the linear n-port $\overline{N}$ in Fig. 1 has the following hybrid-representation:

$$\begin{bmatrix} \bar{\underset{\sim}{i}}_a \\ \bar{\underset{\sim}{v}}_b \end{bmatrix} = \begin{bmatrix} \bar{\underset{\sim}{H}}_{aa} & \bar{\underset{\sim}{H}}_{ab} \\ \bar{\underset{\sim}{H}}_{ba} & \bar{\underset{\sim}{H}}_{bb} \end{bmatrix} \cdot \begin{bmatrix} \bar{\underset{\sim}{v}}_a \\ \bar{\underset{\sim}{i}}_b \end{bmatrix} + \begin{bmatrix} \underset{\sim}{s}_a \\ \underset{\sim}{s}_b \end{bmatrix} \tag{2.1}$$

where

$$\bar{\underset{\sim}{v}}_a \triangleq [\bar{v}_1 \bar{v}_2 \cdots\cdots \bar{v}_\ell]^T, \quad \bar{\underset{\sim}{v}}_b \triangleq [\bar{v}_{\ell+1} \bar{v}_{\ell+2} \cdots\cdots \bar{v}_n]^T$$

$$\bar{\underset{\sim}{i}}_a \triangleq [\bar{i}_1 \bar{i}_2 \cdots\cdots \bar{i}_\ell]^T, \quad \bar{\underset{\sim}{i}}_b \triangleq [\bar{i}_{\ell+1} \bar{i}_{\ell+2} \cdots\cdots \bar{i}_n]^T$$

and $[\bar{\underset{\sim}{S}}_a \; \bar{\underset{\sim}{S}}_b]^T$ denote the source vector due to the independent sources. Efficient computer methods for deriving (2.1) are given in [14]. Hence, we will simply assume that (2.1) is given when describing our algorithm in Section 3. Note that even in the few instances where $\overline{N}$ does not have a hybrid representation, there exit many standard techniques for transforming the circuit N in Fig. 1 into an equivalent circuit N' such that the associated linear n-port $\overline{N}$ has a hybrid representation (2.1). For example, one can always extract a small linear resistor from any nonlinear resistor and imbed it into the linear n-port $\overline{N}$. Hence, the additional "hybrid-representation assumption" does not entail any loss of generality.

Applying the canonical representation from [23], each (piecewise-linear) voltage-controlled resistor can be described analytically by:

$$i_k = a_k + b_k v_k + \sum_{i=1}^{p_k} c_{k_i} |v_k - V_{k_i}| , \quad k = 1,2,\cdots\cdots\ell \tag{2.2}$$

Similarly, each (piecewise-linear) current-controlled resistor can be described by:

$$v_k = a_k + b_k i_k + \sum_{i=1}^{p_k} c_{k_i} |i_k - I_{k_i}|, \quad k = \ell+1,\ \ell+2,\ \cdots\cdots n \tag{2.3}$$

By defining

$$\underset{\sim}{v}_a \triangleq [v_1 v_2 \cdots v_\ell]^T, \quad \underset{\sim}{v}_b \triangleq [v_{\ell+1} v_{\ell+2} \cdots v_n]^T$$

$$\underset{\sim}{i}_a \triangleq [i_1 i_2 \cdots i_\ell]^T, \quad \underset{\sim}{i}_b \triangleq [i_{\ell+1} i_{\ell+2} \cdots i_n]^T$$

we can combine (2.2) and (2.3) into a single vector equation

$$\begin{bmatrix} \underset{\sim}{i}_a \\ \underset{\sim}{v}_b \end{bmatrix} = \underset{\sim}{a}' + \underset{\sim}{B}' \begin{bmatrix} \underset{\sim}{v}_a \\ \underset{\sim}{i}_b \end{bmatrix} + \sum_{j=1}^{n} \sum_{i=1}^{p_j} c_{ji} \underset{\sim}{u}_j \left| \left\langle \underset{\sim}{u}_j, \begin{bmatrix} \underset{\sim}{v}_a \\ \underset{\sim}{i}_b \end{bmatrix} \right\rangle - \beta_{ji} \right| \tag{2.4}$$

where

$$\underset{\sim}{a}' = [a_1 a_2 \cdots a_\ell a_{\ell+1} \cdots a_n]^T$$

$$B' = \text{diag}[b_1 b_2 \cdots b_\ell b_{\ell+1} \cdots b_n]$$

$$\beta_{ji} = \begin{cases} V_{ji}, & j = 1,2,\cdots \ell \\ I_{ji}, & j = \ell+1, \ell+2, \cdots n \end{cases}$$

and $\underset{\sim}{u}_j$ is the jth unit vector in $\mathbb{R}^n$, and $\langle \cdot, \cdot \rangle$ denotes the vector dot product.

From Fig. 1 we have $\bar{v}_k = v_k$, $\bar{i}_k = i_k$, $k = 1,2,\cdots n$. Hence, we can equate the right-hand sides of (2.1) and (2.4) to obtain the equation

$$\underset{\sim}{a} + \underset{\sim}{B}\underset{\sim}{x} + \sum_{j=1}^{n} \sum_{i=1}^{p_j} \underset{\sim}{c}_{ji} \left| \langle \underset{\sim}{u}_j, \underset{\sim}{x} \rangle - \beta_{ji} \right| = \underset{\sim}{0} \tag{2.5}$$

where

$$\underset{\sim}{x} \triangleq \begin{bmatrix} \underset{\sim}{v}_a \\ \underset{\sim}{i}_b \end{bmatrix}$$

$$\underset{\sim}{a} \triangleq \underset{\sim}{a}' - \begin{bmatrix} \bar{\underset{\sim}{s}}_a \\ \bar{\underset{\sim}{s}}_b \end{bmatrix}$$

$$\underset{\sim}{B} \triangleq \underset{\sim}{B}' - \begin{bmatrix} \bar{\underset{\sim}{H}}_{aa} & \bar{\underset{\sim}{H}}_{ab} \\ \bar{\underset{\sim}{H}}_{ba} & \bar{\underset{\sim}{H}}_{bb} \end{bmatrix}$$

$$\underset{\sim}{c}_{ji} \triangleq c_{ji} \underset{\sim}{u}_j$$

If we relabel the double indices in the last term of (2.5), we can recast (2.5) into the following canonical form [24]:

$$\boxed{\underset{\sim}{f}(x) = \underset{\sim}{a} + \underset{\approx}{B}\underset{\sim}{x} + \sum_{i=1}^{p} \underset{\sim}{c}_i \left| \langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i \right| = \underset{\sim}{0}} \tag{2.6}$$

where $\underset{\sim}{c}_i$ and β_i denote simply $\underset{\sim}{c}_{ji}$ and β_{ji} rewritten with new single indices.

Note that

$$\underset{\sim}{\alpha}_j = \begin{cases} \underset{\sim}{u}_1 , & j = 1,2,\cdots p_1, \\ \underset{\sim}{u}_2 , & j = p_1+1,\cdots p_1+p_2 \\ \vdots & \vdots \\ \underset{\sim}{u}_n , & j = p_{n-1}+1, \quad p_{n-1}+p_n. \end{cases}$$

We have just proved that <u>any piecewise-linear resistive circuit can be described by a system of multi-dimensional piecewise-linear equations in the canonical form (2.6)</u>. This compact equation contains only the minimum data needed to specify the circuit. It is clearly far superior to the conventional piecewise-linear approach where a linear equation must be specified and stored in the computer <u>for each</u> region, along with its <u>boundary</u>.[2]

Another noteworthy feature of (2.6) is the special form assumed by the unit vectors $\underset{\sim}{\alpha}_i$. Since each $\underset{\sim}{\alpha}_i$ is simply a "unit vector" along some coordinate axis, each hyperplane

$$\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle = \beta_i \quad , \quad \underset{\sim}{x} \in \mathbb{R}^n$$

is perpendicular to a coordinate axis. Hence the set of "p" hyperplanes in (2.6) partition the domain $\mathbb{R}^n$ of $\underset{\sim}{f}(\underset{\sim}{x})$ into a "rectangular lattice" whose boundaries are parallel to the coordinate axes. This remarkably simple geometrical structure is responsible for the high efficiency of the algorithm to be developed in the following sections.

3. ALGORITHM FOR FINDING ALL SOLUTIONS

We begin with a simple example which illustrates geometrically the basic

[2]The enormous amount of data needed to be stored is in fact one of the most objectionable features of conventional piecewise-linear analysis. This objection is now overcome by representing the data by a compact canonical equation.

idea behind the general algorithm to be presented in detail later.

Example 1.

Consider the simple circuit shown in Fig. 2(a). The v-i characteristics of R1 and R2 are approximated by continuous piecewise-linear segments shown in Fig. 2(b) and 2(c), respectively. Using the formulas in [20], R1 and R2 can be expressed in the canonical form [20] as follow:

$$\text{R1: } i_1 = -\frac{3}{4} + \frac{5}{4} v_1 - \frac{3}{2} |v_1 - 2| + \frac{3}{4} |v_1 - 5| \tag{3.1}$$

$$\text{R2: } i_2 = \frac{9}{4} + \frac{5}{4} v_2 - \frac{3}{4} |v_2 - 3| \tag{3.2}$$

Applying KCL ($i_1 = i_2$) and KVL to Fig. 2(a), we obtain

$$v_1 + v_2 + 2i_1 = 9 \tag{3.3}$$

$$v_1 + v_2 + 2i_2 = 9 \tag{3.4}$$

substituting (3.1) into (3.3), and (3.2) into (3.4), we obtain:

$$-\frac{21}{4} + \frac{7}{4} v_1 + \frac{1}{2} v_2 - \frac{3}{2} |v_1 - 2| + \frac{3}{4} |v_1 - 5| = 0 \tag{3.5}$$

$$-\frac{9}{4} + \frac{1}{2} v_1 + \frac{7}{4} v_2 - \frac{3}{4} |v_2 - 3| = 0 \tag{3.6}$$

This can be recast into the following canonical form:

$$\underset{\sim}{f}(v_1 v_2) = \begin{bmatrix} -\frac{21}{4} \\ -\frac{9}{4} \end{bmatrix} + \begin{bmatrix} \frac{7}{4} & \frac{1}{2} \\ \frac{1}{2} & \frac{7}{4} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} -\frac{3}{2} \\ 0 \end{bmatrix} |v_1 - 2| + \begin{bmatrix} \frac{3}{4} \\ 0 \end{bmatrix} |v_1 - 5| + \begin{bmatrix} 0 \\ -\frac{3}{4} \end{bmatrix} |v_2 - 3|$$

$$= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.7}$$

Figure 2(e) shows that the domain of $\underset{\sim}{f}$ is partitioned by the following three 1-dimensional hyperplanes (straight lines in this case) $v_1 = 2$, $v_1 = 5$, and $v_2 = 3$.

Note that they are parallel to either the v_1 or v_2 axis. Hence, the domain of $\underset{\sim}{f}$ in Fig. 2(e) is partitioned into a rectangular lattice with edges parallel to the coordinate axis.

The image of the lattice in the range space of $\underset{\sim}{f}(v_1,v_2)$ is shown in Fig. 2(f). Note that each of 3 regions bounded by c'a'd'. d'a'b'e' and e'b'f', respectively, contains the origin of the range space as an interior point. By the regionwise linearity of $\underset{\sim}{f}$, we can conclude immediately that the 3 corresponding regions in the domain bounded by cad (region R_1), dabe (region R_2) and cbf (region R_3) contain solutions of (3.7)

Observe that since there are no other regions in the range space in Fig. 2(f) which contain the origin, the regions in the domain which contain a solution of (3.7) are precisely R_1, R_2, and R_3.

Since $\underset{\sim}{f}(\cdot)$ is an affine function in each region, we can simplify (3.7) into a system of 2 linear equations for each of the 3 regions where $\underset{\sim}{f}(\cdot)$ has a solution. For example, in region R_1, (3.7) reduce to:

$$\underset{\sim}{f}(v_1 v_2) = \begin{bmatrix} \frac{5}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{5}{2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} -\frac{9}{2} \\ -\frac{9}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad v_1, v_2 \in R_1 \tag{3.8}$$

solving (3.8), we obtain the following solution of (3.7) in region R_1:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \frac{3}{2} \\ \frac{3}{2} \end{bmatrix}$$

Similarly, we obtain the following solution of (3.7):

Region R_2: $\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$, Region R_3: $\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \frac{17}{3} \\ \frac{2}{3} \end{bmatrix}$.

The above three solutions can be easily verified by the load-line method shown in Fig. 2(d). Here, we combine the 2 nonlinear resistors into an equivalent one-port described by the 3-segment driving-point plot shown in Fig. 2(d) [21].

3.1. The n-dimensional case:

In Example 1, we use visual inspection to determine the regions whose images contain the origin in the range space as an interior point. However, in higher-dimensional cases (dimension ≥ 3), visual inspection becomes very awkward (dimension $= 3$) or even impossible (dimension > 3). We will now develop an algorithm which will extend the preceding geometrical idea to the arbitrary n-dimensional case.

Let $\underset{\sim}{f}(\cdot)$ be represented in the form of (2.6). The partition hyperplanes associated with $\underset{\sim}{f}(\cdot)$ are determined by the set of equations:

$$\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i = 0, \quad i = 1,2,\cdots p \tag{3.9}$$

Consider an arbitrary k-th hyperplane H_k defined by:

$$\langle \underset{\sim}{\alpha}_k, \underset{\sim}{x} \rangle - \beta_k = 0 \tag{3.10}$$

In general, H_k will be further partitioned into several sections[3] by other hyperplanes which intersect it. We consider only one arbitrary section $\sigma_a\sigma_b$ on H_k. (See Fig. 3(a).)

For $\underset{\sim}{x} \in \sigma_a\sigma_b$, we expand the absolute value in the last term of (2.6) and write $\underset{\sim}{f}(\cdot)$ as:

$$\underset{\sim}{f}(\underset{\sim}{x}) = \bar{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}} + \bar{\underset{\sim}{B}}_{k_{\sigma_a\sigma_b}} \underset{\sim}{x} \tag{3.11}$$

where $\underset{\sim}{x}$ is subject to the constraint (3.10) and[4]

$$\bar{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}} = \underset{\sim}{a} + \sum_{\substack{i=1 \\ i\neq k}}^{p} \underset{\sim}{c}_i(\pm\beta_i) \tag{3.12}$$

$$\bar{\underset{\sim}{B}}_{k_{\sigma_a\sigma_b}} = \underset{\sim}{B} + \sum_{\substack{i=1 \\ i\neq k}}^{p} \underset{\sim}{c}_i(\pm\underset{\sim}{\alpha}_i^T) \tag{3.13}$$

The choice of $\pm$ sign in (3.12) and (3.13) depends on the sign of the arguments $\langle \alpha_i, x \rangle - \beta_i$, $i = 1,2,\cdots p$.

Assuming $\bar{\underset{\sim}{B}}^{-1}_{k_{\sigma_a\sigma_b}}$ exists, we get from (3.11)

$$\underset{\sim}{x} = \bar{\underset{\sim}{B}}^{-1}_{k_{\sigma_a\sigma_b}} \left(\underset{\sim}{f}(\underset{\sim}{x}) - \bar{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}} \right) \tag{3.14}$$

---

[3]A section of the k-th hyperplane H_k is a subset of H_k such that for all $\underset{\sim}{x}$ in this subset, $\mathrm{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)$, $i = 1,2,\cdots p$, $i \neq k$ do not change sign.

[4]The term involving $i = k$ drops out in view of (3.10).

Substituting (3.14) into (3.10), we get

$$\left\langle \underset{\sim}{\alpha}_k, \overline{\underset{\sim}{B}}^{-1}_{k_{\sigma_a\sigma_b}} \left[f(\underset{\sim}{x}) - \overline{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}} \right] \right\rangle - \beta_k = 0$$

or

$$\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y} \rangle - \beta'_{k_{\sigma_a\sigma_b}} = 0 \tag{3.15}$$

where

$$\underset{\sim}{y} = \underset{\sim}{f}(\underset{\sim}{x})$$

$$\underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}} = (\overline{B}^{-1}_{k_{\sigma_a\sigma_b}})^T \underset{\sim}{\alpha}_k \tag{3.16}$$

$$\beta'_{k_{\sigma_a\sigma_b}} = \beta_k + \langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \overline{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}} \rangle \tag{3.17}$$

Let $\sigma'_a\sigma'_b$ denotes the image of a section $\sigma_a\sigma_b$ of H_k, then (3.15) is the representation of $\sigma'_a\sigma'_b$ in the range space of $\underset{\sim}{f}$.

In Fig.3(a), let R_a and R_b denote the neighborhood regions separated by $\sigma_a\sigma_b$ and let $\underset{\sim}{x}_a$ and $\underset{\sim}{x}_b$ denote arbitrary interior points of R_a and R_b respectively. Let their images in the range space of $\underset{\sim}{f}$ be R'_a, R'_b, $\underset{\sim}{y}_a$ and $\underset{\sim}{y}_b$ respectively (see Fig. 3(b)).

Assuming that $\underset{\sim}{f}$ is not degenerate (i.e. det $\overline{B}_{k_{\sigma_a\sigma_b}} \neq 0$) in either R_a or R_b, then $\underset{\sim}{y}_a$ and $\underset{\sim}{y}_b$ will be interior points of R'_a and R'_b respectively. The following <u>sign test</u> allows us to determine whether the origin in the range space lies on the same side of $\sigma'_a\sigma'_b$ with $\underset{\sim}{y}_a$:

> Sign test:
>
> $\underset{\sim}{y}_a$ and the origin lie on the same side of $\sigma'_a\sigma'_b$ if and only if
>
> $$\mathrm{sgn}(\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y}_a \rangle - \beta'_{k_{\sigma_a\sigma_b}}) = \mathrm{sgn}(-\beta'_{k_{\sigma_a\sigma_b}}) \tag{3.18}$$
>
> where $\underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}$ and $\beta'_{k_{\sigma_a\sigma_b}}$ are defined in (3.16) and (3.17) respectively.

<u>Proof of the sign test</u>:

Since $\underset{\sim}{f}$ is piecewise-linear and since $\underset{\sim}{f}$ is assumed to be nondegenerate in the neighborhood regions of $\sigma_a\sigma_b$, the image $\sigma_a'\sigma_b'$ is a portion of a linear hyperplane represented by $\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y} \rangle - \beta'_{k_{\sigma_a\sigma_b}} = 0$. Therefore $\underset{\sim}{y}_a$ and the origin lie on the same side of $\sigma_a'\sigma_b'$ if and only if

$$\mathrm{sgn}(\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y}_a \rangle - \beta'_{k_{\sigma_a\sigma_b}}) = \mathrm{sgn}(\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{0} \rangle - \beta'_{k_{\sigma_a\sigma_b}})$$

$$= \mathrm{sgn}(-\beta'_{k_{\sigma_a\sigma_b}})$$

which is exactly (3.18). ⌑

In order to conclude region R_a' contains the origin, we need to perform the sign test on all boundaries of R_a'. Hence we have the following necessary condition:

Solution Validation Test: If the sign test fails on any one of the boundaries of R_a', then R_a contains no solution of (2.6).

The above test allows us to discard a region once the sign test fails on any one of its boundaries. Therefore, carrying out the sign test over all partition hyperplanes defined by (2.6) will allow us to identify all regions which contain a solution of (2.6). Hence this approach guarantees that all solutions of (2.6) will be found.

3.2. Efficient implementation of the sign test

Although the theory behind the sign test is quite simple, its practical implementation is extremely time consuming for arbitrary piecewise-linear equations, i.e., when $\underset{\sim}{f}(\cdot)$ in (2.6) is arbitrary. However, for the subclass of piecewise-linear equations representing the hybrid equations derived in section 1, the unit vectors $\underset{\sim}{\alpha}_i$, $i = 1,2,\cdots p$, assume a particular simple form. In this section, we will exploit this special structure to develop an efficient algorithm for carrying out the sign test.

We will use the following 3-dimensional example as a vehicle to describe the algorithm.

A. Example 2.

Consider the circuit shown in Fig. 4(a). Piecewise-linear resistors R1 and R3 are voltage controlled; their v-i characteristics are shown in Figs. 4(b) and 4(d) respectively. Piecewise-linear resistor R2 is current controlled; its v-i characteristic is shown in Fig. 4(c). Extracting R1, R2, R3 as external ports, we obtain the following hybrid representation for the remaining linear 3-port:

$$\begin{bmatrix} i_1 \\ v_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} + \begin{bmatrix} -5 \\ -5 \\ 5 \end{bmatrix} \tag{3.19}$$

Substituting the equations for R1, R2 and R3 into (3.19), we obtain the following system of 3 piecewise-linear equations:

$$\frac{5}{6}\,|v_1+6| - \frac{5}{6}\,|v_1-6| = v_1 + i_2 + v_3 - 5$$

$$\frac{1}{6}\,|i_2+1| - \frac{1}{6}\,|i_2-5| = i_2 + v_3 - 5$$

$$v_3 - \frac{5}{4}\,|v_3-1| + 2\,|v_3-2| - |v_3-3| = -v_3 + 5$$

These equations can be recast into the following canonical form:

$$\begin{bmatrix} -5 \\ -5 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & -2 \end{bmatrix} \underset{\sim}{x} + \begin{bmatrix} -\frac{5}{6} \\ 0 \\ 0 \end{bmatrix} |\langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle + 6| + \begin{bmatrix} \frac{5}{6} \\ 0 \\ 0 \end{bmatrix} |\langle \underset{\sim}{\alpha}_2, \underset{\sim}{x} \rangle - 6|$$

$$+ \begin{bmatrix} 0 \\ -\frac{1}{6} \\ 0 \end{bmatrix} |\langle \underset{\sim}{\alpha}_3, \underset{\sim}{x} \rangle + 1| + \begin{bmatrix} 0 \\ \frac{1}{6} \\ 0 \end{bmatrix} |\langle \underset{\sim}{\alpha}_4, \underset{\sim}{x} \rangle - 5| + \begin{bmatrix} 0 \\ 0 \\ \frac{5}{4} \end{bmatrix} |\langle \underset{\sim}{\alpha}_5, \underset{\sim}{x} \rangle - 1|$$

$$+ \begin{bmatrix} 0 \\ 0 \\ -2 \end{bmatrix} |\langle \underset{\sim}{\alpha}_6, \underset{\sim}{x} \rangle - 2| + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} |\langle \underset{\sim}{\alpha}_7, \underset{\sim}{x} \rangle - 3| = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.20}$$

where

$$\underset{\sim}{x} = \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix}, \quad \underset{\sim}{\alpha}_1 = \underset{\sim}{\alpha}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \underset{\sim}{\alpha}_3 = \underset{\sim}{\alpha}_4 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ and } \underset{\sim}{\alpha}_5 = \underset{\sim}{\alpha}_6 = \underset{\sim}{\alpha}_7 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The domain $\mathbb{R}^3$ is partitioned by 7 hyperplanes $h_1, h_2 \cdots h_7$ into 36 regions as shown in Fig. 5(a). For example, hyperplane h_1 is described by $\langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle + 6 = 0$. Note that the special structure of $\underset{\sim}{\alpha}_i$ guarantees that the hyperplanes along each coordinate axis are parallel to each other. Now, a brute force implementation of the sign test will require solving for $\underset{\sim}{\alpha}'_{k_{\sigma_a \sigma_b}}$ using (3.16), and $\beta'_{k_{\sigma_a \sigma_b}}$ using (3.17), over all regions. The calculation of $\underset{\sim}{\alpha}'_{k_{\sigma_a \sigma_b}}$ is particularly time consuming because it involves solving a system of linear equations of order n.

