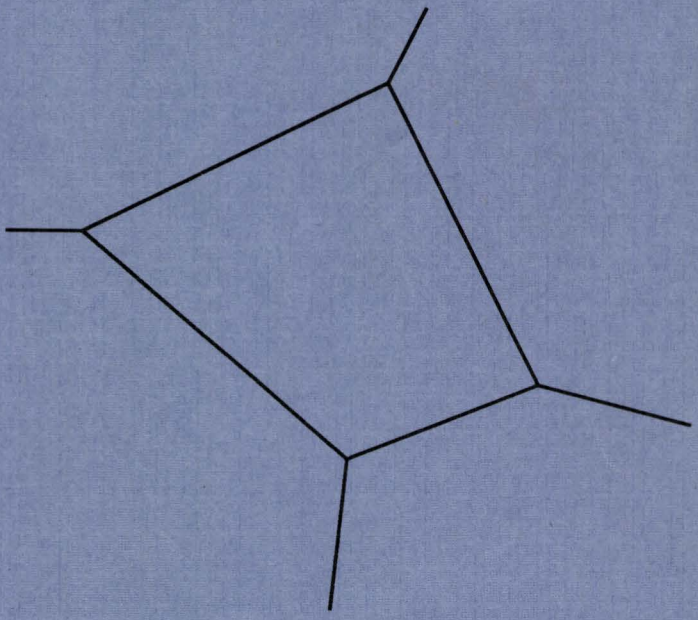
PLATO : a piecewise linear analysis tool for mixed-level circuit simulation

PLATO

A Piecewise Linear Analysis Tool for Mixed-Level Circuit Simulation



M.T. van Stiphout

PLATO -
A Piecewise Linear Analysis Tool
for
Mixed-Level Circuit Simulation

PLATO - A Piecewise Linear Analysis Tool for Mixed-Level Circuit Simulation

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. ir. M. Tels, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 18 mei 1990 om 16.00 uur.

door

Martinus Theodorus van Stiphout

geboren te Geldrop

Dit proefschrift is goedgekeurd
door de promotoren

prof. dr. ing. J.A.G. Jess

en

prof. dr. ir. W.M.G. van Bokhoven

Druk: Dissertatiedrukkerij Wibro, Helmond.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Stiphout, Martinus Theodorus van

PLATO - a piecewise linear analysis tool for mixed-level circuit simulation/
Martinus Theodorus van Stiphout. - [S.l:s.n]. Fig.,tab.
Proefschrift Eindhoven. - Met lit. opg., reg.
ISBN 90-9003390-4
SISO 663.43 UDC 621.3.011.72:519.87 NUGI 832
Trefw.: elektronische schakelingen; simulatie.

Contents

Samenvatting 7

Abstract 9

1. Introduction 11
   1.1 Circuit simulation 11
   1.2 The piecewise linear alternative 13
   1.3 Piecewise linear modeling 14
   1.4 Implementation aspects 15

2. Sparse matrix techniques 19
   2.1 The rank one update 19
   2.2 Sparse implementation 23
   2.3 Forward-backward substitution 27
   2.4 Sparsity analysis 28
   2.5 Recursive bordered block diagonal matrices 32

3. Solving the pwl equations 35
   3.1 Equation building 35
   3.2 Initial solution 35
   3.3 DC solution 46

4. Multirate integration techniques 49
   4.1 Numerical integration 49
   4.2 Multirate integration methods 51

5. Transient analysis 55
   5.1 Introduction 55
   5.2 A uniform stepsize simulator 57

5.3 Pivoting 60
5.4 A multirate simulator 63
5.5 Some optimizations 71
5.6 Event clustering 73

6. Simulation results 77
6.1 Landman circuit 77
6.2 Inverter chains 80
6.3 Ring inverters 80
6.4 A/D converter 80
6.5 Counter 85
6.6 Two complement accumulator 86
6.7 Simple phase locked loop 86
6.8 Neural networks 87
6.9 Program statistics 87

Conclusion 91

Appendix 1: Algorithmic notations 95

References 99

Samenvatting

Sinds de ontwikkeling van de eerste circuit simulatie programma's, ongeveer twintig jaar geleden, zijn veel nieuwe ideeën en abstractie niveaux (denk bijvoorbeeld aan gedragssimulatie, logische simulatie, functionele simulatie, etc) geintroduceerd. Voor diverse niveaux zijn commerciële simulatoren verkrijgbaar. Desondanks wordt circuit simulatie in de industrie nog erg vaak gebruikt. De computersystemen die nodig zijn om de simulatiebehoeften van de ontwerpers met betrekking tot circuit analyse te bevredigen worden echter steeds groter en sneller, aangezien de te simuleren schakelingen nog steeds in grootte toenemen.

In dit proefschrift worden technieken behandeld die een aanzet vormen tot het elimineren van een aantal van de nadelen die aan de huidige circuit simulatoren zijn verbonden. Het gebrek aan flexibiliteit met betrekking tot het introduceren van nieuwe modellen en de toepassing van macromodellen wordt geëlimineerd door de introductie van stuksgewijs lineaire modellen voor alle componenten. Het gebruik van macromodellen voor gedeelten van de schakeling waarvan het gedrag reeds in detail bekend is, kan de simulatie aanzienlijk versnellen. Daarbij levert de uniforme modellering van alle componenten een elegante manier om *mixed-level* simulatie te implementeren. Diverse andere simulatoren ondersteunen *mixed-level* simulatie door voor ieder subcircuit dat op een ander niveau moet worden gesimuleerd een ander algoritme te kiezen. Een belangrijk voordeel van de nieuwe simulator is de sterk verbeterde convergentie in vergelijking met de klassieke iteratieve methoden. Deze eigenschap is karakteristiek voor de algoritmen die we gebruiken voor het oplossen van de stuksgewijs

lineaire vergelijkingen.

Afgezien van stuksgewijs lineaire modellering steunt de simulator op twee belangrijke principes: *multirate* integratie en ijle incrementele technieken. Het idee achter *multirate* integratie is het exploiteren van subcircuits die (tijdelijk) in rust verkeren door ze te integreren met een grote integratie stap. Erg actieve subcircuits kunnen dan worden geintegreerd met kleinere stapgrootten. In principe kan hierdoor veel rekentijd worden bespaard aangezien er voornamelijk wordt gerekend in de actieve circuit gedeelten. Het herhaald oplossen van grote ijle lineaire systemen zoals dat gebeurd in conventionele simulatoren wordt voorkomen door middel van de ijle incrementele methoden. Deze methoden zijn bijzonder geschikt voor operaties die samenhangen met het stuksgewijs lineaire karakter van de algoritmen. Ook passen ze perfect bij het *event driven* integratie schema waarin vaak selectieve veranderingen voorkomen. Merk op dat de simulator weliswaar zonder enig probleem op bijvoorbeeld het logische niveau kan simuleren, maar dat hij nooit de snelheid van een speciaal voor logische simulatie ontworpen programma kan evenaren. We willen hier nogmaals benadrukken dat de simulator op vele niveaux tegelijk kan simuleren. het is dan ook niet eerlijk om een van deze niveaux te selecteren en de resultaten te vergelijken met een speciaal op dat niveau toegesneden programma.

Tot slot moeten we concluderen dat een evaluatie van de methoden zoals ze in dit proefschrift worden voorgesteld een positieve indruk achter laat. De balangrijkste principes blijken efficient te werken hoewel de numerieke integratie nog verder geoptimaliseerd moet worden om een echt snel simulatieprogramma te verkrijgen. Diverse (*mixed-level*) circuits zijn echter met succes gemodelleerd en gesimuleerd.

Abstract

Since the development of the first circuit simulation programs, about two decades ago, many new concepts and simulation levels (such as logic level, behavioral level, functional level, etc.) have been introduced. Simulators are commercially available for various simulation levels. Conventional circuit simulation, however, remains heavily used in industry. Large and fast computers are required to keep up with needs of the designer who wants to simulate circuits with a continuously increasing size.

In this thesis techniques are discussed that potentially are able to eliminate a number of drawbacks of current circuit simulators. First of all the lack of flexibility with respect to the addition of new models and the application of macro models is eliminated by the introduction of piecewise linear models for all components. The introduction of macro models for circuit parts of which the detailed behavior is already known, can speed-up the simulation tremendously. Furthermore the uniform modeling of all components yields an elegant way to implement true mixed-level simulation. Several other simulation tools support mixed-level simulation by the implementation of a variety of algorithms, each of which is aimed at the simulation of a specific sub-circuit at a specific level. Another important issue is the improved convergence as compared to classical iterative methods, which is characteristic for the algorithms that are used to solve the piecewise linear equations.

Apart from the piecewise linear modeling, the simulator is built on two basic concepts: multirate integration and sparse update techniques. The idea behind multirate integration is the exploitation of the latency

of sub-circuits by integrating them with large integration time steps. Very active sub-circuits can be integrated by small time steps. Principally, a large amount of cpu time can be saved since the computational effort is restricted to active circuit parts. The sparse updating schemes are an attempt to eliminate the repeated solution of large sparse linear systems as it occurs in conventional simulators. These techniques are especially suited for matrix operations related to the piecewise linear character of the algorithms. They also fit neatly into the event driven integration scheme with its inherently selective updates. Note that, although the piecewise linear simulator can easily simulate at e.g. the logic level, it will never be able to obtain the simulation speed of dedicated logic simulation programs. Again we should emphasize the ability of the simulator to simulate at several levels simultaneously. This is opposed to selecting a single level and comparing the results with those of tools designed specificly for that particular level.

At the end of this thesis, an evaluation of the techniques as they are proposed and implemented yields a positive impression. The basic concepts appear quite efficient although numerical integration should further be optimized to obtain a really fast simulator. Several (mixed-level) circuits have been successfully modeled and simulated.

1. Introduction

1.1 Circuit simulation

Starting out in the mid-sixties with the simulation of only a few transistors, simulation has evolved to one of the most intensively used design tools in the 1990's. Today the design of complex integrated circuits would be impossible without the use of advanced simulation programs, in some cases able to simulate thousands of transistors. Circuit designers can now choose from numerous simulation tools, ranging from low-level circuit simulation for very accurate analysis of small circuit parts to very coarse logic analysis of an entire integrated circuit. Without denying the importance of all other simulation methods and levels, the basic reference for this thesis will be conventional circuit simulation, which may be regarded as the first and most elementary form of simulation. The piecewise linear simulation techniques presented in this thesis however, are not restricted to any simulation level at all, but, on the contrary, are capable of merging all of them into one mixed-level simulation tool. A short review of circuit simulation techniques seems appropriate. For a thorough treatment of simulation principles see e.g. [1], for a more recent but less exhaustive text see [2].

The circuit simulation problem is commonly described as a set of nonlinear differential-algebraic equations $F(\dot{x}, x, t) = 0$, with initial conditions $x(0) = X_0$, in which x is the vector of circuit variables, $\dot{x}$ the vector of circuit variable derivatives and parameter t, of course, is time. The solution process basicly consists of three parts. At every time point

t_{n+1} we have to solve $F(\dot{x}_{n+1}, x_{n+1}, t_{n+1}) = 0$. The unknown vector $\dot{x}_{n+1}$ is eliminated by substitution of a stiffly stable numerical integration formula. Next the resulting nonlinear system $F(\dot{x}_{n+1}(x_{n+1}), x_{n+1}, t_{n+1}) = 0$ is solved for x_{n+1} by Newton-Raphson iteration. The linear systems arising in this process are solved by LU decomposition.

The above principle has been implemented in several simulation programs [3, 4] and survived nearly two decades of simulation research. Programs like SPICE are still heavily used, both in industry and in academic environments. However a number of drawbacks should be mentioned. First of all the process described above is computationally expensive, especially for large circuits. A large part of the computing resources is used for the repeated solution of the linear system resulting after linearization. A second problem is formed by the limited convergence capabilities of the Newton-Raphson iteration technique. Because of the local convergence, the simulator quite often has to reduce the integration time step to obtain a solution. In many difficult cases the program aborts because the time step becomes prohibitively small. Finally there is a lack of flexibility regarding the introduction of new component models. Such an operation would imply altering the simulators source code, an option which is rather unattractive for the average user.

Several attempts have been made to circumvent the limitations of SPICE-like simulation programs, mostly by rejecting one or more of the principal circuit simulation techniques. Practically all of the so-called third generation simulation tools are concerned with the limited but important class of MOS circuits. The first program to be mentioned is MOTIS [5], a timing simulator for quasi-unidirectional MOS circuits. Based on physical reasoning, MOTIS achieved a tremendous speed up by applying regula falsi and a single relaxation sweep for every time point instead of sparse matrix techniques, numerical integration and Newton-Raphson iteration. Two other programs aimed at the same class of circuitry are SPLICE [6] which used even more the unidirectional nature of MOS digital circuits and DIANA [7], which again uses sparse matrix methods. Both SPLICE and DIANA are mixed mode simulators: they are capable of simulating different circuit parts in two or more modes. A mode can be e.g. register transfer level, logic level, timing simulation or circuit simulation. Mixed mode simulation

is gaining popularity in industry progressively.

