IMPLEMENTATION OF A SPARSE MATRIX PACKAGE FOR SAMSON

by

Demetrios J. Giannopoulos

DRC-01-11-82

April, 1982

Implementation Of A Sparse Matrix Package[1] For SAMSON

(System for Activity • directed Mixed Simulation of Networks)

by
Demetrios J. Giannopoulos

Department of Electrical Engineering

Carnegie-Mellon University

July, 1981

[1]Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science. This work was supported in part by the U.S. Army Research Office under Grant DAAG/29/79/C/0213

Table of Contents

List of Figures

List of Tables

Στους γονεις μου, Γιαννη και Ελενη

To my parents, John and Helen

ACKNOWLEDGEMENTS

I am particularly grateful to my advisor Steve Director for the many fruitful discussions and suggestions which have proved valuable in many ways.

I wish also to thank Karem Sakallah, the writer of SAMSON, for his invaluable help and patience in explaining his program in detail.

Chapter 1
INTRODUCTION

Simulation of integrated circuits requires the solution of a great number of, in general, nonlinear algebraic equations. The handling of these equations demands specific techniques in order to surmount the problems which arise from their bulk. Many circuit simulators have been developed in the past, aiming to efficiently solve circuit equations. Recently, multi-level simulators were developed which, in addition to allowing the description and simulation of electronic networks at various level of detail (e.g. logic-gate and circuit levels), were aimed at optimizing the simulation process. One such simulator recently developed at *CMU* is called SAMSON [Sakallah 81].

The purpose of this work was to develop a sparse matrix package for SAMSON. Specifically this package is employed in the analysis of networks described at the circuit level.The development of the package is based upon the concept of modular simulation, which was motivated by the modular structure of VLSI circuits. This modularity springs from the use of elementary network configurations, such as logic gates,to build up large-scale digital systems.

1.1. A Brief Description of SAMSON

SAMSON is essentially composed of two parts: an input phase, called SAM1, and the simulation part, SAM2. The input phase has a powerful input language permitting mixed level description of networks, and hierarchical network description. As an example, any network may be divided into various subnetworks, which, in turn, can be divided into lower order subnetworks. Usually many subnetworks are similar so that the same description may be advantageously reused to define all of them. The definition of a subnetwork is quite general, it may contain a single transistor, a logic gate or even a more complicated structure such as a Full Adder. The user is free to decide which parts of his circuit will constitute a subnetwork.

Another advantage originated from the generality of the subnetwork concept is the independence of the subnetwork definition from the underlying technology.

The input phase employs our sparse matrix package to analyze all subnetworks included in the network (Ascription. Processed subnetworks may be stored in a library to be used later for simulating other networks which include these subnetworks in their description. This is possible by careful separation of subnetwork modeling and subnetwork interconnection in the network description. Processed subnetworks and tables describing their interconnection are employed by the simulation part of SAMSON. The configuration of the SAMSON system is shown in Figure 1-1.

SAMSON employs the concept of activity-directed circuit simulation in network analysis. The activity of a subnetwork is connected with the rates of change of its variables. In a transient simulation it can be measured by the sizes of the integration steps which are taken to maintain truncation errors within prescribed limits. Therefore, integration steps may serve as a convenient measure of activity. Whereas tranditional circuit simulation constrains all subnetwork to use the same step size, necessarily the smallest, an activity-directed approach allows each subnetwork to follow its own time trajectory. At each instance of time, subnetworks are divided into an alert and a dormant group, based on the relative magnitutes of their steps. Only alert subnetworks are simulated, with dormant subnetworks participating as passive entities whose outputs are extrapolated and propagated to the inputs of alert subnetworks.

1.2. Formulation of Network Equations

Consider a network consisting of a subnetworks. SAMSON classifies the variables associated with subnetwork S_i as follows

- Internal variables, which, include algebraic variables, $x_s(t) * col^2(v^, i^, v^)$, where v^, i^, and v^ are the vectors of branch voltages, branch currents and node voltages respectively; and differential variables $q_{|t}$ the vector of reactive charges and fluxes.

- External variables, which include inputs $u_i(t)$, i.e, the terminal variables (currents or voltages) considered independent (externally specified) when the

[2] col stands for column

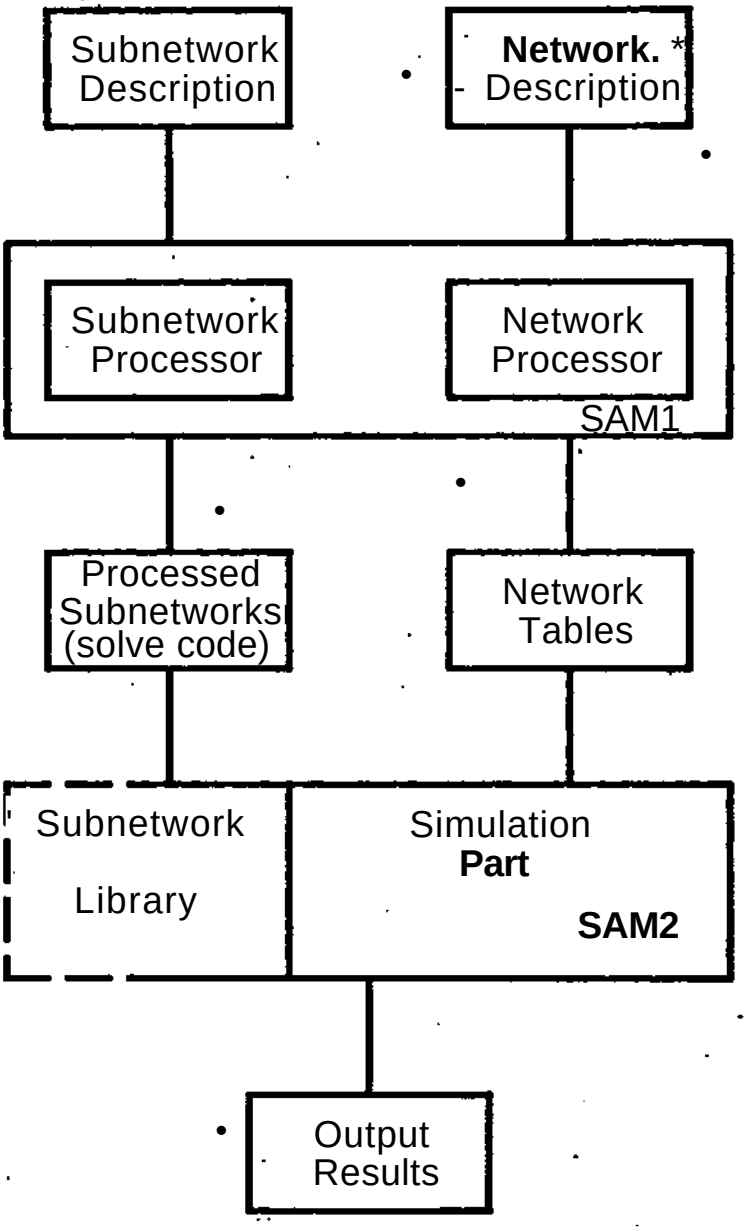


Figure 1-1: Configuration of SAMSON System

subnetwork is solved; and outputs y^ t) , i.e., the terminal variables that are considered dependent, computed when the subnetwork is solved.

SAMSON employs the Tableau formulation [Hachtel 71] of the network equations. According to this approach, the equations describing subnetwork S_i may be written as follows

$$Q_i(*,(t).\ Q|(t), t) \text{ » } 0 \quad \text{(Branch Constitutive Relations)} \tag{1.1}$$

$$h,(\ Xjfl),\ fyt)) \text{ » } Ej\ x_{\dot{y}}(t) \cdot \dot{q}fi) \text{ » } 0 \quad \text{(Reactive Branch Definitions)} \tag{1.2}$$

$$f_s(\ <ij(t), x,(t), y,(t)) \text{ » } 0 \quad \text{(KVL and KCL Constrains)} \tag{1.3}$$

u_i and y_i are n_i-dimension vectors where IX is the number of S^s terminals excluding the reference node.

For a complete network description, in addition to the equations describing each subnetwork, we need the connection equations which define the interconnection pattern between different subnetworks. Since subnetworks are connected at their terminals, connection equations involve only input and output variables. We can reduce equations (1.1) -(1.3) to:

$$G(z(t),g(t),t)\text{«}O \tag{<1.4}$$

$$H(z(t),5(t))\text{»}Ez(t).5(t)\text{»}O \tag{1.5}$$

where z(t) « cd(x(t), y(t), u(t)).

In transient circuit analysfs the approximations z_i and q_i to z(t) and g(t) at a sequence of time points ^ < t, < ... are to be found. At time instant t_n we replace the time derivative with a stiff integration formula [Brayton 72] of order k

$$h_j \dot{q}_n = -\Sigma^{*}_{i=0}\ \alpha_i q_{n-i} \tag{1.6}$$

where h_n»t_n.t_{n-r}

Substituting of (1.6) into equations (1.4) and (1.5) yields

(1.7)

where «$_n$ » col($z_{n>}$ Q_n).
$n>$

In general (1.7) constitutes a set of nonlinear algebraic equations. Upon application of the Newton-Raphson scheme to (1.7) we obtain the coefficient matrix of the linearized system. This coefficient matrix, which will be considered more fully later, has the structure shown in Figure 1-2. Observe that the coefficient matrix will be very sparse, and that many elements are constants of value either +1 or -1, which are called *topological elements.* The other nonzero entries are called *unique entries.*

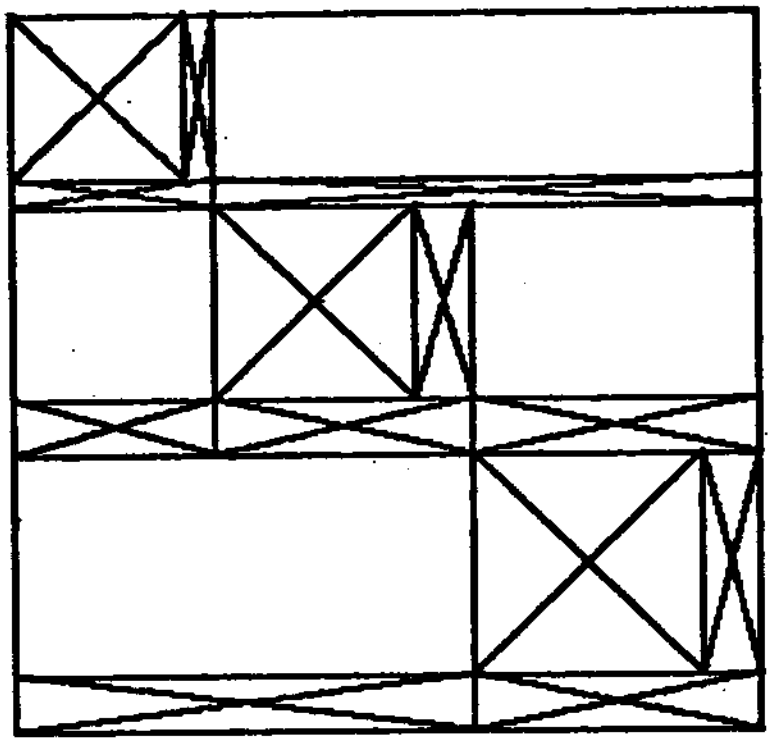


Figure 1-2: Initial Form of Coefficient Matrix

1.3. Motivation for Modular Simulation

Integrated circuits, especially those used to build digital systems,are composed of a large number of a few basic types of subnetworks. Instances of such types most commonly used are the logic gates OR,AND,NOT and the FLIP-FLOP . As a result there is an inherent modularity in the structure of integrated circuits. This modularity implies that several parts of the coefficient matrix are kJentical,corresponding to basic networks of the same type. We can employ this feature of the coefficient matrix, as well as its sparse structure, to obtain a very efficient solution procedure.

Initially the coefficient matrix has the form shown in Figure 1-2. By reordering the variables of the system we can obtain the doubly bordered block diagonal form shown in Figure 1-3. This form is particularly well suited for programming and solving each block separately. Observfe that since each of the blocks on the diagonal of the coefficient matrix corresponds to a subnetwork, and since a given subnetwork may appear more than once in a network,

several diagonal blocks possess the same structure. Thus we need only produce solution code once for each unique subnetwork, and employ the same code for each instantiation of the subnetwork.

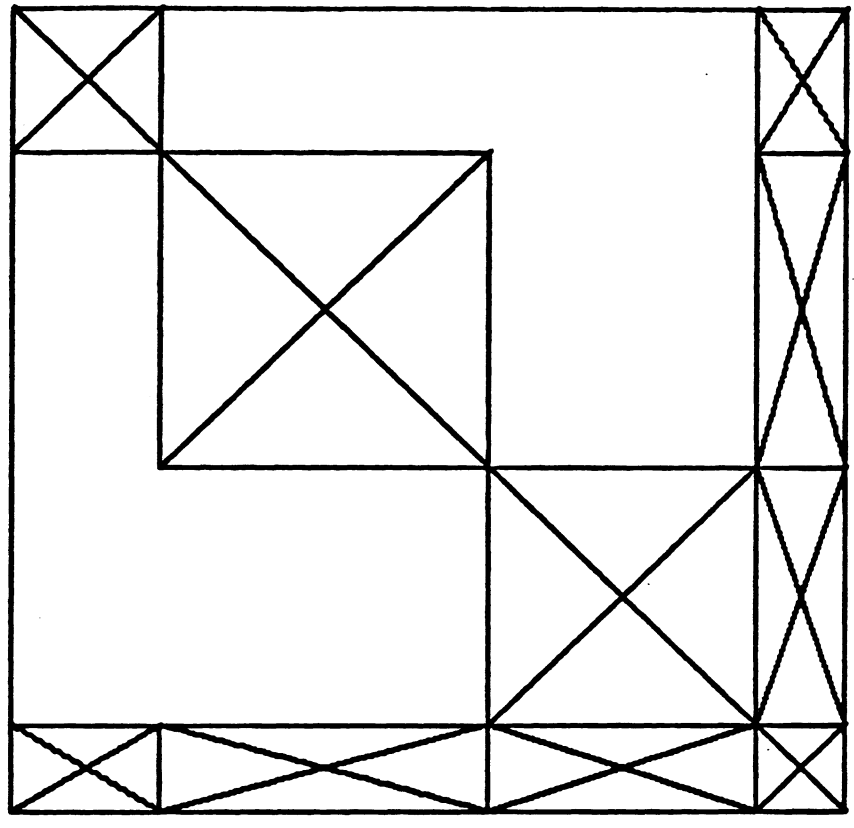


Figure 1-3: Reordered Form of Coefficient Matrix

1.4. Selection of the Approach to the Solution of Sparse Simultaneous Linear Equations

We now consider three aproaches which can be employed for solving the sparse matrix structure resulting from SAMSON's formulation of the network equations.