However, by taking advantage of the special structure of (3.20), the total number of $\underset{\sim}{\alpha}'_{k_{\sigma_a \sigma_b}}$ and $\beta'_{k_{\sigma_a \sigma_b}}$ that needs to be computed can be greatly reduced in view of the following observations:

B. 5 observations

Observation 1:

Consider the center section defined by the rectangle abcd of h_5 in Fig. 5(b). Let a'b'c'd' denote the image of abcd in the range space and let $\underset{\sim}{\alpha}'_5$ be the normal vector of a'b'c'd'. Since abcd serves as a boundary for region 5 as well as for region 14, we can use $\underset{\sim}{\alpha}'_5$ for two sign tests. Therefore for each $\underset{\sim}{\alpha}'_k$ computed by (3.16), we can perform the sign test on two adjacent regions.

Observation 2:

For hyperplane h_6 in Fig. 5(c) and h_7 in Fig. 5(d), let e'f'g'h' and p'q'r's' denote the images of sections efgh and pqrs in the range space respectively. Let $\underset{\sim}{\alpha}'_{14}$ and $\underset{\sim}{\alpha}'_{23}$ denote the normal vector of e'f'g'h' and p'q'r's' respectively. In Fig. 5(a), hyperplanes h_5, h_6, h_7 are parallel, therefore $\underset{\sim}{\alpha}'_{14}$ and $\underset{\sim}{\alpha}'_{23}$ should also be parallel to $\underset{\sim}{\alpha}'_5$. Consider $\underset{\sim}{\alpha}'_{23}$, by the parallelism, there exists a constant $t \neq 0$, such that

$$t\underset{\sim}{\alpha}'_{23} = \underset{\sim}{\alpha}'_5 \qquad (3.21)$$

To determine t, we observe from (3.16) and Fig. 5(d) that

$$\underset{\sim}{\alpha}'_{23} = (\underset{\sim}{B}_{23}^{-1})^T \underset{\sim}{\alpha}_7 \qquad (3.22)$$

Now in (3.20), we have $\underset{\sim}{\alpha}_5 = \underset{\sim}{\alpha}_7 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ (3.23)

Substituting (3.22) and (3.23) into (3.21), we get

$$(\bar{B}_{23})^T \alpha'_5 = t\,\alpha_5 = t \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (3.24)$$

Hence, t can be determined by computing the vector dot product between the previously calculated α'_5 and the last column of $\bar{B}_{23}$. To implement the sign test on region 23, we also need to calculate β'_{23}. By (3.17), we have

$$\beta'_{23} = \beta_{23} + \langle \alpha'_{23}, \bar{a}_{23} \rangle \qquad (3.25)$$

The set of y in the image p'q'r's' must satisfy the equation:

$$\langle \alpha'_{23}, y \rangle - \beta'_{23} = 0 \qquad (3.26)$$

Multiplying (3.25) and (3.26) by t and using (3.21), we obtain

$$t\beta'_{23} = t\beta_{23} + \langle \alpha'_5, \bar{a}_{23} \rangle \qquad (3.27)$$

and

$$\langle \alpha'_5, y \rangle - t\beta'_{23} = 0 \qquad (3.28)$$

It follows from (3.28) that we can use α'_5 instead of α'_{23} in the sign test for region 23 provided we use $t\beta'_{23}$ from (3.27) instead of β'_{23} at same time. Note that (3.27) and (3-28) do not involve α'_{23}. Hence, we have replaced the expensive task of solving a linear system by the simple task of computing a vector dot product via (3.24). Likewise, we have eliminated the task of calculating a new vector by simply rescaling a scalar via (3.27) and (3.28). Note also that we need only one column of $\bar{B}_{23}$ instead of the whole matrix for calculating t via (3.24). It follows from the above observation and Figs. 5(b)-5(e) that only 9 normal vectors (corresponding to the 9 sections comprising h_5) are needed to perform the sign tests for all regions associated with the group of 3 parallel hyperplanes h_5, h_6, h_7.

Observation 3:

Since Observation 2 shows that the number of normal vectors $\alpha'_{k_{\sigma_a \sigma_b}}$ that must be calculated by solving a linear system of equations (hence inefficient) is equal to the number of sections of the associated hyperplane, significant amount of computation time can be saved by choosing a hyperplane having the smallest number of sections.

For example, in Fig. 4(a), hyperplanes h_1, h_2, h_3 and h_4 have 12 sections each, whereas hyperplanes h_5, h_4, and h_7 have only 9 sections each. In this case, we would pick h_5, or any hyperplane parallel to h_5 (h_6 or h_7).

In the general case of (2.6), we let $k_i \geq 0$ denote the number of "parallel" hyperplanes intersecting the x_i axis.[5] Hence the set of all hyperplanes associated with (2.6) is subdivided into "n" groups corresponding to the "n" variables $x_1, x_2, \ldots, x_n$. All hyperplanes belonging to a given group j contains the same number "N_j" of sections, where

$$N_j = \prod_{\substack{i=1 \\ i \neq j}}^{n} (k_i+1) \tag{3.29}$$

Hence, we simply pick a group "k" which contains the smallest number of sections; namely,

$$N_k = \min_{1 \leq j \leq n} N_j \tag{3.30}$$

Observation 4:

Since each normal vector can be used to check the sign test for two adjacent regions (Observation 1) we need only calculate β' (as described in Observation 2) for sections lying on every second parallel hyperplane.

For example, if we start with hyperplane h_5 in Fig. 4(b), then it is not necessary to calculate β' for any of the sections comprising hyperplane h_6. In this case, β' needs to be calculated only for corresponding sections on hyperplanes h_5 and h_7, using the efficient technique described in Observation 2.

Observation 5:

To implement the sign test in each region R_j, we must locate an interior point $\underset{\sim}{x}^* \in R_j$ and calculate $\underset{\sim}{y}^* = \underset{\sim}{f}(\underset{\sim}{x}^*)$ using (3.11). Since all hyperplanes intersecting a coordinate axis $\underset{\sim}{x}_i$ orthogonally are parallel to each other, $\underset{\sim}{x}^*$ can be trivially chosen to be the mid point within each "bounded" region. For example, to find $\underset{\sim}{x}_{23}^*$ for region R_{23} in Fig. 4(d), we note R_{23} is bounded by h_1 ($x_1 = \beta_1$) and h_2 ($x_2 = \beta_2$) in the x_1-direction; by h_3 ($x_2 = \beta_3$) and h_4 ($x_2 = \beta_4$) in the x_2-direction; by h_6 ($x_3 = \beta_6$) and h_7 ($x_3 = \beta_7$) in the

[5] k = 0 if x does not appear within the absolute value signs in (2.6). In terms of the network in Fig. 1, this corresponds to the degenerate case where port "i" is terminated by a linear resistor.

x_3-direction. Hence, we simply choose

$$\underset{\sim}{x}^*_{23} = \begin{bmatrix} \frac{1}{2}(\beta_1+\beta_2) \\ \frac{1}{2}(\beta_3+\beta_4) \\ \frac{1}{2}(\beta_6+\beta_7) \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(6-6) \\ \frac{1}{2}(5+1) \\ \frac{1}{2}(3+2) \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ \frac{5}{2} \end{bmatrix}$$

For unbounded regions, we simply add or subtract the boundary coordinate by a convenient number. For example, to find $\underset{\sim}{x}^*_3$ for region R_3 in Fig. 4(b), we note that R_3 lies to the right of $h_2(x_1 = \beta_2)$ in the x_1-direction; above $h_3(x_2 = \beta_3)$ in the x_2-direction, and below h_5 $(x_3 = \beta_5)$ in the x_3-direction. Hence, a convenient choice of $\underset{\sim}{x}^*_3$ is:

$$\underset{\sim}{x}^*_3 = \begin{bmatrix} \beta_2+1 \\ \beta_3-1 \\ \beta_5-1 \end{bmatrix} = \begin{bmatrix} 6+1 \\ -1-1 \\ 1-1 \end{bmatrix} = \begin{bmatrix} 7 \\ -2 \\ 0 \end{bmatrix}$$

C. Bookkeeping Scheme

In order to take full advantage of the above observations, it is essential to develop an efficient bookkeeping scheme. Again, we will use the example in Fig. 4(a) as a vehicle to illustrate our bookkeeping technique:

We use 3 lists to keep track of regions. Before the "iteration process"[6] starts, the first list W_0 contains all 36 regions; the second list W_s (solution list) and the third list W_1 (working list) are both initially empty. Having chosen h_5 (Observation 4), we begin the iteration by listing all "neighborhood" regions of h_5 belonging to $W_0 \cup W_s$ into W_1; namely,

$$W_1 = \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18\}$$

Next, we compute 9 $\underset{\sim}{\alpha}$'s and β's (corresponding to the 9 sections 1,2,...9 in h_5) and carry out the sign tests for regions 1,2,...18. If a region in W_1 passes the test, it is put into W_s; otherwise it is discarded.

For this example, all 18 regions failed the sign test. Hence, we set W_5 = empty set and put the remaining regions (19,20,...36) into W_0.

Next, we move to h_7 with W_1 containing regions 19 through 36. Now we only need to compute 9 tβ's (recall Observation 2) to accomplish the sign tests.

[6] The "iteration process" here means "computing $\underset{\sim}{\alpha}'$, β' and performing the sign test on related regions.

In this case, only regions 19,20,...27 pass the sign test and we write

W_s = {19,20,21,22,23,24,25,26,27} and W_0 is now empty.

Since W_s contains neighborhood regions of h_6, we return to h_6 to calculate the "9" associated $t\beta$'s needed to carry out the sign test. In this case, we found W_s stays the same.

Having exhausted all hyperplanes in group 3, we proceed to the next group of hyperplanes having the smallest N_j (recall Observation 2). In this case, we can pick either group 1 or group 2 (since $N_1 = N_2 = 12$) and then repeat the iteration. We picked h_1 from group 1 and put its neighboring regions contained within W_s into W_1; namely, W_1 = {19,20,22,23,25,26}.

We calculate 3 more normal vectors to the 3 sections in h_1 (see Fig. 4(d)) by calculating 3 new $\underset{\sim}{\alpha}$'s and β's. The resulting sign tests show W_s remained unchanged. We proceed to h_2 and put W_1 = {21,24,27,20,23,26} (see Fig. 4(d)). Again, we need to calculate 3 more normal vectors to h_2 by calculating 3 more $t\beta$'s. Again the resulting sign tests show W_s remained unchanged.

We proceed next to h_3 with W_1 = {19,20,21,22,23,24} (see Fig. 4(d)). We calculated 3 new $\underset{\sim}{\alpha}$'s and β's to implement the sign tests. The result shows regions 19,20,21 failed the test and these regions are discarded from W_s. Hence the new W_s is {22,23,24,25,26,27}.

We proceed to h_4 with W_1 = {22,23,24,25,26} and after the sign test, we found W_s = {22,23,24}.

Having exhausted all hyperplanes at this point, the iteration is terminated with the conclusion that (3.20) has exactly 3 solutions corresponding to the 3 regions 22,23, and 23 left in W_s.

Finally, using equations (A.2) and (A.3) from Appendix to compute the Jacobians and offset vectors for these regions, we obtain the following solutions:

Region 22   Region 23   Region 24

$$\begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \frac{151}{15} \\ \frac{11}{5} \\ \frac{43}{15} \end{bmatrix}, \quad \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{10} \\ \frac{11}{5} \\ \frac{43}{15} \end{bmatrix}, \quad \begin{bmatrix} v_1 \\ i_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \frac{149}{15} \\ \frac{11}{5} \\ \frac{43}{15} \end{bmatrix}$$

To summarize, we need only solve a total of 9+3+3 = 15 systems of linear equations compared to the 36 needed in the "brute force" method. The additional computation needed to carry out the sign tests is generally insignificant compared to that of solving systems of new linear equations repeatedly, especially when $k_i \gg 1$ for all $i = 1,2,\cdots,n$. In other words, we expect the efficiency of our algorithm to increase as the number of segments per piecewise-linear resistor increases.

3.3. The algorithm

We now summarize our discussions in the previous sections and state the complete algorithm formally for the most general case.

Assume all coefficients in equation (2.6) are given.

Step 0 (initialization)

(1) Let k_i denote the number of parallel hyperplanes orthogonal to coordinate axis x_i where $k_i \geq 0$, $i = 1,2,\cdots n$, compute.

$$N_j = \prod_{\substack{i=1 \\ i\neq j}}^{n} (k_i+1) \qquad i = 1,2,\cdots\cdot n \tag{3.31}$$

Reorder the index j so that $N_1 \leq N_2 \leq \cdots\cdot \leq N_n$

(2) In each group j, reorder the indices in $\{\beta_{ji} | i = 1,2,\cdots\cdot k_j\}$ so that $\beta_{j1} < \beta_{j2} < \cdots\cdot < \beta_{jkj}$.

Comment: We assume all hyperplanes are distinct. This implies that all β_{ji} are different.

Let h_{ji} be the hyperplane which corresponds to β_{ji}. Rearrange the hyperplanes in alternating order:

$$h_{j1}, h_{j3}, h_{j5}, \cdots\cdot h_{j2}, h_{j4}, \cdots\cdot$$

Label all hyperplanes from 1 to p where $p = \sum_{i=1}^{n} k_i$ so that

$$h_1 = h_{11}, h_2 = h_{13}, \cdots\cdot h_p = h_{pkn} .$$

(3) Let W_0 denote a list which contains all regions, and let W_s denote a list which is initially empty.

Set i = 1, go to step 1.

Step 1.

Form a sublist W_1 of $W_0 \cup W_s$ such that W_1 contains all neighborhood regions of the i-th hyperplane in $W_0 \cup W_s$. Replace W_0 by $W_0 - W_1$ and W_s by $W_s - W_1$, where "-" denotes the usual set difference. Let m be the total number of regions in $\overline{W}_1$.

Step 2

If m = 0, go to step 5; otherwise pick an arbitrary region R from $\overline{W}_1$ and consider the section σ of the i-th hyperplane such that σ is the boundary of region R. Find whether $\underset{\sim}{\alpha}_i'$ has previously been computed by checking the parallel sections in the parallel hyperplane group where i-th hyperplane belongs. If $\underset{\sim}{\alpha}_i'$ has been computed, then compute $t\beta_i'$ for the section σ using the technique described in (3.24) and (3.28); otherwise compute $\underset{\sim}{\alpha}_i'$ and β_i' using (3.16) and (3.17) respectively and store the computed $\underset{\sim}{\alpha}_i'$. Go to step 3.

Step 3

Pick an arbitrary point $\underset{\sim}{x}$ in the interior of region R and compute $\underset{\sim}{y} = \underset{\sim}{f}(\underset{\sim}{x})$. Perform the sign test (3.18) on region R. If the result is true, put R on list W_s; otherwise discard region R. Decrease m by 1, go to step 4.

Step 4

Search in the list W_1 the neighborhood region $\tilde{R}$ of R which share the same boundary σ. If it exists, repeat step 3 for $\tilde{R}$ and decrease m by 1. Go to Step 2.

Step 5

Increment i by 1. If $i \leq p$, then go to step 1, otherwise go to step 6.

Step 6

If list W_s is empty, then (2.6) has no solution; otherwise for each region in W_s, compute the Jacobian matrix $\underset{\sim}{J}$ and the offset vector $\underset{\sim}{s}$ using equation (A.2) and (A.3) in the Appendix respectively. The solution in the region is then given by $-\underset{\sim}{J}^{-1}\underset{\sim}{s}$.

4. ILLUSTRATIVE EXAMPLES:

We have programmed the algorithm described in section 3.3 using the "C programming language" on a PDP-11/780 VAX computer running a UNIX time-sharing operating system.[7] The following examples are generated using this program.

Example 3.

Consider the same circuit shown in Fig. 1(a) except that R1 and R2 are represented by (4.1) and (4.2), respectively.

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

$$i_1 = -\frac{125}{8} + \frac{9}{8} v_1 + \frac{7}{8} |v_1+1| - \frac{3}{2} |v_1-2| + \frac{3}{4} |v_1-5| - \frac{1}{8} |v_1-11| - \frac{9}{8} |v_1-13| + 2|v_1-15| \quad (4.1)$$

$$i_2 = \frac{29}{4} + \frac{3}{2} v_2 - \frac{3}{2} |v_2+8| + \frac{3}{2} |v_2+5| - \frac{3}{2} |v_2+3| + \frac{3}{2} |v_2+1| - \frac{3}{4} |v_2-3| - \frac{5}{4} |v_2-8|$$
$$+ \frac{3}{2} |v_2-10| + |v_2-13| - \frac{5}{4} |v_2-16| + \frac{1}{4} |v_2-18| \quad (4.2)$$

The associated circuit equations can be expressed in the following canonical form:

$$f(x) = \begin{bmatrix} -\frac{161}{8} \\ \frac{11}{4} \end{bmatrix} + \begin{bmatrix} \frac{13}{8} & \frac{1}{2} \\ \frac{1}{2} & 2 \end{bmatrix} x + \sum_{i=1}^{16} c_i |\langle \alpha_i, x \rangle - \beta_i| = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (4.3)$$

where

$$x \triangleq \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} ,$$

$$\alpha_i = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{for } i = 1,2,\cdots 6 \text{ and } \alpha_i = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ for } i = 7,8,\cdots 16$$

$$c_1 = \begin{bmatrix} \frac{7}{8} \\ 0 \end{bmatrix}, c_2 = \begin{bmatrix} -\frac{3}{2} \\ 0 \end{bmatrix}, c_3 = \begin{bmatrix} \frac{3}{4} \\ 0 \end{bmatrix}, c_4 = \begin{bmatrix} -\frac{1}{8} \\ 0 \end{bmatrix}, c_5 = \begin{bmatrix} -\frac{9}{8} \\ 2 \end{bmatrix}$$

$$c_6 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, c_7 = \begin{bmatrix} 0 \\ -\frac{3}{2} \end{bmatrix}, c_8 = \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix}, c_9 = \begin{bmatrix} 0 \\ -\frac{3}{2} \end{bmatrix}, c_{10} = \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix}$$

$$c_{11} = \begin{bmatrix} 0 \\ -\frac{3}{4} \end{bmatrix}, c_{12} = \begin{bmatrix} 0 \\ -\frac{5}{4} \end{bmatrix}, c_{13} = \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix}, c_{14} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, c_{15} = \begin{bmatrix} 0 \\ -\frac{5}{4} \end{bmatrix}$$

$$c_{16} = \begin{bmatrix} 0 \\ \frac{1}{4} \end{bmatrix}$$

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

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

Note that the domain $\mathbb{R}^2$ is partitioned into 77 regions by 6 parallel 1-dimensional hyperplanes (vertical lines) $H_1, H_2, \cdots, H_6$ along the x_1-axis, and by 10 parallel hyperplanes (Horizontal lines) $H_7, H_8, \cdots, H_{16}$ along the x_2-axis, as shown in Fig. 6. Hence $k_1 = 6$, $k_2 = 10$ and $N_1 = 11$, $N_2 = 7$ in (3.29). We arrange the hyperplanes in the following alternating order:

$$\{\underbrace{H_7, H_9, H_{11}, H_{13}, H_{15}, H_8, H_{10}, H_{12}, H_{14}, H_{16}}_{\text{first parallel hyperplane group}}, \underbrace{H_1, H_3, H_5, H_2, H_4, H_6}_{\text{second parallel hyperplane group}}\}$$

This completes the initialization step. We also label the regions from R_1 to R_{77} in (Fig. 6) for easy identification.

We start the <u>sign test</u> in the neighborhood regions of H_7, namely; R_1 through R_{14}. Since none of the regions passes the test, they are deleted. We note that H_7 was partitioned into 7 sections by H_1 through H_6. So we have computed 7 α_k's to accomplish the <u>sign test</u>.

Next, we perform the <u>sign test</u> on the neighborhood regions of H_9, which are R_{15} through R_{28}. Note that we need not compute any new α_k' since H_9 is parallel to H_7. Test results showed R_{20}, R_{21}, R_{27} and R_{28} were put in set W_s.