Another attempt to cope with some of the limitations (mainly for large MOS circuits) is the waveform relaxation method, introduced in 1982 [8]. This iterative method repeatedly decomposes the circuit into a number of subsystems, which are analyzed separately for the required time interval. The idea is that latent subcircuits require less computation time than active subcircuits. Most of the gain is due to the limited activity of logic circuitry. Given a network decomposition the conventional solution methods are applied but always on a relatively small circuit. The method works especially well for unidirectional circuits and can yield a speed-up of about 10-50 times compared to SPICE-like programs. The largest drawback is the linear convergence, resulting in much smaller or even no speed-ups for heavily coupled circuits. Derivation of a suitable decomposition can be the second problem. An excellent overview of state of the art waveform relaxation techniques can be found in [9].

1.2 The piecewise linear alternative

The piecewise linear simulation program developed during the past four years started out with the ambitious intention to exploit all positive innovations of current simulation research but avoiding most of the drawbacks. Its basic principles and implementation will be discussed in detail in the following chapters. The first objective is to exploit latent circuit parts but without applying the slowly converging relaxation method. Instead a multirate integration scheme is used in a direct simulation method, i.e. no iterations are necessary. A second but equally important item, is the elimination of Newton's method from the simulation algorithm. This is achieved by introducing piecewise linear modeling techniques. Instead of local convergence, algorithms to solve the piecewise linear equations exhibit global convergence properties. Moreover, the piecewise linear modeling enables us to model a large variety of components in a uniform way, implementing mixed-level simulation in a very natural and elegant manner. Finally, adding new component models becomes trivial, since piecewise linear descriptions can be represented in matrix form and easily fed into the simulator.

1.3 Piecewise linear modeling

A general description of a continuous piecewise linear dynamical system conceived by van Bokhoven [10] is shown in equation (1.1):

$$\begin{bmatrix} 0 \\ \dot{u} \\ p \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x \\ u \\ q \end{bmatrix} + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{1.1}$$

$$\text{where } \dot{u} = \frac{\partial u}{\partial t}, \text{ and} \tag{1.2}$$

$$p^t \cdot q = 0, \; p \geq 0 \text{ and } q \geq 0. \tag{1.3}$$

The system's terminal variables are represented by vector x. Dynamical behavior of a piecewise linear system can be modeled in a very general and powerful way by means of the state variables u. Conditions (1.3) state that vectors p and q contain nonnegative elements only and have zero inner product. Omitting vectors u and $\dot{u}$ and the corresponding submatrices from the general system description and furthermore assuming conditions (1.3) hold and $q = 0$, the resulting system is a simple linear mapping $0 = A_{11}x + a_1$. The region in which this mapping is valid is bounded by the set of inequalities $p = A_{31}x + a_3 \geq 0$. Clearly, a different mapping and corresponding region can be obtained by simply exchanging elements of p and q, i.e. pivoting on an element of matrix A_{33}. It's quite obvious that the maximum number of mappings that can be defined by a single matrix is bounded to 2^n if n is the order of matrix A_{33}. Since we consider mappings that can be stored in a matrix form as shown in equation (1.1), the effect of a pivoting operation with element A_{33}^{kl} is a rank one update for the matrices involved [11]:

$$0 = A'_{11}x + a'_1 \text{ , with} \tag{1.4}$$

$$A'_{11} = A_{11} - \frac{A_{13}^{*l} A_{31}^{k*}}{A_{33}^{kl}} \text{ and } a'_1 = a_1 - \frac{A_{13}^{*l} a_3^k}{A_{33}^{kl}} \tag{1.5}$$

in which the $*$ is used to indicate a row or column. Therefore only continuous mappings can be described by our formalism. The properties of continuous piecewise linear mapping were studied in detail by van Eijndhoven [12]. He discovered that the general matrix description can be cast into a special form by introducing two

additional restrictions: every segment in the mapping must be convex and empty segments are not allowed. For a mapping with two intersecting boundary hyperplanes this matrix looks like:

$$\begin{bmatrix} 0 \\ p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} A^0 & w^0 & w^1 \\ n^{0t} & 1 & -\lambda_{12} \\ n^{1t} & -\lambda_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ q_1 \\ q_2 \end{bmatrix} + \begin{bmatrix} a^0 \\ c^0 \\ c^1 \end{bmatrix} \tag{1.6}$$

where vectors n and scalar c define the boundary planes and vectors w are update vectors. The constants λ determine whether or not a plane continues in a different direction after the intersection with another plane. The matrix can be systematically extended in case more boundary hyperplanes are present. A reasonable collection of elementary piecewise linear models can be constructed by merging the ones available from [10,13,14].

The only viable alternative for the compact matrix notation introduced above, was documented exhaustively by Chua in a number of papers [15-19]. His canonical representation of a piecewise linear function is listed in equation (1.7):

$$f(x) = a + Bx + \sum_{i=1}^{p} c_i \, | \, \alpha_i^t \cdot x - \beta_i \, | = 0 \, . \tag{1.7}$$

in which α_i is the normal vector to the ith hyperplane, β_i is a constant, a and c_i are constant vectors and B is a constant $n \times n$ matrix. Despite thorough analytical treatment, the methods devised by Chua are limited to the determination of driving point and transfer characteristics. The canonical formulation itself is less powerful than the one suggested by van Bokhoven [10].

1.4 Implementation aspects

PLATO (almost an acronym for Piecewise Linear Analysis TOol) [20] was implemented in the C programming language [21] and runs on a number of different UNIX machines (e.g. Apollo and Hewlett-Packard work stations, Alliant FX/8 mini-supercomputer). Apart from PLATO itself, a number of utility programs were developed and implemented to offer what nowadays is referred to as a user interface. User interface technology is developing rather fast. Applications under development are liable to become outdated while still under construction. In this

context the schematic entry tool ESCHER+ may serve as an illustrative example. Initially using some Hewlett-Packard specific graphics package with no notion of workstations and windows at all, the next step was to port it to an Apollo workstation with numerous possibilities for the creation of windows and menus. In this respect an accepted standard like the Graphical Kernel System (GKS) should be considered old-fashioned even before its final definition was established, since it totally ignores the concept of a windowing system and has only limited menu resources. Next it appeared that a portable UNIX compatible windowing system called the X Window System [22] was becoming a de facto standard for user interfaces and computer graphics. Again developments are going fast. Toolkits based upon a set of high level object oriented graphics functions built on top of the lowest level X functions will probably be replaced soon by interactive tools for the design and implementation of user interfaces, generating toolkit source code only as an intermediate. Although the graphics systems become more and more complicated, user interface design time has decreased drastically.

Let us shortly review the separate components that form PLATO's user interface as depicted in figure 1.1:

escher+ an interactive schematic entry program used to construct circuits in a user friendly manner, providing runtime assistance and network checking [23, 24]. Only recently ESCHER+ was enhanced with a logic simulation mechanism and behavioral description capabilities [25].

ndml the ndml++ language compiler [26] taking an extension of the Network Description and Modeling Language [27] as input and generating a form of ndml with reduced complexity, i.e. all expressions evaluated, high level control constructs expanded, etc. Ndml can be generated by text-editing or by the extraction of network data from the escher+ database.

superplog an interactive graphical simulation output postprocessor. It has the ability to manipulate, compare and plot simulation results. Again an example of user interface technology evolution: at the moment a new implementation based on OSF/Motif [28, 29] is being developed.

Many new simulation tools tend to use the old-fashioned SPICE input format, to remain compatible and to please stubborn users. For PLATO the network description and the simulation task description were separated and replaced by more advanced newly designed languages. Especially the powerful constructs that are available in ndml++ are incomparable to the low level error prone SPICE input specification language.

The following chapters describe the techniques that were devised and implemented in the piecewise linear simulator. After an overview of the sparse matrix techniques (chapter 2) that were developed to exploit latency behavior, the solution of the pwl equations is treated in detail (chapter 3). Two more chapters cover the concept of multirate integration (chapter 4) and a rigorous treatment of the transient analysis (chapter 5). The thesis is concluded with some simulation examples and results (chapter 6).

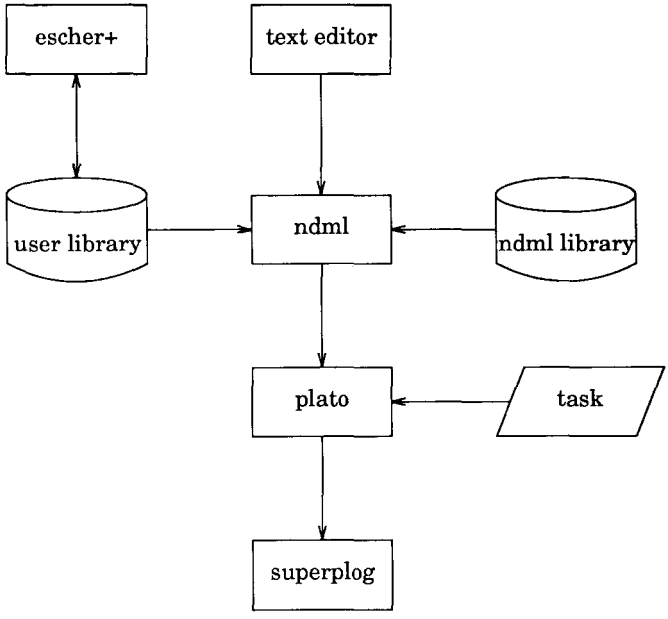


Figure 1.1. *Simulation package overview. Arrows are used to indicate the data flow.*

2. Sparse matrix techniques

A well known method for the computation of the inverse of matrices that have been updated by a rank k matrix is the Sherman-Morrison-Woodbury [30, 31] formula:

$$(A + XCY^t)^{-1} = A^{-1} - A^{-1}X(C^{-1} + Y^tA^{-1}X)^{-1}Y^tA^{-1}. \qquad (2.1)$$

Using this formula the new inverse of an updated matrix can be obtained from the original one in order kn^2 operations, while full recomputation would require order n^3 for an $n \times n$ matrix. In many practical applications like e.g. solving large sets of linear equations, the matrix is not inverted but decomposed into its triangular factors which requires only $\frac{1}{3} \cdot n^3$ operations. The advantages are even greater for matrices with special form or sparse matrices: while the inverse of a sparse matrix will generally be a full matrix, the triangular factors remain sparse. In the following sections Bennetts [32] algorithm for the update of the triangular factors will be introduced. Since the piecewise linear simulator performs only rank one updates we will restrict ourselves to the rank one version of Bennetts algorithm. Next we will review the sparse implementation and the advantages that can be gained from sparse source vectors during the forward-backward solution process.

2.1 The rank one update

Consider the triangular factorization of an $(n \times n)$ nonsingular matrix A:

$$A = LDU, \text{ with} \tag{2.2}$$
$$L_{ii} = U_{ii} = 1 \text{ for } 1 \le i \le n \text{ and}$$
$$L_{ij} = D_{ij} = D_{ji} = U_{ji} = 0 \text{ for } 1 \le i < j \le n.$$

The algorithm devised by Bennett efficiently determines the new triangular factors if matrix A is subject to a rank one update cdr^t:

$$A^* = A + cdr^t = L^*D^*U^*, \tag{2.3}$$

where c and r are vectors $c = (c_1, c_2, ..., c_n)^t$, $r = (r_1, r_2, ..., r_n)^t$ and d is a scalar. By definition A^k is the submatrix of A of order $n - k + 1$, resulting after $k - 1$ elimination steps with elements (A^k_{ij}), $k \le i, j \le n$. So the matrix to be decomposed in the first elimination step $A^1 = A$. The resulting triangular factors are:

$$L^1_{i1} = \frac{A^1_{i1}}{A^1_{11}},\ D^1_{11} = A^1_{11},\ U^1_{1j} = \frac{A^1_{1j}}{A^1_{11}} \text{ for } 1 \le i, j \le n. \tag{2.4}$$

The problem faced after the first elimination step is given by:

$$A^2_{ij} = A^1_{ij} - L^1_{i1}D^1_{11}U^1_{1j} = A^1_{ij} - \frac{A^1_{i1}A^1_{1j}}{A^1_{11}} \text{ for } 2 \le i, j \le n. \tag{2.5}$$