In the first approach, called the *compiled code approach*, loop-free code is generated which can rapidly solve all sets of equations that have the same underlying structure. The main advantage of this approach is extremely fast execution of the code. A disadvantage of this technique is that the generated code is often so long as to require out-of-core storage, thereby slowing execution. Furthermore, implementation of this approach in a high level language is difficult.

The second approach is known as the *looping indexed approach*. This approach actually performs all of the operations (*LU* decomposition, forward and backward substitution) on the coefficient matrix while analyzing its structure, and generates a sequence of ordering arrays and indexing vectors which can then be used to process matrices of the same structure. The advantages of this approach are that the decomposition is less time-consuming than that in the compiled code approach, and that the information generated requires much less storage. However, the use of indirect addressing results in slower execution, and use of this method precludes the use of variability typing (see below).

The last approach, known as the *interpretable code approach*, differs from the compiled code approach in that operation codes and addresses replace the specific code. That is, a table of operation codes and coefficient addresses are generated, which later , interpreted by a program, actually perform the solution. This approach is superior to the compiled code formulation in the amount of storage required, although the storage needed often considerably exceeds that required by the looping indexed approch. Its execution speed lies between the other methods, as does the time for the initial analysis of the structure.

The compiled code approach is considered to be the best if the size of the coefficient matrix does not require out-of-core storage. Modular simulation, however, reduces the amount of the solve code, since only the subnetworks, instead of the whole coefficient matrix, are processed. Thus overcoming the critical problem of space. Also, use of the same code to analyze more than one diagonal block reduces the time needed for the initial analysis of the structure. Evidently, modular simulation in circuit analysis favors the adoption of the compiled code approach.

Chapter 2
SPARSE MATRIX METHODS

In the previous chapter we came to the conclusion that the application of modular simulation reduces the analysis of a simulated circuit to the solution of several sparse systems of linear equations, each of them describing a part of the circuit defined as a subnetwork. We devote this chapter to the techniques applied to solving sets of sparse equations. Storage requirements, the ordering of variables and equations in conjuction with variability typing, and the reduced Crout algorithm used in matrix decomposition are subjects of interest

2.1. Crout Reduction

This section deals with the solution of a system of N linear algebraic equations in N unknowns

$$Ax=b \tag{2.1}$$

based on the Crout method [Duff 77], which consists of factoring A into a product (Figure 2-2)

$$A \ll L\,U \tag{2.2}$$

where L is a lower triangular matrix and a is an upper unit triangular matrix. Crout reduction is nothing more than a scheme for Gaussian elimination which provides a considerable savings in accessing memory. After factorization, system (2.1) can be easily solved by first performing forward elimination

$$Ly*b \tag{2.3}$$

followed by a backward substitution

$$Ux*y. \tag{2.4}$$

Specifically if we denote by a_{ij}, L_{ij} and u_{ij} the components of A, L and U_y respectively, where $1 < i, j \backslash N$, then the elements of the factors L and U can be computed for any given m, $1< m < N$ from the formulae

$$l_{im} = a_{im} - \sum_{\mu=1}^{m-1} l_{i\mu}\, u_{\mu m}, \quad i = m, m+1, \ldots, N. \tag{2.5}$$

$$u_{mj} = (a_{mj} - \Sigma_{\mu=1}^{m-1} l_{m\mu} u_{\mu j}) / l_{mm}, \quad j = m+1, m+2, \ldots, N. \tag{2.6}$$

In forward elimination, (2.3), we compute

$$y_i = (\ldots - \Sigma \ldots \wedge y_i) \ldots, \quad i = 1, 2, \ldots, N \tag{2.7}$$

and then in backward substitution, (2.4),

$$\ldots \tag{2.8}$$

In order to reduce computation time, only the operations in equations (2.5)-(2.8) which involve nontopological components of *L* and *U* are performed, the effect of elements equal +1 is included during code generation.

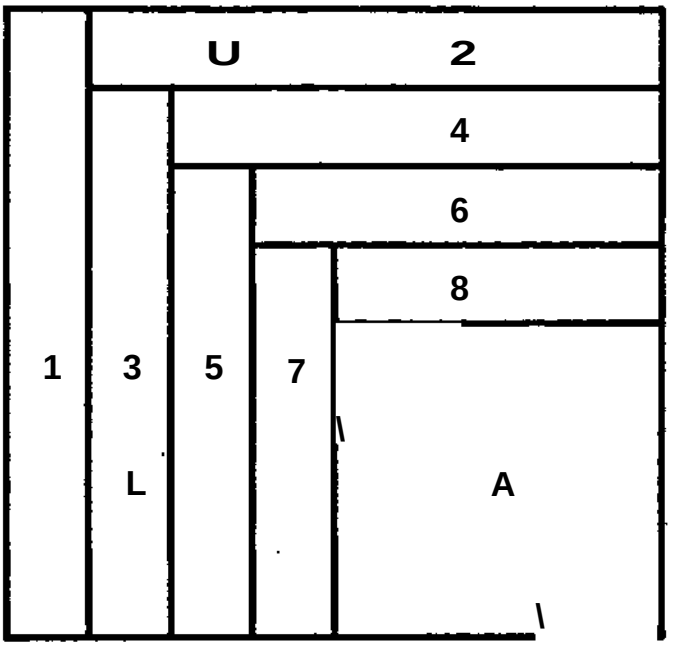


Figure 2-1: Crout Reduction

2.2. Sparse Matrix Storage

The structure of a sparse matrix is particularly favourable for the development of an efficient storage scheme since we need to store only the nonzero elements. We expect a good storage scheme to fulfill two objectives:

- The matrix entries should be readily accessible
- the merory space required should be kept low.

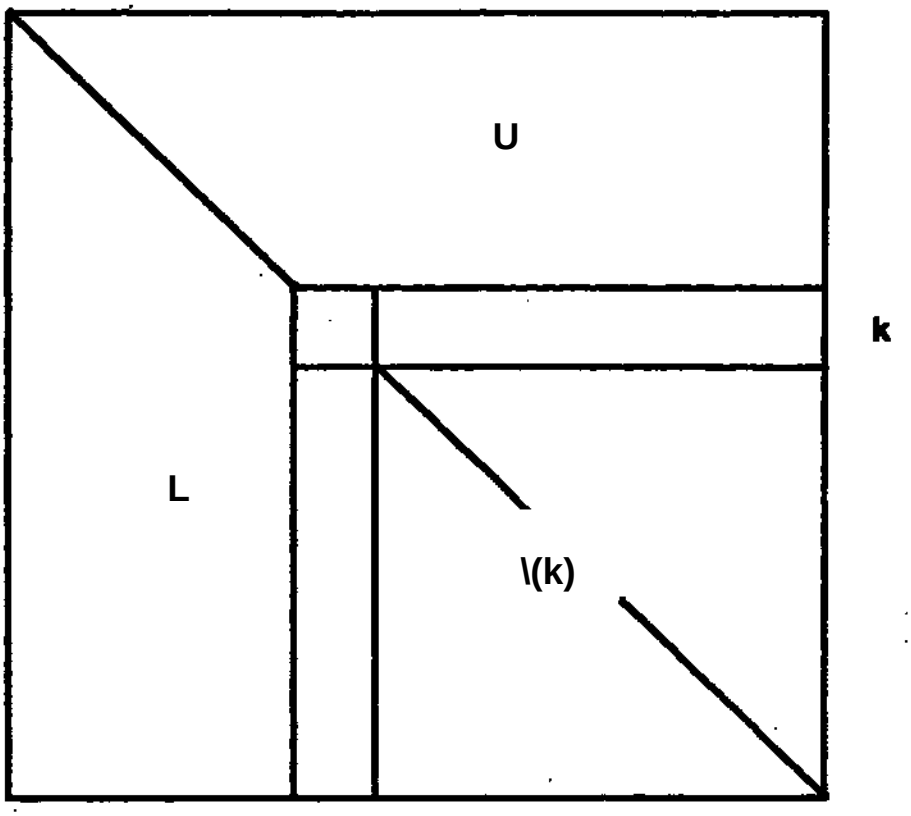
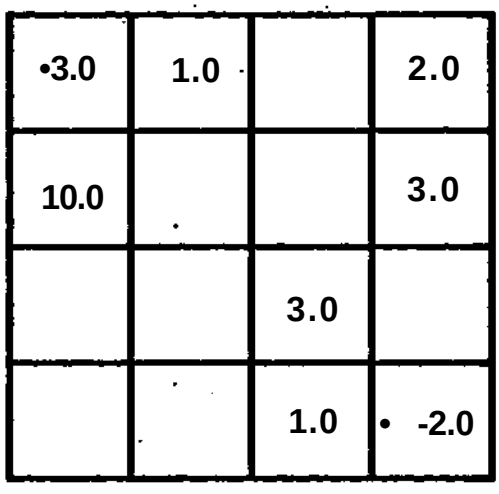


Figure 2-2: LU Decomposition

In general, schemes which result in minimum storage requirements usuatfy lack accessibility. Therefore, there is no optimal scheme for all cases. Storage schemes can be classified into two categories. In *static schemes,the* matrix entries are merely accessed while the sparsity structure is not allowed to change. In *dynamic schemes,* the matrix structure is permitted to grow (or shrink) during the course of a computation.

A common static method requires three arrays for the representation of a sparse matrix
1)a row pointer array, denoted PA, which is used to identify the column indices that belong to a given row
2)a column index array, denoted IA, which holds the column indices of the stored entries and
3)a real array, called Value, which is employed to store the values of the nonzero elements.
This scheme is called a row pointer/column index method and we illustrate it with an example in Tabte 2-1. There is a similar static scheme called column pointer/row index method.These storage schemes are particularly useful for storing sparse matrix information in a file, or when doing operations like matrix-vector multiplication which do not change the sparsity structure.



Value	•	(	-3.0.	1.0.	2.0.	10.0.	3.0.	3.0.	1.0.	-2.0)
IA	. •	(	1.	2.	4.	1.	4.	3.	3.	4)
PA	-	(	1.			4.		6.	7.	9)

Table 2-1: Example of row pointer/column index Static Storage.

In our program we employ this storage scheme in the input phase to store the jacobian of each network, while processing the subnetworks included in network description. A row pointer/ column index static scheme is also used for storing the elements of the factors *L* and *U* of the initial matrix.

An important class of dynamic schemes employ linked data structures. Linking is one of the most flexible representational schemes, it is easy to insert or delete elements and to reutilize any free space caused by such a deletion. With each nonzero entry is associated a data element of the linked list comprising

1)row index

2)column index

3)reference to the next element in the row

4)reference to the next element in the column.

K the sparsity structure is altered all we need is an adjustment of the references. This possibity renders the dynamic storage schemes very attractive for storing a sparse matrix when pivoting is being performed. In fact, this scheme is adopted for ordering the subnetwork

jacobians in our program. Figure 2-3 ilustrates this class of storage scheme using the same matrix we employed in Table 2-1.

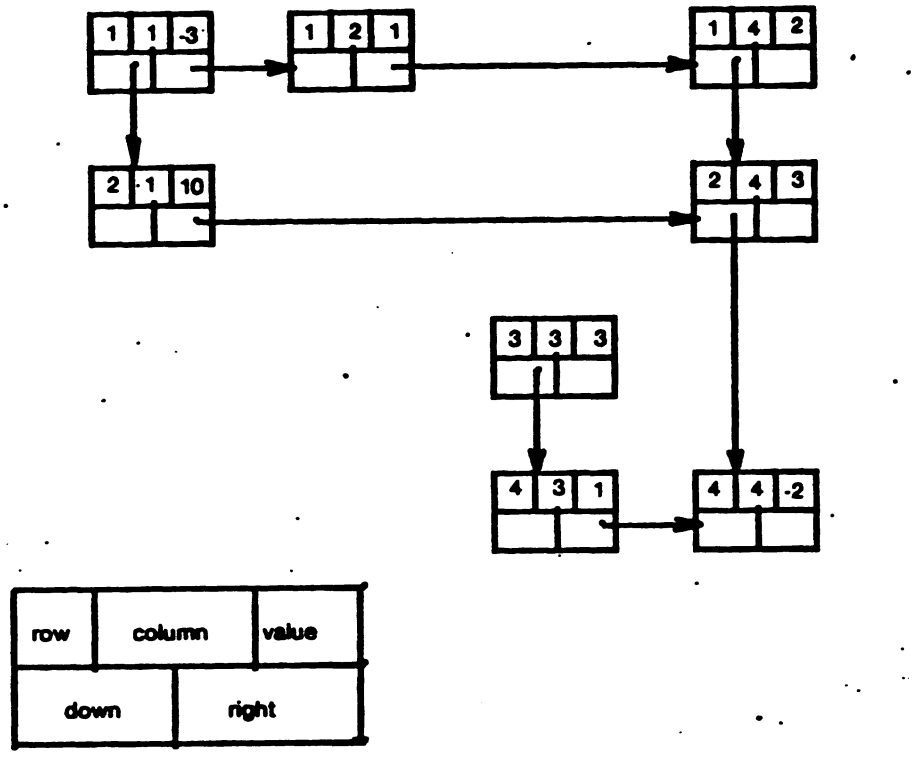


Figure 2-3: Example of Dynamic Storage.