Continuing the iteration on H_{11}, we found regions R_{29} through R_{35} were put in set W_s; on H_{15}, regions R_{57} through R_{63} were put in W_s. At the end of iteration on H_{15}, W_0 contains R_{71} through R_{77} and W_s contains the following regions:

$$\{R_{20}, R_{21}, R_{27}, R_{28}, R_{29}, R_{30}, R_{31}, R_{32}, R_{33}, R_{34}, R_{35}, R_{57}, R_{58}, R_{59}, R_{60}, R_{61}, R_{62}, R_{63}\}$$

We continue to iterate on H_8 through H_{16}, and eliminated R_{21}, R_{28}, R_{29}, R_{35}, R_{57}, R_{59}, R_{60} and R_{63} from W_s, and R_{71} through R_{77} from W_0. At the end of iteration on the first parallel hyperplane group, W_0 is empty (i.e., we have scanned all regions once) and W_s contains the following regions:

$$\{R_{20}, R_{27}, R_{30}, R_{31}, R_{32}, R_{33}, R_{34}, R_{58}, R_{61}, R_{62}\}$$

Note that these are the <u>only</u> regions left to be tested in the second parallel hyperplane group.

The first hyperplane in the second parallel group is H_1. Since W_0 is now an empty set, the only neighborhood regions of H_1 on $W_0 \cup W_s$ are R_{30} and R_{59}. Therefore we need only to compute 2 new α_k's.

Note that the α_k' calculated for the section serving as boundary of R_{30} can be used for R_{31} through R_{34}. Similarly, the α_k' calculated for the section serving as boundary of R_{58} can be used for R_{61} and R_{62}. Therefore, for all sections of the hyperplanes in the second group, only 3 new α_k's need to be calculated (the third one was for R_{20}).

At the end of iteration on the second parallel group, W_s contains R_{30}, R_{31} and R_{32}, and step 6 gives the following three solutions: $\begin{bmatrix} \frac{3}{2} \\ \frac{3}{2} \end{bmatrix}$ in R_{30}, $\begin{bmatrix} 4 \\ 1 \end{bmatrix}$ in R_{31}, and $\begin{bmatrix} \frac{17}{3} \\ \frac{2}{3} \end{bmatrix}$ in R_{32} .

Remark:

To help us keep track of the results of the sign test, some regions in Fig. 6 are marked with one or more asterisks. A "*" near a boundary means the sign test associated with that boundary in the region is "positive".[8] For example, in region R_{33}, three *'s are marked close to the boundaries H_{11}, H_{10} and H_5. This means that for boundaries H_{11}, H_{10} and H_5, the results of the sign test are all positive. However, since there is no * for H_4, the sign test is negative there. Hence, R_{33} contains no solution of (4.3).

Example 4.

Consider the four-transistor multi-state circuit shown in Fig. 7(a) [15]. Each transistor is modeled by a controlled source in series with a p-n junction diode as shown in Fig. 7(b). The diode I_D-V_D characteristic is approximated by a continuous piecewise-linear function with two segments as shown in Fig. 7(c). The canonical representation of the piecewise-linear function is:

$$I_D = f(v_D) = -1.29052\text{x}10^{-2} + 3.9708313\text{x}10^{-2} v_D + 3.9708313\ |v_D - 0.325|$$

The associated circuit equations can be expressed in the following canonical form:

$$\underset{\sim}{f}(\underset{\sim}{x}) = \begin{bmatrix} -1.27712 \\ -1.69119 \\ -1.27712 \\ -1.67119 \end{bmatrix} + \begin{bmatrix} 2.42347 & 1.18058 & 0 & 0 \\ 1.47556 & 2.62869 & 0.28796 & 0.19854 \\ 0 & 0 & 2.42347 & 1.18058 \\ 0.28796 & 0.19854 & 1.47556 & 2.62859 \end{bmatrix} \underset{\sim}{x} \qquad (4.4)$$

[8]For simplicity, we say that sign test is positive for a given section σ if (3.18) holds in σ. Otherwise, it is negative.

$$+\sum_{i=1}^{4} \underset{\sim}{c}_i \, |\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x}\rangle - \beta_i| = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where

$$\underset{\sim}{x} \triangleq \begin{bmatrix} v_{D1} \\ v_{D2} \\ v_{D3} \\ v_{D_4} \end{bmatrix}, \; \underset{\sim}{\alpha}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \; \underset{\sim}{\alpha}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \; \underset{\sim}{\alpha}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \; \underset{\sim}{\alpha}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

$$\underset{\sim}{c}_1 = \begin{bmatrix} 2.42347 \\ 1.42156 \\ 0 \\ 0.27796 \end{bmatrix}, \; \underset{\sim}{c}_2 = \begin{bmatrix} 1.13692 \\ 2.62869 \\ 0 \\ 0.19854 \end{bmatrix}, \; \underset{\sim}{c}_3 = \begin{bmatrix} 0 \\ 0.27796 \\ 2.42347 \\ 1.42156 \end{bmatrix}, \; \underset{\sim}{c}_4 = \begin{bmatrix} 0 \\ 0.19854 \\ 1.13692 \\ 2.62869 \end{bmatrix},$$

$$\beta_1 = \beta_2 = \beta_3 = \beta_4 = 0.325.$$

Using our program, all nine solutions of the circuit are found and the result is listed in Table 1.

The number of regions eliminated by the sign test does not look impressive in this example because we have approximated each diode by only 2 segments so that the reader can check the results manually. However, the efficiency of our algorithm becomes apparent as we increase the number of segments of the piecewise-linear characteristic, as shown in Figs. 7(d)-7(g) for 3,4,5 and 6 segments respectively.

The calculations corresponding to different number of breakpoints (Column 1) and segments (Column 2) is summarized in Table 2. Note that for large k_i, the number of linear system of equations that must be solved using our algorithm is significally smaller than that of using the "brute-force" combinatorial method; namely, $\prod_{i=1}^{n} (k_i+1)$. Note that the higher k_i is, the more efficient our algorithm becomes.

5. ANALYSIS OF ILL-CONDITIONED CASES

So far we have assumed that system (2.6) behaves rather well in the sense that the algorithm can be carried out without difficulty. For example, the matrix $\overline{\underset{\sim}{B}}_{k_{\sigma_a \sigma_b}}$ in (3.13) is assumed to be nonsingular; and the scalar $\beta'_{k_{\sigma_a \sigma_b}}$ in (3.18) is assumed to be nonzero, etc. But this may not be true in general.

In this section, we will exhibit some ill-conditioned examples where the above assumptions are violated so that the sign test can not be performed.

We will analyze these ill-conditioned cases in detail and offer a remedy in each case.

5.1. Ill-conditioned case 1: matrix $\bar{B}_{k_{\sigma_a\sigma_b}}$ is singular (Example 5)

Consider the circuit shown in Fig. 8(a). Both R1 and R2 are voltage-controlled with constitutive relation $i_j = g(v_j)$, $j = 1,2$, where $g(\cdot)$ is shown in Fig. 8(b). The circuit equation can be expressed in the following canonical form:

$$f(x) = \begin{bmatrix} -\frac{5}{2} \\ -\frac{5}{2} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x + \begin{bmatrix} -\frac{1}{2} \\ 0 \end{bmatrix} |\langle \alpha_1, x \rangle - 1| + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix} |\langle \alpha_2, x \rangle - 2|$$

$$+ \begin{bmatrix} 0 \\ -\frac{1}{2} \end{bmatrix} |\langle \alpha_3, x \rangle - 1| + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix} |\langle \alpha_4, x \rangle - 2| = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.1}$$

where $x \triangleq \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$, $\alpha_1 = \alpha_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\alpha_3 = \alpha_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

The partition in the domain of $f(\cdot)$ and its image in the range space are shown in Figs. 8(c) and 8(d) respectively. The singularity of matrices $\bar{B}_{AC}$, $\bar{B}_{AB}$, $\bar{B}_{BD}$, and $\bar{B}_{CD}$ give rise to the following degenerate behavior: the interiors of regions R_2, R_4, R_5, R_6 and R_8 as well as their boundaries AB, AC, BD and CD have shrunk into a single point P in the range space. Since point P does not coincide with the origin in the range space, there is no solution of (5.1) in these degenerate regions.

However, the <u>sign test</u> is applicable in the 4 corner regions and the test results show that region R_9 contains a solution of (5.1). This conclusion can also be verified by inspection of Fig. 8(d), where the image of R_9 is the only region which contains the origin. The corresponding solution is

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}.$$

This example suggests the following method for overcoming "ill-conditioned case 1:" If we encounter a singular $\bar{B}_{k_{\sigma_a\sigma_b}}$, consider instead the equation

$$\bar{B}_{k_{\sigma_a\sigma_b}} x + \bar{a}_{k_{\sigma_a\sigma_b}} = 0 \tag{5.2}$$

where $\bar{a}_{k_{\sigma_a\sigma_b}}$ is computed from (3.12). Using standard techniques from linear system theory [26], determine if $\bar{a}_{k_{\sigma_a\sigma_b}}$ is in the range space of $\bar{B}_{k_{\sigma_a\sigma_b}}$. If it is, then all solutions of (5.2) lying within the definition of regions R_a and R_b (referring to Fig. 3(a)) are solutions of (2.6). This unusual situation corresponds to the case where point p in Fig. 8(d) coincides with the origin.

On the other hand, if $\bar{a}_{k_{\sigma_a\sigma_b}}$ is not in the range space of $\bar{B}_{k_{\sigma_a\sigma_b}}$, then (2.6) has no solution in regions R_a and R_b. This situation corresponds to the case in Fig. 8(d) where point p does not coincide with the origin.

For efficient computer implementations we will now derive a useful property for checking the singularity of $\bar{B}_{k_{\sigma_a\sigma_b}}$. Although matrix $\bar{B}_{k_{\sigma_a\sigma_b}}$ is a constant matrix, it is obtained from (2.6) by restricting $x \in \mathbb{R}^n$ to certain section $\sigma_a\sigma_b$ (i.e., equation (3.13) in a given region. In fact, $\bar{B}_{k_{\sigma_a\sigma_b}}$ is just the Jacobian matrix of f(•) evaluated in section $\sigma_a\sigma_b$. Since $\bar{B}_{k_{\sigma_a\sigma_b}}$ will vary from one section to another; let us write $\bar{B}_{k_{\sigma_a\sigma_b}}$ as follows:

$$\bar{B}_{k_{\sigma_a\sigma_b}} = \frac{d}{dx} f(x)\Big|_{x\in\sigma_a\sigma_b} \tag{5.2}$$

Note that if we let $\langle \alpha_k, x\rangle - \beta_k = 0$ in (2.6) before we compute the Jacobian matrices of $f(\cdot)$ in regions R_a and R_b (Fig. 3(a)), then the results will be $J_a|_{x\in\sigma_a\sigma_b}$ and $J_b|_{x\in\sigma_a\sigma_b}$, respectively. Since $f(\cdot)$ is continuous, we must have $J_a|_{x\in\sigma_a\sigma_b} = J_b|_{x\in\sigma_a\sigma_b} = \bar{B}_{k_{\sigma_a\sigma_b}}$.

Since $\sigma_a\sigma_b$ is of 1 lower dimension than $\mathbb{R}^n$, we have

$$\text{null}(\bar{B}_{k_{\sigma_a\sigma_b}}) \subset \text{null}(J_a) \cap \text{null}(J_b) \tag{5.3}$$

where null(•) denotes "the null space of" (•). We can interpret (5.3) as follow:

<u>Property 1</u>.

For any <u>continuous</u> piecewise-linear function $f(\cdot)$, if $\det\left[\frac{d}{dx} f(x)\Big|_{x\in\sigma_a\sigma_b}\right] = 0$

then, using the notation of Fig. 3(a), $\underset{\sim}{f}(\cdot)$ is singular in both R_1 and R_2.

Property 1 implies that if det $\underset{\sim}{J}_a \neq 0$ or det $\underset{\sim}{J}_b \neq 0$, then det $\overline{\underset{\sim}{B}}_{k_{\sigma_a\sigma_b}} \neq 0$.

To illustrate the application of this property, consider the regions in Fig. 8(c). Since $\overline{\underset{\sim}{B}}_{AC}$, $\overline{\underset{\sim}{B}}_{AB}$, $\overline{\underset{\sim}{B}}_{BD}$ and $\overline{\underset{\sim}{B}}_{CD}$ are singular, by Property 1, $\underset{\sim}{J}_2$, $\underset{\sim}{J}_4$, $\underset{\sim}{J}_5$, $\underset{\sim}{J}_6$ and $\underset{\sim}{J}_8$ must also be singular, as is easily verified by inspection of the Jacobian matrix in each region of Fig. 8(c). On the other hand, since $\underset{\sim}{J}_9$ is nonsingular (as well as J_1, J_3 and J_7), it follows from Property 1 that $\overline{\underset{\sim}{B}}_{DC}$, $\overline{\underset{\sim}{B}}_{DF}$ (as well as $\overline{\underset{\sim}{B}}_{Aa}$, $\overline{\underset{\sim}{B}}_{Ab}$, $\overline{\underset{\sim}{B}}_{Bc}$, $\overline{\underset{\sim}{B}}_{Bd}$, $\overline{\underset{\sim}{B}}_{Cg}$ and $\overline{\underset{\sim}{B}}_{Ch}$) must also be nonsingular. This conclusion allows our program to perform the sign test in regions R_9 (as well as in R_1, R_3 and R_7).

5.2. Ill-Conditioned Case 2 $(\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y}_a \rangle - \beta'_{k_{\sigma_a\sigma_b}} = 0)$ and Ill-Conditioned Case 3 $(\beta'_{k_{\sigma_a\sigma_b}} = 0)$

From here on, we assume that matrix $\overline{\underset{\sim}{B}}_{k_{\sigma_a\sigma_b}}$ is nonsingular, and that we have computed $\underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}$ and $\beta'_{k_{\sigma_a\sigma_b}}$ from (3.16) and (3.17) respectively. Consider Figs. 3(a) and 3(b), let

$$\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, y \rangle = \beta'_{k_{\sigma_a\sigma_b}} \tag{5.4}$$

denote the equation of the hyperplane containing the section $\sigma'_a\sigma'_b$ in the range space. Let $\underset{\sim}{x}_a$ be an arbitrary interior point in region R_a and let $\underset{\sim}{y}_a = \underset{\sim}{f}(\underset{\sim}{x}_a)$. Let $\underset{\sim}{J}_a$ denote the Jacobian matrix of $\underset{\sim}{f}(\cdot)$ in region R_a.

Property 2.

det $\underset{\sim}{J}_a = 0$ if and only if

$$\langle \underset{\sim}{\alpha}'_{k_{\sigma_a\sigma_b}}, \underset{\sim}{y}_a \rangle = \beta'_{k_{\sigma_a\sigma_b}} \tag{5.5}$$

Proof:

Necessity (only if): If det $\underset{\sim}{J}_a = 0$, then the interior of R'_a collapses into its boundaries. The degree of degeneration depends on the rank of $\underset{\sim}{J}_a$, and the highest dimension of R'_a can not exceed n-1 where n is the dimension of $\underset{\sim}{J}_a$.

Sufficiency (if): Let $\langle \alpha_k, x \rangle = \beta_k$ be the equation of the hyperplane in the domain containing $\sigma_a\sigma_b$. Write $f(x) = J_a x + s_a$ for $x \in R_a$ and suppose that $\det J_a \neq 0$.

Since $\sigma_a\sigma_b$ is a subset of R_a and $f(\cdot)$ is continuous, we can rederive equation (3.16) and (3.17) with J_a in place of $\bar{B}_{k_{\sigma_a\sigma_b}}$ and s_a in place of $\bar{a}_{k_{\sigma_a\sigma_b}}$. Thus we have

$$\alpha'_{k_{\sigma_a\sigma_b}} = (J_a^{-1})^T \alpha_k \tag{5.6}$$

$$\beta'_{k_{\sigma_a\sigma_b}} = \beta_k + \langle \alpha'_{k_{\sigma_a\sigma_b}}, s_a \rangle \tag{5.7}$$

Substituting y_a by $J_a x + s_a$ and $\beta'_{k_{\sigma_a\sigma_b}}$ by (5.7), equation (5.5) becomes

$$\langle \alpha'_{k_{\sigma_a\sigma_b}}, J_a x_a + s_a \rangle = \beta_k + \langle \alpha'_{k_{\sigma_a\sigma_b}}, s_a \rangle$$

Cancelling $\langle \alpha'_{k_{\sigma_a\sigma_b}}, s_a \rangle$ from both sides, we get $\langle \alpha'_{k_{\sigma_a\sigma_b}}, J_a x_a \rangle = \beta_k$, or

$$\langle J_a^T \alpha'_{k_{\sigma_a\sigma_b}}, x_a \rangle = \beta_k$$

But (5.6) implies $J_a^T \alpha'_{k_{\sigma_a\sigma_b}} = \alpha_k$. Therefore we have $\langle \alpha_k, x_a \rangle = \beta_k$, which is absurd since x_a was assumed to be an interior point in R_a. Hence we must have $\det J_a = 0$. ¤

Example 6.

Consider the circuit shown in Fig. 9(a). Let R1 and R2 be the same as in Example 4. The circuit equation can be expressed in the following canonical form:

$$f(x) = \begin{bmatrix} -\frac{5}{2} \\ -\frac{5}{2} \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} x + \begin{bmatrix} -\frac{1}{2} \\ 0 \end{bmatrix} |\langle \alpha_1, x \rangle - 1| + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix} |\langle \alpha_2, x \rangle - 2|$$

$$+ \begin{bmatrix} 0 \\ -\frac{1}{2} \end{bmatrix} |\langle \alpha_3, x \rangle - 1| + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix} |\langle \alpha_4, x \rangle - 2| = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.8}$$

The partition in the domain of $\underset{\sim}{f}(\cdot)$ and its image in the range space are shown in Figs. 9(b) and 9(c) respectively. Note that ill-conditioned case 2 manifests itself in Fig. 9(c) with region R_5 degenerating into the line segment A'D'. This ill-conditioning follows of course from Property 2, since det $\underset{\sim}{J}_5 = 0$.

In general, if ill-conditioned case 2 occurs, we need to examine the value of $\beta'_{k_{\sigma_a\sigma_b}}$.

If $\beta'_{k_{\sigma_a\sigma_b}} \neq 0$, which is geometrically related to the fact that the hyperplane in the range space containing $\sigma'_a\sigma'_b$ (Fig. 3(a)) does not pass through the origin, there is no solution in $\sigma_a\sigma_b$ or in its degenerate neighborhood region.

If $\beta'_{k_{\sigma_a\sigma_b}} = 0$ (i.e., ill-conditioned case 3), the hyperplane in the range space containing $\sigma'_a\sigma'_b$ must pass through the origin. In this case, we need to consider the following subcases:

Case a: $\underset{\sim}{J}_a$ or $\underset{\sim}{J}_b$ or both are singular.

If $\underset{\sim}{J}_a$ is singular, form a set $N_{J_a} \triangleq \{x \in \mathbb{R}^n | \underset{\sim}{J}_a \underset{\sim}{x} + \underset{\sim}{s}_a = \underset{\sim}{0}\}$. ($\underset{\sim}{s}_a$ is calculated Using (A.3) in the Appendix). Then all $\underset{\sim}{x} \in N_{J_a} \cap R_a$ are solutions of (2.6).

If $\underset{\sim}{J}_b$ is singular, form $N_{J_b} \triangleq \{\underset{\sim}{x} \in \mathbb{R}^n | \underset{\sim}{J}_b \underset{\sim}{x} + \underset{\sim}{s}_b = \underset{\sim}{0}\}$ and all $\underset{\sim}{x} \in N_{J_b} \cap R_b$ will be solutions of (2.6).

Case b. Both $\underset{\sim}{J}_a$ and $\underset{\sim}{J}_b$ are nonsingular.
Solve (5.2) directly to obtain $\underset{\sim}{x}^* = -\bar{\underset{\sim}{B}}^{-1}_{k_{\sigma_a\sigma_b}} \bar{\underset{\sim}{a}}_{k_{\sigma_a\sigma_b}}$ (recall that by Property 1, $\bar{\underset{\sim}{B}}_{k_{\sigma_a\sigma_b}}$ is nonsingular). If $\underset{\sim}{x}^* \in \sigma_a\sigma_b$, then it is the solution of (2.6). Otherwise, continuity of $\underset{\sim}{f}(\cdot)$ implies that (2.6) has no solution on $\sigma_a\sigma_b$, as well as in either R_a or R_b.

Let us illustrate the above method using Example 6. Since $\underset{\sim}{J}_5$ is singular, we form (case a) $N_{J_5} = \{\underset{\sim}{x} \in \mathbb{R}^2 | x_1 + x_2 - 1 = 0\}$. Since $N_{J_5} \cap R_5$ = empty set, there is no solution of (5.8) in R_5. This can also be verified graphically in Fig. 9(c). Note that even though the line containing segment A'D' passes through the origin, segment A'D' itself does not contain the origin.

The sign test remains valid in the remaining regions. The resulting calculation shows that (5.8) has a single solution

$$\begin{bmatrix} v_1 \\ \vdots \\ v_2 \end{bmatrix} = \begin{bmatrix} \frac{2}{3} \\ \frac{2}{3} \end{bmatrix}$$, which is located in region R_1.

In the next example, we will show a similar case where $\beta'_{k_{\sigma_a \sigma_b}} = 0$. However, this time the circuit exhibits infinitely many solutions.

Example 7.