The updated triangular factors after the first step are readily identified as:

$$D^1_{11}{}^* = A^1_{11}{}^* = A^1_{11} + c_1dr_1 = D^1_{11} + c_1dr_1 \tag{2.6}$$

$$L^1_{i1}{}^* = \frac{A^1_{i1}{}^*}{A^1_{11}{}^*} = \frac{A^1_{i1} + c_idr_1}{D^1_{11} + c_1dr_1} = \frac{D^1_{11}L^1_{i1} + c_idr_1}{D^1_{11} + c_1dr_1}$$

$$= L^1_{i1} + \frac{dr_1}{D^1_{11} + c_1dr_1}(c_i - L^1_{i1}c_1) \tag{2.7}$$

$$U^1_{1j}{}^* = \frac{A^1_{1j}{}^*}{A^1_{11}{}^*} = \frac{A^1_{1j} + c_1dr_j}{D^1_{11} + c_1dr_1} = \frac{D^1_{11}U^1_{1j} + c_1dr_j}{D^1_{11} + c_1dr_1}$$

$$= U_{1j}^1 + \frac{dc_1}{D_{11}^1 + c_1 dr_1}(r_j - U_{1j}^1 r_1) \tag{2.8}$$

Our task is now to express the updated A_{ij}^{2*} as a new rank one update problem in terms of the original matrices and update. Then we have obtained a similar problem but with smaller dimension. The method can then be recursively applied until the updated triangular factors have been obtained. Our new problem is denoted by:

$$A_{ij}^{2*} = A_{ij}^{1*} - L_{i1}^1 {}^* D_{11}^1 {}^* U_{1j}^{1*} = A_{ij}^2 + c_i^2 d^2 r_j^2 \tag{2.9}$$

where superscripts indicate modifications like the ones defines for A. Substituting equations (2.6), (2.7) and (2.8) leads to:

$$A_{ij}^{2*} = A_{ij}^1 + c_i dr_j - \tag{2.10}$$

$$\left[L_{i1}^1 + \frac{c_i dr_1 - L_{i1}^1 c_1 dr_1}{D_{11}^1 + c_1 dr_1}\right]\left[D_{11}^1 + c_1 dr_1\right]\left[U_{1j}^1 + \frac{c_1 dr_j - U_{1j}^1 c_1 dr_1}{D_{11}^1 + c_1 dr_1}\right]$$

or

$$A_{ij}^{2*} = A_{ij}^1 - L_{i1}^1 D_{11}^1 U_{1j}^1 + c_i dr_j - c_1 dr_j L_{i1}^1 - \tag{2.11}$$

$$\frac{c_i dr_1 - L_{i1}^1 c_1 dr_1}{D_{11}^1 + c_1 dr_1}(D_{11}^1 U_{1j}^1 + c_1 dr_j) \ ,$$

finally resulting in:

$$A_{ij}^{2*} = A_{ij}^2 + \frac{dD_{11}^1}{D_{11}^1 + c_1 dr_1}(c_i - L_{i1}^1 c_1)(r_j - U_{1j}^1 r_1) \ . \tag{2.12}$$

Summarizing the new expressions for update vectors c and r, the new value for d and the updated entries in L and U are given:

$$c_i^2 = c_i - L_{i1}^1 c_1 \tag{2.13}$$

$$r_j^2 = r_j - U_{1j}^1 r_1 \tag{2.14}$$

$$d^2 = \frac{dD_{11}^1}{D_{11}^1 + c_1 dr_1} \tag{2.15}$$

$$L_{i1}^{1\,*} = L_{i1}^{1} + \frac{dr_1}{D_{11}^{1} + c_1 dr_1} c_i^2 \tag{2.16}$$

$$D_{11}^{1\,*} = D_{11}^{1} + c_1 dr_1 \tag{2.17}$$

$$U_{1j}^{1\,*} = U_{1j}^{1} + \frac{dc_1}{D_{11}^{1} + c_1 dr_1} r_j^2 \tag{2.18}$$

The results obtained above for the first Gaussian elimination step can simply be generalized to yield the final algorithm. Factors p and q are introduced for computational convenience.

Algorithm 2.1. *Bennetts algorithm for the update of the triangular factors after a rank one update on the original matrix.*

$d = 1.0$;
for $i = 1$ **to** n **do**
    $D_{ii} = D_{ii} + r_i \cdot d \cdot c_i$;
    $p = c_i \cdot d \;/\; D_{ii}$;
    $q = r_i \cdot d \;/\; D_{ii}$;
    $d = d - p \cdot D_{ii} \cdot q$;
    for $j = i + 1$ **to** n **do**
        $c_j = c_j - L_{ji} \cdot c_i$;
        $L_{ji} = L_{ji} + q \cdot c_j$;
        $r_j = r_j - U_{ij} \cdot r_i$;
        $U_{ij} = U_{ij} + p \cdot r_j$;
    od;
od;

It is obvious that the number of operations performed by Bennetts algorithm is far less than a recomputation of the triangular factors would require. Numerically however, some care must be taken. In the process of factorization we have the freedom to reject a pivot if its value is too small. We then apply some pivoting strategy, like partial or complete pivoting, to obtain a better suited pivot involving row and/or column swaps. Using algorithm 2.1, however, we are committed to the pivot order that was used for the initial LU decomposition. This order may very well be unsuitable for any of the succeeding matrices. In fact it is even possible for pivots to become zero. Therefore the pivot values must be monitored carefully during the update process and the LU decomposition should be recomputed if any value becomes too small.

2.2 Sparse implementation

A close examination of Bennetts algorithm immediately reveals its potential for a sparse implementation. The inner as well as the outer loop body can be completely skipped if $c_i = r_i = 0$. If $c_i = 0$ and $r_i \neq 0$, only row vector r and column L_{*i} are liable to change. In the opposite case, $c_i \neq 0$ and $r_i = 0$, only column vector c and row U_{i*} are affected. The often used wildcard notation in L_{*i} and U_{i*} means that the asterisk can be replaced by any legal index value, i.e. indicating column i of matrix L and row i of matrix U respectively.

The efficiency of the method described when applied to piecewise linear networks was already shown for the restricted case of piecewise linear resistive networks [11]. Update vectors originating from pivoting are generally sparse, very often containing only one or a few non zero elements. Since matrix A is sparse also, the update process will be able to skip many entries compared to the full version of the algorithm. Inevitably some fill-in is generated during the update process, as is depicted in figure 2.1.

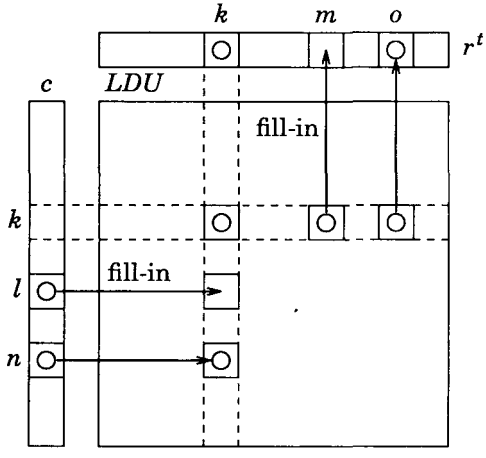


Figure 2.1. *Update of a sparse matrix with sparse update vectors. The triangular factors are stored in one matrix. Circles represent nonzeros. Arrows indicate fill-in.*

For the sparse implementation we assume the following datastructure. The L, D and U matrices are stored in one sparse matrix:

$$A_{ij} = \begin{cases} L_{ij} & \text{if } i > j \\ D_{ij} & \text{if } i = j \\ U_{ij} & \text{if } i < j \end{cases} \tag{2.19}$$

Only nonzero elements are stored connected by column and row links in a bi-threaded list [33]. Two pointer arrays are provided to access rows and columns. The diagonal elements can be accessed directly through a third array of pointers. An important advantage of this representation is the ease with which new elements (fill-in) can be inserted into the matrix. A drawback may be the extra memory requirements compared to more conventional methods. A visualization of the sparse matrix datastructure is shown in figure 2.2. The sparse update vectors are stored in linear linked lists again containing only the nonzero elements. In the sequel a sparse vector is sometimes treated as a set of its nonzero elements. The corresponding redefinition of some mathematical operators for sparse vectors is documented in appendix 1.

A sparse implementation of Bennetts algorithm is given in algorithm 2.2. To avoid the tedious pointer manipulations inevitably arising from linked list processing, most list operations are described on a higher level. In particular the fill-in operations *fill_in_in_vector* and *fill_in_in_matrix* are efficiently implemented using the C address operator [21]. The exact definition of the operations in algorithm 2.2 is:

— *fill_in_in_vector*(v,i) : create a fill-in in vector v at index i.

— *fill_in_in_matrix*(M,r,c) : create a fill-in in matrix M at position (r, c).

— *index*(v) : retrieve index of element pointed to by v, i.e.

$$index(v) \triangleq \begin{cases} v \rightarrow index \in [1, n] & \text{if } v \neq \varnothing \\ N > n & \text{if } v \equiv \varnothing \end{cases}$$

— *min_index*(v_1, v_2) : retrieve smallest index, i.e.

$$min\_index(v\,1, v\,2) \triangleq min(index(v\,1), index(v\,2)).$$

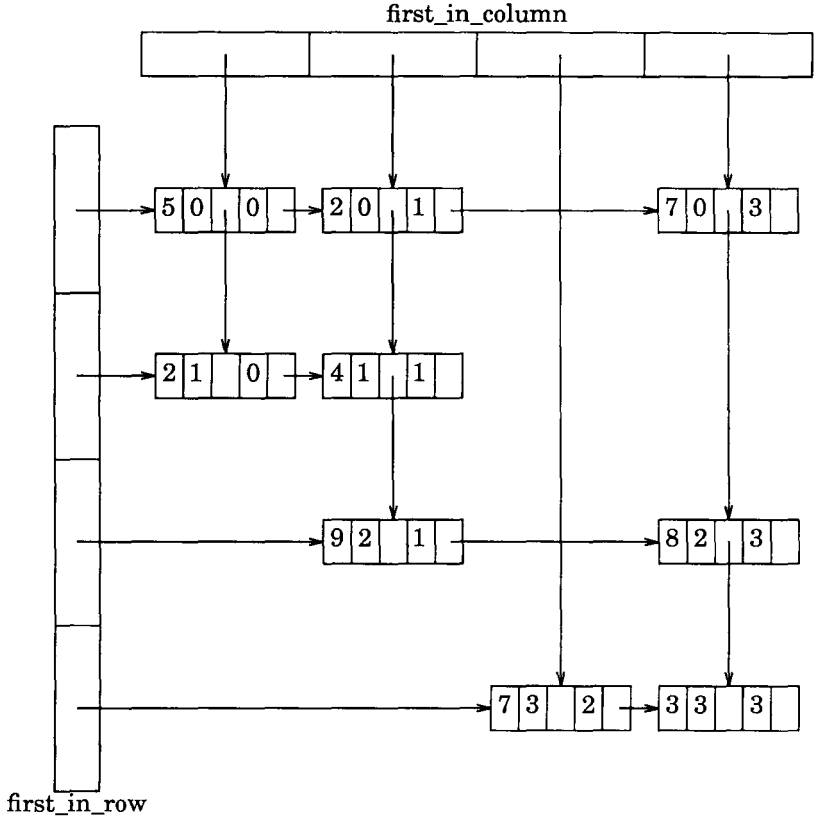


Figure 2.2. *Sparse matrix datastructure. Matrix elements consist of two pointers, a value field and a row and column index. The pointer array D (not shown here) is initialized as soon as the optimal equation ordering is determined. It provides direct access to the diagonal elements.*

Algorithm 2.2. *A sparse version of Bennetts algorithm for the update of the triangular factors.*

```
d = 1;
while c ≠ ∅ and r ≠ ∅ do
        k = min_index(c, r);
        if k ≡ index(c) then ck = c→value; c = c→next else ck = 0 fi;
        if k ≡ index(r) then rk = r→value; r = r→next else rk = 0 fi;
        D[k]→value += ck·d·rk;
        Dk = D[k]→value;
        pk = ck·d / Dk;
        qk = rk·d / Dk;
        d −= pk·Dk·qk;
        /* Update c vector and L–column */
        vec = c;
        el = D[k]→next_row;
        while vec ≠ ∅ or el ≠ ∅ do
                if index(vec) > index(el) and ck≠0 then
                        vec = fill_in_in_vector(c, index(el));
                elseif index(vec) < index(el) and rk≠0 then
                        el = fill_in_in_matrix(L, index(vec), k);
                fi;
                if index(vec) ≡ index(el) then
                        vec→value −= el→value·ck;
                        el→value += qk·vec→value;
                        vec = vec→next;
                        el = el→next_row;
                elseif index(vec) > index(el) then
                        el = el→next_row;
                else
                        vec = vec→next;
                fi;
        od;
        /* Update r vector and U–row */
        vec = r;
        el = D[k]→next_col;
        while vec ≠ ∅ or el ≠ ∅ do
                /* Process r vector and U–row in a way similar */
                /* to the c vector and L–column – omitted */
        od;
od;
```

2.3 Forward-backward substitution

Let us assume the linear system to be solved is given by:

$$Ax = LDUx = b$$
$$Ly = b \text{ and } DUx = y.$$

Given the LDU factors the system is readily solved using forward-backward substitution. In algorithms 2.3 and 2.4 two variants are given. The first conventional one uses already computed values of y and x to obtain the current ones. The second version immediately processes every new value of y and x by updating the source vector. In the case of full matrices both algorithms are equally appropriate. For sparse matrices and, especially, sparse source vectors, we prefer to use the row based version.

Algorithm 2.3. *Column based forward-backward substitution.*

for $i = 1$ **to** n **do**
$$y_i = b_i - \sum_{j=1}^{i-1} L_{ij} \cdot y_j;$$
od;
for $i = n$ **downto** 1 **do**
$$x_i = D_{ii}^{-1} \cdot y_i - \sum_{j=i+1}^{n} U_{ij} \cdot x_j;$$
od;

The advantages for sparse source vectors can be tremendous. Assume for instance $b_i = 0$ for $1 \leq i \leq k \leq n$. It is obvious that the top k entries of y will be zero as well. If source vector b is sparse, in the case of our applications very often only a single element, we can expect the intermediate vector y to be sparse also. This can be exploited during the forward substitution by implementing a sparse version of algorithm 2.4. The computational activity in such an algorithm would be guided by the occurrence of nonzero source vector elements instead of by scanning the entire diagonal. Unfortunately the solution vector x tends to develop many fill-ins during the backward substitution process depending on the type of circuit at hand. This phenomenon will be dealt with in more detail in the next section. In general the backward substitution can best be implemented as a full one choosing either algorithm 2.3 or 2.4.