2.3. Variability Typing

The introduction of variability typing has been proven to significantly improve efficiency in the solution of sparse systems. This concept is based on the fact that some of the entries of *L* and *U* computed by (2.5) and (2.6) depend on the values of circuit elements,some depend on time and the others are functions of system variables. Since some entries need to be computed more often than others, we classify elements into groups. Each group of entries may be computed separately. As a result the operations needed by *LU* decomposition may be performed in nested loops. The outer loop will include the computation of those elements defined to be of *C*-type, i.e., these which depend only on circuit elements. The first inner loop will calculate the matrix entries of *P*-type, with values changing with design parameters. The next inner loop will contain elements of *T*-type altered when time increases. The innermost loop will include entries of *X*-type, computed at each Newton step,which depend on unknowns. In Figure 2-4 we show the flow of computation of the jacobian entries. *T*-type

entries arise from the discretization of partial differential equations while *X*-type elements come into existence from the linearization of nonlinear equations. Finally, in order to efficiently handle the large number of topological elements which arise from the tableau formulation we define the plusone and minusone types. The introduction of topological elements increases the process time of a subnetwork, but the saving in simulation time is considerable since only operations involving nontopological elements are performed.

```
BEGIN   { Interactive Loop }
  Compute C-type entries ;
  WHILE design parameters change DO
    BEGIN   { Optimization Loop }
      Compute P-type entries;
      WHILE t < T DO    { t is time and T is interval of simulation }
        BEGIN   { Time Loop }
          Compute T-type entries;
          WHILE solution of equations does not converge DO
            BEGIN   { Newton Loop }
              Compute X-type entries
            END   { Newton Loop }
        END   { Time Loop }
    END    { Optimization Loop }
END;   { Interactive Loop }
```

Figure 2-4: Flow of Computation of Jacobian Entries

We are interested, when generating code, in reducing the number of operations performed in the Newton loop. So the pivoting strategy we shall choose , (see next section) must discourage a pivot selection resulting in many operations needed to be performed in inner loops. Therefore, it is necessary to associate with each operation a variability type to indentify where this operation must be done, and then to allocate a weight to each type. This weight will constitute a measure of cost for performing an operation of a specific type. We can also use the allocated weights to order variability types in the sense that a higher order type possesses a greater weight. Each operation then is characterized by the highest involved variability type.The various types are listed in Table 2-2 in increasing order.

```
Type Plusone : topological entries of value +1.
Type Minusone : topological entries of value -1.
Type C : constant entries.
Type P : entries changing with design parameters.
Type T : entries changing with time.
Type X : entries depending on the unknowns.
```

Table 2-2: Variability Types

In the input phase, we use two different structures to store the initial matrix describing a

subnetwork, and its factors L and $U_{\%}$ because decomposition may change the variability type of an entry. In this case the loading of the entry should be done in an iteration level different from the loop where operations for the decomposition, referred to that entry, are performed. Furthermore, we do not store topdogical entries and we keep only one value for all the T-type entries.

Since we use variability typing, we need three more arrays to completely represent the structure of a sparse structure employing a static scheme. We will need: 1)An array.denoted VType, parallel to the Value array holds the variability types of the stored elements. 2) Another array, denoted EntryLocation, parallel to the index array points to the Value array. 3)An array, called Nominal array/is also employed. It is particularly useful in ordering because it holds the nominal values the user assign to entries(X type) depending on the unknowns so that we can compute the roundoff error factor from (2.11) when pivoting.

2.4. Pivot Selection

The principle objective of pivoting strategies is to find an ordering of the rows and columns that keeps the number of arithmetic operations required for solution at a minimum. Another objective is minimizing the number of nonzero elements added to the matrix structure during solution. A third objective in pivoting is to ensure numerical stability.

It would be desirable to find the best ordering (among $(n!)^2$ possible orderings for a full nxn matrix)in some well defined sense. However, in practice, this is impossible. Pivoting strategies can be classified into a priori and local methods. The first category includes methods which order columns (or rows) before the elimination process takes place. In local strategies, at each step in the factorization procedure, the pivot is selected from among all the nonzeros in the reduced submatrix,which has been updated during previous elimination stages.

A priori strategies yield results inferior to local methods. A common a priori pivoting technique is to order the columns in increasing number of nonzeros in each column, or to order them in increasing total number of nonzero elements in rows having a nonzero entry in the given column. We choose the pivot within each column, to be the nonzero entry of the pivotal column that is located on the row with minimal row count (number of nonzeros in the row). Although a priori methods are not the best ones, their adoption in pivoting to preserve sparsity results in great improvement over not ordering at all.

Among local pivoting techniques the most popular is that suggested by Markowitz [Markowitz 57]. At each stage of elimination, this ordering selects as pivot the nonzero element that minimizes the product of the number of other nonzeros in the candidates row and column. Markowitz strategy has proven to be very effective. In general, there is a trade-off between the time spent in choosing a pivoting strategy and the resulting fill-in, local orderings are computationally more expensive than a priori methods because they require simulation of the elimination process for all candidate pivots.

Markowitz strategy can be significantly improved if we incorporate in it variability typing of nonzero entries [Gustavson 70]. We expect that variability typing, as well as the* distinction' between topological and unique elements, will require extra processing time. If we recall that we can use subnetworks from a library,which have already been processed.the added overhead is rather unimportant. Next, we describe the ordering we follow in our implementation based upon the Markowitz selection criterion, in conjuction with variability typing. The context of modular simulation imposes a few modifications on the basic algorithm.

Since the task of ordering is to minimize fill-in and operation count while retaining accuracy, we allocate to each candidate pivot a weigthed pivot cost, denoted *PC.* The cost for selecting the candidate pivot in i^{m1} row and j^m column is defined as

$$PC[i,j] = fiMfU] \cdot *\,[U] + OH \qquad (2.9)$$

where ji_v $w > 0$ are input parameters, *M* is a weighted operation count, *R* is a roundoff error factor and *OH* is an overhead cost. We also define

$$M[i,j] = \Sigma_{k=1}^{m_{ij}} vw_k \qquad (2.10)$$

where vw_k stands for the weight associated with the variability type assigned to the k^m of the m_{ij} multiplications(divisions) in eliminating the i^m row and j^m column. Moreover

$$[U](2_l|A[i,l]|)/(nz_i|4[i,j]|), \quad l\{ci/A[i,ciH0\} \qquad (2.11)$$

with I running over the nz_l nonzero column indices in row i and *A[* i, i] standing for the numerical value of the entry in 1^{th} row and j^m column.

The need for the overhead cost, denoted by *OH* in (2.9), originated from the application of modular simulation in circuit analysis, which requires the solution of the connection equations describing the interaction between different subnetwork, in the Newton loop. Therefore, we are interested In reducing the the number of the operations involved in the solution of the connection equations. Among the variables describing a subnetwork there are some representing a voltage or a current referred to a terminal node. These variables can be either

an input or an output. A specific subnetwork accepts the action of other subnetwork through its input variables while it acts upon the other parts of the network through its output variables. The purpose of introducing the overhead cost is to impose on the ordering strategy to pivot at the end on columns corresponding to output variables and on rows which involve an input variable. So we assign to OH a very large value for nonzero elements in such a row or column, whereas it takes on a zero value for all the other entries. The large value of *OH* in (2.9) increases significantly the total pivot cost for the specific entries discouraging their early selection as pivots. It will become clear in the next chapter, where we give a detailed description of a Subnetwork, that this arrangement will considerably reduce the operation count in computing the coefficients of connection equations.

Next we describe the k* of N identical stages needed in the process of pivoting on an N *x* N matrix. At the end of the (k - 1)* stage, k - 1 eliminations have been made and the reduced matrix A^w (Figure 2-2) has been updated at the previous stage of elimination. So in the k^m step the structure of A^w is known, comprising nonzero locations, variability types and values of the matrix entries. We compute from (2.11) the roundoff-error factor for all the nonzero elements of A^w and consider as candidate pivots only those entries for which

$$TOL \quad fl[i,j] < 1 \qquad (2.12)$$

where $0 < TOL < 1$ is a threshold factor. The remaining elements are rejected because their small magnitude may cause numerical stability problems. Then we simulate the execution of the operations involved in the elimination process for each candidate pivot. We consider as nontrivial only the operations (multiplications and divisions) involving nonzero elements of a type other than plusone or minusone. The weighted operation count *M* increases for each nontrivial operation by the weight corresponding to the type of the specific operation. After computing the weighted operation count for each candidate pivot from (2.9) we choose as pivot the entry in I^{th} row and 7^{th} column for which

$$PC[I, J] = \min_{[i,j]} PC[i, j]$$

where [i, j] refers to the location of a candidate pivot satisfying (2.12). The next step is the execution of the numerical Gaussian elimination of row / and column J, creating the reduced (k-1)x(k-1) matrix $A^{fr+1)}$. At this point the k* stage of elimination has been completed.

Chapter 3
SUBNETWORKS IN CIRCUIT ANALYSIS

The purpose of this chapter is to describe, in detail, the use of subnetworks in circuit analysis. As mentioned in Chapter 1, a network is considered to consist of a number of interconnected subnetworks. By carefully studing the interconnection some interesting results can be obtained regarding the solution of connection equations.

3.1. Formulation of Network Equations

We consider first the formulation of the equations which describe a network consisting of a subnetworks. From Chapter 1, subnetwork is completely described by the set of equations of the form (1.1) -(1.3). Let $w_{\%}$ » col($x_{i9}q_{\cdot}$) the /^-dimensional vector of the internal variables.After the pivoting process has been done (1.1)-(1.3) can be written as follows

$$D_j M_f + t_f * 0 \tag{3.1}$$

where $A_{\cdot}$ is a nonsingular $p_{\cdot} x p_{\cdot}$ matrix,$B_{\cdot}$ is a $ft_{\cdot} x II_{\cdot}$ topological matrix, $D_{\cdot}$ is a $II_{\cdot} x p_{\cdot}$ matrix and $a_{\cdot}$ is a $^{\cdot}$-dimensional vector.

Separation of $w_{\%}$ and y_{I} in (3.1) has been made in order to pivot on the columns corresponding to output' variables at the end. We can combine $w_{\cdot}$ and $y_{\cdot}$ into the $a_{\cdot}$-dimensional vector $z_{\cdot}$ = col(w_v y_{I}), where $n_{\cdot} * p_{\cdot} + n_{\cdot}$and then (3.1) may be expressed as

$$J_{\cdot} Z_j + T_j U_{\cdot} - r_j \tag{3.2}$$

where $J_{\cdot}$ is a nonsigular $n_{\cdot} x n_{\cdot}$ Jacobian matrix, $r_{\cdot}$ is an $n_{\cdot}$-dimensionaJ vector whose lower $II_{\cdot}$ components are zero and $T_{\cdot}$ is an $n_{\cdot} x n_{\cdot}$ matrix which is zero in the top $p_{\cdot}$ rows while the lower $II_{\cdot}$ rows include topological elements of value zero or one. A subnetwork is shown in Figure 3-1 while Figure 3-2 illustrates the structure of (3.1).

While individual subnetworks can be completely described by equation of the form (3.2), we

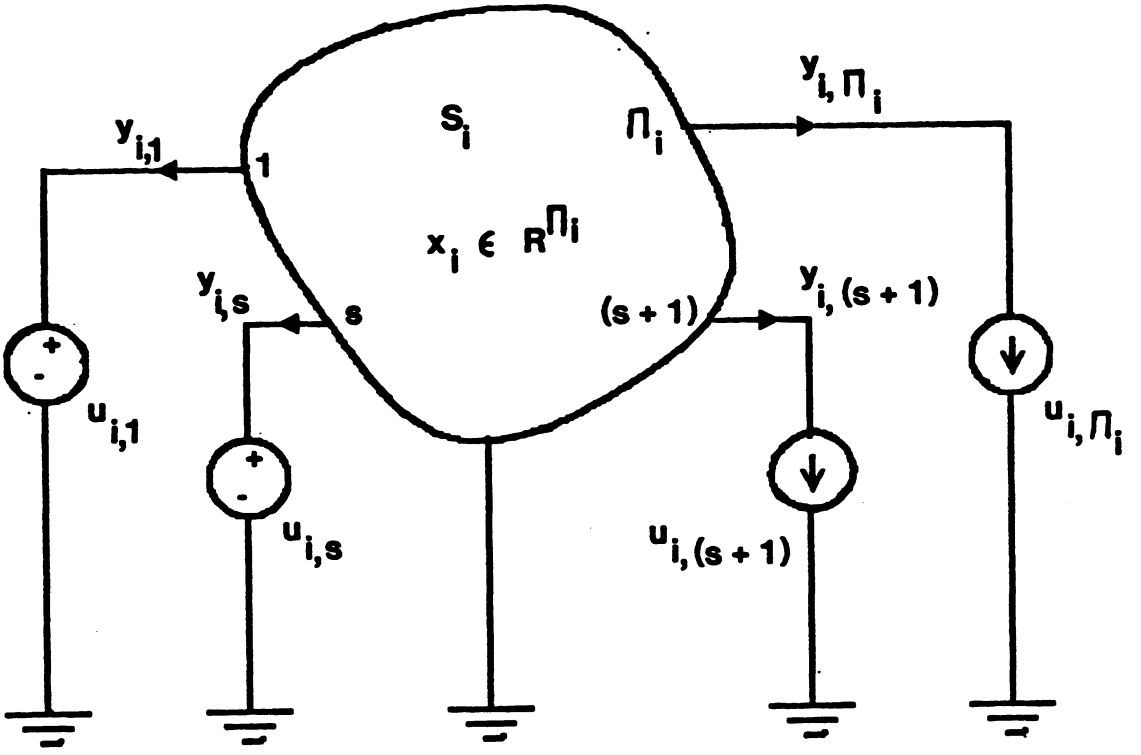


Figure 3-1: Subnetwork

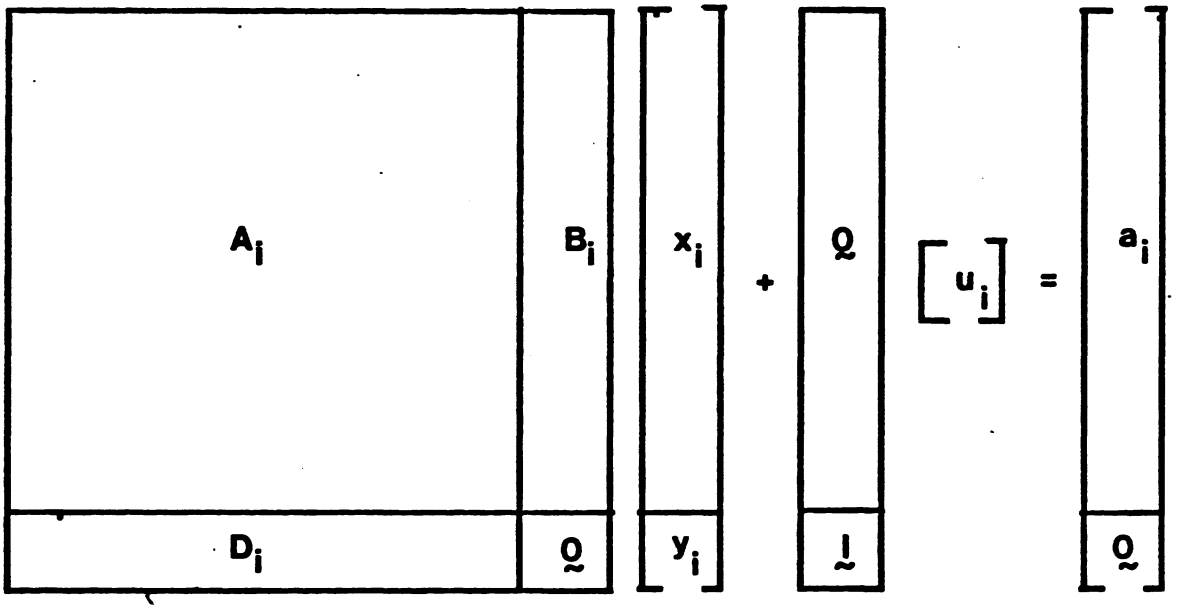


Figure 3-2: Structure of Subnetwork Equations

need an additional set of equations to completely characterize the whole network. These

equations define the interconnection pattern of the various subnetworks comprising the network. Subnetworks are connected at their terminals, so connection equations involve only input and output variables. Each input variable is expressed in terms of the output variables of other subnetworks [Sakallah 80]. This implies that exactly one current-activated terminal and one or more voltage-activated terminals can be involved in a connection equation. A terminal is current-activated or voltage-activated when the corresponding input is, respectively, a current source or voltage source. An interconnection of three subnetworks is depicted in Figure 3-3. The input vector of the S_f subnetwork can be expressed as

$$u, \bullet Z^{\wedge}{}_{-1} A_{||} z_{|} - n, \qquad (3.3)$$

where A is a $II.xn_1$ augmented topdogical fan-in matrix defining the interconnection structure between $S_§$ and S_{Jf} and n, is a n_f-dimensional vector of primary (external) excitation/Therefore, the network is completely described by the following set of equations -

$$JjZj + T, u_§ « r,, \quad I \bullet 1,2,.... a \qquad (3.4)$$

$$2^{*}. \quad < A.. Z, \bullet U; « /I, , \quad I » 1, £^ ..., O \qquad \text{W.O)}.$$

Equations (3.4) and (3.5) written in matrix form for the network in Figure 3-3 are depicted in Figure 3-4.