Consider again the simple circuit shown in Fig. 2(a). Let R1 and R2 be the same as in Example 1, but the value of the linear resistor is changed from 2Ω to 0.5 Ω and the value of the dc voltage source is changed from 9v to 6v (see Fig. 10(a)). Now the load line formed by the linear resistor and the dc voltage source coincides with the middle segment of the driving-point plot of the one port made up of the series connection of R1 and R2 (see Fig. 10(b)). Therefore the circuit must have infinitely many solutions. The circuit equation is expressed in the following canonical form:

$$\underset{\sim}{f}(\underset{\sim}{x}) = \begin{bmatrix} -\frac{39}{8} \\ 3 \end{bmatrix} + \begin{bmatrix} 1 & \frac{13}{8} \\ -\frac{5}{4} & \frac{5}{4} \end{bmatrix} \underset{\sim}{x} + \begin{bmatrix} 0 \\ \frac{3}{2} \end{bmatrix} |\langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle - 2| + \begin{bmatrix} 0 \\ -\frac{3}{4} \end{bmatrix} |\langle \underset{\sim}{\alpha}_2, \underset{\sim}{x} \rangle - 5|$$

$$+ \begin{bmatrix} -\frac{3}{8} \\ -\frac{3}{4} \end{bmatrix} |\langle \underset{\sim}{\alpha}_3, \underset{\sim}{x} \rangle - 3| = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The partition in the domain of f(•) and its image in the range space are shown in Figs. 10(c) and 10(d) respectively.

Note that (5.5) holds in this case because region R_2 degenerates into a half line A'b' (or equivalently B'c') in the range space. Now since $\underset{\sim}{J}_2$ is singular, we form

$$N_{J_2} \triangleq \{\underset{\sim}{x} \in \mathbb{R}^2 | \underset{\sim}{J}_2 \underset{\sim}{x} + \underset{\sim}{s}_2 = \underset{\sim}{0}\} = \{\underset{\sim}{x} \in \mathbb{R}^2 | x_1 + 2x_2 - 6 = 0\}$$

and $N_{J_2} \cap R_2 = \{\underset{\sim}{x} \in \mathbb{R}^2 | \underset{\sim}{x} = \begin{bmatrix} q \\ 3 - \frac{q}{2} \end{bmatrix}, 2 \le q \le 5, q \in \mathbb{R}\}$. Therefore, the solutions of (5.9) are given by

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} q \\ 3 - \frac{q}{2} \end{bmatrix}, \quad 2 \le q \le 5, \quad q \in \mathbb{R}$$

This is shown in the shaded region (including the boundaries) in Fig. 10(c).

6. Computational Efficiency

We will discuss the computational efficiency of our algorithm in this section. First, we assume the given system (2.6) is well-behaved so that we can exclude ill-conditioned cases. Then we will compare the efficiency of our algorithm with that of the "brute-force" combinatorial algorithm [14], where $\underset{\sim}{J}\underset{\sim}{x} + \underset{\sim}{s} = \underset{\sim}{0}$, must be solved in each region. A reasonable figure of merit to be used in the comparison is the total number of linear systems of equations needed to be solved until all solutions are found.

Let n be the number of parallel hyperplane groups. Let $k_i, i = 1,2,\ldots n$, be the number of parallel hyperplanes in the i-th group. Then the total number of linear systems needed to be solved in the "brute-force" combinatorial algorithm is equal to the total number of regions; namely,

$$\prod_{i=1}^{n} (k_i+1) \tag{6.1}$$

For the algorithm stated in section 3.3, the number for the worst case is found to be

$$\sum_{j=1}^{n} N_j + \text{total number of solutions} \tag{6.2}$$

where N_j is defined in (3.29) or (3.31).

For circuits exhibiting multiple solutions, the exact number of solutions is usually impossible to predict. Indeed, comparing Example 7 with Example 1, we note that as we change the value of the linear resistor and the constant voltage source slightly, the number of solutions can change drastically. From the practical point of view, however, the number of solutions is usually very much smaller compared to the first term in (6.2).

Hence, comparing only the first term in (6.2) with (6.1), we get

$$\prod_{i=1}^{n} (k_i+1) - \sum_{j=1}^{n} \prod_{\substack{i=1 \\ i\neq j}}^{n} (k_i+1) = \prod_{i=1}^{n} (k_i+1)\left[1 - \sum_{j=1}^{n} \frac{1}{k_j+1}\right] \tag{6.3}$$

Equation (6.3) implies that if $k_j + 1 > n$, $j = 1,2,\ldots.n$, then the left-hand side of (6.3) will always be positive. Since k_j is also equal to the number of breakpoints in the j-th piecewise-linear resistor in the circuit, then $k_j + 1 > n$ means that if the total number of segments in each piecewise-linear resistor is greater than the total number of piecewise-linear resistors, then the worst case figure of our algorithm will be smaller than that of the combinatorial algorithm. Fortunately, the worst case number $\sum_{j=1}^{n} N_j$ is seldom achieved in practical circuits. As illustrated in Examples 2, 3, and 4, most of the regions are eliminated by the sign test before the iteration reaches the second group of parallel hyperplane.

We have already given the comparison data in Table 1 for Example 4. The corresponding data for Examples 2 and 3 is given in Table 3.

In Table 4 we list the total CPU time consumed for each example. Since the UNIX operating system is a time-sharing system, the actual time consumed depends on the current load on the system at that time. Hence, we give only a range of the total CPU time. The data is obtained from 10 tries at various loading conditions. Although this quantity is not exact, it does give a realistic "ball park" figure.

7. CONCLUDING REMARKS

The algorithm presented on Section 3.3 and the combinatorial algorithm in [14] both scan all regions defined by the piecewise-linear function $\underset{\sim}{f}$ in (2.6). Therefore, both will find all solutions .

The worst case figure of our algorithm is given by the first term in (6.2). This overly conservative upper bound is achieved only when there is a solution to (2.6) in every region. In practice (e.g. Examples 2, 3 and 4), many regions will usually be eliminated during the early phase of the iteration; i.e., from the very first few groups of parallel hyperplanes. Hence, our algorithm is indeed quite efficient in solving practical circuits.

Although originally developed for nonlinear circuits, our algorithm is applicable to any system of piecewise-linear equations which can be expressed in the canonical form (2.6), where $\underset{\sim}{\alpha}_i$ denotes unit vectors.

Finally we remark that since it is possible for a piecewise-linear equation to have a solution in every region, any algorithm capable of finding all solutions must necessarily scan through all possible regions.

References

[1] R. O. Nielsen and A. N. Willson, Jr., "A fundamental result concerning the topology of transistor circuits with multiple equilibria," Proceedings of the IEEE, vol. 68, no. 2, pp. 196-208, Feb. 1980.

[2] T. T. Dao, "Recent multi-valued circuits," IEEE COMPCON conference Digest, pp. 194-199, Feb. 1981.

[3] D. Etiemble, "Multi-valued integrate circuits for signal transmission," IEEE COMPCON conference Digest, pp. 205-208, Feb. 1981.

[4] Z. G. Vranesic, "Application and scope of multiple-valued LSI technology," IEEE COMPCON conference Digest, pp. 213-216, Feb. 1981.

[5] Z. G. Vranesic and K. S. Smith, "Engineering aspects of multi-valued logic systems," Computer, vol 9, pp. 34-41, 1974.

[6] D. C. Rine, Computer Science and Multiple-valued Logic, North-Holland Publishing Co., 1977.

[7] R. M. Bozorth, Ferro-Magnetism, D. Van Norstrand Company, Princeton, NJ, 1951.

[8] V. M. Fridkin, Ferroelectric Semiconductors, (translated from Russian), Consultants Bureau, New York, NY, 1980.

[9] T. Van Duzer and C. W. Turner, Principles of Superconductive Devices and Circuits, Elsevier, New York, NY 1981.

[10] H. Hartnagel, Gunn-Effect Logic Devices, Heinemann Educational Books Ltd., London, 1973.

[11] L. O. Chua and Y. W. Sing, "A nonlinear lumped circuit model for Gunn Diodes," International Journal of Circuit Theory and Applications, vol. 6, pp. 375-408, Oct. 1978.

[12] M. Latif and P. Bryant, "Multiple equilibrium points and their significance in the second breakdown of bipolar transistors," IEEE J. Solid-State Circuits, vol. SC-16, no. 1, pp. 8-15, Feb. 1981.

[13] L. O. Chua, "Efficient computer algorithms for piecewise-linear analysis of resistive nonlinear networks," IEEE Trans. Circuit Theory, vol. CT-18, pp. 73-75, Jan. 1971.

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

[15] L. O. Chua and A. Ushida, "A switching-parameter algorithm for finding multiple solutions of nonlinear resistive circuits," Inter. J. Circuit Theory and Applications, vol. 4, pp. 215-239, 1976.

[16] K. S. Chao, D. K. Liu and C. T. Pan, "A systematic search method for obtaining multiple solutions of simultaneous nonlinear equations," IEEE Trans. Circuits and Systems, vol. CAS-22, pp. 748-753, Sept. 1975.

[17] 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.

[18] M. J. Chien, "Searching for multiple solutions of nonlinear systems," IEEE Trans. Circuits and Systems, vol. CAS-26, no. 10, pp. 817-827, Oct. 1979.

[19] 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.

[20] 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, pp. 1-5.

[21] L. O. Chua, Introduction to Nonlinear Network Theory, New York: McGraw Hill, 1969.

[22] L. O. Chua and P. A. Medlock, MECA - A User Oriented Computer Program for Analyzing Resistive Nonlinear Networks, vol. 1, User's Manual. Lafayette, Ind.: Purdue University, School of Electrical Engineering, Rept. TR-EE69-7, Apr. 1969.

[23] 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.

[24] 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.

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

Appendix

Let $\underset{\sim}{f}$ be represented in the canonical form (2.6). Since $\underset{\sim}{f}$ is piecewise-linear, it is <u>affine</u> in each region R_j. Hence, for any $\underset{\sim}{x} \in R_j$, $\underset{\sim}{f}$ can be written as $\underset{\sim}{J}_j \underset{\sim}{x} + \underset{\sim}{s}_j$. The matrix $\underset{\sim}{J}_j$ (often called the <u>Jacobian matrix</u> of $\underset{\sim}{f}$ in region R_j) and the vector $\underset{\sim}{s}_j$ (often called the <u>offset vector</u> of $\underset{\sim}{f}$ in region R_j) can be computed from the coefficients of (2.6). Here we derive two simple formulas for computing $\underset{\sim}{J}_j$ and $\underset{\sim}{s}_j$.

We can write (2.6) in the following form:

$$\underset{\sim}{f}(\underset{\sim}{x}) = \underset{\sim}{a} + \underset{\sim}{B}\underset{\sim}{x} + \underset{\sim}{C} \begin{bmatrix} |\langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle - \beta_1| \\ \vdots \\ |\langle \underset{\sim}{\alpha}_p, \underset{\sim}{x} \rangle - \beta_p| \end{bmatrix} \tag{A.1}$$

where
$$\underset{\sim}{C} = \begin{bmatrix} c_{11} & c_{21} & \cdots & c_{p1} \\ \vdots & \vdots & & \vdots \\ c_{1n} & c_{2n} & \cdots & c_{pn} \end{bmatrix}$$

Since we expect to express $\underset{\sim}{f}(\cdot)$ in the form of $\underset{\sim}{J}_j x + \underset{\sim}{s}_j$ for $\underset{\sim}{x}$ in the interior of region R_j, we can differentiate (A.1) with respect to $\underset{\sim}{x}$ to find $\underset{\sim}{J}_j$. Since $\underset{\sim}{x}$ is assumed to be an interior point in R_j, none of the terms $\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i$, $i = 1,2,\ldots.p$, will be zero. Carrying out the differentiation, we get:

$$\underset{\sim}{D}\underset{\sim}{f}(\underset{\sim}{x})\Big|_{x \in R_j} = \underset{\sim}{B} + \underset{\sim}{C}\, \text{diag}[\{\text{sgn}(\langle \underset{\sim}{\alpha}_i, x \rangle - \beta_i)\}_1^p] \begin{bmatrix} \underset{\sim}{\alpha}_1^T \\ \underset{\sim}{\alpha}_2^T \\ \vdots \\ \underset{\sim}{\alpha}_p^T \end{bmatrix}\Bigg|_{\underset{\sim}{x} \in R_j} \tag{A.2}$$

where $\text{diag}[\, \text{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)\}_1^p]$ is a pxp diagonal matrix with the i-th diagonal element being $\text{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)$. Here, $\text{sgn}(\cdot)$ denotes the sign function defined as follows

$$\mathrm{sgn}(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{if } z < 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}, \quad z \in \mathbb{R}$$

For any $\underset{\sim}{x}$ in the interior of R_j, the terms $\mathrm{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)$, $i = 1,2,\ldots,p$, assume fixed values. Hence, $\underset{\sim}{J}_j$ is precisely the right-hand side of (A.2).

To find $\underset{\sim}{s}_j$, we observe:

$$\underset{\sim}{s}_j = \underset{\sim}{f}(\underset{\sim}{x})\Big|_{\underset{\sim}{x} \in R_j} - \underset{\sim}{J}_j \underset{\sim}{x}\Big|_{\underset{\sim}{x} \in R_j}$$

$$= \underset{\sim}{a} + \underset{\sim}{c}\left\{\begin{bmatrix} |\langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle - \beta_1| \\ \vdots \\ |\langle \underset{\sim}{\alpha}_p, \underset{\sim}{x} \rangle - \beta_p| \end{bmatrix} - \mathrm{diag}[\{\mathrm{sgn}(\underset{\sim}{\alpha}_i, \underset{\sim}{x} - \beta_i)\}_1^p] \cdot \begin{bmatrix} \underset{\sim}{\alpha}_1^T \\ \vdots \\ \underset{\sim}{\alpha}_p^T \end{bmatrix} \underset{\sim}{x} \right\}\Bigg|_{\underset{\sim}{x} \in R_j}$$

$$= \underset{\sim}{a} + \underset{\sim}{c}\, \mathrm{diag}[\{\mathrm{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)\}_1^p] \left\{\begin{bmatrix} \langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle - \beta_1 \\ \vdots \\ \langle \underset{\sim}{\alpha}_p, \underset{\sim}{x} \rangle - \beta_p \end{bmatrix} - \begin{bmatrix} \langle \underset{\sim}{\alpha}_1, \underset{\sim}{x} \rangle \\ \vdots \\ \langle \underset{\sim}{\alpha}_p, \underset{\sim}{x} \rangle \end{bmatrix}\right\}\Bigg|_{\underset{\sim}{x} \in R_j}$$

$$= \underset{\sim}{a} - \underset{\sim}{c}\, \mathrm{diag}[\{\mathrm{sgn}(\langle \underset{\sim}{\alpha}_i, \underset{\sim}{x} \rangle - \beta_i)\}_1^p] \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}\Bigg|_{x \in R_j} \qquad \text{(A.3)}$$

FIGURE CAPTIONS

Fig. 1. Extraction of 2-terminal nonlinear resistors to form a linear n-port $\overline{N}$.

Fig. 2. Figures for Example 1.
(a) Circuit containing 2 piecewise-linear resistors.
(b) Constitutive relation of piecewise-linear resistor R1.
(c) Constitutive relation of piecewise-linear resistor R2.
(d) Driving-point plot of 1-port N and the load line showing 3 intersections Q_1, Q_2 and Q_3.
(e) Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (3.7).
(f) Partitions in the range space of $\underset{\sim}{f}(\cdot)$. Note that region e'b'a'd' encloses the origin.

Fig. 3. A portion of a partition in the general case.
(a) Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (2.6). H_k denotes an arbitrary hyperplane and $\sigma_a\sigma_b$ denotes an arbitrary section on H_k.
(b) Corresponding partitions in the range space.

Fig. 4. Figures for Example 2.
(a) Circuit containing 3 piecewise-linear resistors.
(b) Constitutive relation of piecewise-linear resistor R1.

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

(c) Constitutive relation of piecewise-linear resistor R2.

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

(d) Constitutive relation of piecewise-linear resistor R3.

R1: $i_3 = v_3 - \frac{5}{4}|v_3 - 1| - 2|v_3 - 2| - |v_3 - 3|$

Fig. 5. Figures for Example 2.
(a) Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (3.20). Note the rectangular lattice structure.
(b) Part of the partition for $-\infty < v_3 < 1$.
(c) Part of the partition for $1 < v_3 < 2$.
(d) Part of the partition for $2 < v_3 < 3$.
(e) Part of the partition for $3 < v_3 < \infty$.

Fig. 6. Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (4.3). Each asterisk "*" implies the sign test in the corresponding region is positive.

Fig. 7. Figures for Example 4.
(a) A 4-transistor multi-state circuit.
(b) Simplified Ebers-Moll model of a NPN transistor.
(c) Piecewise-linear approximation of diode v-i characteristic: (2-segment case):

$m_0 = 0,\ m_1 = 7.94167\text{x}10^{-2};\ V_1 = 0.325\text{v}$

$I_D = -1.29052\text{x}10^{-2} + 3.9708313\text{x}10^{-2} v_D + 3.9708313\ |v_D - V_1|$

(d) Piecewise-linear approximation of diode v-i characteristic: (3-segment case):

$$m_0 = 0,\ m_1 = 4.95062\text{x}10^{-2},\ m_2 = 2.26\text{x}10^{-1};$$

$$V_1 = 0.325,\ V_2 = 0.372$$

$$I_D = -4.08734956\text{x}10^{-2} + 1.13002391\text{x}10^{-1}\, v_D + 2.47530803\text{x}10^{-2}\,|v_D - V_1| + 8.82493106\text{x}10^{-2}\,|v_D - V_2|$$

(e) Piecewise-linear approximation of diode v-i characteristic: (4-segment case):

$$m_0 = 0,\ m_1 = 3.886\text{x}10^{-2},\ m_2 = 9.661\text{x}10^{-2},\ m_3 = 2.246\text{x}10^{-1};$$

$$V_1 = 0.32,\ V_2 = 0.355,\ V_3 = 0.377$$

$$I_D = -4.0600305\text{x}10^{-2} + 1.12315982\text{x}10^{-1}\, v_D + 1.943118\text{x}10^{-2}\,|v_D - V_1| + 2.88747296\text{x}10^{-2}\,|v_D - V_2| + 6.40100727\text{x}10^{-2}\,|v_D - V_3|$$

(f) Piecewise-linear approximation of diode v-i characteristic: (5-segment case):

$$m_0 = 0,\ m_1 = 2.316\text{x}10^{-2},\ m_2 = 4.682\text{x}10^{-2},\ m_3 = 1.449\text{x}10^{-1},\ m_4 = 2.996\text{x}10^{-1};$$

$$V_1 = 0.306,\ V_2 = 0.3375,\ V_3 = 0.366,\ V_4 = 0.3875$$

$$I_D = -5.48808681\text{x}10^{-2} + 1.48309806\text{x}10^{-1}\, v_D + 1.15780376\text{x}10^{-2}\,|v_D - V_1| + 1.18300472\text{x}10^{-2}\,|v_D - V_2| + 4.90265238\text{x}10^{-2}\,|v_D - V_3| + 7.58751971\text{x}10^{-2}\,|v_D - V_4|$$

(g) Piecewise-linear approximation of diode v-i characteristic: (6-segment case):

$$m_0 = 0,\ m_1 = 2.513\text{x}10^{-2},\ m_2 = 2.666\text{x}10^{-2},\ m_3 = 3.765\text{x}10^{-2},\ m_4 = 8.603\text{x}10^{-2},\ m_5 = 1.865\text{x}10^{-1};$$

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

$$I_D = -3.33570322\text{x}10^{-2} + 9.32400146\text{x}10^{-2}\, v_D + 1.25666608\text{x}10^{-2}\,|v_D - V_1| + 2.2353727\text{x}10^{-3}\,|v_D - V_2| + 8.49354618\text{x}10^{-3}\,|v_D - V_3| + 2.41900658\text{x}10^{-2}\,|v_D - V_4| + 5.02251145\text{x}10^{-2}\,|v_D - V_5|$$

Fig. 8. Figures for <u>Example 5</u>.

(a) Circuit diagram.

(b) Curve of a continuous piecewise-linear function:

$$g(v) = -\frac{1}{2} + v - \frac{1}{2}|v-1| + \frac{1}{2}|v-2|$$

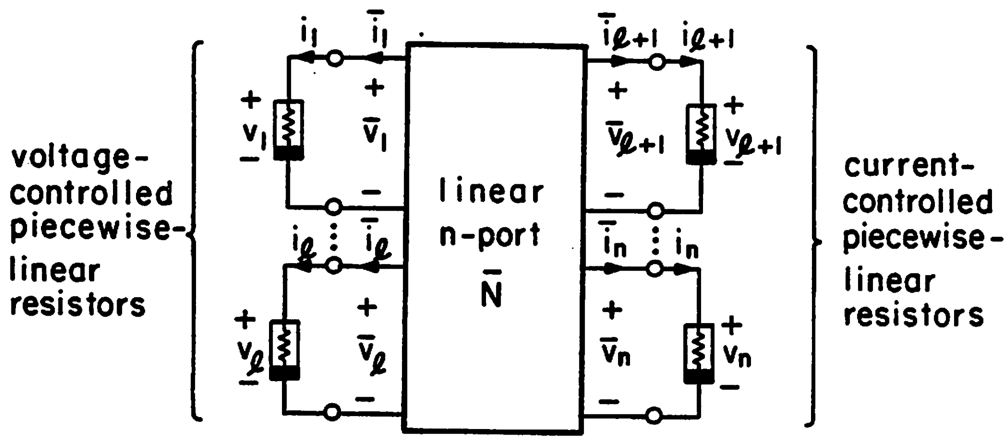
(c) Partitions in the domain of $f(\cdot)$ defined by (5.1). The Jacobian matrix in each region is also shown.

(d) Partitions in the range space of $f(\cdot)$. Note that regions R_2, R_4, R_5, R_6 and R_8 degenerate into a single point P.

Fig. 9. Figures for Example 6.
(a) Circuit diagram.
(b) Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (5.8).
(c) Partitions in the range space of $\underset{\sim}{f}(\cdot)$. Note that region R_5 degenerates into a line segment A'D'.