Algorithm 2.4. *Row based forward-backward substitution.*

```
for i = 1 to n do
        y_i = b_i;
        for j = i + 1 to n do
                b_j = b_j - L_ji·y_i
        od;
od;
for i = n downto 1 do
        x_i = D_ii^-1·y_i;
        for j = i - 1 downto 1 do
                y_j = y_j - U_ij·x_i;
        od;
od;
```

2.4 Sparsity analysis

In the previous section we considered forward-backward substitution. In algorithm 2.5 the sparsity of both the matrix A and the source vector b are exploited. We observe that the algorithm can be very efficient provided the y and x vectors remain sparse. However the number of operations can grow quadratically as soon as this assumption fails. In this section we will attempt to validate the approach in algorithm 2.5 by investigating the density of the solution vector x. We first introduce the directed graph associated with a sparse matrix.

Definition 2.1
The digraph $G(V,E)$ of the $n \times n$ matrix A has vertices 1,2, ..., n and an edge (i,j) from vertex i to vertex j for every off-diagonal matrix entry A_{ij}. An entry is a structural nonzero element in a sparse matrix. Structural nonzero elements are either nonzero elements or nonzero elements that became zero by cancellation.

Let us first consider the impact of irreducibility on the sparsity of solution vector x.

Definition 2.2
An $n \times n$ matrix A is said to be reducible if it can be permuted to a block triangular form:

Algorithm 2.5. *Sparse row based forward-backward substitution.*

```
/* Solve Ly = b */
y = ∅;
while b ≠ ∅ do
        el = D[vec→index]→next_row;
        while el ≠ ∅ do
                new = fill_in_in_vector(b, el→row_index);
                new→value -= el→value·b→value;
                el = el→next_row;
        od;
        /* Generate y in reverse order */
        y = prepend(head(b), y);
od;
/* Solve Ux = y */
x = ∅;
while y ≠ ∅ do
        y→value = y→value / D[y→index]→value;
        el = first_in_column[y→index];
        while el→row_index < y→index do
                /* Create temporary vector in reverse order */
                merge_vec = prepend(new_element(),merge_vec);
                merge_vec→value = -el→value·y→value;
                merge_vec→index = el→row_index;
                el = el→next_row;
        od;
        y = merge(merge_vec, y);
        x = append(head(y), x);
od;
```

$$\begin{bmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ . & & . & \\ A_{N1} & . & . & A_{NN} \end{bmatrix} \tag{2.20}$$

where the diagonal matrices A_{ii} are square and irreducible and $N>1$.

It can be shown that a matrix A is irreducible if and only if, there is a path from i to j in graph G for any two nodes $1 \leq i, j \leq n$ [34].

A well known theorem on the directed graph of the factorization $A = LU$ is [35]:

Theorem 2.1
Suppose an $n \times n$ matrix A is irreducible and nonsingular and has a triangular factorization LU. If there is a so-called legal path $(i, k_1, k_2, k_3, \ldots, k_m, j)$, $k_l < i$ and $k_l < j$ for all $l = 1, 2, \ldots, m$, between nodes i and j in the directed graph of A then there will be an edge from node i to node j in the directed graph of the triangular factors. A path is legal if it is elementary (i.e. each node in the path occurs exactly once) and satisfies the above condition $k_l < i, j$.
Proof. Proof is rather straightforward. Consider the first elimination step concerning the legal path. Before the elimination there exist edges (i, k_1) and (k_1, k_2). The elimination will at least introduce an edge (i, k_2) because of fill-in in the original matrix. So now there is a path $(i, k_2, k_3, \ldots, k_m, j)$. Eventually this strategy yields edge (i, j). Q.e.d.

Assuming the triangular factors have been determined, the following two theorems are useful.

Theorem 2.2
Every column of L except the last one has at least one entry beneath the diagonal.
Proof. Suppose there is a column l of L with no entry below the diagonal. According to definition 3.2, there must exist a path $(k, \ldots, l)$, $k < l$. However this implies the existence of an entry A_{kl} in the modified matrix which causes a contradiction. Q.e.d.

Theorem 2.3
Every row of U except the last one has at least one entry to the right of the diagonal.
Proof. Suppose there is a row u of U with no entry to the right of the diagonal. According to definition 3.2 there must exist a path (u, k) with $k > u$. This would imply an entry A_{uk} to exist which contradicts the assumption. Q.e.d.

We are now ready to inspect the sparsity of the intermediate solution vector y and the solution vector x. First we solve $L \cdot y = b$ assuming that b has at least one entry. In this case it is easy to see that y_n will be structurally nonzero. Suppose b has a single entry b_k. If $k = n$ then y_n is nonzero. If $k < n$ there will be a fill-in b_l, $l > k$ in b due to theorem 3.2. Continuing this procedure will yield the above statement. If both

L and b are sparse, the vector y may very well be sparse too.

During the backward substitution we solve $U \cdot x = y$. At step k we have:

$$U_{kk} \cdot x_k = y_k - \sum_{j=k+1}^{n} U_{kj} \cdot x_j \tag{2.21}$$

Since y_n is structurally nonzero, x_n will be also. From theorem 3.3 we know that $U_{n-1,n} \neq 0$ so x_{n-1} must be structurally nonzero also. Repeating this process leads us to the conclusion that the entire solution vector is structurally nonzero.

Our final conclusion for irreducible matrices is that in general no gain can be expected from a sparse implementation of the backward substitution process, since the solution vector x is structurally nonzero. On the sparsity of the intermediate solution y, however, no such negative conclusion can be drawn. Generally, if the source vector b is sparse we expect y to be sparse also because of the limited fill-in that can occur. In practice this assumption is confirmed. The average amount of fill-in during forward substitution is small. Contrary to the general conclusion for solution vector x, it usually remains quite sparse if source vector b originated from a pwl related operation (see the next chapter; pwl is a shorthand for piecewise linear). For these cases a sparse implementation of the backward substitution is applied.

How can we determine whether a given matrix is irreducible? A simple method is to determine the strongly connected components, e.g. by using Tarjan's algorithm [36]. This algorithm not only constructs the components but also imposes an ordering using the "lowlink" values. By renumbering the nodes using the given ordering, the associated matrix is transformed to a block triangular form. If the matrix is irreducible we will find only a single strong component containing all nodes. In fact the number of strong components can be seen as a measure for the connectivity in the graph and, subsequently, for the circuit the graph has been derived from. A circuit in which most signals have only limited propagation will tend to have lots of strong components resulting in a highly reducible matrix and minimal fill-in during forward-backward substitution. The average analog circuit however will typically have few components because of resistive and capacitive coupling between the components. A typical piecewise linear circuit featuring many high level components will show less coupling.

2.5 Recursive bordered block diagonal matrices

In an attempt to minimize computational effort for matrix operations when dealing with large circuits, several authors have investigated specific matrix forms such as the bordered block diagonal (BBD) matrix in figure 2.3. The basic idea is that subcircuits are associated with submatrices which can be solved almost independently. This way latent circuit behavior might be exploited. A way to generate a single level BBD matrix is a method called node tearing pioneered by Kron [37, 38]. Here we try to create a number of block matrices of reasonable size. The objective is to keep the connecting border as thin as possible. Unfortunately this optimization problem is NP-complete so we have to resort to heuristic algorithms, e.g. [39].

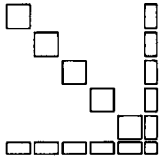


Figure 2.3. *Single level bordered block diagonal matrix.*

The decomposition into subcircuits is known a priori if the simulator is provided with a hierarchical circuit description. Now we can construct a multi level decomposition known as a recursive (or nested) block bordered matrix as depicted in figure 2.4 Such a matrix structure allows among other advantages for selective recomputation of the LU decomposition in case a subsystem is changing. It is also claimed to be efficient and suitable for the application of parallel processing [40]. Only recently an overview of algorithms for recursive BBD matrices was given by Vlach [41] in which he elaborates on an earlier paper [42].

A previous version of the piecewise linear simulator fully maintained the circuit hierarchy as it was defined by the user throughout the entire program. For every level in the hierarchy, matrices relating the child modules to their parent module had to be assembled and kept up to date causing a lot of computational overhead [13]. The BBD matrix structure was introduced in the new simulation program in an attempt to eliminate this drawback [43]. Since the pwl simulator does not

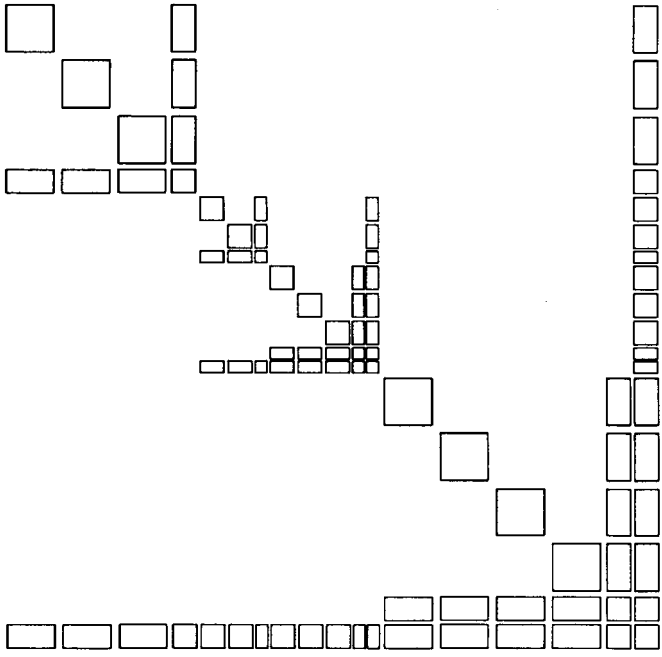


Figure 2.4. *Recursive bordered block diagonal matrix.*

restrict its components to two-terminal devices, the application of nodal analysis is rather complicated compared to conventional simulators such as SPICE. Therefore the system matrix contains nearly all currents. A basic disadvantage in our implementation is the large number of superfluous current variables that were introduced while descending down the circuit hierarchy. Every time a net connects several child modules to the current father module, new current variables and the corresponding current equation have to be generated. This is very inefficient since all intermediate variables have to be solved for no other purpose than to retain the BBD matrix structure.

When sparse matrix techniques applied in the simulator evolved towards sparsity directed computations as described in the previous sections, the BBD structure became obsolete. In fact the structure was

never used explicitly. Although in principle the BBD structure guarantees that large parts of the matrix can be skipped when performing updates due to subsystem activity, the same phenomenon is observed in practice. The dyadic update tends to produce little fill-in in the update vectors and the computation "dies" very fast. Eventually it appeared that circuit latency can be exploited efficiently without casting the system matrix into a special form.

In our view the reason for this result is twofold. The reordering of a matrix to a BBD structure in fact results in a very specific pivot order which may very well be sub-optimal with respect to sparsity considerations. Secondly it can be observed that the processing of updates on a microscopic level is more efficient than the macroscopic treatment of block sub-matrices along a path upwards in the circuit hierarchy. The same statement holds if we consider the LU decomposition process when implemented on a parallel computer with e.g. a multifrontal scheme [44]. Summarizing we can conclude that the BBD matrix structure is appealing since the carefully constructed circuit hierarchy is maintained or retrieved. A significant gain in processing time, however, seems rather unlikely.

3. Solving the pwl equations

3.1 Equation building

At start up time the first action of the piecewise linear simulator is to examine the circuit topology and assemble a set of equations. The circuit description accepted may be a hierarchy but internally it is reduced immediately to a single level, built-up from leafcells and their mutual interconnections. Every leafcell is characterized by a matrix description as defined in equation (1.1). The system matrix is formed by the current linear mapping of every leafcell together with the interconnection equations. In this context input specifications are viewed upon as a special kind of leafcells. Consequently the pwl system is stored in a distributed form thus localizing leafcell data and minimizing memory requirements. A consequence of this approach is the more complicated form of some of the basic solution methods applied in the simulator as will be shown in this chapter. An alternative would be to actually construct the overall pwl system by eliminating internal variables at the cost of larger and denser matrices.

3.2 Initial solution

For the initial solution let us impose initial condition $u = 0$ for all leafcells. The general leafcell description (1.1) then reduces to

$$\begin{bmatrix} 0 \\ p \end{bmatrix} = \begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix} \begin{bmatrix} x \\ q \end{bmatrix} + \begin{bmatrix} a_1 \\ a_3 \end{bmatrix} \tag{3.1}$$

For the moment assume that the description is self contained, i.e. the pwl system consists of a single leafcell only with all inputs specified properly. By further assuming that $q = 0$ the terminal variables x can be rapidly solved from $0 = A_{11} \cdot x + a_1$. With x known we can simply rewrite the system as

$$p = M \cdot q + m, \quad \text{with } p \geq 0,\ q \geq 0,\ p^t \cdot q = 0, \tag{3.2}$$

known as the Linear Complementarity Problem (LCP). An extensive discussion of this problem can be found in e.g. [45], and will not be repeated here. Variables p and q are often referred to as the basic and nonbasic variables respectively. Our task is to find vectors p and q such that the conditions are satisfied. A number of algorithms are available to solve the LCP. Best suited for our application are the pivoting type algorithms like the ones devised by Katzenelson [46] and Lemke [45].