3.2. Solution of Network Equations

LU factorization is employed to solve the system of linear equations (3.4) and (3.5). The structure of the coefficient matrix after forward elimination is shown in Figure 3-5. It can be easily seen that it is possible to carry out the forward elimination as well as the backward substitution step for each block on the diagonal separately in any order). Each diagonal block(except that at the lower left corner which is referred to as the subnetwork interconnection block) constitutes the Jacobian matrix of a subnetwork processed during the input phase. Equation (3.4) may be written as

$$z_1 \bullet JT_i^{1} (r._1 - r._1 fij) \quad I \bullet 1,..., a \qquad (3.6)$$

where J_i can be factored as

$$J_i = L_i U_i$$

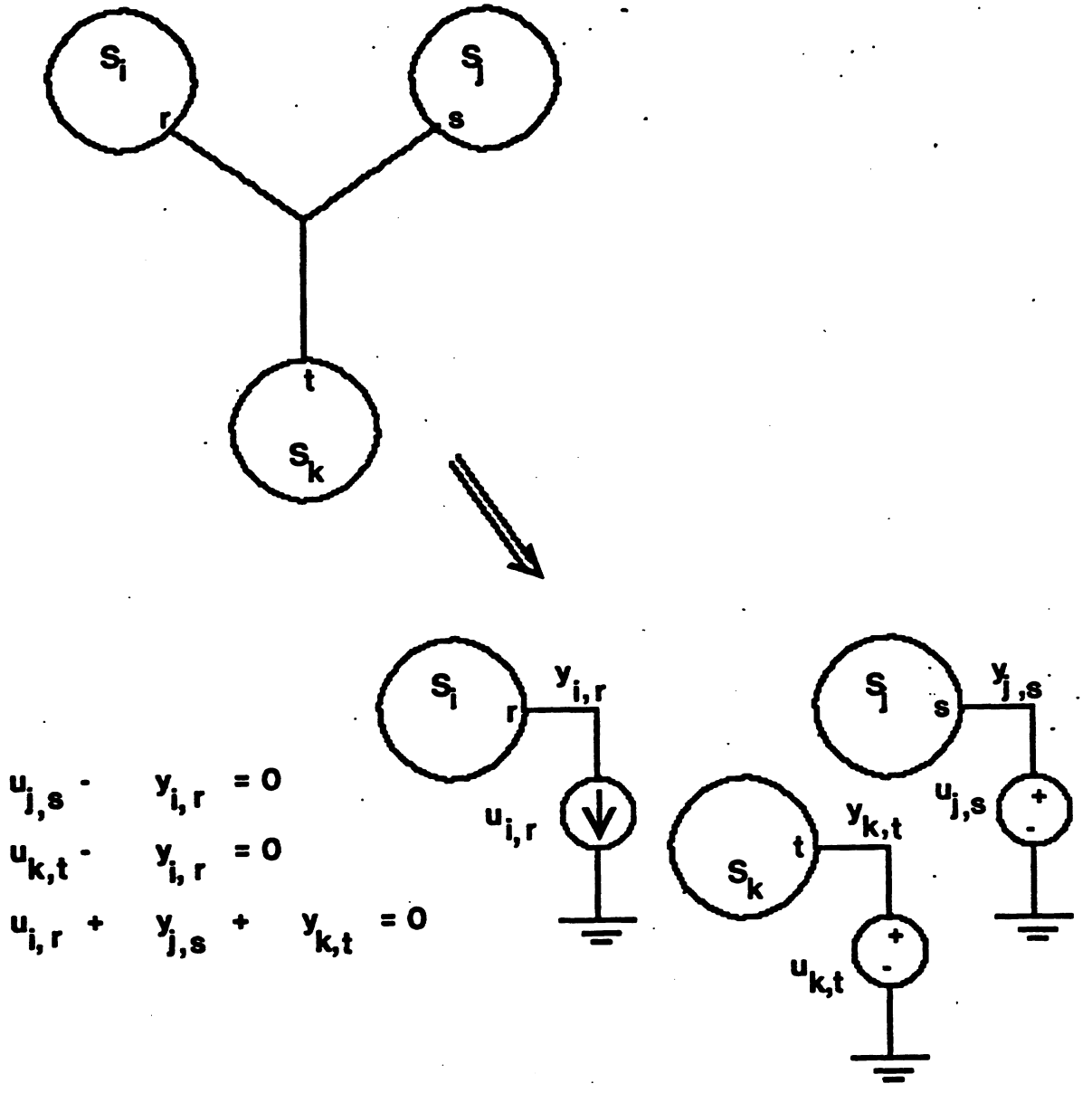
Assuming the input vectors u_1, i * 1 , . . . , *a* are available, (3.6) provides us with the values of internal and output variables for all subnetworks. When (3.6) is substituted in (3.5) we get

$$2f_s \; j \; A.._{1,j} \bullet u.\% \; m \; n_§ \bullet 2?_{m\;1}\; 4_§ j \quad i « 1, ... a \qquad (3.7)$$

where

$$A_{ij} - -A_{ij}.^{\wedge} r._j \; t \; FpixfM \qquad (3.8)$$

and



$$u_{j,s} - y_{i,r} = 0$$

$$u_{k,t} - y_{i,r} = 0$$

$$u_{i,r} + y_{j,s} + y_{k,t} = 0$$

Figure 3-3: Connection Equations

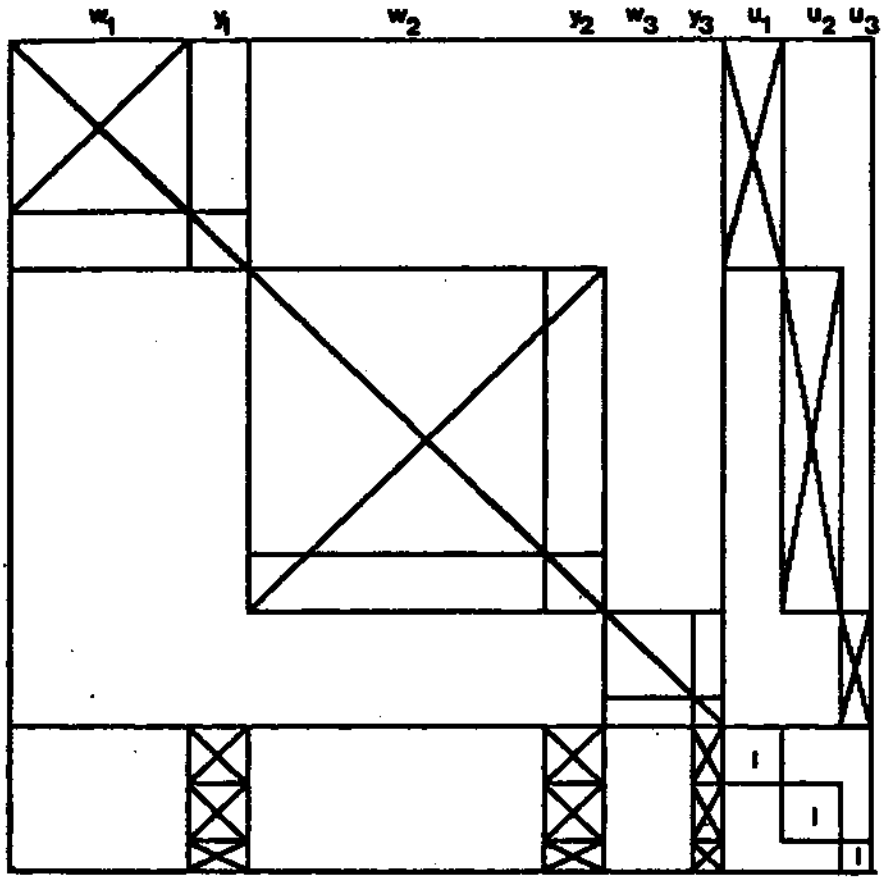


Figure 3-4: Structure of Network Equations

$$\delta_{I\,i} = A_{i\,j}\, J_{i\,j}^{-1}\, r_{1} \quad i \quad R \prod_{i} \tag{3.9}$$

The system of linear equations (3.7) involves only the input vectors u_y Therefore we can solve (3.7) to obtain the values of u_1. Matrix A_{ij} in (3.8) and vector 6_{ij} in (3.9) represent the action of subnetwork S_i on subnetwork S_j.

In order to describe analytically the computational steps required to solve (3.4) and (3.5) we introduce the connection matrix C *t* ft^2! * 1 ^{n}i *x* fl^{2}i » 1 ^{n}i and the vectors *u* and c e ***Fri*** « 1 " i $_f$ where *u* * col(u_v $u_{2\%}$.. >, u_Q) is the composite input of the network. The index in *u* of the first component of u_1 is denoted by $£_i$. We define the fan-out list of $y_{X\ r>}$ the r^ output variable of S_i, as

$$FO_{i,\,r} = \{1 < j < Z^{\wedge}{}_{s\,1}\, 1\, I_k \mid j \text{ is the index in the composite input vector } u \text{ of an output variable affected by } y_{i,\,f}\}$$

We also introduce the n_i-dimensional vector $X_{i,\,r}$, which is zero except for the $(p_i + r)^m$ component which is +1 if $y^{\wedge}{}_r$ is a current, and -1 if it is a voltage. Note that $FO_{i,\,r}$ and $X_{i,\,r}$

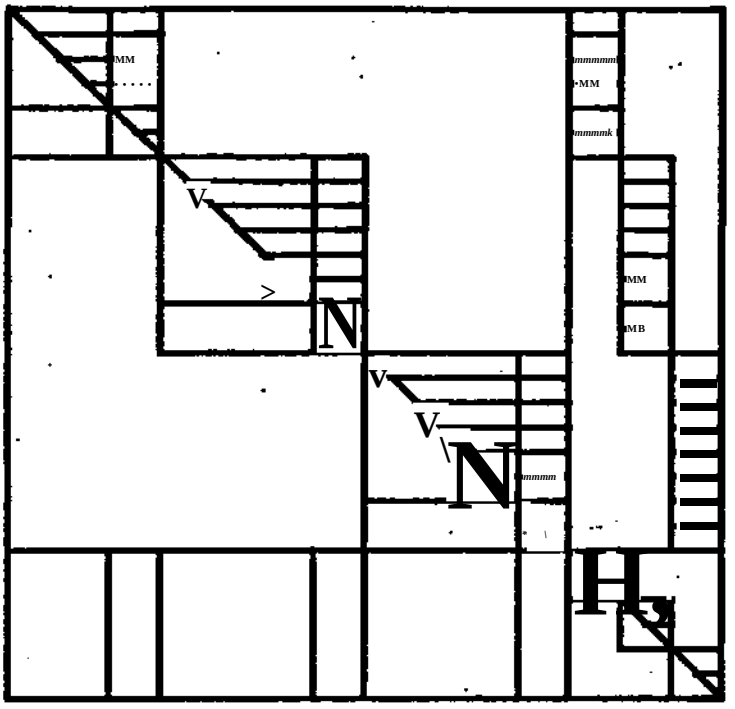


Figure 3-5: Matrix of Figure 3-4 after Forward elimination

contain the same connection information in a different representation for all the output variables in the network.

The following steps are to solve the network equations

Step 0: [Initialize]

C * l,c « col(n$_v$n$_2$, . . . , ^ 1

Step 1: [Factorize] i * 1,2,. . . , a

$$L_i U_i = J_i$$

Step *t.* [Forward Elimination] i • 1,2,. . . , *a*

$$L.J. = \mathbf{r}_i$$

Step 3: [Propagate Outputs] i « 1,.... a, r « 1$_t$ ...$_f$ n , j e FO$_{i,r}$

$$L_i \tilde{\Gamma}_i = \Gamma_i$$

$$X_{i,r}^T U_i = \lambda_{i,r}^T$$

$$C[\xi_i .. \xi_i + \Pi_i - 1] = X_{i,r}^T \tilde{\Gamma}_i$$

cQ] » cO]$_-$ + *zfa* • rj(+ if y_u is a voltage,-if it is a current)

Step 4: [Solve Connection Equations]

Cu » c

Step 5: [Back Substitute] i » 1,2,.... a

$$L_i \tilde{r}_i = r_i - \Gamma_i u_i$$

$$u_i z_i * T \backslash$$

The algorithm above is a direct translation of (3.6) • (3.9). Implementation details are given in next section.

3.3. Implementation Details and Examples

When the simulation part of SAMSON employs the code, which is generated during the input phase, to solve the simulated network any information about the sparsity of the matrices describing the subnetworks is redundant. Therefore for the representation of each subnetwork matrix in SAM2 we need only a vector holding the values of the unique entries. Figure 3-6 depicts the jacobian matrices of three subnetworks.

We can save a considerable amount of memory space by storing only the values of the unique entries (see Table 3-1). But more impressive is the reduced operation count as a result of the execution of operations involving only nontopological elements. Recall that the effect *til* operations involving topological elements has been taken into consideration during the code generation. Table 3-2 is indicative of the number of operations which do are not carried out. Factorization of the jacobian constitutes the first step of our algorithm for the solution of network equations.