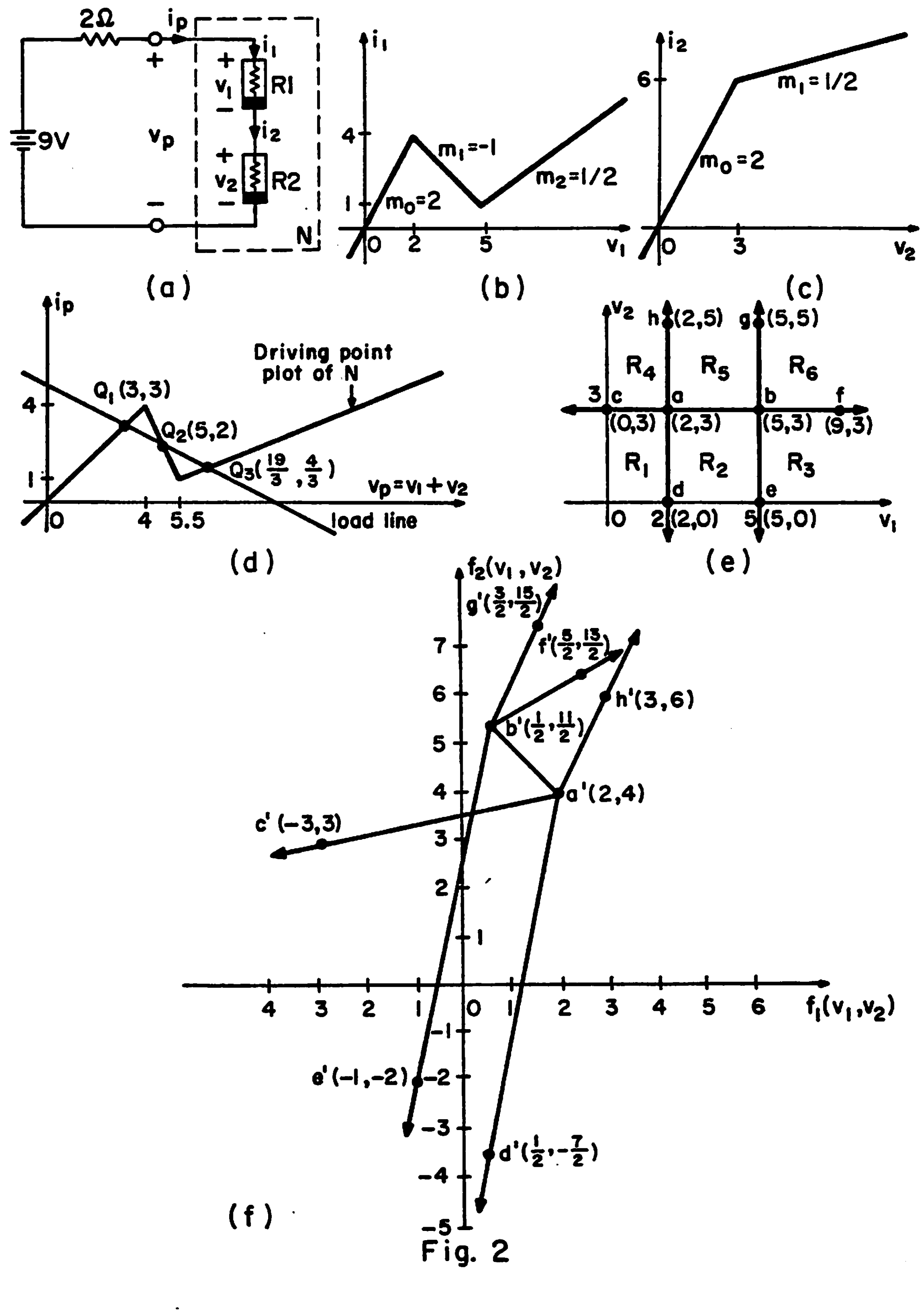
Fig. 10. Figures for Example 7.
(a) Circuit diagram.
(b) Driving-point plot of 1-port N and the load line. Note infinitely many solutions exist for this circuit.
(c) Partitions in the domain of $\underset{\sim}{f}(\cdot)$ defined by (5.9).
(d) Partitions in the range space of $\underset{\sim}{f}(\cdot)$. Note that region R_2 degenerates into a half line passing through the origin.

LIST OF TABLE CAPTIONS



voltage-controlled piecewise-linear resistors
current-controlled piecewise-linear resistors
linear n-port
N̄

Fig. 1



2Ω
9V
i_p
v_p
i_1
v_1
R1
i_2
v_2
R2
N
(a)
i_1
$m_0=2$
$m_1=-1$
$m_2=1/2$
v_1
(b)
i_2
$m_0=2$
$m_1=1/2$
v_2
(c)
i_p
$Q_1(3,3)$
$Q_2(5,2)$
$Q_3(\frac{19}{3},\frac{4}{3})$
Driving point plot of N
$v_p=v_1+v_2$
load line
(d)
v_2
h (2,5)
g (5,5)
R_4
R_5
R_6
c (0,3)
a (2,3)
b (5,3)
f (9,3)
R_1
R_2
R_3
d (2,0)
e (5,0)
v_1
(e)
$f_2(v_1,v_2)$
$g'(\frac{3}{2},\frac{15}{2})$
$f'(\frac{5}{2},\frac{13}{2})$
$h'(3,6)$
$b'(\frac{1}{2},\frac{11}{2})$
$a'(2,4)$
$c'(-3,3)$
$e'(-1,-2)$
$d'(\frac{1}{2},-\frac{7}{2})$
$f_1(v_1,v_2)$
(f)

Fig. 2

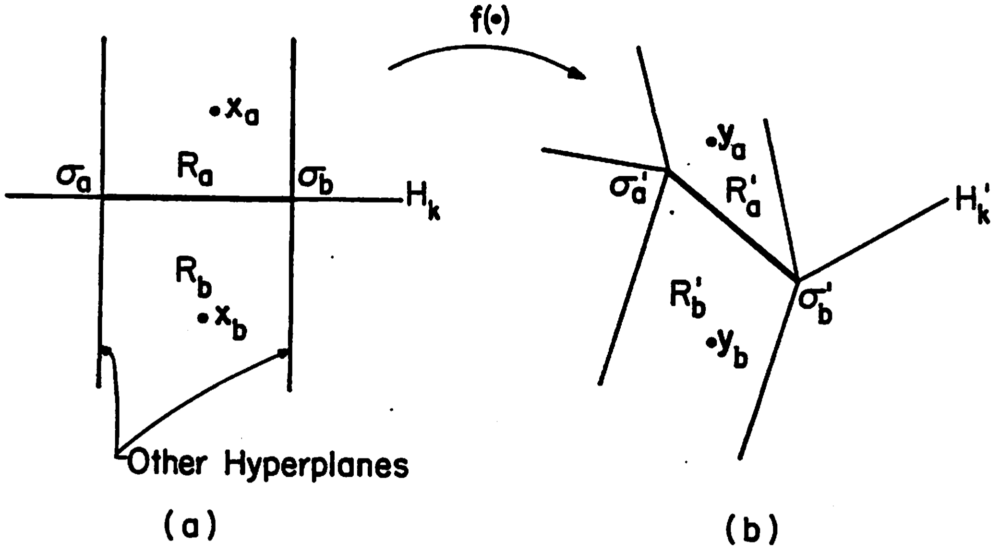

f(•)
x_a
σ_a
R_a
σ_b
H_k
R_b
x_b
Other Hyperplanes
(a)
y_a
σ_a'
R_a'
H_k'
R_b'
σ_b'
y_b
(b)

Fig. 3

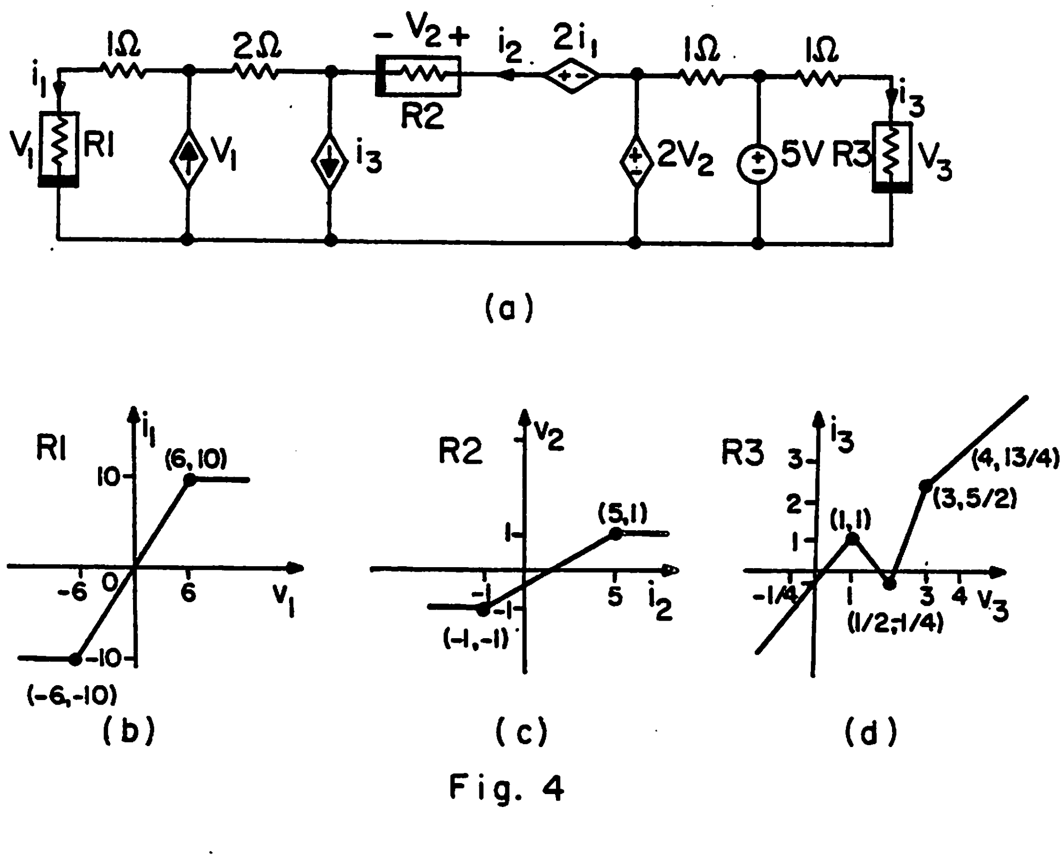
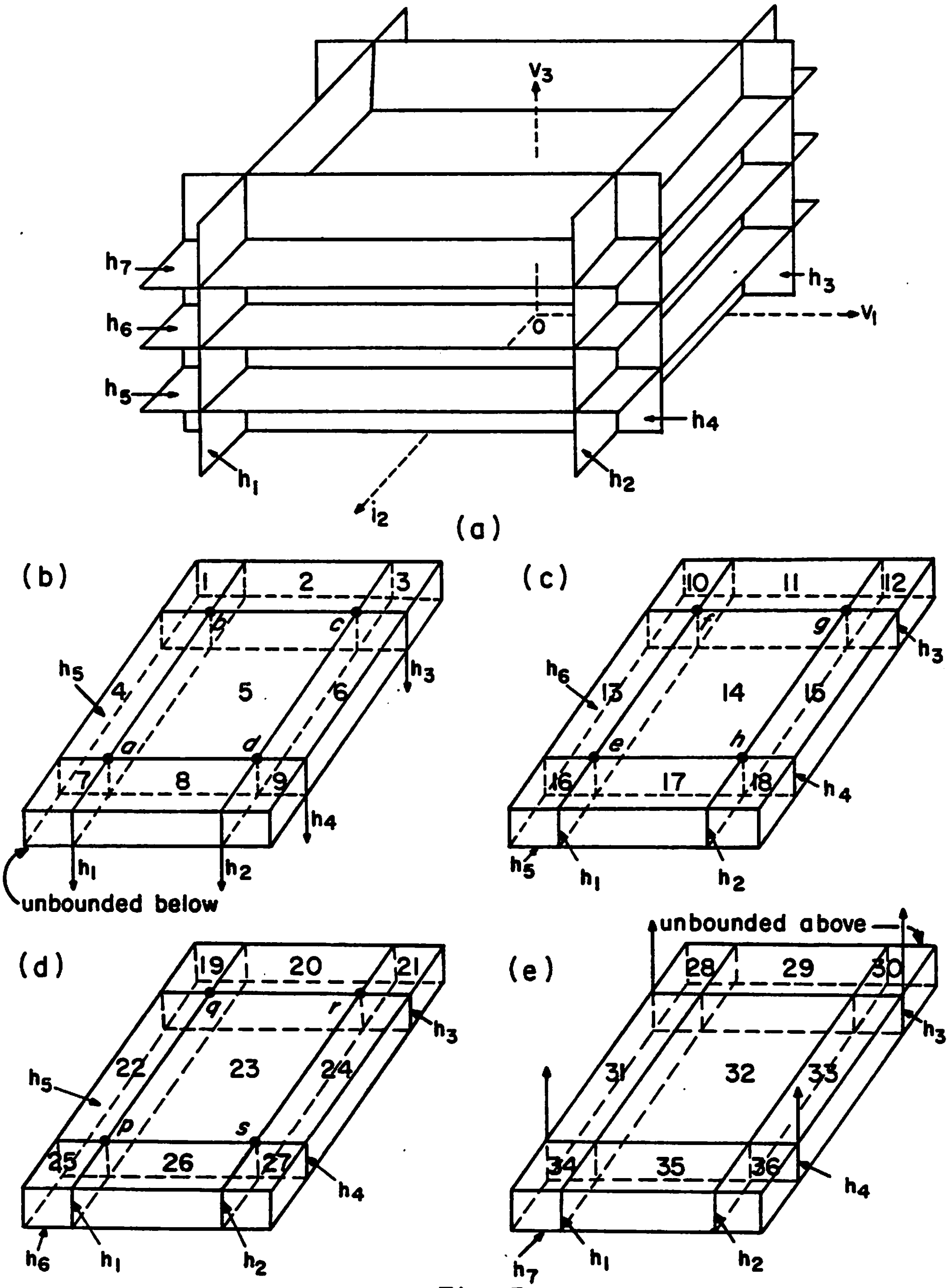
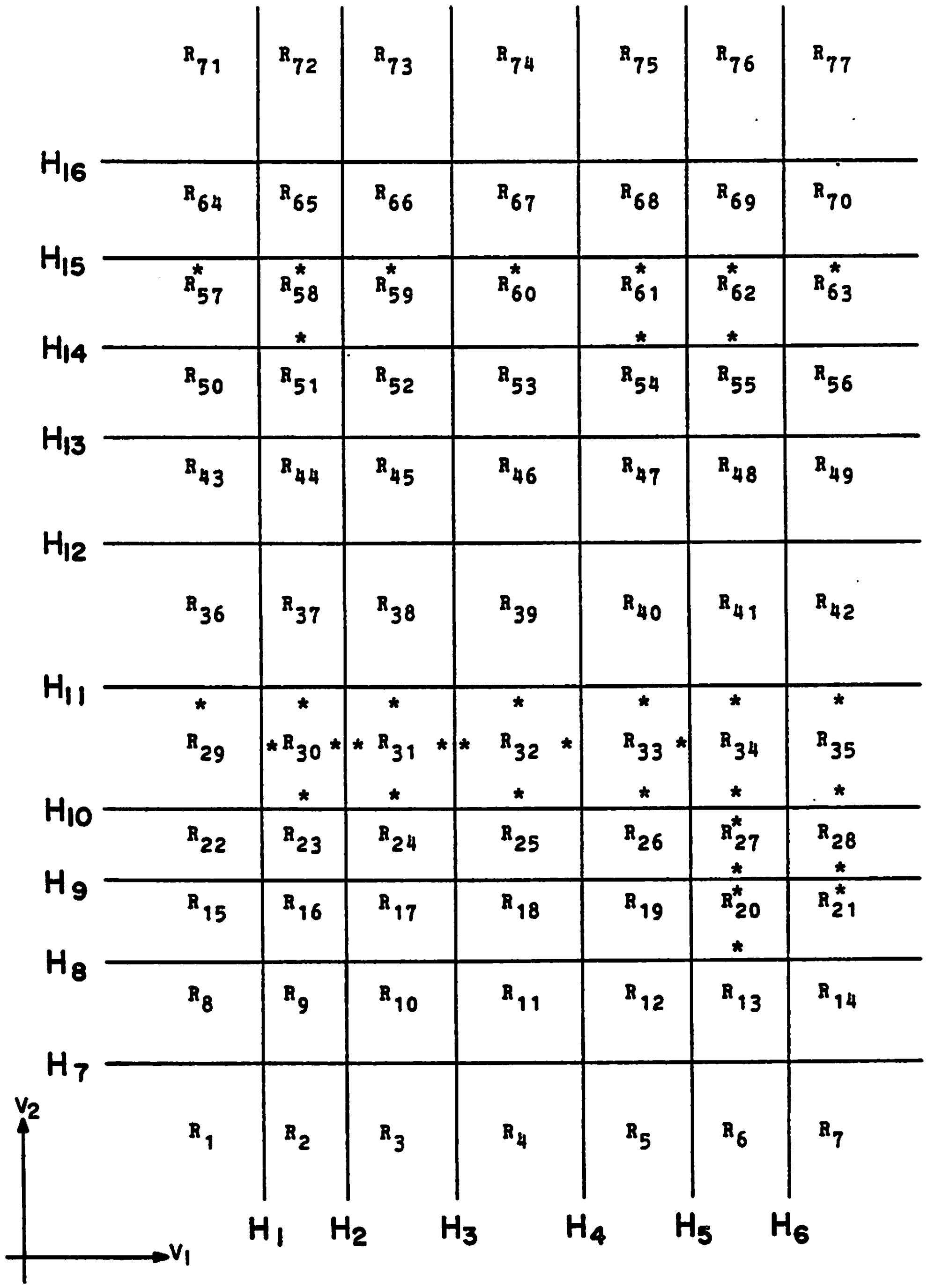

1Ω
2Ω
− V2 +
i2
2i1
1Ω
1Ω
i1
V1
R1
V1
i3
R2
2V2
5V
R3
i3
V3
(a)
R1
i1
10
(6, 10)
-6
0
6
V1
-10
(-6,-10)
(b)
R2
V2
(5,1)
1
-1
5
i2
-1
(-1,-1)
(c)
R3
i3
3
2
1
(1,1)
(4, 13/4)
(3, 5/2)
-1/4
1
3
4
V3
(1/2;1/4)
(d)

Fig. 4



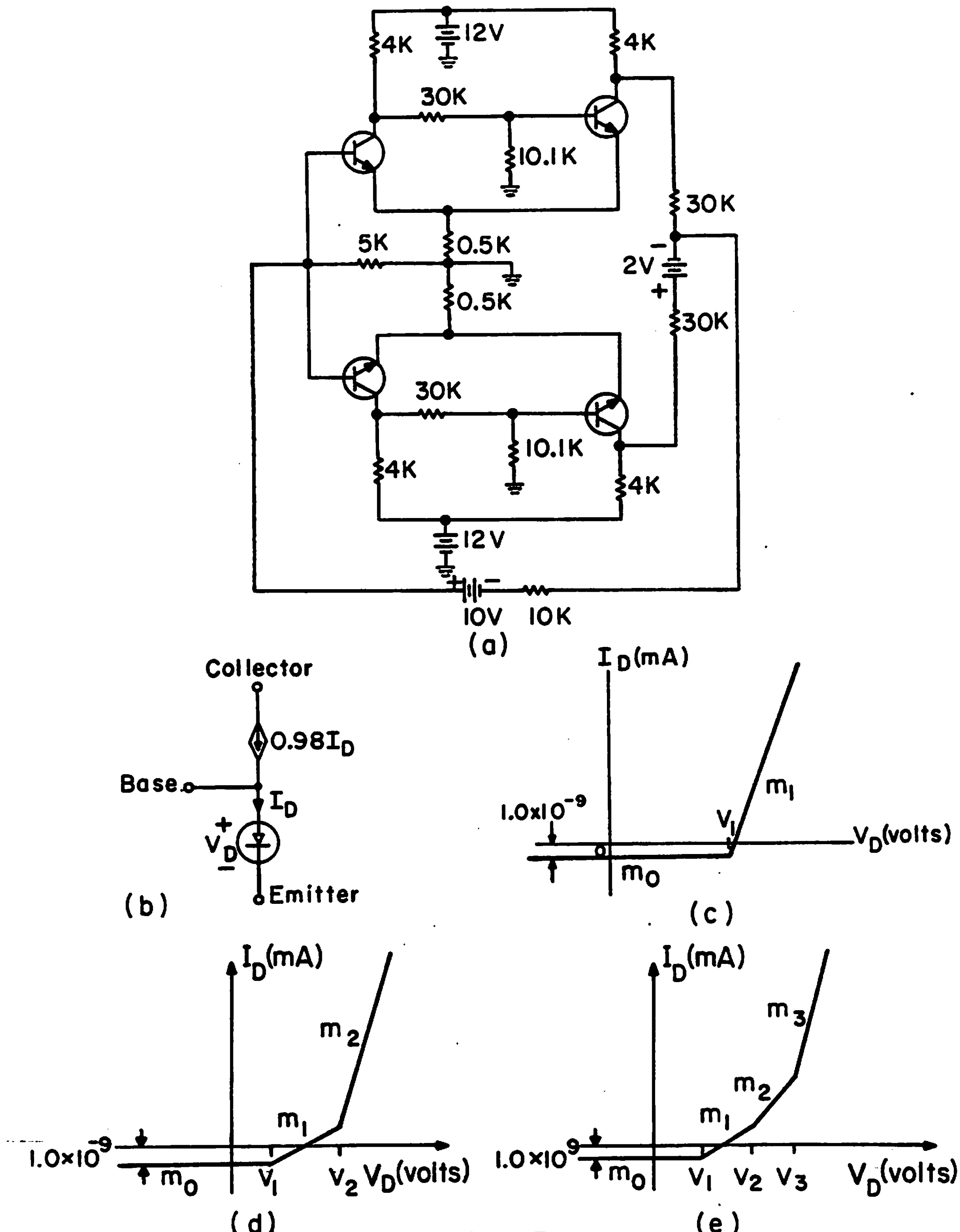
v3
h7
h6
h5
h3
h4
v1
0
h1
h2
i2
(a)
(b)
1
2
3
b
c
h5
4
5
6
h3
a
d
7
8
9
h4
h1
h2
unbounded below
(c)
10
11
112
f
g
h6
13
14
15
h3
e
h
16
17
18
h4
h5
h1
h2
(d)
19
20
21
q
r
h3
h5
22
23
24
p
s
25
26
27
h4
h6
h1
h2
(e)
unbounded above
28
29
30
h3
31
32
33
34
35
36
h4
h7
h1
h2