The LCP problem is fully classified by matrix M. Therefore let us consider a few relevant matrix classes [47]. It can be shown that a unique solution for the LCP exists, if and only if M belongs to matrix class P. If M belongs to class P, we can use Katzenelson's algorithm to obtain a solution. An odd number of solutions exists if matrix M belongs to class SSM. The Lemke algorithm can be shown to find a solution for matrices M belonging to matrix class L (SSM is a subset of class L). For this class, termination of the algorithm without finding a solution implies that no solution is feasible. Matrix class L is large enough to model a large variety of circuits, including hysteresis-like behavior, flip-flops, etc.

In our piecewise linear simulator however the piecewise linear problem does not appear in the original form of the LCP. The circuit is composed with a number of components, each having its private pwl description, and their mutual interconnections. In particular the system description $Ax + a = 0$ is assembled using the leafcell jacobians A_{11}, the leafcell source vectors a_{11} and the interconnection equations. Consequently matrix M is not at our disposal. It would have to be computed by eliminating all internal variables. Therefore pivoting must be restricted to the matrices A_{33} of the components. Other entries in matrix M are prohibited to act as pivot. This restriction rules out the algorithms mentioned before because they tend to choose

arbitrary off-diagonal pivots which generally are not available in our datastructure.

An algorithm that meets our requirements was devised by van de Panne [48]. Its functionality is equivalent to that of Lemke but it uses only diagonal or block pivots thus keeping the piecewise linear system complementary. A block pivot is the inversion of a principal submatrix of M. Whereas a regular pivot exchanges exactly one basic variable with one nonbasic variable, application of a block pivot will exchange several basic against nonbasic variables. In fact a diagonal pivot is a block pivot of dimension 1. Note that in our implementation block pivots are always performed as a sequence of single pivots.

Let us give a short review of the algorithm. A solution to the LCP is found directly for $m \geq 0$ in which case we have $q = 0$ and $p = m \geq 0$. If any component of vector m is negative, we add a non-negative vector e and a positive constant λ to system (3.2):

$$p = M \cdot q + m + \lambda \cdot e, \tag{3.3}$$

$$e_k = \begin{cases} 1 & \text{if } p_k < 0 \\ 0 & \text{if } p_k \geq 0 \end{cases}$$

We initialize $\lambda = \underset{k}{\text{MAX}}(-p_k)$ such that $p \geq 0$ and let it decrease to zero, while p is kept non-negative. A solution is found as soon as this is achieved. Assume the decrease of λ is stopped by blocking row k: row p_k becomes zero. To resolve the blocking we now try to pivot with matrix element M_{kk}. Two situations can occur:

1. element M_{kk} is nonzero or
2. element M_{kk} is zero.

In the first case we perform a pivoting operation with matrix element M_{kk}. Again we must distinguish between two cases:

1a pivot $M_{kk} > 0$: after pivoting the blocking is resolved and the decrease of λ can continue.

1b pivot $M_{kk} < 0$: after pivoting the blocking is resolved but now we must increase λ. The algorithm stops without finding a solution if no upper bound can be found for λ.

In the second case the pivot is rejected. Instead we keep λ fixed and

start increasing q_k. A column currently being manipulated is referred to as the active column. The increase stops if some row p_l is blocked. Now q_l becomes the active variable while q_k remains fixed. Again we try to resolve the blocking by pivoting and matrix element M_{ll} is our pivot candidate. If a single pivot is not feasible the algorithm tries to find a block pivot incorporating as many fixed variables as possible. If such a block pivot succeeds the corresponding fixed variables are released and the algorithm proceeds with some previous fixed variable. The direction in which the new current variable is moved can change depending on the sign of the last pivot. Apart from pivoting, a currently active variable is released also if it can be decreased to zero. The algorithm stops without finding a solution if either the increase or the decrease of the active variable is unbounded.

Since the form of the LCP solved by van de Panne is not directly available in our simulator, the algorithm must slightly be adapted. The changes (mainly caused by the fact that our pwl system is available in a distributed form) have been listed by van Eijndhoven [13]:

- the size of a step in the direction of the active variable (λ or a nonbasic variable) may be influenced by more than one piecewise linear component.
- since matrix M is not available we cannot determine the value of the selected pivots directly.

In this implementation all leafcells with a negative component of p are marked improper. Leafcells for which $p \geq 0$ are called proper. Next the van de Panne algorithm is applied sequentially to all improper leafcells, each receiving its private λ and e, meanwhile assuring that proper leafcells remain proper. In the current simulation program, all improper leafcells are solved simultaneously receiving only local e vectors and a single global λ for all improper leafcells.

The problems listed above are solved by utilizing partial derivatives of the terminal variables x and basic variables p. After extension of equations (3.1) with parameter λ and vector (e_1, e_2, e_3), partial differentiation to λ for all improper leafcells leads us to the following relations:

$$\begin{bmatrix} 0 \\ \tilde{p} \end{bmatrix} = \begin{bmatrix} A_{11} \\ A_{31} \end{bmatrix} \cdot \tilde{x} + \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} \tag{3.4}$$

$$\tilde{x} = \frac{\partial x}{\partial \lambda}, \tilde{p} = \frac{\partial p}{\partial \lambda}.$$

Using the concatenation of all e_1 vectors as a source vector, $\tilde{x}$ can rapidly be solved from the system matrix by forward-backward substitution. Now, $\tilde{p}$ is readily determined:

- $\tilde{p} = A_{31} \cdot \tilde{x} + e_3$ for all improper leafcells and
- $\tilde{p} = A_{31} \cdot \tilde{x}$ for all proper leafcells.

The maximum step θ that can be made in the current direction of λ can easily be computed from the current values of p and $\tilde{p}$ for all leafcells related to a nonzero component in $\tilde{x}$ and the condition that $p + \theta \cdot \tilde{p} \geq 0$. In case a nonbasic variable q_k of some leafcell is active instead of λ, the above equations change to:

$$\begin{bmatrix} 0 \\ \tilde{p} \end{bmatrix} = \begin{bmatrix} A_{11} \\ A_{31} \end{bmatrix} \cdot \tilde{x} + \begin{bmatrix} A_{13}^{*k} \\ A_{33}^{*k} \end{bmatrix} \tag{3.5}$$

$$\tilde{x} = \frac{\partial x}{\partial q_k}, \tilde{p} = \frac{\partial p}{\partial q_k}.$$

After $\tilde{x}$ has been computed we find for $\tilde{p}$:

- $\tilde{p} = A_{31} \cdot \tilde{x} + A_{33}^{*k}$ for the active leafcell and
- $\tilde{p} = A_{31} \cdot \tilde{x}$ for all other leafcells.

From the last equations we can easily solve our second problem: the determination of the sign of for instance pivot $(p_k, q_k) = A_{33}^{kk}$. The requested sign can be derived directly from the sign of $\tilde{p}_k$.

At this point the initial solution algorithm can be presented in some more detail. Before doing so let us define some useful symbols:

L	set of all leafcells.
I	set of improper leafcells: $I = \{ l \in L \mid \exists_k [p_k^l < 0] \}$.
p^l	vector of basic variables related to leafcell $l \in L$

x^l set of terminal variables related to leafcell $l \in L$.

$\{\tilde{x}\}$ set of nonzero entries in $\tilde{x}$.

$L_{\tilde{x}}$ set of leafcells related to nonzero entries in $\tilde{x}$: $L_{\tilde{x}} = \{ l \in L \mid x^l \cap \{\tilde{x}\} \neq \emptyset \}$.

The actual algorithm is split into two parts. The first part actually serves as an initialization for the van de Panne algorithm which forms the second part. The error vector e is added as an extra column to the leafcell matrix to simplify the implementation. Parameter λ becomes part of the p vector but in fact is a global variable, generally represented in several p vectors

Before listing the implementation of the modified van de Panne algorithm we define some key variables:

tos index indicating the top of the van de Panne stack.

stack stack used by van de Panne to store blocked rows. A stack element is a 3-tuple (leafcell, active column, direction). The current active column is on top of the stack. All other columns (and λ) on the stack are kept fixed. Columns can be deleted either by pivoting or because the corresponding column variable became 0. It is also possible for fixed columns to become active again. The blocked rows (or manipulated columns)are stored in stack[1..tos]. Position 0 is reserved for λ. Legal stack operations are *push* and *pop*.

leaf the leafcell currently being manipulated; active leafcell (stack[tos].leaf).

row the current blocking row.

column the current active column (stack[tos].column).

dir current direction of the active column (stack[tos].dir). Legal values are +1 (up) and –1 (down).

θ feasible step in the direction of the active variable.

Algorithm 3.1. *Initial solution: startup phase.*

Decompose matrix A into A = LDU; / Note: L is a matrix here */*
Solve Ax = b;
$I = \emptyset$;
for all $l \in L$ **do**
    initialize $u^l = q^l = 0$;
    compute $\dot{u}^l$;
    compute p^l;
    if *any component* $p_k^l < 0$ **then** $I = I \cup l$ **fi**;
od;
determine leafcell i and *row r such that*: $\forall_{i \in I,\, l \in I}\, [\, p_r^i \leq p_*^l \,]$;
$\lambda = -p_r^i$;
for all $i \in I$ **do** $p^i = p^i + \lambda \cdot e$ **od**;
push (*i*, "λ *column*", *down*);
push (*i*, *r*, *up*);
vdpanne (λ);

Implementing the solution process as described in algorithm 3.2 two potential problems can be expected. First, assuming van de Panne's algorithm has selected a pivot A_{33}^{kl} for a specific leafcell because $\delta p_k \,/\, \delta q_l \neq 0$ while the actual value in the leafcell equals zero, the pivot operation appears to be infeasible. Fortunately this problem can always be solved as is shown in the next theorem.

Theorem 3.1
Suppose a pivot selected by van de Panne's algorithm in our piecewise linear simulation program appears to be zero in the local leafcell matrix A_{33}. If the pivot element is A_{33}^{kl} we can always select a row m from equations $0 = A_{11}x + A_{13}q + a_1$ with nonzero element A_{13}^{ml} and add it to row p_k.

Proof. Assume a set of N pwl components each defined by a matrix (3.1). Further assume the existence of a set of interconnection equations and an adequate set of input specifications. Let the overall pwl system be defined by

$$\begin{bmatrix} 0 \\ \hat{p} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{13} \\ \hat{A}_{31} & \hat{A}_{33} \end{bmatrix} \begin{bmatrix} \hat{x} \\ \hat{q} \end{bmatrix} + \begin{bmatrix} \hat{a}_1 \\ \hat{a}_3 \end{bmatrix} \tag{3.6}$$

where

Algorithm 3.2. *Initial solution: van de Panne.*

```
vdpanne(λ):
    carry_on = true;
    while carry_on do
        /* Compute derivatives of x */
        generate source vector b;
        solve LDU · x̃ = b;
        assemble L_x̃;

        /* Determine maximal number of pivots */
        Si = 0;
        for i = tos downto 1 do
            s = sign(stack[i].leaf, stack[i].column);
            if s ≠ 0 then
                S = s;
                Si = i;
            fi;
        od;
        if Si = 0 then /* Cannot pivot */
            determine_theta();
            if θ ≡ ∞ then abort; /* Cannot solve */
            if θ > 0 then
                x = x + dir·θ·x̃;
                for all l∈L_x̃ do update u̇^l and p^l od;
                update λ or q^leaf[column];
            fi;
            if column ≡ "λ column" and λ ≡ 0 then
                carry_on = false /* Solution found */
            elseif column ≠ λ and q[column] ≡ 0 then
                /* Current active column became 0 */
                pop();
                stack[tos].dir = − stack[tos].dir;
            else /* New blocking row */
                push(leaf, row, up)
            fi;
        else
            perform_pivots();
            for i = tos downto Si do pop() od;
            dir = dir·S;
        fi;
    od;
```

sign (l, r):
determine $\tilde{p}_r$;
now set the sign of $A_{33}[r,column]$ *to*:

$$sign = \begin{cases} -1 & \text{if } \tilde{p}_r < 0 \\ 0 & \text{if } \tilde{p}_r = 0 \\ 1 & \text{if } \tilde{p}_r > 0 \end{cases}$$

determine_theta ():
for all $l \in L_{\tilde{x}}$ **do**
determine $\tilde{p}$;
find θ^l *such that* $\theta^l = \underset{k}{\text{MAX}}(\theta_k^l)$ *where* $p_k + dir \cdot \theta_k \cdot \tilde{p} \geq 0$;
od;
now set $\theta = \underset{l \in L_{\tilde{x}}}{\text{MIN}}(\theta^l)$;
set leaf to the corresponding leafcell;
set row to the corresponding row of leaf;

perform_pivots ():
pivot (*stack* [*Si*].*column*, *column*);
for i = *Si* **to** *tos* −1 **do**
pivot (*stack* [i +1].*column*, *stack* [i].*column*);
od;