Subnetwork 1

```
  12345678901234
 1-   +   +     1
 2-+  f   f+    2
 3  ++       +  3
 4   -   •    +4
 6    -        +5
 6     +  C     6
 7      +  C    7
 8   +  Tf t  f 8
 9 -      ff+   9
10  +f T ftttt  10
11       XXxXxx 11
12  -f       ff 12
13 -  f   f fxxxl3
14-   f   fftxxxl4
  12345678901234
```

Orderina Parameters

M '' 0.01
9 «< 0.01
TOL « 0.0
VW_{CTYPE} • 5.0
VW_{PType} * 6.0
Z^{**}_{XType} « S'.0

Notation

• :: topological entry +1
- :: topological entry -1
f :: fill-in +1 or -1
C :: C type entry
c :: C type fill-in
P :: *P* type entry
P '.: *P* type fill-in
T :: *T* type entry
t !:. *T* type fill-in
X :: X type entry
x :: X type fill-in

Subnetwork 2

```
  123456789012345678901 23
 1+                     1
 2+-                    2
 3 •-        +          3
 4   -           +      4
 5    •     -          +6
 6     +      +         +6
 7      +    C          7
 8       +       C      8
 9        +           C  9
10         +    C        10
11      T   +t           11
12 -+        f    +      12
13         T   +       t  13
14          T   +t         14
15  -         f  + f       15
16  +-        f   ff    + 1 6
17  -         f    f+      17
18  -         f    ff    . +18
19  ..             XxX    xl9
20               XxX Xx x20
21     + T    f  ttt  ttt21
22   -           fff  ftt22
23           +t-+t  tt-+xxx23
  12345678901234567890123
```

Figure 3-6: Examples of Jacobian Structures

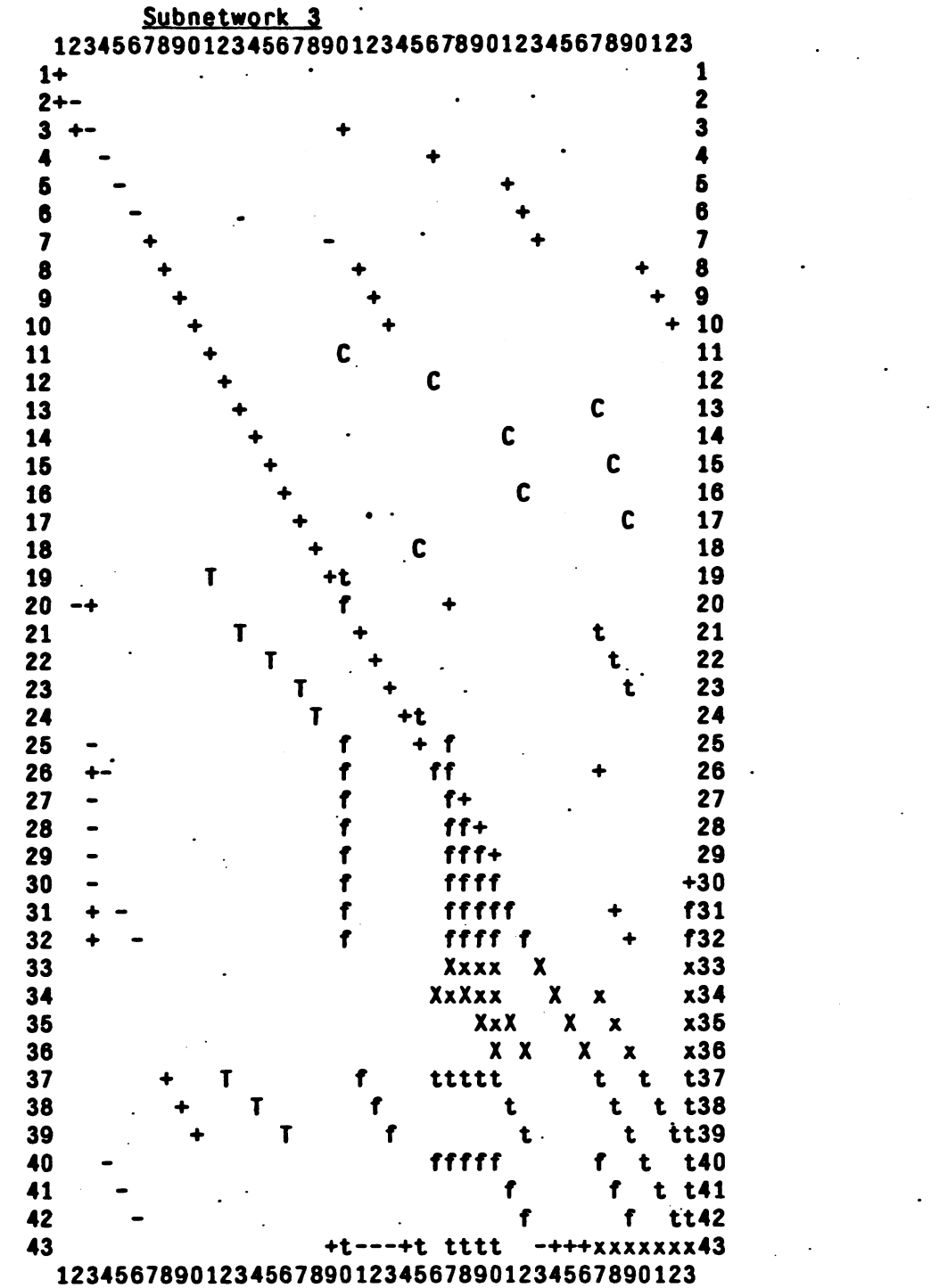


Figure 3-6, concluded

	Subnetwork1	Subnetwork2	Subnetwork3
Dimension of Jacobian	14	23	43
Nonzero entries before decomposition	30	40	83
Unique entries before decomposition	6	10	19
Nonzero entries after decomposition	62	92	201
Unique entries after decomposition	22	36	80

Subnetworks depicted in Figure 3-6

Table 3-1: Examples of Jacobian Sparsity

	Subnetwork1	Subnetwork2	Subnetwork3
Operations in nonzero entries	39	46	100
Operations in unique entries	13	13	33

Subnetworks depicted in Figure 3-6

Table 3-2: Examples of Operation Count in Jacobian Decomposition

3.3.1. Forward Elimination

In the second step, the forward elimination, we also take advantage of the execution of nontrivial operations (as they have been already defined). Table 3-3 shows to the operation count of the forward elimination for the three subnetworks in Figure 3-6. We may reduce the number of operations even more if we also treat the right-hand side r_i as a sparse vector. In a

tableau formulation the sparsity of the right-hand side is particulariy significant making its exploitation worthwhile. Moreover, we need less space to store only the nonzero elements of r,. Notice that we can employ the vector of unknowns z, to store temporarily r_y

	Subnetwork	Subnetwo rk2	Subnetwork3
Operations* nonzero entries	4Q	57	-\QQ
Operations in unique entries	- j ^	23	52

Subnetworks depicted in Figure 3-6

Table 3-3: Examples of Operation Count in Forward Elimination

3.3.2. Propagation **of** the **Outputs**

The next step, specified in the algorithm description as "propagation of the output variables", consists of the elimination of output variables from the connection equations. Observe that the *LU* factorization of the diagonal blocks causes fill-in in the connection matrix C. To compute this **fill-in** in matrix C we need a partial forward substitution (L^{-1} « I\) followed by another partial forward elimination (X^{T}_{f} I/, • $X7_{f}$). At this point it becomes clear why we pivoted at the end on columns corresponding to output variables and on rows containing an input variable. The applied ordering reduces the number of nonzero components of F, and X; _ to a minimum. As a result, less operations are to be performed in partial eliminations and the number of multiplications involved in loading the connection matrix decreases.

T^{-1}, and XT_r are not implemented as a matrix and a vector, respectively. We represent them as a set of variables, created in code generation. Each variable corresponds to a nonzero entry of T^{-1}, and $X/_f$. So we reserve space only for the nonzero elements. Furthermore, we employ the same variables to hold the values of all vectors XT_f, r * 1, ... II,. The variables representing I^{-1}\ and XT_f are sparse when the Jacobian of the subnetwork is of large dimension (greater than 50).

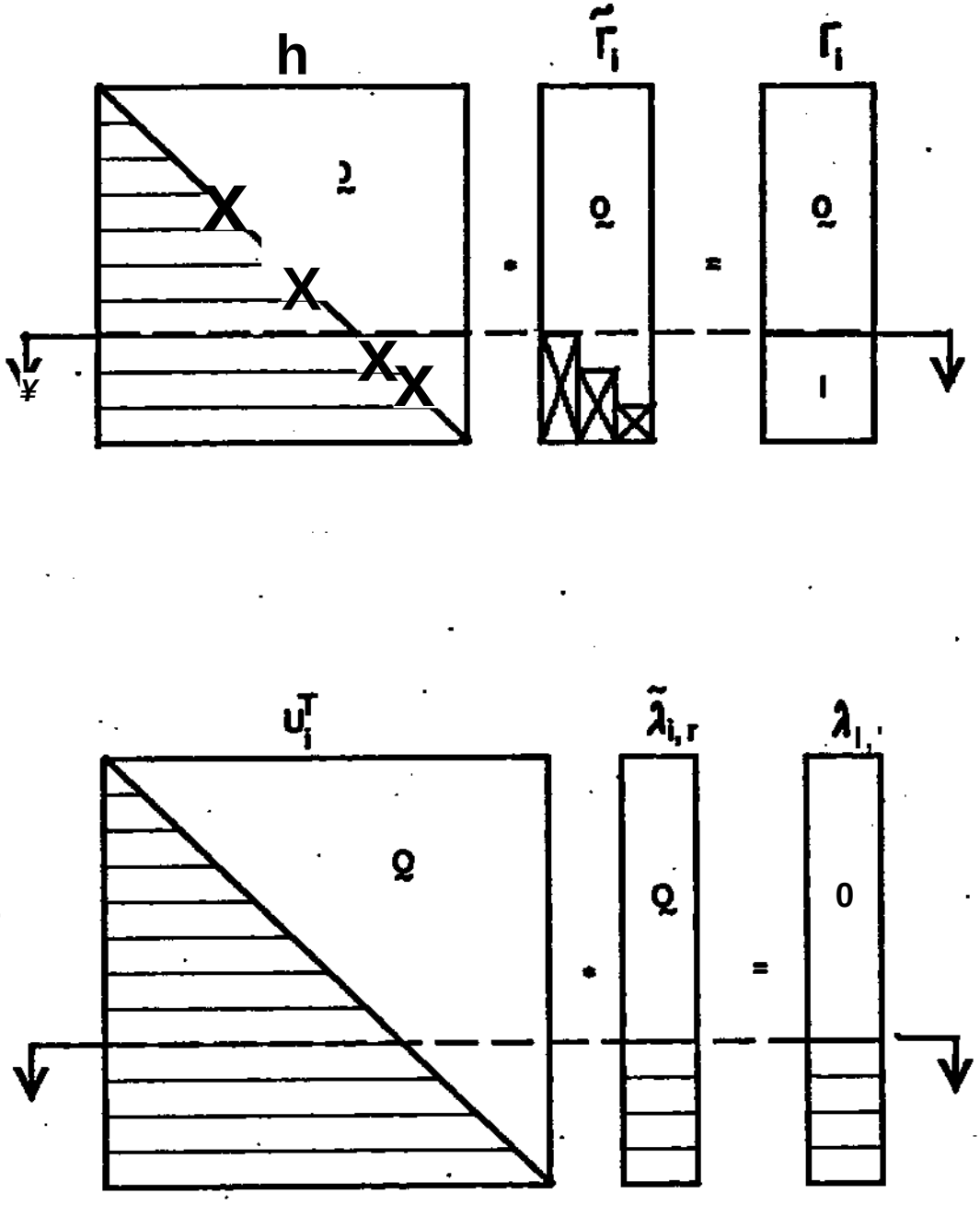


Figure 3-7: Partial Forward Substitution

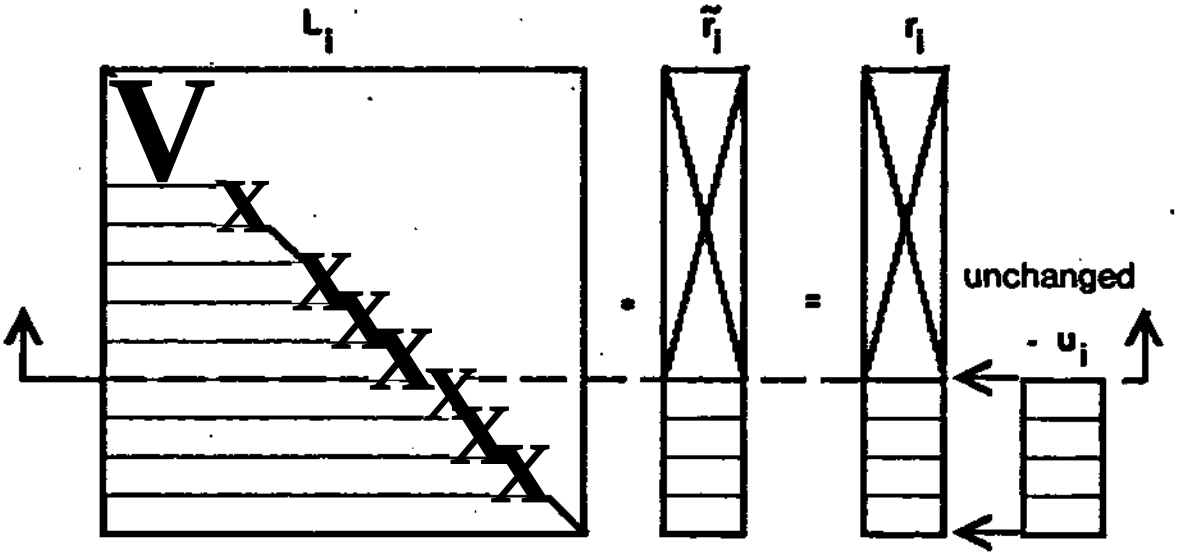


Figure 3-8: Correcting Forward Elimination

3.3.3. Solution of Connection Equations

The fourth step in the algorithm for solving network equations is the solution of connection equations (C *u* » c). SAMSON employs the concept of activity-directed circuit simulation, making possible an efficient solution of connection equations. This is because when we solve for the connection equations, we do not have to eliminate the output variables of dormant subnetworks, since they can be extrapolated. Therefore the structure of the connection matrix C is dynamically altered depending on the status, at that time instant, of the various interconnected subnetworks. Consider the network shown in Rgure 3-9, consisting of three subnetworks S_y S_2 and S_3. The connection equations are shown in Rgure 3-10. Observe that the connection matrix C, i.e., the coefficient matrix of the vector of output variables *u* » (u_v u_T a_3) is a unit matrix before elimination of the output variables. Assume that subnetwork S« is alert, and consider the elimination of the output variable y_o. It causes fill-in in the rows involving this variable, and the matrix C takes the form shown in Rgure 3-11. The fill-in is exactly the same in each row, and equal to X_1 $_2\bar{T}_y$ it is also independent of the