Fig. 5



R71 R72 R73 R74 R75 R76 R77
H16
R64 R65 R66 R67 R68 R69 R70
H15
R57 R58 R59 R60 R61 R62 R63
H14
R50 R51 R52 R53 R54 R55 R56
H13
R43 R44 R45 R46 R47 R48 R49
H12
R36 R37 R38 R39 R40 R41 R42
H11
R29 R30 R31 R32 R33 R34 R35
H10
R22 R23 R24 R25 R26 R27 R28
H9
R15 R16 R17 R18 R19 R20 R21
H8
R8 R9 R10 R11 R12 R13 R14
H7
R1 R2 R3 R4 R5 R6 R7
V2
V1
H1 H2 H3 H4 H5 H6

Fig. 6



12V
4K
4K
30K
10.1K
30K
5K
0.5K
0.5K
2V
30K
30K
10.1K
4K
4K
12V
10V
10K
(a)
Collector
0.98I_D
Base
I_D
V_D
Emitter
(b)
I_D(mA)
m1
1.0x10^-9
V1
V_D(volts)
m0
(c)
I_D(mA)
m2
m1
1.0x10^-9
m0
V1
V2
V_D(volts)
(d)
I_D(mA)
m3
m2
m1
1.0x10^9
m0
V1
V2
V3
V_D(volts)
(e)

Fig. 7

Fig. 7 cont'd

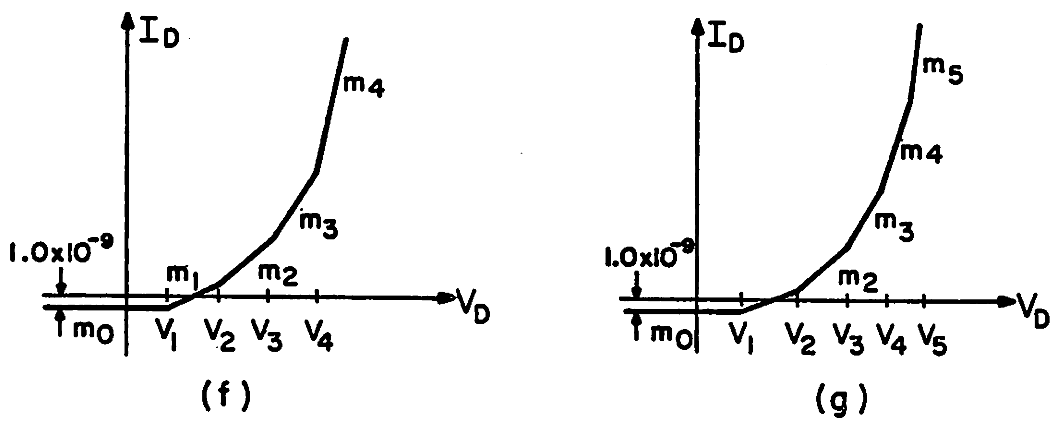
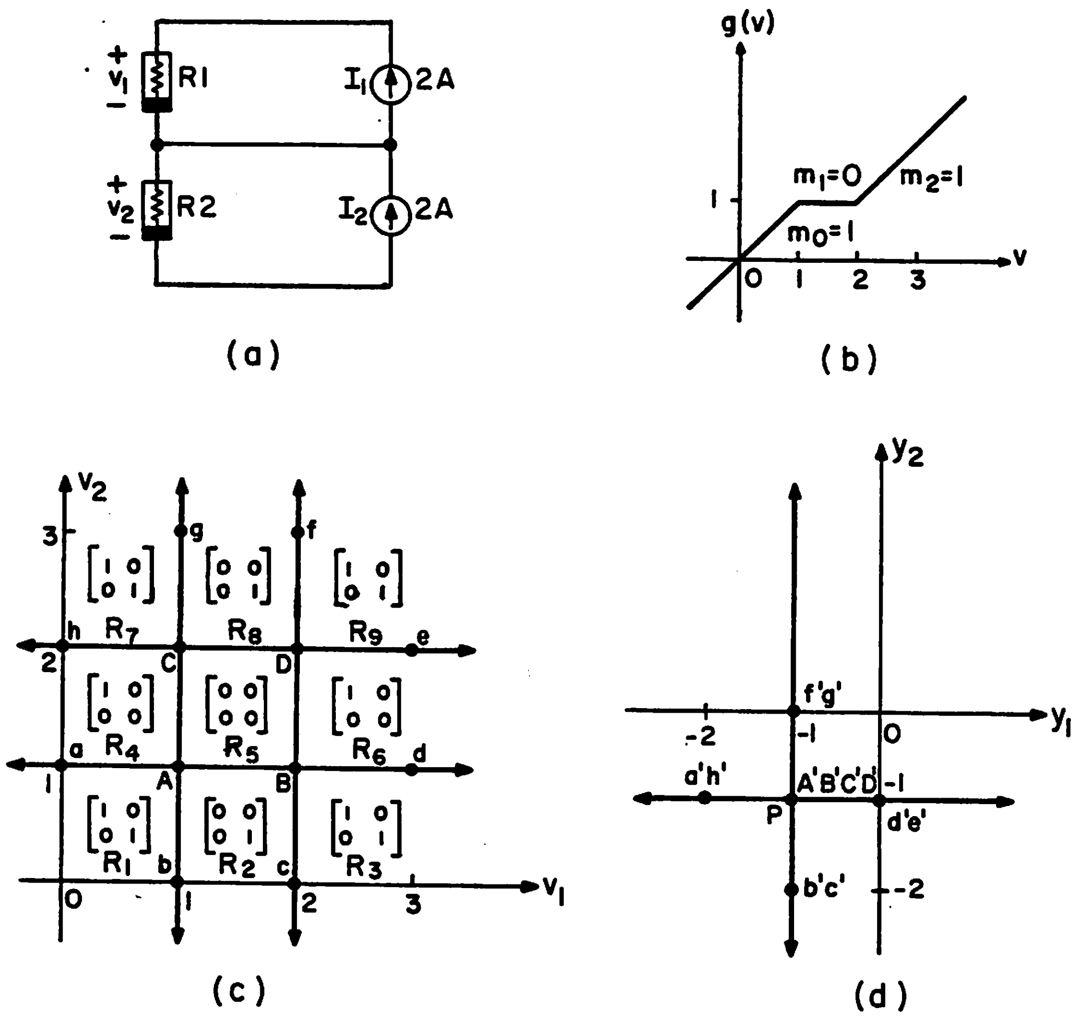


Fig. 7



v_1 R1 I_1 2A
v_2 R2 I_2 2A
(a)
g(v)
m_1=0
m_2=1
m_0=1
v
(b)
v_2
v_1
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 R_9
(c)
y_2
y_1
f'g'
a'h'
A'B'C'D'
P
d'e'
b'c'
(d)

Fig. 8

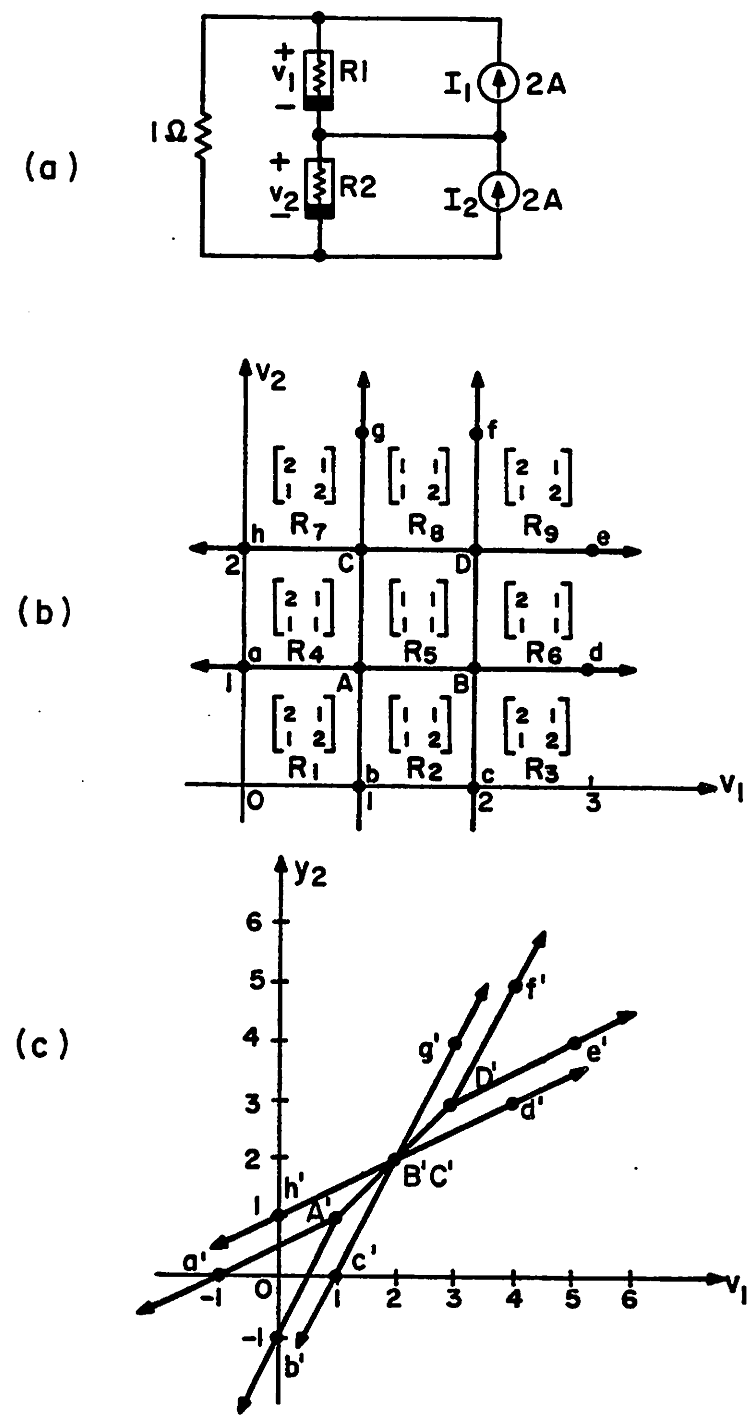
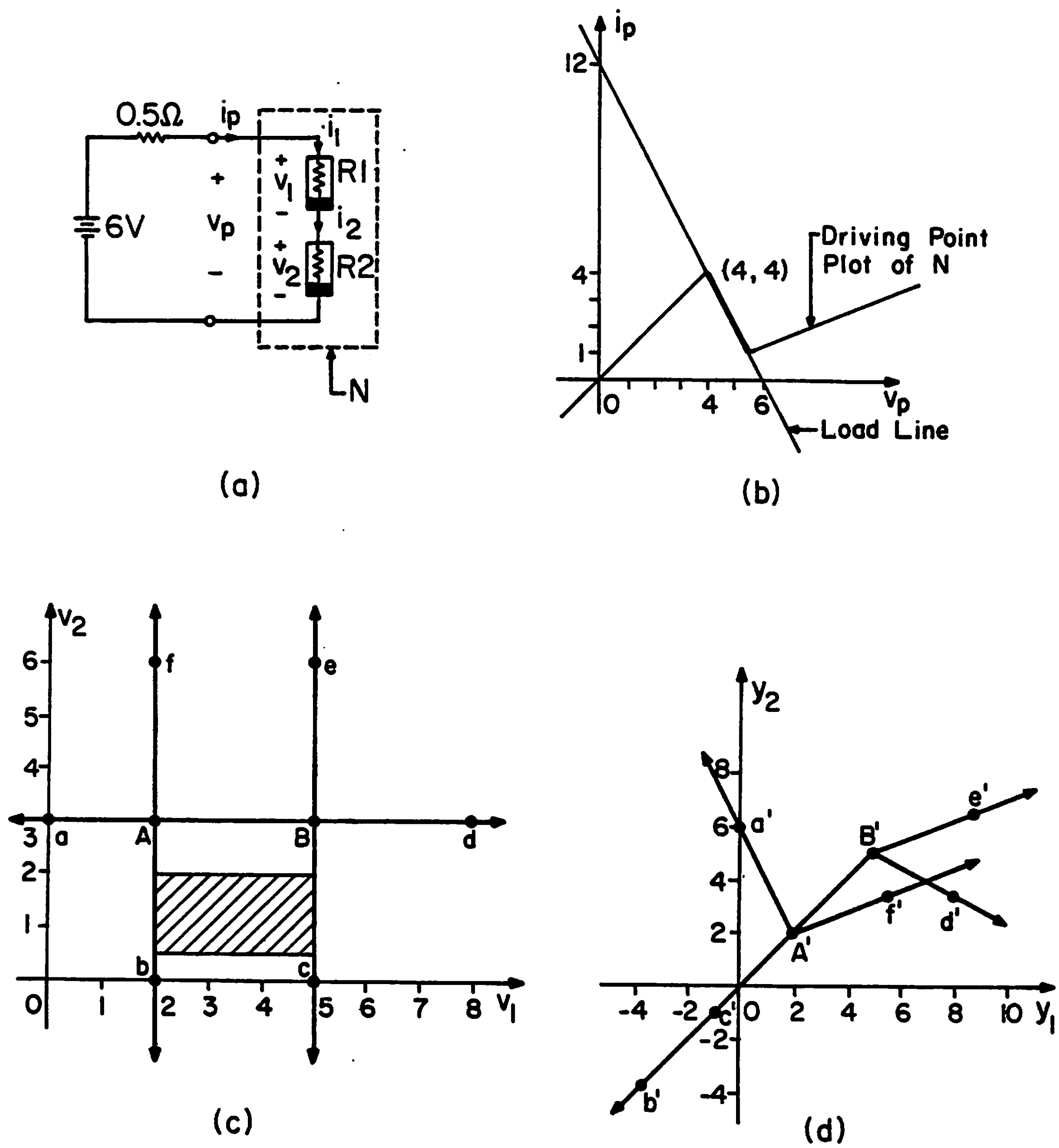

(a)
1Ω
+
v1
−
R1
I1
2A
+
v2
−
R2
I2
2A
(b)
v2
g
f
R7
R8
R9
h
C
D
e
2
R4
R5
R6
a
A
B
d
1
R1
R2
R3
b
c
v1
0
1
2
3
(c)
y2
f'
g'
D'
e'
d'
B'C'
h'
A'
a'
c'
b'
v1
0
1
2
3
4
5
6
-1

Fig. 9



0.5Ω
i_p
i_1
R1
v_1
6V
v_p
i_2
R2
v_2
N
(a)
i_p
12
4
1
0
4
6
v_p
(4, 4)
Driving Point Plot of N
Load Line
(b)
v_2
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
v_1
a
A
B
d
b
c
f
e
(c)
y_2
8
6
4
2
-2
-4
-4
-2
0
2
4
6
8
10
y_1
a'
A'
B'
e'
f'
d'
b'
c'
(d)

Fig. 10

Table 1. Solutions of Fig. 7(a).

solutions	v_{d1}	v_{d2}	v_{d3}	v_{d4}
1	0.38392	-3.79264	0.37543	-2.84029
2	0.38859	-4.31084	0.33696	0.34565
3	0.33398	0.35187	0.38142	-3.51440
4	0.33197	0.35608	0.33452	0.35074
5	-1.06411	0.37066	0.38539	-3.95558
6	-0.72552	0.37066	0.33345	0.35298
7	0.39388	-4.89790	-1.52344	0.37066
8	0.33051	0.35914	-1.11032	0.37066
9	-0.52530	0.37066	-0.97985	0.37066

Table 2. Summary of computation for Example 4.

no. of breakpoints k_i	no. of segments $k_i + 1$	total no. of regions $\prod_{i=1}^{n} (k_i+1)$	no. of linear systems solved by:		
			"brute-force" combinatorial method	our algorithm	
				worst case $\sum_{j=1}^{n} N_j$	actual case
1	2	16	16	32	28
2	3	81	81	108	72
3	4	256	256	256	133
4	5	625	625	500	229
5	6	1296	1296	864	362

Table 3. Summary of computation for Example 2 and 3.

Examples	total no. of regions	no. of linear systems solved by:		
		brute-force combinatorial method	our algorithm	
			worst case $\sum_{j=1}^{n} N_j$	actual case
Example 2	36	36	33	15
Example 3	77	77	18	10

Table 4. Approximate CPU time used in each example.

Examples		CPU in seconds
Example 1		0.10 - 0.13
Example 2		0.37 - 0.48
Example 3		0.83 - 0.93
Example 4	$k_i = 1$	0.40 - 0.62
	$k_i = 2$	2.00 5.07
	$k_i = 3$	4.53 - 8.77
	$k_i = 4$	14.40 - 21.65
	$k_i = 5$	36.65 - 48.22
Example 5		0.13 - 0.22
Example 6		0.12 - 0.15
Example 7		0.10 - 0.20