- $\hat{x} = (x^1, x^2, \cdots, x^N, x^i)$,
- $\hat{p} = (p^1, p^2, \cdots, p^N)$,
- $\hat{q} = (q^1, q^2, \cdots, q^N)$,
- $\hat{a}_1 = (a_1^1, a_1^2, \cdots, a_1^N, a^i)$ and
- $\hat{a}_3 = (a_3^1, a_3^2, \cdots, a_3^N)$

in which superscripts are used to identify the "donating" leafcells. The extra variables x^i denote the additional internal connection variables. Further assume that all input and interconnection relations are merged together with the leafcell A_{11} matrices to yield a square matrix $\hat{A}_{11}$. Notice that since both $\hat{A}_{11}$ and $\hat{A}_{33}$ are square, matrix $\hat{A}$ formed by matrices $\hat{A}_{ij}$, $i, j \in [1, 3]$ is square also. For $N = 2$, matrix $\hat{A}$ is depicted in figure 3.1

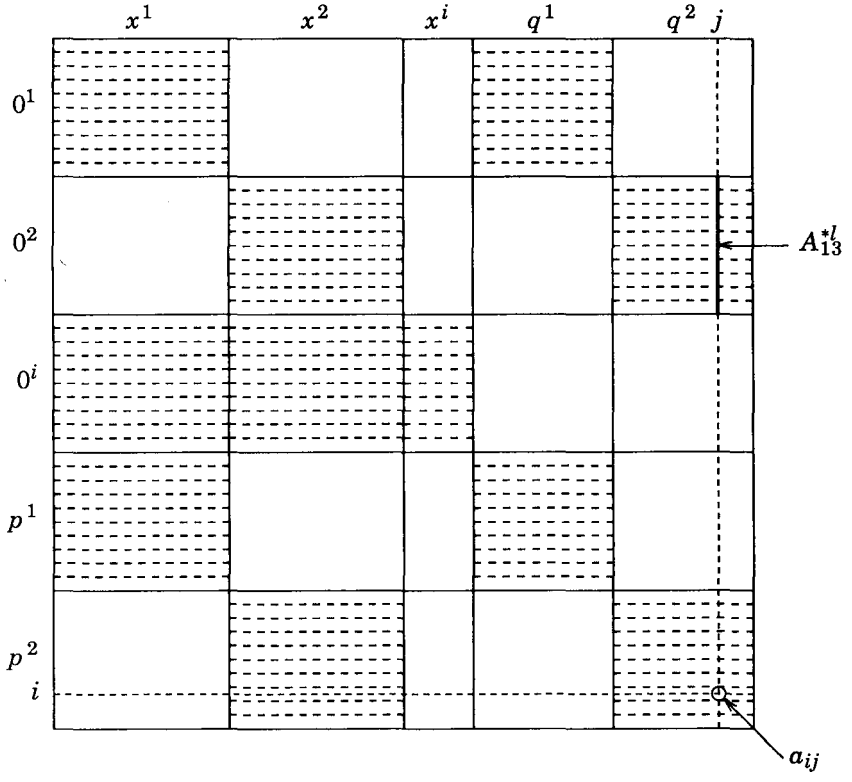


Figure 3.1. *Matrix $\hat{A}$ for $N = 2$. The original leafcell matrices are indicated by the dashed areas. Non dashed areas are supposed to contain zeros only. The additional interconnection equations are represented by 0^i.*

Now define a bipartite graph $G(V, E)$ as follows:

- the set of nodes formed by e_i, $i \in [1,n]$ for every row in $\hat{A}$ numbered in ascending order.
- the set of nodes formed by c_j, $j \in [1,n]$ for every column in $\hat{A}$ numbered in ascending order.

- the set of nodes $V = \{e_i, c_j\}$, $i,j \in [1,n]$.
- for every nonzero entry $\hat{a}_{ij}$ in matrix $\hat{A}$ there is an edge (e_i, c_j) between node e_i and node c_j in the graph.

Elimination of the internal variable in column c_k with row e_l will transform graph G to $\hat{G}$:

- nodes c_k and e_l will be deleted from G as well as all edges connected to either of them.
- add an edge between every equation node e_i $(i \neq l)$ and column node c_j corresponding with a matrix fill-in $\hat{a}_{ij}$ for which there was a path (c_j, e_l, c_k, e_i) in graph G.

Clearly, unless there is a path between e_i and c_j, no fill-in can occur at position $\hat{a}_{ij}$. Elimination of all internal variables will yield a sequence of graphs G, $\hat{G}^1$, $\hat{G}^2$, $\cdots$. Although it isn't explicitly constructed, van de Panne operates on a system equivalent to the one represented by the final graph.

Now consider the case in which a specific leafcell contributes a zero matrix element A_{33}^{kl} to $\hat{A}$ in say $\hat{a}_{ij}$. Furthermore assume column A_{33}^{*l} of that leafcell is identical to zero. This implies that for all paths (e, c_j), e corresponds to an equation resident in matrices $\hat{A}_{3*}$. We are certain that this equation will never be used to eliminate an internal variable. Eliminating all internal variables by linear matrix operations, the only way for entry $\hat{a}_{ij}$ to become nonzero is if there is a path from j to i containing at least one e node that can be used in an elimination, while we just concluded that such a path cannot exist. So if the entry is created, our assumption is falsified and a path from j to i does exist. This means that the row mentioned in the theorem can always be found. Q.e.d.

The second problem is also related to our hierarchical datastructure. As we can see from the implementation of function *perform_pivots*, van de Panne's advice about the (block) pivot to be performed is followed closely. No problems occur as long as a block pivot is located within a single leafcell. If however a block pivot is scattered over more than one leafcell, the operation is infeasible. In this case we have to resort to diagonal pivots only.

Theorem 3.2
A block pivot found by van de Panne's algorithm for the Linear Complementarity Problem as it arises in our piecewise linear simulation program can always be performed by a set of diagonal pivots.
Proof. Suppose van de Panne finds a scattered block pivot consisting of $\hat{A}_{33}$ entries $\hat{a}_{i_2 j_1}$ and $\hat{a}_{i_1 j_2}$. These entries were equal to zero in the initial $\hat{A}$ matrix. As we concluded in the previous theorem, these entries can only become nonzero if the A_{33}^{*l} columns of the corresponding leafcells contain at least one nonzero. In this case it is obvious that there always exists a diagonal block pivot that can replace the original one. Q.e.d.

3.3 DC solution

In conventional circuit simulation programs such as SPICE [4] the most elementary action is the determination of the quiescent state of the circuit. Capacitors and inductors are simply replaced by open circuits or short cuts. Nonlinear components are handled by applying the Newton-Raphson technique. It is this technique that often causes problems because of its limited local convergence properties. To cope with these problems the technique has been modified in a number of ways some of which are listed in [34, 49]. Recently an improvement was achieved by applying a new type of vector norm in which certain equations receive higher priority than others in an attempt to guide the damping of the Newton updates in a more appropriate way than is done by the regular L_2 norm [50].

An important advantage of the application of piecewise linear techniques is the improved convergence compared to Newton-Raphson based methods. Due to the piecewise linear approximation of all nonlinear functions a solution, if feasible, can always be determined thanks to powerful algorithms yielding global instead of local convergence. The pwl problem to solve is given by:

$$\begin{bmatrix} 0 \\ \dot{u} \\ p \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ u \\ q \end{bmatrix} + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

where $\dot{u} = \dfrac{\partial u}{\partial t}$,

in which we attempt to force $\dot{u}$ to zero. The simplest solution we can think of is to add the state vector u to the unknown circuit variables and use the relations for $\dot{u}$ to retain a well determined system:

$$\begin{bmatrix} 0 \\ p \end{bmatrix} = \begin{bmatrix} A'_{11} & A'_{13} \\ A'_{31} & A_{33} \end{bmatrix} \begin{bmatrix} x' \\ q \end{bmatrix} + \begin{bmatrix} a'_1 \\ a_3 \end{bmatrix}$$

with

$$A'_{11} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, A'_{31} = \begin{bmatrix} A_{31} & A_{32} \end{bmatrix},$$

$$A'_{13} = \begin{bmatrix} A_{13} \\ A_{23} \end{bmatrix} \text{ and } x' = \begin{bmatrix} x \\ u \end{bmatrix}, a'_1 = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

Unfortunately this method often fails. For various models both the matrices A_{21} and A_{22} equal zero thus yielding a singular A'_{11}. Furthermore the addition of a number of unknowns and equations to the system matrix solely for the dc analysis does not seem the most elegant solution.

Another approach much more in the spirit of the simulator would be to exchange the position of u and $\dot{u}$ by pivoting on the A_{22} matrix before starting the initial solution process. This way we would automatically obtain the desired dc solution since the u (and in the new situation the $\dot{u}$) vector is explicitly assumed zero. Afterwards we can retrieve the original set of equations by pivoting on the A_{22} matrix one more time. Although this method works well for a number of components like capacitors, inductors, etc., a new problem arises if matrix A_{22} does not have full rank. In that case the following strategy is followed:

- assuming matrix row A_{21}^{k*} is nonzero, try to find an element $A_{12}^{*k} \neq 0$ and add the corresponding equation to $\dot{u}_k$. If no such entry exists or if A_{21}^{k*} equals zero

- try to find a block pivot $\begin{bmatrix} A_{22}^{kk} & A_{23}^{kl} \\ A_{32}^{lk} & A_{33}^{ll} \end{bmatrix}$ or if required a more dimensional block pivot using van de Panne-like techniques (see figure 3.2).

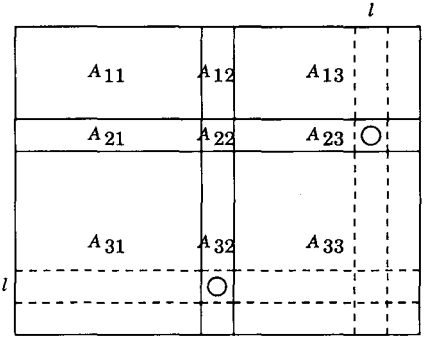


Figure 3.2. *Block pivot for dc analysis in case* $A_{21} \equiv A_{22} \equiv 0$. *In this example matrix* A_{22} *has dimension* 1.

The latter action looks somewhat tricky: after the correct segment of the pwl mapping has been determined, the block pivot not only swaps u and $\dot{u}$ but also puts the leafcell into another (incorrect) segment rather than in the one just found causing a rather undesired side effect. The restoring pivoting operation must be selected with care. Selection of the wrong block pivot might cause $\dot{u}$ to remain nonzero after the pivoting, due to nonzero values in the source vector. Another argument against this type of pivoting is that we may pivot to a segment of the mapping that is never entered during normal operation.

A very robust method guaranteed to find a solution is the application of transient analysis. With all stimuli constant and no oscillations assumed to occur, a dc solution can be found for $t \to \infty$. The method has been applied successfully in conventional simulation [51]. The major disadvantage of this approach is the computational effort that is required. All analog components have to be integrated numerically until variations of the circuit variables remain below some error criterion while application of one of the former methods yields a direct solution.

4. Multirate integration techniques

4.1 Numerical integration

Unfortunately the general circuit simulation problem is too complicated to allow for a closed form analytical solution. Instead we must resort to numerical methods such as the Newton-Raphson iteration mentioned in the introduction and numerical integration. Assuming we have determined a solution for the piecewise linear equations, the resulting problem is now reduced to a set of relatively simple linear differential equations: $\dot{u} = Au + a$.

Contrary to the conventional circuit simulation problem, a direct solution method should not be ruled out immediately. A related method called the approximate exponential function (AEF) [52] method has been proven quite successful for the simulation of MOS VLSI circuits. Generally a set of homogeneous differential equations $\dot{u} = Au$ can be solved be searching solutions of the form $u(t) = e^{\lambda t}v$, in which λ is an eigenvalue of matrix A and v the corresponding eigenvector. For sufficient independent eigenvectors and eigenvalues we can construct a general solution. Disadvantage of this method, supposed we are able to find a solution, are the need to compute (at least once but probably for every segment) all the eigenvalues of the generally very large system and the corresponding eigenvectors. The determination of all the eigenvalues of a linear system is a comprehensive operation. Furthermore the direct determination of exponential curves may be very sensitive to error propagation. Our conclusion is that the direct method appears unattractive and the application of numerical

integration seems more appropriate.

The integration scheme as it is implemented in PLATO is flexible and easily extendible. At the moment four integration formulas are available, see table 4.1.

TABLE 4.1. *Integration formulas for fixed stepsize.*

method	formula	lte
BDF1	$u_{n+1} - u_n - h\dot{u}_{n+1} = 0$	$-\frac{1}{2}h^2u^{(2)}(\xi)$
TR	$u_{n+1} - u_n - \frac{1}{2}h(\dot{u}_n + \dot{u}_{n+1}) = 0$	$-\frac{1}{12}h^3u^{(3)}(\xi)$
BDF2	$u_{n+1} - \frac{1}{3}(4u_n - u_{n-1} + 2h\dot{u}_{n+1}) = 0$	$-\frac{2}{9}h^3u^{(3)}(\xi)$
ACT2	$\frac{1}{6}(5u_{n+1} - 4u_n - u_{n-1})$ $-\frac{h}{9}(5\dot{u}_{n+1} + 2\dot{u}_n + 2\dot{u}_{n-1}) = 0$	$-\frac{2}{9}h^3u^{(3)}(\xi)$

Since the problems to be solved are usually stiff, only A-stable methods are suited for use in a circuit simulation program. As a default PLATO uses the backward Euler (BDF1) formula. This is an easy to implement one-step integration formula. Two unpleasant features are its low order (one) and the very high damping properties (see the results for the Landman circuit in chapter 6). Note that damping is usually a desirable property, but unpractical when dealing with oscillatory problems. An overview of the damping and time error qualities of all methods discussed here can be found in [53]. Another simple but robust one-step method is the trapezoidal rule (TR) which has order two. Its damping qualities are much better compared as to the backward Euler method (again see chapter 6). Unfortunately the decay of round off errors in the presence of large eigenvalues is slow [54]. In a variable stepsize method this oscillatory behavior prevents a further increase of the time step.