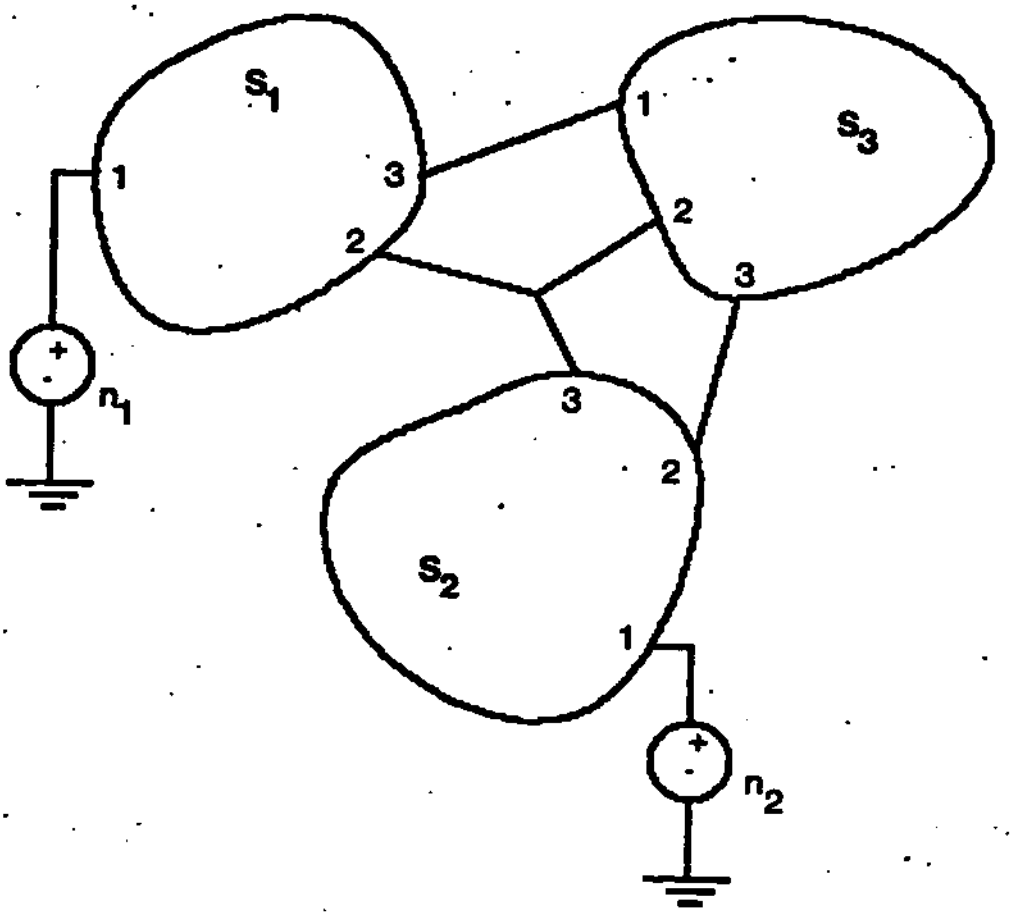
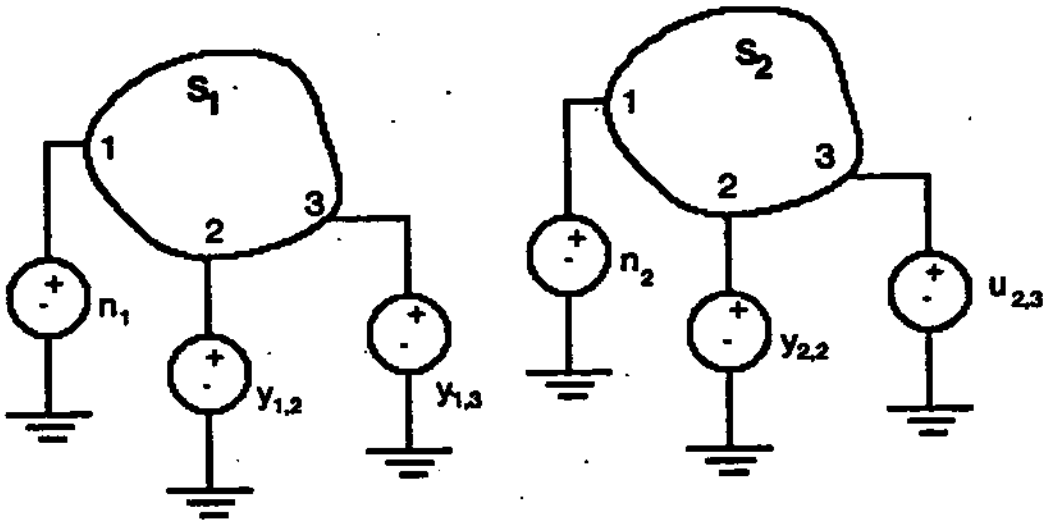


Figure 3-9: Example of a Network

interconnection of the subnetworks. Therefore, it is convenient to associate with each output variable an interaction vector holding the values of fill-in resulting from the elimination of this variable. Interaction vectors are then employed to load the connection matrix. Notice that we generate code, separately for each subnetwork, to compute entries of the connection matrix caused by the elimination of the output variables, independently of the interconnection pattern of the network.As a result, we can use a processed subnetwork from a library to solve any network whose description includes this subnetwork , without regard for the interconnection of the subnetworks. Except for the connection matrix, the right-hand side vector c also needs to be loaded.The required values are stored as the last components in interaction vectors. Figure 3-12 illustrates the dynamic change of the connection matrix of our example.

While we shall solve the connection equations using Crout reduction, the special form of the connection matrix requires a new pivoting technique, since traditional orderings cannot be applied. Since the pivoting sequence is determined only once, at the beginning of



$$
\begin{aligned}
&-n_1 + U_{vi} * 0 \\
&y_{2,3} + y_{3,2} + u_{1,2} = 0 \\
&Y_{3.1} + u_{1,3} = 0 \\
&-n_2 + U_{2.1} = 0 \\
&y_{3,3} + u_{2,2} = 0 \\
&-y_{1,2} + u_{2,3} = 0 \\
&-y_{1,3} + u_{3.1} = 0 \\
&-y_{1,2} + u_{3,2} = 0 \\
&-y_{2,2} + u_{3,3} = 0
\end{aligned}
$$

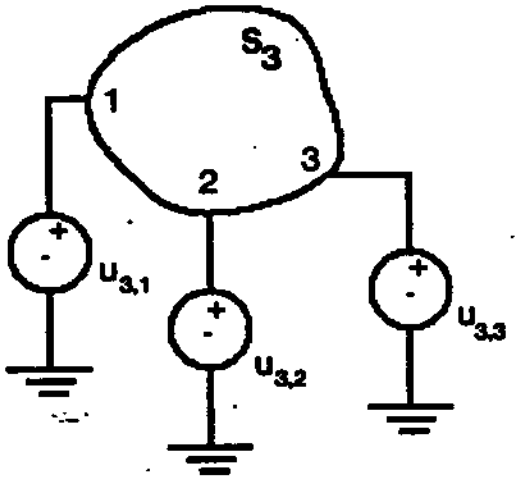


Figure 3-10: Connection Equations of the Network depicted in Figure 3-9

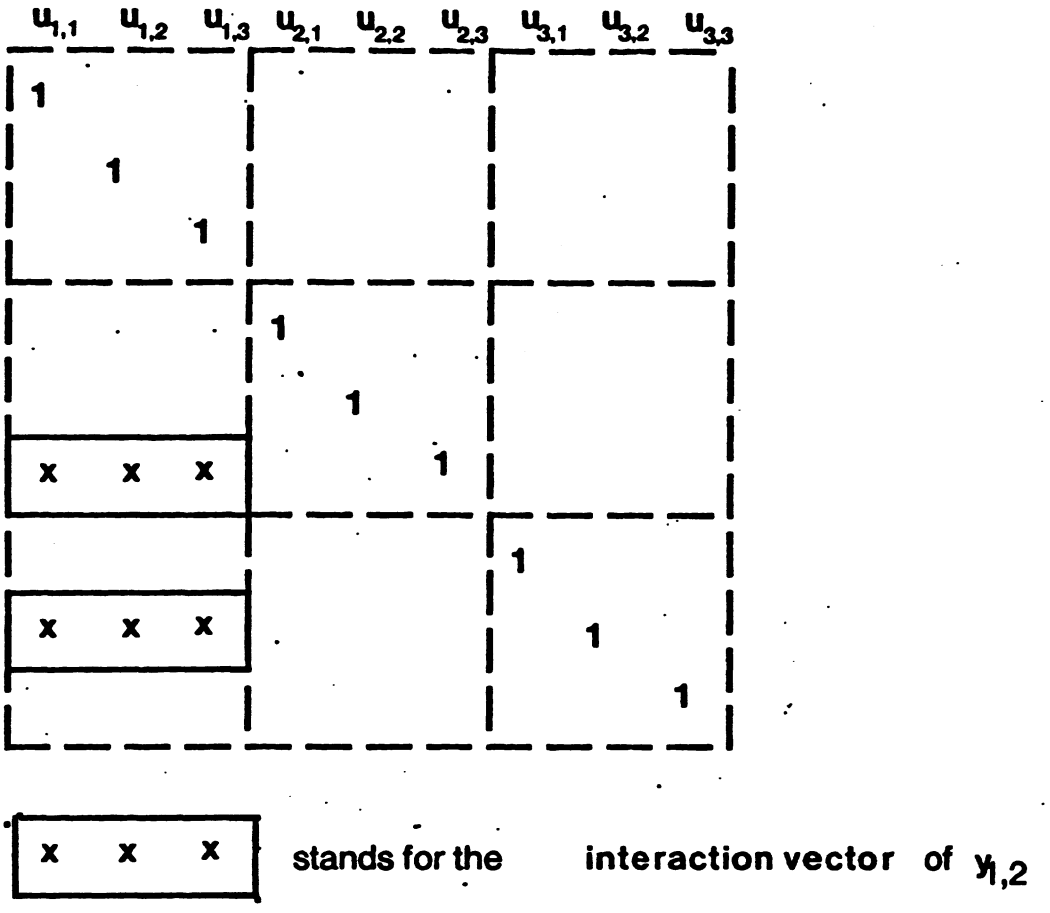


Figure 3-11: Example of Elimination of Output Variable

simulation, we are restricted to pivoting only on diagonal elements of the unit submatrices , which are independent of the status of subnetworks. All other entries may become zero if the subnetwork is dormant. Block pivoting is another feature of the applied pivoting., we pivot consecutively on the elements of each unit submatrix on the diagonal of the connection matrix *C*. Block ordering enables us to employ the dormancy of subnetworks in the solution of the connection equations. Row and column order is not changed inside each unit block. Otherwise we need to update for any change the data structure of the network, which is time consuming. Furthermore, we have to reorder the components of the interaction vectors since the code for computing these vectors has been generated under the assumption of a certain order for the input variables of each subnetwork. So when we order the connection matrix, in fact, we decide for the order of the diagonal unit blocks without altering internally the order of rows and columns.

The selection strategy we follow is Markowitz criterion adjusted to our requirements. We compute the operation count for eliminating each one nonzero entry of all diagonal unit

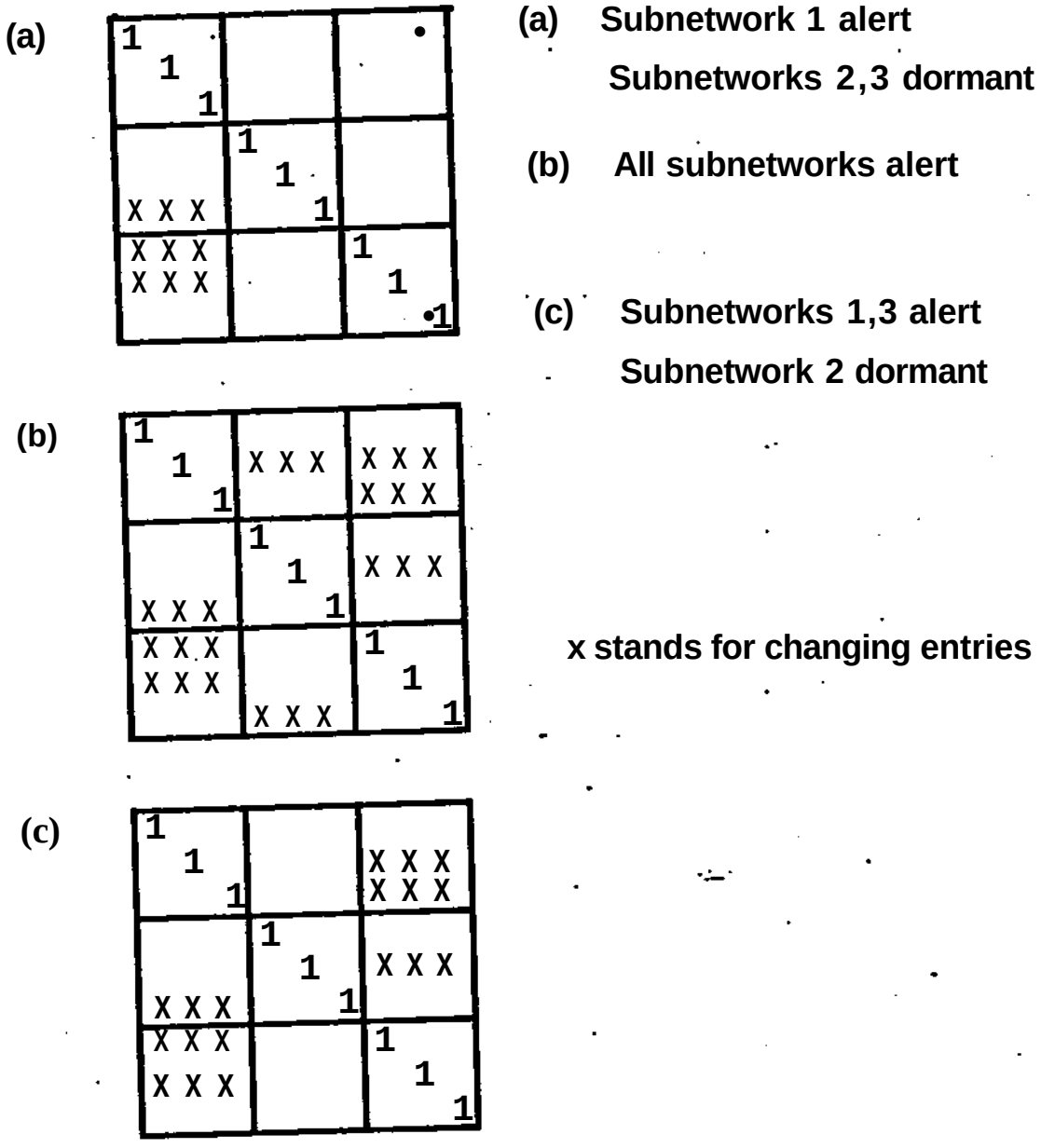


Figure 3-12: Example of Dynamic Change of Connection Matrix

blocks assuming all subnetworks to be active. Let us denote $oc_{i,J}$ the operation count for pivoting on the nonzero element in i^{th} row of J^{th} block. We define the block operation count for J^{th} block as

$$BOC_J = \Sigma_{i=1}^{\Pi_i} oc_{i,J}$$

We select as a piovotal block the PJ^{th} one whith minimal block operation count

$$BOC_{PJ} = \min_{1 \le J \le \sigma} BOC_J$$

Then we update the reduced matrix taking under consideration produced by the elimination of the pivotal block. This procedure is repeated σ times to order all blocks.