PROGRAM LISTING

```
/*          Copyright (c) 1981        Robin L.P. Ying
**
** The routines in this package are able to find All Solutions
** of a given Piece-Wise Linear Function (ASPWLF)
**                        f(x) = 0
** provided that f(.) is represented in the piecewise-linear
** canonical form with boundary hyperplanes parallel to the
** coordinate axes.  The algorithm used is described in this
** section 3.3 of this memo.
**
** This package is written in the standard C-language described in
** "The C Programming Language" by B.W. Kernighan and D.M. Ritchie.
** 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:
**     aspwlf.h:  containing definitions of data structures and global
**                variables.
**       main.c:  handling command line options.
**     aspwlf.c:  containing the main routines for solving the given
**                piecewise-linear system.
**       init.c:  containing routines for initializations.
**      queue.c:  containing queue lists manipulating routines.
**      print.c:  containing printing routines.
**      error.c:  printing run-time error messages.
**    support.c:  containing various supporting routines.
**    interac.c:  an user oriented interactive program which will
**                creat a C-program defining the piecewise-linear
**                function.
**     Makefile:  file maintenance program.
**
** The folowing routines are needed from the double-precision IMSL
** library:
**        leqt2f(), ludatf(), luelmf(), lureff(), uertst(),
**        ugetio(), vxadd(), vxmul(), vxsto().
**
** Compile and run:
**        The following steps are contained in "Makefile" for compiling
**        and running this package:
**
** Step   1. create Alib:
**           cc -c main.c asplwf.c init.c print.c queue.c error.c support.c
**           ar ru Alib *.o
**           ranlib Alib
**
**        2. create interac:
**           cc interac.c -o interac
**
**        3. create a.out:
**           cc ex.c Alib -limsld -lF77 -lI77 -lm
**           [ ex.c can be created by running "interac" ]
*/
```

```
/*
** aspwlf.h -- this is the header file for all ASPWLF routines
**        except "support.c", "interac.c" and "ex.c" which defines
**        pwlf().
**
** 5 data structures are defined in this module:
**        RGN:    each region;
**        RGNQ:   queue of regions;
**        HYP:    each hyperplane;
**        HYPQ:   queue of hyperplanes;
**        QLST:   list of the HYPQs.
**
** Dimension of arrays:
**    variables in pwlf():
**        a[dim], B[dim][dim], C[dim][hyp], D[dim][hyp], e[hyp].
**    variables in struct RGN:
**        sgnsq[hyp], bdry[hyp], px[dim], py[dim].
**    variables in aspwlf():
**        ah[dim], Bh[dim][dim], Bht[dim][dim], alphah[dim],
**        sol[dim], sgnsv[hyp], markx[rjmax][dim], ynrml[rjmax][dim],
**        wk[dim*(dim+3)].
**
**    All these arrays are dynamically allocated using palloc().
*/

#define FORMAT1  "%6.3f "          /* define printing formats */
#define FORMAT2  "%2.0f "
#define FORMAT3  "%13.6e "

#define RNIL (RGN  *)     0177777
#define HNIL (HYP  *)     0177777
#define QNIL (HYPQ *)     0177777

typedef struct    region {
        int       *sgnsq;          /* sign sequence */
        int       *bdry;           /* boundaries */
        double    *px;             /* point in region */
        double    *py;             /* f(px) */
        int       id;              /* region identifier */
        struct    region *link;    /* region queue link */
} RGN;

typedef struct    {
        RGN       *head;           /* head of queue */
        RGN       *tail;           /* tail of queue */
        int       n;               /* # of elements on queue */
} RGNQ;

typedef struct    hplane {
        int       id;              /* hyperplane identifier */
        struct    hplane *link;    /* hyperplane queue link */
} HYP;

typedef struct    hqueue {
        HYP       *head;           /* head of queue */
        HYP       *tail;           /* tail of queue */
        int       n;               /* # of elements on queue */
        int       rj;              /* # of sections */
        int       axis;            /* coordinate axis */
        struct    hqueue *link;    /* link */
} HYPQ;

typedef struct    {
```

```
        HYPQ    *head;          /* head of list */
        HYPQ    *tail;          /* tail of list */
        int     n;              /* # of elements on list */
} QLST;

RGNQ    *W[4];
QLST    *Q;
int     rjmax;
extern  double  *a, *B, *C, *D, *e, epsilon;
extern  int     dim, hyp, aflg, pflg, tflg, imsl, ier, sigdgt;
```

```
/*
** main.c -- handles command line flags.
**
** Command line flags:
** -p :  turns on the "pflg" so that all information of hyperplanes
**       and regions will be printed.
**
** -t :  turns on the "tflg" so that the solution obtained in STEP 6
**       will be tested.
**
** -a :  turns on the "aflg" as well as "pflg" & "tflg" so that every
**       detail of the iteration will be printed.
**
** -i :  turns on the "imsl" flag so that aspwlf() will use the IMSL
**       routine LEQT2F() to solve linear systems.
**
** -s int : resets the significant digit to "int" decimal digits
**       (int < 0 is ignored); if int = 0 then the accuracy test in
**       the IMSL routine is disabled; the default value of int is 9;
**       this option automatically turns on the "-i" flag.
*/

int     aflg=0, pflg=0, tflg=0, imsl=0, sigdgt=9;
double  epsilon;

main (argc, argv)

int     argc;
char    **argv;
{
        register int    i, flg=0, tmp;

        while (--argc > 0 && (*++argv)[0] == '-') {
            while (*++*argv) switch (**argv) {
                case 'a':               /* turn on aflg */
                    aflg = 1;
                    pflg = 1;
                    tflg = 1;
                    continue;
                case 'p':               /* turn on pflg */
                    pflg = 1;
                    continue;
                case 't':               /* turn on tflg */
                    tflg = 1;
                    continue;
                case 'i':               /* turn on imsl */
                    imsl = 1;
                    continue;
                case 's':               /* reset sigdgt */
                    imsl = 1;
                    tmp  = atoi(argv[1]);
                    if ( tmp > 0 ) {
                        sigdgt = tmp;
                        flg = 1;
                    }
                    goto next;
                default:                /* other char has no effect */
                    continue;
            }
        next:
            argc--;
        }

        if ( imsl )
```

```
        printf("**-< using IMSL routine >-**\n");

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

    if ( flg || pflg ) {
        printf("**-< significant digit is set to %d >-**\n",sigdgt);
        printf("**-< epsilon = %8.1e >-**\n",epsilon);
    }

    aspwlf();                          /* start */
}
```

```
/*
** aspwlf.c -- contains 6 routines:
**      aspwlf():       the main iteration routine.
**      conW2():        construct list W[2].
**      compah():       compute ah[] and offset[].
**      compBh():       compute Bh[] and jcbn[,].
**      sgntst():       perform sign test.
**      cpnrml():       compute ynrml[].
*/

#include "aspwlf.h"

double   *ah, *Bh, *Bht, *alfah, *sol, *ynrml, *markx, *wk;
int      *sgnsv, kk, mxcnt, itr=0, ier;


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

aspwlf ()
{
        HYPQ    *gethq(), *hq;
        HYP     *gethyp();
        RGN     *getrgn();
        double  inprdct(), betah, scale;
        int     cpnrml(), n, nhq,
                nsol=0,                 /* # of solutions */
                btaflg,                 /* for betah==0 */
                erflg1,                 /* for Bh[,] singular */
                erflg2,                 /* for puting h back to hq */
                hqflg;                  /* for 1st h on hq */
        register RGN    *rgn;
        register HYP    *h;
        register int    i, j,           /* running indices */
                        flg1, flg2;     /* for matching & nbhd */

   /* STEP 0: initialize & allocate spaces */
        init();
        ah    = (double *) palloc(dim*sizeof(double));
        Bh    = (double *) palloc(dim*dim*sizeof(double));
        Bht   = (double *) palloc(dim*dim*sizeof(double));
        alfah = (double *) palloc(dim*sizeof(double));
        sol   = (double *) palloc(dim*sizeof(double));
        ynrml = (double *) palloc(dim*rjmax*sizeof(double));
        markx = (double *) palloc(dim*rjmax*sizeof(double));
        sgnsv = (int     *) palloc(hyp*sizeof(int));
        if ( imsl )
            wk = (double *) palloc(dim*(dim+3)*sizeof(double));

   /* BEGIN ITERATION */
     while ( Q->n != 0 ) {                      /* main loop */
        hq      = gethq(Q);
        nhq     = hq->n;
        mxcnt   = hqflg = 0;
        while ( hq->n != 0 ) {                  /* 2nd loop */
            h   = gethyp(hq);
            kk  = h->id;
            erflg2 = 0;
```

```
/* STEP 1: construct set W[2] from ( W[0] union W[1] ) */
        if ( aflg ) {
            printf("\n\n@ STEP 1: hq->axis: %d, ",hq->axis);
            printf("hyp->id: %d,\nW[0]->n: %d\n",kk,W[0]->n);
        }
        for (i=0; i < 2; i++) {
            conW2(W[i],W[2]);
            if ( aflg ) printf("W[%d]->n=%d\n",i+1,W[i+1]->n);
        }

/* STEP 2 */
        while ( W[2]->n != 0 ) {        /*  3rd loop   */
            /*
            ** pick a region from W[2], save its sgnsq[],
            ** set kk-th element in rgn->sgnsq[] to zero.
            */
            rgn = getrgn(W[2]);
            for (i=0; i < hyp; i++)
                sgnsv[i] = rgn->sgnsq[i];
            rgn->sgnsq[kk] = 0;
            if ( aflg )
                printf("\n\n@ STEP 2: rgn on W[2]: %d\n",rgn->id);
            compah(ah,rgn);             /* compute ah[] */
            scale = 1.0;                /* reset scale */
            if ( !hqflg )               /* 1st hyp on hq */
                erflg1 = cpnrml(rgn);
            /*
            ** try matching 'markx[]' with rgn->px[],
            ** if not, compute ynrml[].
            */
            else {
                erflg1 = flg2 = 0;
                for (j=0; j < mxcnt; j++) {
                    flg1 = 0;
                    for (i=0; i < dim; i++) {
                        if ( i == hq->axis ) continue;
                        else if (rgn->px[i] != markx[j*dim+i]) {
                            flg1 = 1;
                            break;
                        }
                    }
                    if ( !flg1 ) {      /* matched */
                        for (i=0; i < dim; i++)
                            alfah[i] = ynrml[j*dim+i];
                        /* compute hq->axis-th column of Bh[,] */
                        compBh(sol,rgn,1,hq->axis);
                        scale = inprdct(alfah,sol,dim);
                        flg2 = 1;
                        break;
                    }
                }
                if ( !flg2 ) erflg1 = cpnrml(rgn);
            }
            /* restore the kk-th bit in rgn->sgnsq[] */
            rgn->sgnsq[kk] = sgnsv[kk];
            switch ( erflg1 ) {
            case 0:                     /* compute betah */
                betah = scale*e[kk] + inprdct(alfah,ah,dim);
                btaflg = 0;
                if ( betah == 0 ) {  /* put rgn to W[1] */
                    btaflg = 1;
                    if ( pflg ) error(3,"aspwlf()",rgn,kk,Bh,ah);
                    putrgn(W[1],rgn);
                }
```

```
                    break;
                case 1:                     /* numerical error occurred
                                               in solving alfah[] */
                    erflg2 = 1;
                    if ( nhq == 1 )         /* can not revover error */
                        error(0,"aspwlf()",rgn,kk);
                    else {
                        if ( aflg )
                            printf("\n** put back to W[0]: %d\n",rgn->id);
                        putrgn(W[0],rgn);
                    }
                    break;
                case 2:                     /* rgn degenerated */
                    if ( pflg ) error(4,"aspwlf()",rgn);
                    putrgn(W[3],rgn);
                    break;
                }
                if ( (erflg1!=0) || btaflg ) goto nbhd;

/* STEP 3: perform sign test */
                if ( aflg ) {
                    printf("\n   alfah[]: ");
                    prdvctr(alfah,dim,FORMAT1);
                    printf("\n   betah=%6.3f\n",betah);
                    printf("\n\n@ STEP 3: rgn on 1st sign test:");
                    printf(" %d\n",rgn->id);
                }
                /* if sign test is true, put the region on W[1] */
                if ( sgntst(rgn,alfah,betah) ) putrgn(W[1],rgn);

/* STEP 4: get neighborhood region */
                /*
                ** scan W[2], search for the neighborhood region
                ** (all but the sgnsq[kk] matches) and perform sign
                ** test again.
                */
    nbhd:
                n = W[2]->n;
                for (j=0; j < n; j++) {
                    rgn = getrgn(W[2]);
                    flg1 = 0;
                    for (i=0; i < hyp; i++) {
                        if ( i == kk ) continue;
                        else if ( rgn->sgnsq[i] != sgnsv[i] ) {
                            flg1 = 1;
                            break;
                        }
                    }
                    if ( flg1 )             /* not nbhd region */
                        putrgn(W[2],rgn);
                    else {                  /* nbhd region */
                        if ( aflg ) {
                            printf("\n\n@ STEP 4: rgn on 2nd sign test:");
                            printf(" %d\n",rgn->id);
                        }
                        switch ( erflg1 ) {
                        case 0:
                            if ( btaflg || sgntst(rgn,alfah,betah) )
                                putrgn(W[1],rgn);
                            break;
                        case 1:
                            if ( aflg )
                                printf("\n** rgn put back to W[0]: %d",
                                    rgn->id);
```

```
                        putrgn(W[0],rgn);
                        break;
                    case 2:
                        if ( pflg ) error(4,"aspwlf()",rgn);
                        putrgn(W[3],rgn);
                        break;
                    }
                    break;
                }
            }
        }                                       /* end of 3rd loop */
        hqflg = 1;
        if ( erflg2 ) {                         /* try to fix error */
            if ( aflg )
                printf("\n** end3: put #%d hyp back to queue.",h->id);
            puthyp(hq,h);
            nhq = hq->n;
            if ( aflg ) printf("\n** end3: nhq=%d",nhq);
        }

/* STEP 5 */
    }                                           /* end of 2nd loop */
  }                                             /* end of main loop */

/* STEP 6: compute solutions */
    if ( pflg ) printf("\n\n@ STEP 6: compute solution.");
    if ( W[3]->n != 0 ) {                       /* check W[3] */
        printf("\n\n** The following are degenerate regions:\n");
        /* print degenerated region id */
        while ( W[3]->n != 0 ) {
            rgn = getrgn(W[3]);
            printf("\tregion %d\n",rgn->id);
            if ( pflg ) {
                compah(ah,rgn);
                compBh(Bh,rgn,0);
                printf("Jacobian[,]:");
                prdmtrx(Bh,dim,dim,FORMAT1);
                printf("Offset[]: ");
                prdvctr(ah,dim,FORMAT1);
                printf("\n\n");
            }
        }
    }
    if ( W[1]->n != 0 )                         /* check W[1] */
        while ( W[1]->n != 0 ) {
            rgn = getrgn(W[1]);
            /* compute offset[], use ah[] as offset[] */
            compah(ah,rgn);
            /* compute jcbn[,], use Bh[,] as jcbn[,] */
            compBh(Bh,rgn,0);
            if ( pflg ) {
                printf("\n\n* region %d:\n",rgn->id);
                printf("Jacobian[,]:");
                prdmtrx(Bh,dim,dim,FORMAT1);
                printf("Offset[]: ");
                prdvctr(ah,dim,FORMAT1);
            }
            /* compute solution */
            if ( !imsl ) {
                lineqn(Bh,sol,ah,dim,0,&scale);
                for (i=0; i < dim; i++)
                    sol[i] = 0 - sol[i];
            }
            else {
```

```
                    transp(Bh,Bht,dim,dim);
                    leqt2f_(Bht,&imsl,&dim,&dim,ah,&sigdgt,wk,&ier);
                    if ( ier > 128 ) {
                         error(6,"aspwlf()",rgn,kk,Bh);
                         goto end;
                    }
                    else
                         for (i=0; i < dim; i++)
                              sol[i] = 0 - ah[i];
               }
               nsol++;
               /* print solution */
               printf("\n\n** solution %d:\t", nsol);
               prdvctr(sol,dim,FORMAT3);
               /* test solution */
               if ( tflg ) {
                    for (i=0; i < dim; i++)
                         rgn->px[i] = sol[i];
                    cmputy(rgn);
                    printf("\n     -> pwf(solution) = ");
                    prdvctr(rgn->py,dim,FORMAT1);
               }
   end:;
          }
          else                                   /* W[1] is empty */
               printf("\n\t** No solution **\n");
          printf("\n\n** Total number of normal vectors computed: %d\n",itr);
}


/*
** conW2() -- construct W[2] from W[0] or W[1]; called by aspwlf().
*/

conW2 (w, wi)

register RGNQ      *w, *wi;
{
          register RGN      *rgn;
          register int      i, n;

          n = w->n;                              /* save # of regions on w */
          /*
          ** for each region on queue w, test the the specified bdry
          ** bit, if it is on, then put the region on queue wi,
          ** otherwise return it to queue w.
          */
          for (i=0; i < n; i++) {
               rgn = getrgn(w);
               if ( aflg ) printf("conW2: rgn from W[0&1]: %d\n",rgn->id);
               if ( rgn->bdry[kk] == 1 ) {
                    if ( aflg ) printf("conW2: rgn put on W[2]: %d\n",rgn->id);
                    putrgn(wi,rgn);
               }
               else {
                    if ( aflg )
                         printf("conW2: rgn put back to W[0&1]:\t%d\n",rgn->id);
                    putrgn(w,rgn);
               }
          }
}
```

```
/*
** compah() -- compute vector ah[] and offset[] since they
**         share the same codes; called by aspwlf().
*/

compah (vctr, rgn)

register double    *vctr;
register RGN       *rgn;
{
        register int       i, j, n;

        for (i=0; i < dim; i++) {
             vctr[i] = a[i];
             n = i*hyp;
             for (j=0; j < hyp; j++)
                  vctr[i] -= C[n+j] * e[j] * rgn->sgnsq[j];
        }
}


/*
** compBh() -- compute matrix Bh[,] and jcbn[,] since
**         they share the same codes; called by aspwlf().
*/

compBh (mtrx, rgn, flag, axis)

double    *mtrx;
RGN       *rgn;
int       flag, axis;
{
        register int       i, j, k, m, n, p;

        /* compute the whole matrix */
        if ( !flag ) {
             for (i=0; i < dim; i++) {
                  m = i*dim;
                  n = i*hyp;
                  for (j=0; j < dim; j++) {
                       mtrx[m+j] = B[m+j];
                       p = j*hyp;
                       for (k=0; k < hyp; k++)
                            mtrx[m+j] += C[n+k] * D[p+k] * rgn->sgnsq[k];
                  }
             }
        }
        /* compute the axis-th column of Bh[,] only */
        else {
             for (i=0; i < dim; i++) {
                  m = axis*hyp;
                  n = i*hyp;
                  mtrx[i] = B[i*dim+axis];
                  for (k=0; k < hyp; k++)
                       mtrx[i] += C[n+k] * D[m+k] * rgn->sgnsq[k];
             }
        }
}


/*
** sgntst() -- perform sign test; called by aspwlf().
*/
```

```
sgntst (rgn, alfa, beta)

register RGN        *rgn;
register double     *alfa, beta;
{
        double   Abs(), inprdct();
        int      Sgn();
        register int        sa, sb;
        register double     tmp;

        tmp = inprdct(alfa,rgn->py,dim) - beta;
        if ( Abs(tmp) < epsilon ) {
            if ( pflg ) error(4,"sgntst()",rgn);
            putrgn(W[3],rgn);
            return(0);
        }
        else {
            sa = Sgn(tmp);
            sb = 0 - Sgn(beta);
            if ( aflg ) printf("\tsa = %d, sb = %d",sa,sb);
            if ( sa == sb ) {
                if ( aflg ) printf("\n\trgn put on W[1]: %d\n",rgn->id);
                return(1);
            }
            else
                return(0);
        }
}


/*
** cpnrml() -- compute normal in y-space, store it in ynrml[];
**          returns  0: if successful,
**                   1: if numerical error occured,
**                   2: if Bh[,] is singular;
**          called by aspwlf().
*/

cpnrml (rgn)

register RGN        *rgn;
{
        double   det;
        short    dep;
        register int        i, err=0;

        compBh(Bh,rgn,0);                  /* compute matrix Bh[,] */
        /* compute alfah[] */
        for (i=0; i < dim; i++)
            sol[i] = D[i*hyp+kk];          /* use sol[] as alfa[] */
        if ( !imsl ) {
            transp(Bh,Bht,dim,dim);
            lineqn(Bht,alfah,sol,dim,0,&det);
        }
        else {
            leqt2f_(Bh,&imsl,&dim,&dim,sol,&sigdgt,wk,&ier);
            if ( ier > 128 ) {             /* numerical error */
                if ( pflg ) error(5,"cpnrml()",rgn,kk,Bh);
                rowech(Bh,wk,dim,dim,&det,&dep);
                if ( dep == 0 )
                    err = 1;
                else
```

```
                    err = 2;            /* Bh[,] is singular */
            }
            else
                for (i=0; i < dim; i++)
                    alfah[i] = sol[i];
        }
        if ( !err ) {
            itr++;
            for (i=0; i < dim; i++) {
                ynrml[mxcnt*dim+i] = alfah[i];
                markx[mxcnt*dim+i] = rgn->px[i];
            }
            if ( aflg ) {
                printf("\n* CPNRML: hyp->id: %d",kk);
                printf("\n   mxcnt=%d",mxcnt);
                printf("\n   markx[]: ");
                prdvctr(markx+mxcnt*dim,dim,FORMAT1);
                printf("\n   ynrml[]: ");
                prdvctr(ynrml+mxcnt*dim,dim,FORMAT1);
                printf("\n   Bh[,]:");
                prdmtrx(Bh,dim,dim,FORMAT3);
            }
            mxcnt++;
        }
        return(err);
}
```

```
/*
** init.c -- contains 7 routines:
**         init():          call rest routines to initialize.
**         nrmliz():        normalize D[,] & e[].
**         phgrps():        find parallel hyperplane groups.
**         dtrmnx():        compute trgn, rj.
**         dsub():          compute x[] & bd[].
**         lodrgn():        load all region information.
**         cmputy():        compute y[]=pwlf(x[]).
*/

#include "aspwlf.h"

int        trgn;                          /* total # of regions */
int        *bd;                           /* bd[trgn][hyp] */
double     *x;                            /* x[trgn][dim] */
int        *dcol;                         /* dcol[hyp] */
int        *ngrph;                        /* ngrph[dim] */


/*
** init() -- takes care of all necesary initializations described
**         in STEP 0; called by main().
*/

init ()
{
        register int      i, j, k;

        pwlf();                       /* initializing pwl function */
        prtcoef();                    /* print coefficients */

        for (i=0; i < 4; i++) {       /* allocate spaces */
             W[i] = (RGNQ *) palloc(sizeof(RGNQ));
             W[i]->head = W[i]->tail = RNIL;
             W[i]->n = 0;
        }
        Q = (QLST *) palloc(sizeof(QLST));
        Q->head = Q->tail = QNIL;
        Q->n = 0;
        dcol  = (int *) palloc(hyp*sizeof(int));
        ngrph = (int *) palloc(dim*sizeof(int));

        nrmliz();                     /* normalize D[,] and e[] */
        phgrps();                     /* find parallel hyperplane groups */

        trgn = 1;                     /* compute trgn */
        for (i=0; i < Q->n; i++)
             trgn *= ngrph[i];

        /*
        ** allocate space for x[]; if Q->n < dim, then those
        ** unassigned x[trgn][j], j > Q->n, will stay 0.
        */
        j = trgn*dim;
        x = (double *) malloc(j*sizeof(double));
        if ( Q->n < dim ) for (i=0; i < j; i++)
             x[i] = 0;

        /*
        ** allocate space for bd[]; all entries of bd[] are
        ** initialized to zero.
        */
        j  = trgn*hyp;
```

```
        bd = (int *) malloc(j*sizeof(int));
        for (i=0; i < j; i++)
             bd[i] = 0;

        dtrmnx();           /* determine x[] & bd[] in each region */
        lodrgn();           /* load all information for each region */

        if ( pflg ) prtq();

        /* free spaces */
        free(x); free(bd);
}


/*
** nrmliz() -- 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().
*/

nrmliz ()
{
        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 ) {
                       if ( !flag ) {      /* the only nonzero */
                            flag++;
                            /* normalizing */
                            if ( *dtmp != 1.0 ) {
                                 e[j] /= *dtmp;
                                 *dtmp = 1.0;
                            }
                            dcol[j] = i;   /* the i-th row in the j-th
                                              column is nonzero */
                       }
                       else                /* >= 2 nonzero entries */
                            error(1,"nrmliz()");
                  }
             }
             /* all entries in column j are 0 */
             if ( !flag ) error(1,"nrmliz()");
        }
}


/*
** phgrps() -- identical columns in D[,] represent parallel
**         hyperplanes; this routine groups those columns in sets
**         (each set corresponds to a HYPQ), allocates spaces for
**         each HYPQ and puts those HYPQs on the QLST Q; called by
**         init().
*/

phgrps ()
{
        int           flag, n, *tested;
        register int       i, j, k, count;
```

```
	register HYPQ	*hq;
	register HYP	*h;

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

	i = 0;
	count = 0;
	while ( count < hyp && i < hyp ) {
		/* allocate space & initialization */
		hq = (HYPQ *) palloc(sizeof(HYPQ));
		hq->head = hq->tail = HNIL;
		hq->n = 0;
		hq->axis = dcol[i];

		h = (HYP *) palloc(sizeof(HYP));
		h->id = i;			/* assign id */
		puthyp(hq,h);			/* put on list */

		tested[i]++;
		count++;
		flag = 0;			/* reset flag */
		k = -1;				/* reset k */
		/*
		** find parallel columns by searching for the same
		** dcol[j].
		*/
		for (j=i+1; j < hyp; j++)
		if ( !tested[j] ) {		/* if not tested */
			/* if parallel */
			if ( dcol[j] == dcol[i] ) {
				h = (HYP *) palloc(sizeof(HYP));
				h->id = j;
				puthyp(hq,h);
				tested[i]++;
				count++;
			}
			/* get the 1st nonparallel untested column */
			else if ( !tested[j] && !flag ) {
				k = j;
				flag++;
			}
		}
		n = hq->n + 1;			/* save the length */
		puthq(Q,hq);			/* put list on Q */
		ngrph[Q->n-1] = n;

		if ( Q->n > dim )		/* fatal error */
			error(1,"phgrps()");
		if ( k == -1 )
			break;			/* all are parallel */
		else
			i = k;			/* k = 1st nonparallel column */
	}
}


/*
** dtrmnx() -- this routine is called by init() and does the following
**	things:
**	1. compute rj for each HYPQ;
**	2. find rjmax;
**	3. call dsub() to compute x[] & bd[];
```

```
**        4. sort Q so that the rj for each HYPQ on Q is in increasing
**           order.
*/

dtrmnx ()
{
        HYPQ    *gethq(), *tmp;
        double  *dtmp;
        register int        i, j, k, n, period;
        register HYPQ      **vhq;

        rjmax = 0;                          /* initialize */
        n = Q->n;                           /* save the length */

        /* vhq[] contains pointers of HYPQ */
        vhq = (HYPQ **) palloc(n*sizeof(int));

        period = 1;                         /* starting period */
        for (i=0; i < n; i++) {
             vhq[i] = gethq(Q);
             vhq[i]->rj = trgn/ngrph[i];    /* compute rj */
             if ( rjmax < vhq[i]->rj )
                  rjmax = vhq[i]->rj;       /* get maximum */
             dsub(vhq[i],period);
             period *= ngrph[i];            /* change period */
        }

        /* SHELL sorting so that vhq[]->rj is in increasing order */
        for (k = n/2; k > 0; k /= 2) {
             for (i=k; i < n; i++)
             for (j  = i-k;
                  j >= 0 && vhq[j]->rj > vhq[j+k]->rj;
                  j -= k) {
                 tmp = vhq[j];
                 vhq[j] = vhq[j+k];
                 vhq[j+k] = tmp;
             }
        }

        /* put sorted objects back to Q */
        for (i=0; i < n; i++)
             puthq(Q,vhq[i]);

}


/*
** dsub() -- use SHELL sort to sort a HYPQ so that the
**         corresponding e[] (i.e. beta) is in increasing order;
**         compute x[] & bd[], note that x[] is actually x[trgn][dim],
**         only trgn x[][i]'s, 0 <= i <= dim-1, are assigned each time
**         this routine being called by dtrmnx().
*/

dsub (hq, p)

HYPQ      *hq;
int       p;
{
        HYP     *gethyp(), **vh, *tmp;
        double  *xi;
        register int        i, j, k, axis, n, r;
```

```
	n = hq->n;                          /* save the length */
	vh = (HYP    **) palloc(n*sizeof(int));
	xi = (double *) palloc((n+1)*sizeof(double));
	for (i=0; i < n; i++)
		vh[i]  = gethyp(hq);

	axis = dcol[vh[0]->id];             /* save axis */

	/* SHELL sorting so that beta is in increasing order */
	for (k = n/2; k > 0; k /= 2) {
		for (i=k; i < n; i++)
		for (j = i-k;
			j >= 0 && e[vh[j]->id] > e[vh[j+k]->id];
			j -= k) {
			tmp = vh[j];
			vh[j] = vh[j+k];
			vh[j+k] = tmp;
		}
	}

	/* compute xi[] */
	xi[0] = e[vh[0]->id] - 1.0;         /* left-most point */
	for (i=1; i < n; i++) {
		j = vh[i-1]->id;
		k = vh[i]->id;
		if ( e[j] == e[k] )
			error(2,"dsub()");
		else                            /* middle points */
			xi[i] = (e[j] + e[k]) / 2.0;
	}
	xi[n] = e[vh[n-1]->id] + 1.0;       /* right-most point */

	/* assign x[] & bd[] */
	r = 0;                              /* trgn counter */
	while ( r != trgn ) {
		for (i=0; i <= n; i++)
		for (j=0; j < p; j++) {
			x[r*dim+axis] = xi[i];
			k = r*hyp;
			if ( i == 0 )               /* left-most */
				bd[k+vh[i]->id] = 1;
			else if ( i == n )          /* right-most */
				bd[k+vh[i-1]->id] = 1;
			else {
				bd[k+vh[i-1]->id] = 1;
				bd[k+vh[i]->id] = 1;
			}
			r++;
		}
	}

	/*
	** put sorted hyp's back to hq in the alternating order:
	** '1,3,5,7,....,2,4,6,8,....'
	*/
	for (j=0; j < 2; j++)
		for (i=j; i < n; i+=2)
			puthyp(hq,vh[i]);
}
```

```
/*
** lodrgn() -- allocate space for each RGN; compute the sign
**        sequence; assign bdry, px, py, id; place RGN on the
**        queue W[0].
*/

lodrgn ()
{
        int        Sgn();
        register int       i, j, k, m, n;
        register RGN       *rgn;

        if ( aflg ) printf("\nRegions' information: - lodrgn()\n");

        for (k=0; k < trgn; k++) {
            /* allocate spaces */
            rgn =          (RGN *) palloc(sizeof(RGN));
            rgn->sgnsq = (int *) palloc(hyp*sizeof(int));
            rgn->bdry  = (int *) palloc(hyp*sizeof(int));
            rgn->px = (double *) palloc(dim*sizeof(double));
            rgn->py = (double *) palloc(dim*sizeof(double));

            m = k*dim;
            n = k*hyp;
            for (i=0; i < dim; i++)         /* assign rgn->px[] */
                rgn->px[i] = x[m+i];
            for (j=0; j < hyp; j++)         /* assign rgn->bdry[] */
                rgn->bdry[j] = bd[n+j];
            /*
            ** compute sign sequences.
            ** note that since columns of D[,] are unit vectors,
            ** only one component of rgn->px[] is needed.
            */
            for (j=0; j < hyp; j++) {
                rgn->sgnsq[j] = Sgn(rgn->px[dcol[j]]-e[j]);
            }
            cmputy(rgn);                    /* compute rgn->py[] */
            rgn->id = k + 1;                /* set region id */
            if ( pflg ) {                   /* print regions */
                printf("\n* region %d",k+1);
                prtrgn(rgn,":");
            }
            putrgn(W[0],rgn);               /* place region on W[0] */
        }
}


/*
** cmputy() -- compute y[] = pwlf(x[]) for each given region;
**        called by lodrgn().
*/

cmputy (rgn)

register RGN       *rgn;
{
        register int       i, j, k, m, n;

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