Two more complicated families of formulas are the backward differentiation methods [55,56] and a set of methods showing a local property called A-contractivity [53] which for convenience will be referred to as ACT methods. PLATO only uses the two-step second order A-stable formulas (BDF2 and ACT2). In fact there are several ACT2 methods. The one chosen is regarded as an optimal trade-off with respect to damping properties and phase errors [57].

Disadvantage of both formulas is the need to administer more than one previous value for u as well as $\dot{u}$. Another disadvantage is related to the variable stepsize implementations of the two-step methods. In this case the coefficients depend on the current and previous time steps and have to be recomputed for every step change. As a last remark we have to note that the phase errors introduced by BDF2 can be large. The best method regarding damping properties as well as time errors is the ACT2 method.

In most practical applications numerical integration is applied with automatically controlled variable time steps. This extension has no influence on the stability properties of the one-step integration methods. Two-step methods however can become less stable. In particular the two-step BDF method can become unstable for specific problems with increasing steps [58]. The problem does not occur with decreasing steps but in that case the method often suffers from too much damping. A much better perspective is offered by the variable step ACT methods. Two-step second order ACT methods can be constructed that are stable for any step sequence [57]. Formulas for the variable step methods can be found in for instance [54, 57, 59].

4.2 Multirate integration methods

An integration method for a set of ordinary differential equations is called multirate if different (subsets of) equations are integrated with different time steps. The potentials of such a method are obvious. The computation time required can be minimized by integrating slowly varying subsets of equations with a large time step and fast varying subsets with a smaller one.

Only few applications of multirate formulas appear in the literature. Most important example is the waveform relaxation method [8] in which the sub-circuits are integrated independently with promising results for a specific class of circuits. Another entirely different application of multirate integration was implemented in a SPICE-like circuit simulator called SAMSON [60, 61]. Again the electronic circuit is decomposed into a number of sub-circuits, each receiving its private time step. Subcircuits are called alert or dormant depending on whether they are active or not. Computation time is saved by avoiding the discretization and linearization steps for dormant models. Instead

the dormant sub-circuits are approximated by extrapolation. This extrapolation can be done very efficiently by using the prediction-based differentiation formulas as derived by van Bokhoven [62]. Apart from the previous two, only a few other applications have been reported, e.g. [63-65].

Even less authors have attempted to analyze the properties of multirate integration methods. A first rather optimistic paper was published by C.W. Gear in 1980 [66]. A more elaborated followup article appeared in 1984 [67]. Only recently topics like stiff stability of multirate methods were investigated and reported on by S. Skelboe [68-70]. The main conclusion from these papers is that in general stability cannot be guaranteed, even if implicit A-stable methods of first and second order are applied. In order to be successful, implementations must carefully monitor the integration and in case of instability take appropriate action.

An analysis of multirate methods is generally restricted to a set of ordinary differential equations, separated in two subsystems a fast one (equation (4.1)) and a slow one (equation (4.2)):

$$\dot{y} = f(t,y,z), \quad y(t_a) = y_a \tag{4.1}$$
$$\dot{z} = g(t,y,z), \quad z(t_a) = z_a \tag{4.2}$$

with $t \in [t_a, t_b]$, $f : \mathbb{R}\times\mathbb{R}^N\times\mathbb{R}^M \rightarrow \mathbb{R}^N$, $g : \mathbb{R}\times\mathbb{R}^N\times\mathbb{R}^M \rightarrow \mathbb{R}^M$ and f as well as g Lipschitz continuous in y and z. The idea is to integrate the fast subsystem with q time steps from t_{n-q} to t_n for every time step $t_n - t_{n-q}$ of the slow subsystem. Such a step is referred to as a compound step. Several strategies for performing a compound step have been investigated:

- fastest first algorithm: first integrate the fast system (4.1) from t_{n-q} to t_n using q steps. Next the slow system (4.2) is integrated using a single time step.
- slowest first algorithm: first integrate the slow system using extrapolation to estimate the value of y at time t_n. Then integrate the fast system.

Another variation is related to the final step of a compound step. This can be semi-implicit (evaluate fast and slow systems separately) or implicit (evaluate together). The stability properties of multirate formulas based on the backward differentiation methods were

investigated by Skelboe using the 2×2 test problem:

$$\begin{bmatrix} \dot{y} \\ \dot{z} \end{bmatrix} = A \begin{bmatrix} y \\ z \end{bmatrix} \text{ where } A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}. \tag{4.3}$$

For this problem absolute stability is guaranteed for BDF1 and BDF2, for $h > 0$ and $q > 0$ if:

1. A is real and the eigenvalues of A are in the left-hand half-plane and
2. $a_{11} \leq 0$.

In practice these conditions are often satisfied, e.g. when dealing with Resistor-and-Grounded-Capacitor (RGC) networks [9]. For $a_{11} > 0$ it was possible to construct examples showing instabilities. Note that these results were based on a fixed time step. The integration scheme implemented in PLATO is the fastest first algorithm based on variable time steps resulting in an event driven simulation scheme. A principal problem occurs if the stepsize chosen for a specific leafcell appears too large and has to be decreased. Leafcells with smaller stepsizes have already been integrated up to the current time point. To prevent rejection of numerous integration steps and the cost of large back-ups, time steps are continuously monitored and adjusted. Furthermore time step selection is rather conservative and based on a second order error criterion, even for second order integration formulas.

As with many numerical methods, theoretical analysis tends to be pessimistic but in practice results are often quite satisfying. Very often physical properties will satisfy the conditions under which methods are reliable. In particular the integration results obtained by PLATO appear quite accurate. The same conclusion was made in [61] where results were compared with the eternal simulation reference SPICE. Finally we should realize that all numerical integration methods are limited with respect to their application area. In this context the results for several methods applied to the Landman circuit in chapter 6 are quite illustrative.

5. Transient analysis

5.1 Introduction

Transient analysis is probably the most frequently used simulation task in circuit simulation. Unfortunately, transient analysis is also the most cpu time consuming activity in the field. The most straightforward way to implement transient analysis in a piecewise linear simulator is to apply an integration formula with fixed uniform stepsize. In this context a uniform stepsize by definition implies that all leafcells obtain the same time step, whereas fixed implies that the time step for any leafcell remains constant during the integration process. (In literature uniform often means fixed but here we distinguish between those two notions). After having determined an initial solution, replace all dynamical elements by companion models using a stiffly stable integration formula. Next, make a time step equivalent to the stepsize chosen and check whether the pwl equations remain valid. If the solution curve has crossed a segment boundary, the piecewise linear equations have to be re-solved, e.g. using Katzenelson's algorithm. Otherwise proceed with the next time step.

A number of objections can be formulated against this approach:

1. the user has to provide the simulator with an appropriate stepsize which is not always a trivial problem.
2. applying a fixed stepsize can be extremely inefficient. We cannot take advantage of the fluctuating activity in the circuit. The stepsize chosen has to be small enough to integrate the fastest

deviations accurately and is not allowed to grow if circuit activity diminishes.

3. we cannot estimate the truncation error if a boundary hyperplane is crossed. After a segment boundary has been crossed, the pwl equations will have to be solved again. This will generally cause few problems since the available algorithms show excellent convergence properties. Afterwards however it is impossible to estimate the error introduced. This error may be severe in case of hysteresis-like behavior (as is introduced in almost any realistic logic circuit) or step functions. So the stepsize must be relatively small compared to the circuit dynamics.

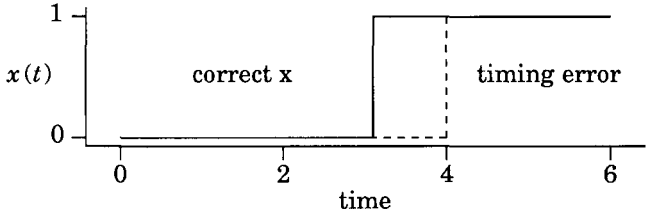


Figure 5.1. *Step function.*

4. discontinuous signals are prohibited. Applying a step function to a circuit in which e.g. all capacitors are replaced by companion models may cause capacitor voltages to change discontinuously.

Of course, the latter two objections become less severe with smaller stepsizes at the cost of decreased efficiency. PLATO can be forced to use a fixed stepsize for benchmark purposes.

The method can be improved by applying a uniform variable step integration rather than a fixed step. Decreasing circuit activity will then yield larger stepsizes. Although the repeated computation of suitable stepsizes causes some overhead, mainly in recomputing the LU decomposition of the system matrix, the efficiency gains will outnumber the cost in most practical cases. Furthermore the overhead can be reduced by applying a conservative strategy with respect to stepsize changes, e.g. only significant changes are effectuated. Secondly, we can improve upon the accuracy of the algorithm because the need to integrate across segment boundaries is eliminated. Stepsizes can be chosen such that the next time point coincides with the crossing of a

boundary.

5.2 A uniform stepsize simulator

The simulation program solves for the circuit variable derivatives $\dot{x}$ rather than the circuit variables themselves. The time derivatives are used to explicitly determine the moment when the solution reaches a segment boundary. Therefore we introduce the divided differences $\bar{x}_{n+1}, \bar{u}_{n+1}, \bar{p}_{n+1}, \bar{q}_{n+1}$ and $\bar{\dot{u}}_{n+1}$:

$$\begin{aligned} \bar{x}_{n+1} &\triangleq \frac{x_{n+1}-x_n}{h_n} \qquad \bar{u}_{n+1} \triangleq \frac{u_{n+1}-u_n}{h_n} \qquad \bar{\dot{u}}_{n+1} \triangleq \frac{\dot{u}_{n+1}-\dot{u}_n}{h_n} \\ \bar{p}_{n+1} &\triangleq \frac{p_{n+1}-p_n}{h_n} \qquad \bar{q}_{n+1} \triangleq \frac{q_{n+1}-q_n}{h_n} \end{aligned} \tag{5.1}$$

Subtracting (1.1) for time points $t = t_{n+1}$ and $t = t_n$ where $h_n = t_{n+1} - t_n$ yields:

$$\begin{bmatrix} 0 \\ \bar{\dot{u}}_{n+1} \\ \bar{p}_{n+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \bar{x}_{n+1} \\ \bar{u}_{n+1} \\ \bar{q}_{n+1} \end{bmatrix} \tag{5.2}$$

in which the constant source vector $(a_1, a_2, a_3)^t$ has disappeared. In the sequel the subscripts $n+1$ are often omitted. We now eliminate the unknown $\bar{\dot{u}}_{n+1}$ by substituting a linear multistep integration formula with constant stepsize:

$$0 = \sum_{i=-1}^{i=p} \alpha_i u_{n-i} + h \sum_{i=-1}^{i=p} \beta_i \dot{u}_{n-i} \tag{5.3}$$

This yields the following equation for $\bar{u}_{n+1}$:

$$\bar{u}_{n+1} = -h\frac{\beta_{-1}}{\alpha_{-1}} D^{-1} A_{21} \bar{x}_{n+1} - h\frac{\beta_{-1}}{\alpha_{-1}} D^{-1} A_{23} \bar{q}_{n+1} - \frac{\beta_{-1}}{\alpha_{-1}} D^{-1} \dot{u}_n \tag{5.4}$$

$$-\frac{1}{h\alpha_{-1}} D^{-1} \left\{ (\alpha_0 + \alpha_1) u_n + \sum_{i=1}^{i=p} \alpha_i u_{n-i} + h \sum_{i=0}^{i=p} \beta_i \dot{u}_{n-i} \right\} ,$$

where

$$\delta = -\frac{\beta_{-1}}{\alpha_{-1}}h \tag{5.5}$$

and

$$D = I - \delta A_{22}\,. \tag{5.6}$$

Combining this result with $0 = A_{11}\bar{x} + A_{12}\bar{u} + A_{13}\bar{q}$ yields:

$$0 = J_{11}\cdot\bar{x} + \bar{b}_1 \tag{5.7}$$

in which

$$J_{11} = A_{11} + \delta A_{12}D^{-1}A_{21} \tag{5.8}$$

and

$$\bar{b}_1 = -\frac{(\beta_{-1} + \beta_0)}{\alpha_{-1}}A_{12}D^{-1}\dot{u}_n - \frac{(\alpha_{-1} + \alpha_0)}{h\alpha_{-1}}A_{12}D^{-1}u_n \tag{5.9}$$

$$-\frac{1}{h\alpha_{-1}}A_{12}D^{-1}\sum_{i=1}^{i=p}(\alpha_i u_{n-i} + h\beta_i\dot{u}_{n-i})$$

from which $\bar{x}$ is rapidly solved. Here we assume that J_{11} is a square matrix representing the entire system. In reality J_{11} corresponds to a single leafcell only. It has to be inserted in the system matrix from which $\bar{x}$ is solved. With $\bar{x}$ known, $\bar{u}$, $\dot{\bar{u}}$ and $\bar{p}$ can be solved successively. Equation (5.7) will be referred to as the leafcells companion model. Table 5.1 shows the specific values for δ and $\bar{b}_1$ for all integration formulas applied in the simulator. Of course the above results are valid under the assumption that he inversion of D is feasible.