Among the existing approaches to the solution of a sparse system of linear equations we select to follow the looping indexed approach to solve the connection equations. This approach requires much less storage than the compiled code and the interpretable code approaches. Its choice is also favored by the fact that the solution of connection equations does not demand use of variability typing. The other two formulations have a faster execution speed but they require more work in performing the decomposition. This overhead in processing the sparse matrix can be balanced by their fast execution only for large matrices. The connection matrix is frequently a matrix of small dimension.

After ordering we use the Crout algorithm to compute the entries of factors L_C and U_C of the connection matrix. Unfortunately, for reasons explained below, the data representation of the interconnection pattern does not favor the employment of dormancy in the solution of interconnection equations. Specifically, we associate with each output variable a fan-out list and an interaction vector. The elements of an interaction vector constitute consecutive entries in a row of the connection matrix *C*. A fast loading of the matrix *C* requires the components of each interaction vector to be successively stored, that is a row pointer/column index storage scheme. On the other hand, when we perform the Crout reduction we need to compute only entries in columns corresponding to alert subnetworks, since no fill-in is caused in columns of the dormant subnetworks. Therefore elements in a column should be consecutively stored by the applied storage scheme. The column pointer/row index scheme fulfills this requirement. This brief analysis resulted in a contradiction for the storage scheme. A solution would be a different implementation of the interaction vectors in which the components of a vector would constitute elements of a column. In our implementation we employed a row pointer / column index storage scheme and we solved the connection equations loading the connection matrix with zeros in the place of the interaction vectors of the dormant sunetworks.

3.3.4. Backward Substitution

Backward substitution (U_{ℓ} $z_{\S}$ * $\tilde{r}_{i}$) is the last step we implemented. Before executing this elimination, we have to incorporate in $\tilde{r}j$ the effect of the input vector u_x on the unknowns z_{l}, solving the equation L, ?| • r_{ℓ} • l\ u_j. This is very efficiently carried out by loading the negative of the calculated a_{i} in the bottom part of r_{l} and then repeating the forward substitution on the bottom part only as shown in Figure 3-8. We perform, again, only nontrivial operations saving computational time. In the partial elimination we recompute only the bottom components of $\tilde{r}_{i}$ since we pivoted at the end on rows comprising an input variable.

Chapter 4
A PRACTICAL EXAMPLE OF MODULAR SIMULATION

This chapter is devoted to the modular simulation of a circuit. Our aim is to demonstrate the advantages of modular simulation , such as hierarchical design and computation efficiency. Furthermore, it is particularly well suited to an activity-directed circuit simulation scheme.

Since our program is still in the experimental stage, we chose to simulate the circuit shown in block form in Figure 4-1. This circuit, although simple, shows the benefits derived from the adoption of modular simulation.

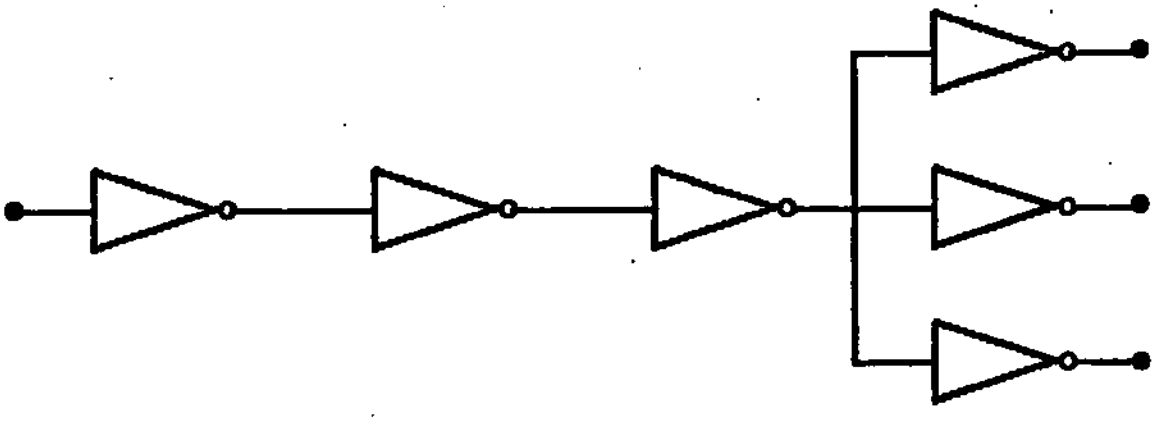


Figure 4-1: Simulated Network in Block Form

Figure 4-2 shows the structure of the network at the circuit level. It consists of six interconnected inverters, each built with n-channel MOSFETS's. We need to define the boundary for each subnetwork which will constitute the basic unit to be simulated separately from other subnetworks. We can consider different levels of subnetworks. For instance, we

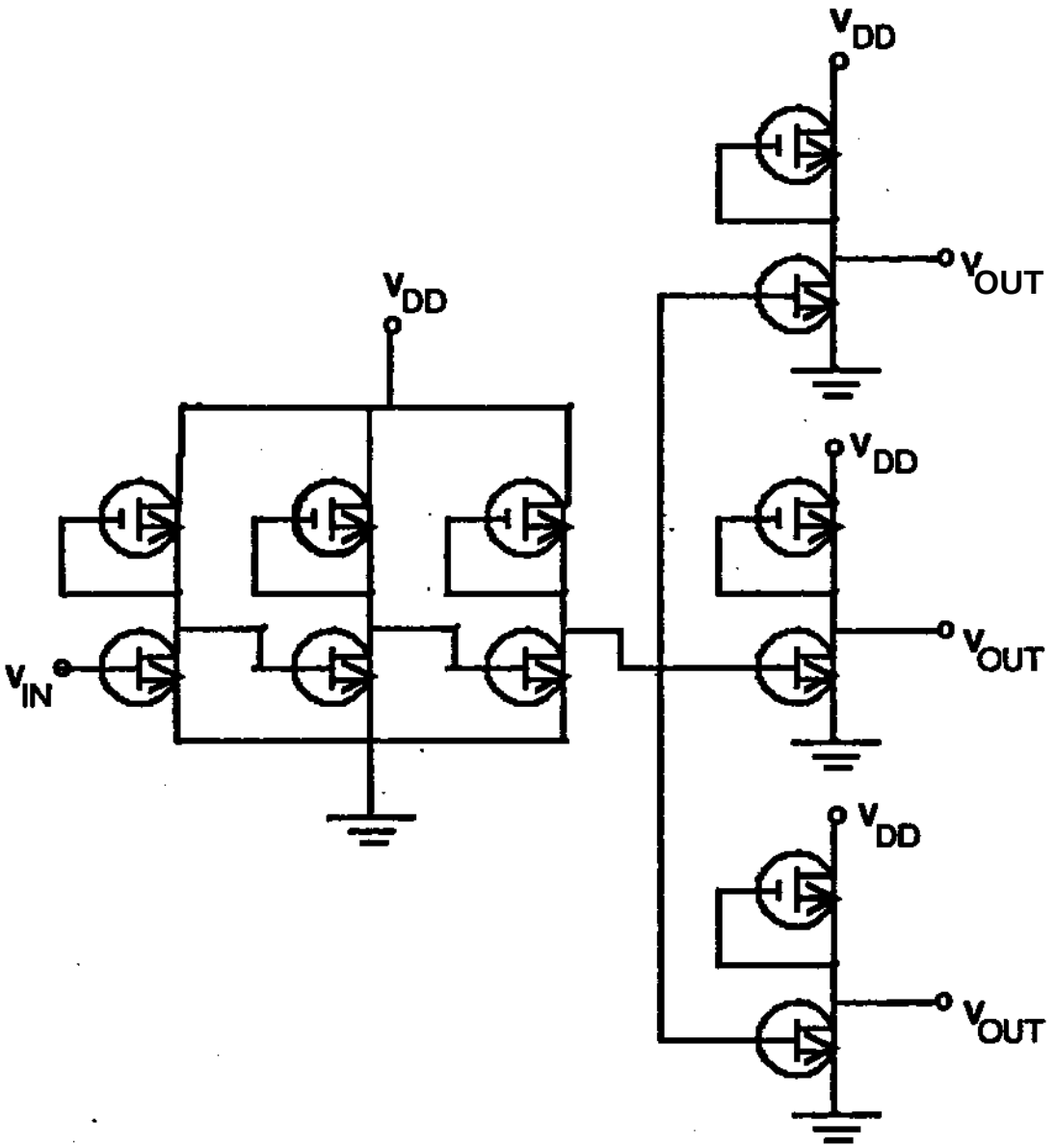


Figure 4-2: Simulated Network in circuit form

may define each inverter to be a subnetwork. We can also consider an inverter to consist of two subnetworks of lower level, each composed of a transistor. This hierarchy is very convenient for describing the whole network. First we define a transistor subnetwork, which describes a transistor in detail (topology, values of elements). Then we only need the interconnection pattern between transistor subnetworks to define an inverter subnetwork. Finally, we may consider the whole structure as a subnetwork in order to emloy it in building more complex networks. The simulated circuit is shown at different levels of subnetwork representation in Figures 4-3 and 4-4.

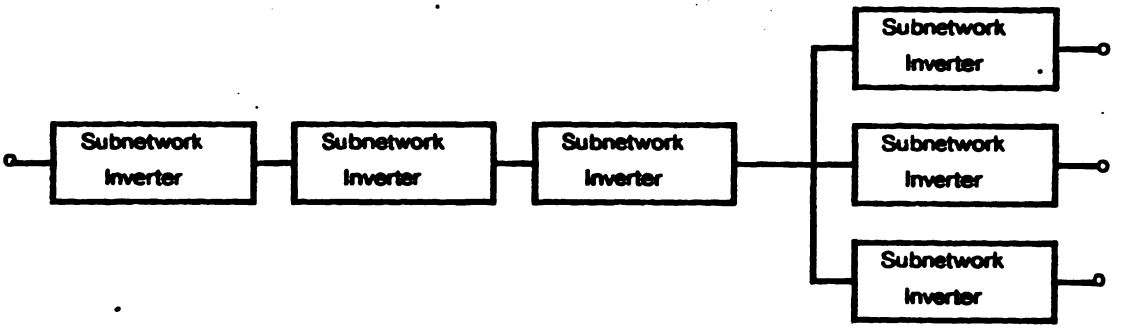


Figure 4-3: Circuit Representation at Inverter Level

We can also take advantage of modular simulation in network solution by employing a processed subnetwork to analyze all subnetworks defined to be this kind of subnetwork. For instance, if we consider the given circuit to consist of inverter subnetworks possessing the same structure, we may generate code (see Appendix A) for solving one of the six inverters, and use it to analyze all inverters of the network. Notice that, in this case, the lower level subnetwork representation employing transistor as basic unit has been used only for circuit description, while it has been ignored in code generation.

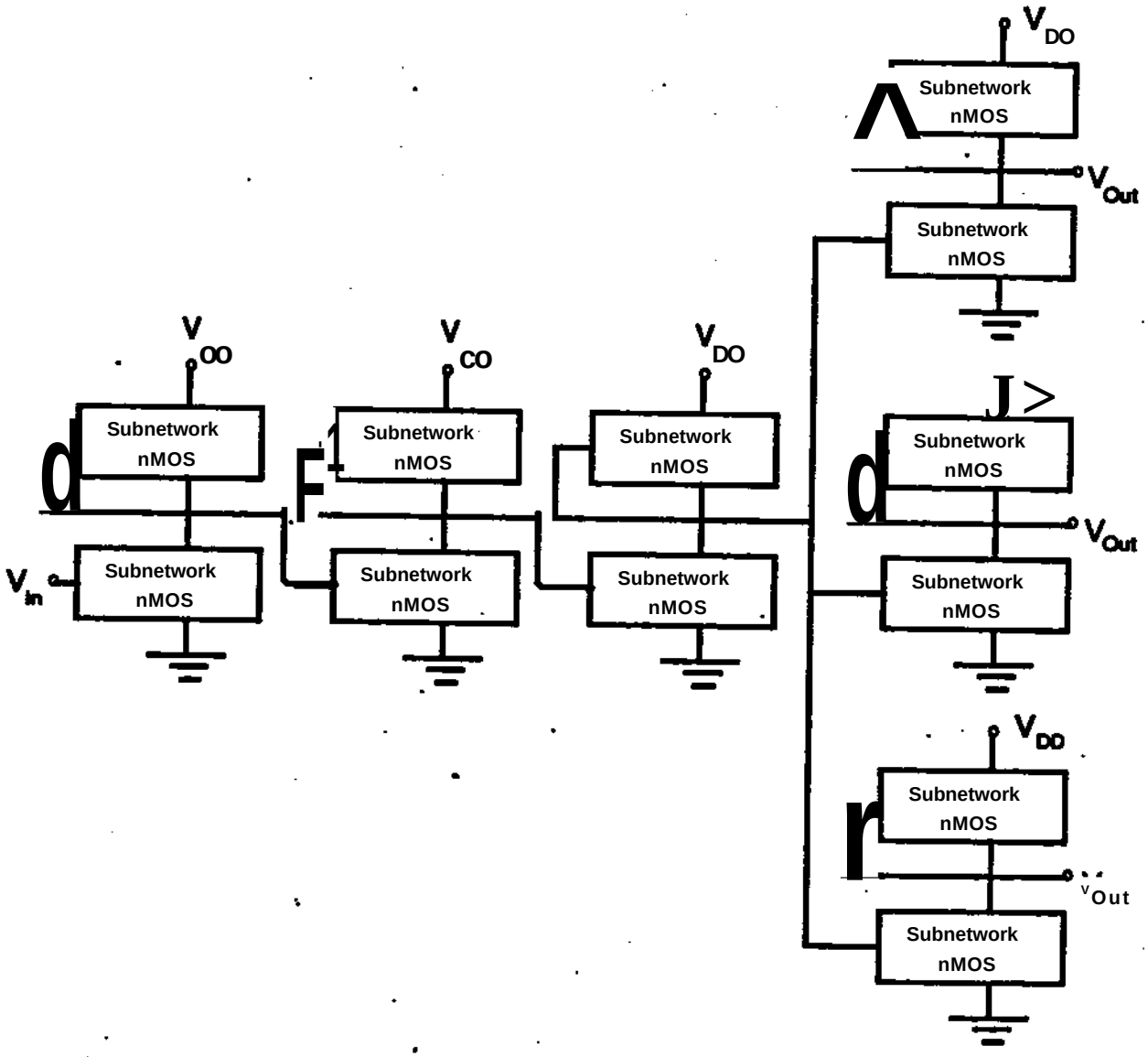


Figure 4-4: Circuit Representation at Transistor Level

Chapter 5
EPILOGUE

5.1. Conclusions

Our work has been the development of a sparse matrix package, which could be efficiently employed in an activity directed simulation program at the efficient solution of equations describing a network. We took advantage of ttie inherent repetitiveness of a limited number of elementary network configurations in the structure of integrated electronics, and applied the principle of modular simulation.