```
                for (k=0; k < hyp; k++) {
                    rgn->py[i] += C[n+k] * rgn->sgnsq[k]
                            * (rgn->px[dcol[k]]-e[k]);
                }
            }
}
```

```
/*
** queue.c -- containing 6 queue-lists manipulating routines:
**        putrgn(), getrgn(), puthyp(), gethyp(), puthq(), gethq().
**
**        putrng() & getrgn():      RGNQ.
**        puthyp() & gethyp():      HYPQ.
**        puthq() & gethq():        QLST.
*/

#include "aspwlf.h"


/*
** putrgn() -- places RGN at the end of RGNQ, it always assumes
**        queue is not empty.
*/

putrgn (rgnq, rgn)

register RGNQ      *rgnq;
register RGN       *rgn;
{
        rgn->link = RNIL;
        /* if queue was initially empty */
        if (rgnq->head == RNIL) {
            rgnq->head = rgn;
            rgnq->tail = rgn;
        }
        /* if queue was not empty, append at the end */
        else {
            rgnq->tail->link = rgn;
            rgnq->tail = rgn;
        }
        rgnq->n++;
}


/*
** getrgn() -- gets one RGN from the front of RGNQ and returns
**        a pointer to that RGN; it returns NIL if the RGNQ is empty.
*/

RGN       *getrgn (rgnq)

register RGNQ      *rgnq;
{
        register RGN       *rgn;

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


/*
** puthyp() -- places HYP at the end of HYPQ, it always assumes
**        queue is not empty.
*/
```

```
puthyp (hq, h)

register HYPQ     *hq;
register HYP      *h;
{
        h->link = HNIL;
        /* if queue was initially empty */
        if (hq->head == HNIL) {
            hq->head = h;
            hq->tail = h;
        }
        /* if queue was not empty, append at the end */
        else {
            hq->tail->link = h;
            hq->tail = h;
        }
        hq->n++;
}


/*
** gethyp() -- gets one HYP from the front of HYPQ and returns
**         a pointer to that HYP; it returns NIL if the HYPQ is empty.
*/

HYP       *gethyp (hq)

register HYPQ     *hq;
{
        register HYP      *h;

        h = HNIL;         /* if queue is empty, return NIL */
        /* if queue is not empty, get one from the front */
        if (hq->head != HNIL) {
            h = hq->head;
            hq->head = hq->head->link;
            hq->n--;
        }
        return(h);
}


/*
** puthq() -- places HYPQ at the end of QLST, it always assumes
**         queue is not empty.
*/

puthq (qlst, hq)

register QLST     *qlst;
register HYPQ     *hq;
{
        hq->link = QNIL;
        /* if queue was initially empty */
        if (qlst->head == QNIL) {
            qlst->head = hq;
            qlst->tail = hq;
        }
        /* if queue was not empty, append at the end */
        else {
            qlst->tail->link = hq;
            qlst->tail = hq;
        }
        qlst->n++;
```

```
}

/*
** gethq() -- gets one HYP from the front of QLST and returns
**         a pointer to that HYP; it returns NIL if the QLST is empty.
*/

HYPQ    *gethq (qlst)

register QLST    *qlst;
{
        register HYPQ    *hq;

        hq = QNIL;        /* if queue is empty, return NIL */
        /* if queue is not empty, get one from the front */
        if (qlst->head != QNIL) {
            hq = qlst->head;
            qlst->head = qlst->head->link;
            qlst->n--;
        }
        return(hq);
}
```

```
/*
** print.c -- containing 4 printing routines:
**        prtcoef(), prtrgn(), prthq(), prtq().
**
**        Routine "prtcoef()" is for printing the coefficients
**        of the piecewise-linear function; prtrgn(), prth() &
**        prtq() are called if the "pflg" flag is set.
*/

#include "aspwlf.h"


/*
** prtcoef() -- print coefficients of the pwlf(.).
*/

prtcoef ()
{
        printf("\nCoefficients of the piecewise-linear function:");

        printf("\n\na[]:\t");
        prdvctr(a,dim,FORMAT1);

        printf("\n\nB[,]:");
        prdmtrx(B,dim,dim,FORMAT1);

        printf("\nC[,]:");
        prdmtrx(C,dim,hyp,FORMAT1);

        printf("\nD[,]:");
        prdmtrx(D,dim,hyp,FORMAT2);

        printf("\ne[]:\t");
        prdvctr(e,hyp,FORMAT1);

        printf("\n");
}


/*
** prtrgn() -- print the sign sequence, boundaries x[] and y[]
**        in the given region.
*/

prtrgn (rgn, str)

register RGN      *rgn;
register char     *str;
{
        register int      k;

        printf("%s",str);

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

        printf("\nboundries: ");
        for (k=0; k < hyp; k++)
             printf("%2d ",*(rgn->bdry+k));

        printf("\nx[]: ");
        prdvctr(rgn->px,dim,FORMAT1);
```

```
        printf("\ny[]: ");
        prdvctr(rgn->py,dim,FORMAT1);

        printf("\n");
}


/*
** prthq() -- print the given HYPQ.
*/

prthq (hq, str)

register HYPQ     *hq;
register char     *str;
{
        HYP     *gethyp();
        register HYP      *h;
        register int      nhq;

        printf("\n%s-queue: ",str);
        nhq = hq->n;

        while ( nhq != 0 ) {
            h = gethyp(hq);
            printf("%d,",h->id);
            puthyp(hq,h);
            nhq--;
        }
        printf("\n");
}


/*
** prtq() -- print the id of each hyperplane on the structure QLST.
*/

prtq ()
{
        HYP     *gethyp();
        HYPQ    *gethq();
        register HYP      *h;
        register HYPQ     *hq;
        register int      nq;

        printf("\nQ-list:");
        nq = Q->n;
        while ( nq != 0 ) {
            hq = gethq(Q);
            printf("\n* n=%d, rj=%d",hq->n,hq->rj);
            prthq(hq,"* hyp");
            puthq(Q,hq);
            nq--;
        }
        printf("\n");
}
```

```
/*
** error.c -- prints error messages.
*/

#include "aspwlf.h"

error (flag, str, rgn, hid, mtrx, vctr)

register int        flag, hid;
register char       *str;
register double     *mtrx, *vctr;
register RGN        *rgn;
{
        if ( flag < 3 )
            printf("\n\n**ERROR: [in routine: %s]:\n\t",str);
        else
            printf("\n\n**WARNING: [from routine: %s]:\n\t",str);

        switch ( flag ) {
        case 0:                         /* in aspwlf() */
            printf("can not recover numerical error.");
            printf("\n\toccured at\tregion: %d;",rgn->id);
            printf("\thyperplane: %d",hid);
            break;
        case 1:                         /* in nrmliz() & phgrps() */
            printf("Matrix D[,] is not compatible with ");
            printf("hybrid representation.");
            break;
        case 2:                         /* in dsub() */
            printf("Vector e[] is not compatible with");
            printf("hybrid representation.");
            break;
        case 3:                         /* in aspwlf() */
            printf("betah = 0");
            printf("\n\toccured at\tregion: %d;",rgn->id);
            printf("\thyperplane: %d",hid);
            printf("\nah[] = ");
            prdvctr(vctr,dim,FORMAT1);
            printf("\nBh[,]:");
            prdmtrx(mtrx,dim,dim,FORMAT1);
            break;
        case 4:                         /* in sgntst() & aspwlf() */
            printf("region %d is a degenerate region.\n",rgn->id);
            break;
        case 5:                         /* in cpnrml() */
            printf("matrix Bh[,] ");
            break;
        case 6:
            printf("Jacobian matrix J[,] ");
            break;
        }

        switch ( flag ) {
        case 5:
        case 6:
            if ( ier == 129 )
                printf("is algorithmically singular.");
            else if ( ier == 131 ) {
                printf("is too ill-conditioned for iterative\n");
                printf("\t\timprovement to be effective.");
            }
            printf(" [from IMSL]");
            printf("\n\toccured at\tregion: %d;",rgn->id);
            printf("\thyperplane: %d",hid);
```

```
		printf("\nmatrix:");
		prdmtrx(mtrx,dim,dim,FORMAT3);
		if ( flag == 6 )
			printf("\n**-< solution not computed >-**");
		break;
	}

	if ( flag > 2 )
		printf("\n**-< program continued >-**\n");
	else {
		printf("\n\n**-< program aborted >-**\n");
		exit(1);
	}
}
```

```
/*
** support.c -- contains supporting routines to the ASPWLF
**        programs:
**        Abs(), Sgn(), inprdct(), transp(), prdmtrx(),
**        privctr(), prdvctr(), lineqn(), rowech(), palloc().
*/


/*
** 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 (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 (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 (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 (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]);
}
```

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

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));
	}
}
```

```
/*
** interac.c -- outputs a C-program which defines an N-dimensional
**         piecewise-linear function "pwlf()"; the input is taken
**         from user's terminal.
*/

#include <stdio.h>

#define  pf      printf                     /* abbreviations */
#define  fpf     fprintf
#define spf      sprintf

FILE     *fp;
int      dim, hyp;
char     str[BUFSIZ];

char     cc[]     = "/bin/cc -p -O";
char     lib[]    = "Alib -limsld -lF77 -lI77 -lm";
char     flg1[]   = "\t'-i': use IMSL routine\n";
char     flg2[]   = "\t'-t': test solution\n";
char     flg3[]   = "\t'-p': print hyp & rgn\n";
char     flg4[]   = "\t'-a': print all iteration details\n";

main (argc, argv)

int      argc;
char     *argv[];
{
         FILE     *fopen();
         char     *ctime(), *l_get(), *s_get(), buf[BUFSIZ];
         int      i_get(), tim[2];
         register char    *str, *tl;
         register int     k, trgn;

         if ( argc == 1 || (fp = fopen(argv[1], "w")) == NULL ) {
              pf("Usage: interac file.c\n");
              exit(1);
         }

         pf("\nEnter title: ");
         tl = s_get();

         time(tim);
         fpf(fp,"/*\t- %-s -\n", argv[1]);
         fpf(fp,"**\n** %s\n**\n** %24.24s\n*/\n\n", tl,ctime(tim));

         pf("\n** Enter coefficients of the PWL function **");
         pf("\n\tEnter the row dimension of a[]: ");
         dim = i_get();
         pf("\tEnter the column dimension of D[,]: ");
         hyp = i_get();
         fpf(fp,"\nint\tdim=%d, hyp=%d;\n",dim,hyp);
         fpf(fp,"\ndouble\t*a, *B, *C, *D, *e;\n");

         fpf(fp,"\npwlf ()\n{");
         sub1("a","double","dim","1");
         sub1("B","double","dim","dim");
         sub1("C","double","dim","hyp");
         sub1("D","double","dim","hyp");
         sub1("e","double","hyp","1");
         fpf(fp,"\n");

         pf("\n    Enter vector a[]:");
         sub2(1,"a",dim,0,0);
```

```
        pf("\n    Enter matrix B[,]:");
        sub2(2,"B",dim,dim,0);
        pf("\n    Enter matrix C[,]:");
        sub2(2,"C",dim,hyp,0);
        pf("\n    Enter matrix D[,]:");
        sub2(2,"D",dim,hyp,0);
        pf("\n    Enter vector e[]:");
        sub2(1,"e",hyp,0,0);

        fpf(fp,"\n\tprintf(\"\\n%s\\n\");\n}\n", tl);
        fclose(fp);
        pf("\n** output file is %s **\n", argv[1]);

        /* continue excution */
        pf("\nContinue to excute ? [y,n] ");
        str = l_get();
        if (*str != 'y') exit(1);

        /* compiling */
        spf(buf,"%s %s %s",cc,argv[1],lib);
        pf("\n%s\n", buf);
        system(buf);

        /* excuting */
        pf("\n\007Ready to excute, command line flags are:\n");
        pf("%s%s%s%s",flg1,flg2,flg3,flg4);
        pf("Invoke flag(s): ");
        str = s_get();
        spf(buf,"./a.out %s", str);
        system(buf);
}


/*
** sub1() -- write to the output program the lines containing
**        "malloc()".
*/

sub1 (s1, s2, s3, s4)

register char     *s1, *s2, *s3, *s4;
{
        fpf(fp,"\n\t%s = (%s *) malloc(%s*%s*sizeof(%s));",
            s1,s2,s3,s4,s2);
}


/*
** sub2() -- write to the output program the lines of arrays.
*/

sub2 (flg, s, row, col, rgn)

char      *s;
register int      row, col, rgn;
{
        double   d_get();
        int      i_get();
        register int      i, j;

        switch ( flg ) {
```

```
        case 1:                              /* a[], e[] */
            for (i=0; i < row; i++) {
                pf("\n\t%s[%d] = ",s,i+1);
                fpf(fp,"\n\t%s[%d] = %16.9e;",s,i,d_get());
            }
            break;
        case 2:                              /* B[,], C[,], D[,] */
            for (i=0; i < row; i++) {
                pf("\n row %d:",i+1);
                for (j=0; j < col; j++) {
                    pf("\n\t%s[%d,%d] = ",s,i+1,j+1);
                    fpf(fp,"\n\t/* %s[%d,%d] */",s,i+1,j+1);
                    fpf(fp," %s[%d] = %16.9e;",s,i*col+j,d_get());
                }
            }
            break;
        default:                             /* x[,], bd[,] */
            for (j=0; j < col; j++) {
                pf("\n\t%s[%d] = ",s,j+1);
                fpf(fp,"\n\t/* %s[%d,%d] */",s,rgn+1,j+1);
                if (flg == 3)                /* for x */
                    fpf(fp," %s[%d] = %16.9e;",s,rgn*dim+j,d_get());
                else                         /* for bd */
                    fpf(fp," %s[%d] = %d;",s,rgn*hyp+j,i_get());
            }
            break;
        }
        fpf(fp,"\n");
}


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

int     i_get ()
{
        int     atoi();

        fgets(str,sizeof str,stdin);
        return(atoi(str));
}


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

double  d_get ()
{
        double  atof();

        fgets(str,sizeof str,stdin);
        return(atof(str));
}


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

char    *l_get ()
{
        register char   *c;
```

```
        c = (char *) malloc(BUFSIZ*sizeof(char));
        fgets(c,sizeof c,stdin);
        return(c);
}


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

char    *s_get ()
{
        register char   *c;

        c = (char *) malloc(BUFSIZ*sizeof(char));
        gets(c);
        return(c);
}
```

```
# File maintenance for ASPWLF programs.

CFLAGS = -O -p
E =      ex.c

FILE =   Makefile\
         aspwlf.h main.c aspwlf.c init.c print.c queue.c error.c\
         interac.c support.c

OBJS =   main.o aspwlf.o init.o print.o queue.o error.o support.o

Alib:            $(OBJS)
         ar ru Alib $(OBJS); ranlib Alib

main.o:          aspwlf.h main.c
aspwlf.o:        aspwlf.h aspwlf.c
init.o:          aspwlf.h init.c
print.o:         aspwlf.h print.o
queue.o:         aspwlf.h queue.c
error.o:         aspwlf.h error.c
support.o:       support.c

interac:
         cc -O -o interac interac.c; strip interac

run:
         cc $(CFLAGS) $(E) Alib -limsld -lF77 -lI77 -lm
```