Adoption of modular simulation makes it possible to describe a network in terms of a few subnetworks, the analysts of a network then will requires only the solution of the submatrices describing the subnetworks. Therefore, we only need to generate solve code only for the subnetworks, instead of the whole coefficient matrix. This arrangement significantly reduces storage space requirements and permits the analysis of very large systems without storage problems. An important convenience we provide the user is the possibility of employing previously processed subnetworks from a library, which further decreases processing time. The user has absolute control over subnetwork definition and can define elements described by any nonlinear relation. Hence, subnetworks can be used for network analysis independently of the underlying technology.

Another advantage of modular simulation is the possibility of employing the latency (inactivity) exhibited by most subnetworks in the analysis of integrated circuits.

5.2. Further Research

The solution of the connection equations remains open to further research. There are alternatives for solving this problem that would allow fast loading of the connection matrix *C* and efficient employment of subnetwork latency. The first alternative is to seek another representation describing the interaction between different subnetworks. A second alternative would be a different storage scheme for the connection matrix. The last approach is to generate solve code for the connection equations.

A subject for further investigation is the pivoting parameters. In our implementation they were assigned constant values. It would be interesting to find out how these parameters could vary, according to the structure of the subnetworks, to yield a more efficient pivoting strategy.

Finally, our implementation may be improved if the sparsity of the right-hand side vector and the vector of unknowns is employed in the analysis of a subnetwork.

Appendix A
Generated Code for a Subnetwork Analysis

```
Procedure nMOSIl(Var R.JVGS,JVDS:Real ;Var JIDS:Real;VGS:Real;-
      Var VOS:Real;IDS.PL.PW.PVTO.PCox.PLMod:Real); Extern Pascal;

Procedure CINVl(Var S :. PSubnet; A : Action; L : CodeLevel);
PROCEDURE LoadJ;   { Load Jacobian }
VAR
 TrReal;  {Temporary}              ••
BEGIN
 WITH St.JMatrixt DO
  CASE L OF
   InteractiveLoop:
    BEGIN
     Rt.V[l] :« Ramp * (      5.000000);
     V[10] :* -0.5 • Pt.V[6] • Pt.V[2] • Pt.V[2] • Pt.V[3] • P+.V[4];
     V[9] :» -0.5 • Pt.V[12] • Pt.V[8] • Pt.V[8] • Pt.V[9] • Pt.V[10];
     V[2] :« -0.5 • Pt.V[12] • Pt.V[8] • Pt.V[8] • Pt.V[9] • P+.V[10];
     V[5] :« -Pt.V[l]
    END;
   OptimizeLoop:
    BEGIN
    END;
   TimeLoop:
    BEGIN
     V[l]:»-Beta;
     Rt.V[ll] := -Gammat.V[l];
     Rt.V[13] :« -Gammat.V[3];
     Rt.V[14] :« -Gammat.V[4];
     Rt.V[21] :« -Gammat.V[2]
    END;
   NewtonLoop:
    BEGIN
     nMOSIl(Rt.V[19].T.V[6].V[7], 0.0000000    ,Xt.V[17],Xt.V[19],
            Pt.V[3],Pt.V[4],Pt.V[5].Pt.VC6].Pt.V[7]);
     nMOSIl(Rt.V[20].V[8],V[4].V[3].Xt.V[16].Xt.V[18].Xt.V[20].
            Pt.V[9].Pt.V[10].Pt.V[ll].Pt.V[12],Pt.V[13])
    END
  END, {CASE}
END;  {LoadJ}
PROCEDURE Factor;  { LU Decomposition of Jacobian }
BEGIN
 WITH St.JMatrixt DO
```

```
  BEGIN
   CASE L OF
    InteractiveLoop:
     BEGIN
     LUt.V[16]:«V[10];
      LUt.V[18]:«V[9];
      LUt.V[20]:«V[2];
      LUt.V[22]:«V[5]
     END;
    OptimizeLoop:
     BEGIN
     END;
    TimeLoop:
     BEGIN
      LUt.V[23]:=V[1];
      LUt.V[ll]:=V[1];
      LUt.V[30]:=V[1];
      LUt.V[33]:=V[1];
      LUt.V[24]:=-LUt.V[23]»LUt.V[16];
      LUt.V[27]:=-LUt.V[24];
      LUt.V[31]:=-LUt.V[30]*LUt.V[20];
      LUt.V[35]:=-LUt.V[33]*LUt.V[22];
      LUt.V[26]:=-LUt.V[35];
      LUt.V[12]:=-LUt.V[ll]*LUt.V[18];
      LUt.V[13]:=-LUt.V[12];
      LU*.V[28]:=-LUt.V[27]-LUt.V[26];
      LUt.V[14]:=-LUt.V[13];
      LUt.V[29]:=-LUt.V[28];
      LUt.V[15]:=+LUt.V[31]+LUt.V[12];
      LUt.V[17]:=(-1.0)/LUt.V[15];
      LU*.V[19]:=(+LUt.V[14])/LUt.V[15];
      LUt.V[21]:=+LUt.V[17];
      LUt.V[25]:=(-1.0+LUt.V[19])/LUt.V[21]
     END;
    NewtonLoop:
     BEGIN
      LU+.V[4]:=V[8];
      LUt.V[l]:=V[6];
      LUt.V[6]:=-LUt.V[4];
      LUt.V[3]:«-LUt.V[l];
      LUt.V[7]:»V[4]-LUt.V[6];
      LUt.V[2]:«V[7];
      LUt.V[5]:*(+LUt.V[3])/LUt.V[2];
      LUt.V[8]:«V[3];
      LUt.V[9]:«(+LUt.V[4])/LUt.vt8];
      LUt.V[10y.«(+LUt.V[7])/LUt.V[8];
      LUt.V[32]:*+LUt.V[31]-LU*.V[9];
      LUt.V[34]:—LUt.V[32]»LUt.V[17];
      LUt.V[36]:»+LUt.V[29]+LUt.V[5]-LUt.V[10]-LUt.V[32]*LUt.V[19]
                 -LUt.V[34]*LUt.V[25]
     END
   END   {CASE}
  END  {WITH}
END; tFactor}
```

```
PROCEDURE FwdSubstituteCvar X.R:Vector);
BEGIN
 WITH St.Xt DO  •    •
  BEGIN
   V[1]:=R↑.V[1];
   V[2]:»-Rt.V[2]+V[l]:
   V[3]:«-Rt.V[3]+V[2];
   V[4]:»-Rt.V[4];
   V[5]:«Rt.V[6];
   V[6]:«Rt.V[6];
   V[7]:=Rt.V[7];
   V[8]:»Rt.V[8];
   V[9]:=Rt.V[9];
   V[10]:«Rt.V[10];
   V[ll]:»Rt.V[ll]-LUt.V[23]»V[7]:
   V[12]:«Rt.V[12]+V[2]-V[3];
   V[13]:»Rt.V[13]-LUt.V[30]«V[9]:
   V[14]:»Rt.V[14]-LUt.V[33]»V[10];
   V[15]:«Rt.V[15]+V[3]+V[12];
   V[16]:*-Rt.V[16]+V[3]-V[4]+V[12];
   V[17]:»Rt.V[17]+V[3]+V[12];
   V[18]:»-Rt.V[18]-V[3]-V[12]+V[17];
   V[19]:»(Rt.V[19]-LUt.V[l]*V[17]-LUt.V[3]«V[18])/LUt.V[2];
   V[20]:»(Rt.V[20]-LUt.V[4]*V[16]-LUt.V[6J*V[17]
           -LUt.V[7]*V[18])/LUt.V[8];
   V[21]:«(Rt.V[21]-V[6]-LUt.V[U]*V[8]+V[13]-LUt.V[12]»V[16]
           -LUt.V[13]*V[17]-LUt.V[14]»V[18])/LUt.V[15];
   V[22]:«(Rt.V[22]+V[4]+V[16]-V[17]+V[18]+V[21])/LUt.V[21];
   V[23]:«(Rt.V[23]-V[II]-LUt.V[27]"V[12]+V[13]-V[14]-LUt.V[26]*V[15]
           -LUt.V[28]*V[17]-LUt.V[29]*V[18]+V[19]-V[20]
           -LUt.V[32]*V[21]-LUt.V[34]*V[22])/LUt.V[36]
  END   {WITH}
END;  {FwdSubstitute}
PROCEDURE FwdPass;
VAR
G122.G123.G223.L23:real;
BEGIN
 WITH St DO
  BEGIN
   FwdSubstitute(X.R);
   G122:»(1.0)/LUt.V[21];
   G123:»(-LUt.V[34]*G122)/LUt.V[36];
   G223:«(1.0)/LUt.V[36]:
   L23:»-LUt.V[25];
   InteractionVect.V[l]t.V[l]:«-G122-L23«G123;
   InteractionVect.V[l]t.V[2]:=-L23*G223;
   InteractionVect.V[l]t.V[3]:=-Xt.V[22]-L23*Xt.V[23];
   InteractionVect.V[2]t.V[l]:«+G123;
   InteractionVect.V[2]t.v[2]:*+G223;
   InteractionVect.v[2]t.v[3]:»+Xt.v[23]
  END  {WITH}
 END;   {FwdPass}
PROCEDURE BwdSubstitute(var X:Vector);
BEGIN
```

```
 WITH S↑,X↑ DO
  BEGIN
   V[23]:=V[23];
   V[22]:=V[22]-LU↑.V[25]*V[23];
   V[21]:=V[21]-LU↑.V[19]*V[23]-LU↑.V[17]*V[22];
   V[20]:=V[20]-LU↑.V[10]*V[23]-LU↑.V[9]*V[21];
   V[19]:=V[19]-LU↑.V[5]*V[23];
   V[18]:=V[18]+V[23];
   V[17]:=V[17]-V[18];
   V[16]:=V[16]+V[21]-V[17];
   V[15]:=V[15]-V[17];
   V[14]:=V[14]-LU↑.V[35]*V[15];
   V[13]:=V[13]-LU↑.V[31]*V[21];
   V[12]:=V[12]-V[17];
   V[11]:=V[11]-LU↑.V[24]*V[12];
   V[10]:=V[10]-LU↑.V[22]*V[15];
   V[9]:=V[9]-LU↑.V[20]*V[21];
   V[8]:=V[8]-LU↑.V[18]*V[16];
   V[7]:=V[7]-LU↑.V[16]*V[12];
   V[6]:=V[6]-V[22]-V[13];
   V[5]:=V[5]-V[19]+V[11];
   V[4]:=V[4]+V[16];
   V[3]:=V[3]+V[12];
   V[2]:=V[2];
   V[1]:=V[1]
  END  {WITH}
END;  {BwdSubstitute}
PROCEDURE BwdPass;
 BEGIN
  WITH S↑,X↑ DO
   BEGIN
    V[22]:=(R↑.V[22]+V[4]+V[16]-V[17]+V[18]+V[21])/LU↑.V[21];
    V[23]:=(R↑.V[23]-V[11]-LU↑.V[27]*V[12]+V[13]-V[14]-LU↑.V[26]*V[15]
            -LU↑.V[28]*V[17]-LU↑.V[29]*V[18]+V[19]-V[20]
            -LU↑.V[32]*V[21]-LU↑.V[34]*V[22])/LU↑.V[36];
    BwdSubstitute(X)
   END  {WITH}
 END;  {BwdPass}
Begin { CINV1 }
 Case A Of
  LoadJacobian : LoadJ;
  LoadRHS : { For time being -- no action };
  LUFactor : Factor;
  ForwardPass : FwdPass;
  BackwardPass : BwdPass
 End   { Case }
End.  { CINV1 }
```

References

[Brayton 72] Brayton, R. K., Gustavson, F. G., and Hachtel, G. D.
A New Implicit Algorithm for Solving Differential- Algebraic Systems Using Implicit Backward Differentiation Formulas.
Proceedings of IEEE 60(1)^8-108, January, 1972.

[Duff 77] Duff, I. S.
A Survey of Sparse Matrix Research.
Proceeding of IEEE 65(4):182-193, April, 1977.

[Gustavson 70] Gustavson, F. G., Liniger, W. and Willoughby, R.
Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations.
Journal of the ACM (17):87-109,1970.

[Hachtel 71] G. D. Hachtel, R. K. Brayton, and F. G. Gustavson.
The Sparse Tableau Approach to Network Analysis and Design.
IEEE Transactions on Circuit Theory, January, 1971.

[Markowitz 57] Markowitz, H. M.
The elimination form of the inverse and its application to Linear Programming.
Management Science (3):255-269,1957.

[SakallahSO] Sakallah, K. and Director, S. W.
An Activity-Directed Circuit Simulation Algorithm.
In *Proc. of the 1980 IEEE International Cont on Circuit and Computers.* IEEE, 1980.

[Sakallah 81] Sakallah, K.
Mixed Simulation of Electronic Integrated Circuit.
PhD thesis, Carnegie-Mellon University, 1981.