TABLE 5.1. *δ and $\bar{b}$ for several integration rules.*

	δ	$\bar{b}_1$
BE	h	$A_{12}D^{-1}\dot{u}_n$
TR	$\frac{1}{2}h$	$A_{12}D^{-1}\dot{u}_n$
BDF2	$\frac{2}{3}h$	$\frac{2}{3}A_{12}D^{-1}\dot{u}_n + \frac{1}{3}A_{12}D^{-1}\bar{u}_n$
ACT2	$\frac{2}{3}h$	$\frac{14}{15}A_{12}D^{-1}\dot{u}_n - \frac{1}{5}A_{12}D^{-1}\bar{u}_n + \frac{4}{15}A_{12}D^{-1}\dot{u}_{n-1}$

Let us for the moment restrict ourselves to one step integration methods like Backward Euler (BE) and the Trapezoidal rule (TR). The implications of multistep (in particular two-step) methods with variable stepsizes will be investigated later on. The general system

description (5.2) with the substitution of an integration formula now becomes:

$$\begin{bmatrix} 0 \\ \bar{p} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{q} \end{bmatrix} + \begin{bmatrix} \bar{b}_1 \\ \bar{b}_2 \end{bmatrix} \tag{5.10}$$

$$= \begin{bmatrix} A_{11} + \delta A_{12} D^{-1} A_{21} & A_{13} + \delta A_{12} D^{-1} A_{23} \\ A_{31} + \delta A_{32} D^{-1} A_{21} & A_{33} + \delta A_{32} D^{-1} A_{23} \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{q} \end{bmatrix} + \begin{bmatrix} A_{12} D^{-1} \dot{u}_n \\ A_{32} D^{-1} \dot{u}_n \end{bmatrix}$$

Observe that the update $\delta A_{12} D^{-1} A_{21}$ on matrix A_{11} is a rank r update where r is the rank of matrix A_{22}. In many cases matrix A_{22} will have dimension one in which case a change of stepsize requires only a simple rank one update on the system matrix. For simplicity, we will assume that r equals one since the operation principle of the algorithms is independent of r.

At time $t = 0$ we have determined values for x_0, $\dot{u}_0$ and p_0 with $q_0 = 0$ and $u_0 = 0$ as initial values. The integration is then started with a Forward Euler integration step: the initial $\bar{x}$ is determined using the source vector $\bar{b} = A_{12} \dot{u}_0$ after which we can derive a suitable stepsize from $\bar{\dot{u}}_0$. From then on an implicit integration rule can be applied, using $\bar{\dot{u}}_n$ to estimate a time step.

Let us now summarize the algorithm for a simple uniform variable stepsize piecewise linear simulator and identify the basic operations and updates. A very coarse version of the algorithm might resemble the one shown in algorithm 5.1

The actions that occur very often are:

- rank one updates on the system matrix due to pivoting operations.
- full vector forward-backward substitutions.
- a change of stepsize and the recomputation of leafcell data.
- a full LU decomposition on the system matrix.

The largest part of the computational effort is accounted for by the last two operations. Since a change of stepsize causes rigorous changes to the system matrix, updating the LU decomposition is rather unattractive and full LU decomposition is inevitable. In the following sections we will first discuss pivoting operations during transient

Algorithm 5.1. *Simple uniform stepsize simulator.*

```
t = 0;
determine x(t);
while t < T do
   solve A·x̄ = LU·x̄ = b̄;
   determine new time step Δt;
    t = t + Δt;
   update circuit and leafcell variables till current time t;
   if solution has reached a boundary then
      solve pwl equations meanwhile keeping LU up to date;
   else /* time step bounded by integration accuracy */
      determine new step size;
      if step size has changed then
         recompute leafcell jacobians and source vectors;
         reassemble the system matrix;
         compute new LU decomposition;
      fi;
   fi;
od;
```

analysis. Next we will try to eliminate the full LU decomposition as much as possible by the introduction of a multirate integration scheme.

5.3 Pivoting

As explained in the previous section, the time integration is interrupted if the solution hits a boundary hyperplane. At this time the pwl equations have to be re-solved which involves pivoting. Note that although a pivoting operation generally changes $\dot{x}$, i.e. the direction in which the circuit variables propagate as a function of time, the applied stepsize may very well remain appropriate, since $\dot{u}$ will often change continuously in time. Since pivoting itself is no reason to restart the integration, it's more efficient to keep the current stepsize while solving the pwl equations and validate it afterwards.

Unlike in the situation during the initial analysis, the submatrix A_{11} has been updated with a rank one update due to the substitution of an integration formula as indicated by equation (5.7). In fact every submatrix A_{ij}, $i, j \in \{1, 3\}$ has been updated according to equation

(5.10). The pivot value and the update vectors are not immediately available since the simulator does not explicitly store matrices J_{ij}. Only matrix J_{11} is merged explicitly into the system matrix. Nevertheless it is still possible to compute the update vectors when performing a pivoting operation and obtain the new jacobian matrix by performing a rank one update.

Theorem 5.1
Let $P(k;l)$ denote a pivot operation on submatrix element A_{33}^{kl} and let L denote the substitution of a multistep integration formula. Operator $\circ$ is the well known "compose" operator in operator theory. Now $P \circ L = L \circ P$, i.e. the order in which operations P and L are applied is irrelevant.

Proof. Let us define column and row vectors (see figure 5.2)

$$c_1 \triangleq A_{13}^{*l},\ c_2 \triangleq A_{23}^{*l},\ r_1 \triangleq A_{31}^{k*},\ r_2 \triangleq A_{32}^{k*} \tag{5.11}$$

and pivot element

$$p \triangleq A_{33}^{kl}. \tag{5.12}$$

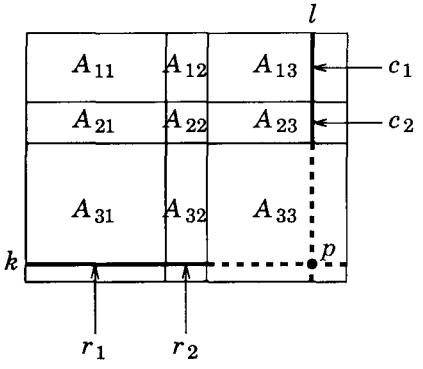


Figure 5.2. *Update vectors for pivoting operation.*

Now define the following matrices where A is a shorthand for matrices A_{ij} (see also figure 5.3):

A' effect of operation P with pivot p on matrix A

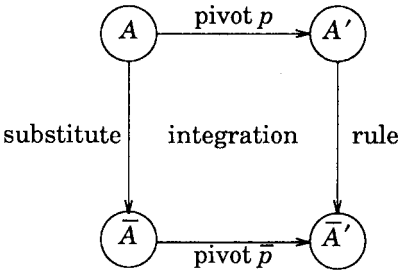


Figure 5.3. *Pivoting operations.*

$\bar{A}$ apply integration rule (5.3) to matrix A

A'' apply integration rule (5.3) to matrix A'

$\bar{\bar{A}}$ effect of operation P with pivot $\bar{p}$ on matrix $\bar{A}$

and show that $\bar{\bar{A}}$ equals A'' equals $\bar{A}'$.

First we construct A'':

$$A'' = A'_{11} + \delta A'_{12} D'^{-1} A'_{21}, \tag{5.13}$$

$$D = I - \delta A_{22}, \quad D' = I - \delta A'_{22}.$$

Now substitute the expressions for A':

$$A'_{11} = A_{11} - \frac{c_1 r_1}{p}, \quad c'_1 = \frac{c_1}{p} \tag{5.14}$$

$$A'_{12} = A_{12} - \frac{c_1 r_2}{p}, \quad c'_2 = \frac{c_2}{p}$$

$$A'_{21} = A_{21} - \frac{c_2 r_1}{p}, \quad r'_1 = -\frac{r_1}{p}$$

$$A'_{22} = A_{22} - \frac{c_2 r_2}{p}, \quad r'_2 = -\frac{r_2}{p}$$

$$p' = \frac{1}{p}$$

The result contains a complicated matrix inverse which can be eliminated by applying the Sherman-Morrison-Woodbury formula:

$$(A + XCY^t)^{-1} = A^{-1} - A^{-1}X(C^{-1} + Y^tA^{-1}X)^{-1}Y^tA^{-1} \tag{2.1}$$

with

$$C \equiv \frac{\delta}{p},\ A \equiv D,\ X \equiv c_2,\ Y^t \equiv r_2\ . \tag{5.15}$$

Using the fact that

$$K = (\frac{p}{\delta} + r_2D^{-1}c_2)^{-1} \text{ and } k = r_2D^{-1}c_2 \tag{5.16}$$

are constants and some juggling with terms we finally obtain:

$$A'' = A_{11} + \delta A_{12}D^{-1}A_{21} \tag{5.17}$$

$$- K(c_1 + \delta A_{12}D^{-1}c_2)(\frac{r_1}{\delta} + r_2D^{-1}A_{21})\,.$$

The derivation of an expression for $\bar{\bar{A}}$ is a bit simpler. Combining

$$\bar{\bar{A}}_{11} = \bar{A}_{11} - \frac{\bar{c}_1\bar{r}_1}{\bar{p}} \tag{5.18}$$

with

$$\begin{aligned}
&\bar{A}_{11} = A_{11} + \delta A_{12}D^{-1}A_{21} && (5.19)\\
&\bar{A}_{13} = A_{13} + \delta A_{12}D^{-1}A_{23} \ \rightarrow\ \bar{c}_1 = c_1 + \delta A_{12}D^{-1}c_2\\
&\bar{A}_{31} = A_{31} + \delta A_{32}D^{-1}A_{21} \ \rightarrow\ \bar{r}_1 = r_1 + \delta r_2D^{-1}A_{21}\\
&\bar{A}_{33} = A_{33} + \delta A_{32}D^{-1}A_{23} \ \rightarrow\ \bar{p} = p + \delta r_2D^{-1}c_2
\end{aligned}$$

we almost automatically find that $\bar{\bar{A}}$ equals A'' which completes the proof.

5.4 A multirate simulator

A way to eliminate the costly full LU decomposition every time the integration time step changes is to apply a multirate integration scheme as introduced in the previous chapter. Instead of integrating the entire circuit with a uniform variable stepsize, every leafcell is assigned its own optimal stepsize. This way fast varying leafcells obtain small stepsizes while slowly varying leafcells are integrated with large ones. Naturally it is possible and even desirable for a

number of leafcells to share the same stepsize and have simultaneous events. If a leafcell changes its activity, its stepsize has to be adapted resulting in a matrix update of small rank.

The transient analysis proceeds in an event driven manner. We distinguish between two types of events:

- dynamic events and
- pwl events.

Let us first examine the dynamic events. These occur every time the validity of a companion model expires. Suppose some leafcell l has a dynamic event at time $t_{ev} = t_{n+1}$. First we update the x-, u-, $\dot{u}$- and p-variables corresponding to leafcell l using the divided differences which are approximations of their time derivatives $\bar{x}$, $\bar{u}$, $\bar{\dot{u}}$ and $\bar{p}$. Next a new stepsize h_{n+1} is computed for the next integration step by estimating the local truncation error using $\bar{\dot{u}}$. The previous stepsize h_n is maintained if there is no significant difference between the new stepsize h_{n+1} and h_n. In this case only the $\bar{b}$ vector changes due to the update of $\dot{u}$ (see equation (5.9)). Vector $\bar{b}$ is constructed by merging all leafcell contributions $\bar{b}_1$ into one system source vector. The changes in $\bar{b}$ denoted by $\Delta\bar{b}$ are now used to compute an update for $\bar{x}$ denoted by $\Delta\bar{x}$. A more complicated situation arises if the stepsize requires adjustment. Now not only $\bar{b}$ but also J is subject to a change causing a full recomputation of $\bar{x}$ by forward-backward substitution. As we will see later on, this recomputation can be replaced by an update. Finally the new event of leafcell l is determined at $t_{ev} + h_{n+1}$, assuming no pwl event is expected to occur within time interval $[t_{ev}, t_{ev} + h_{n+1}]$.

We already noticed that substitution of an integration formula causes a rank r update on the A_{11} matrix if A_{22} has dimension r. The jacobian matrix is updated with a matrix containing the chosen stepsize. Thus a change of stepsize will also change the jacobian J_{11}. Suppose

$$J_{11}{}^{(n)} = A_{11} + \delta^{(n)} A_{12} (D^{-1})^{(n)} A_{21} \tag{5.20}$$

represents the jacobian based on h_n in the time interval $t_n \le t \le t_{n+1}$ and

$$J_{11}{}^{(n+1)} = A_{11} + \delta^{(n+1)} A_{12} (D^{-1})^{(n+1)} A_{21} \tag{5.21}$$

is the new jacobian after h_n has changed to h_{n+1}. An update ΔJ for the

jacobian ${J_{11}}^{(n)}$ is easily computed:

$$\begin{aligned}\Delta J &= {J_{11}}^{(n+1)} - {J_{11}}^{(n)} \\ &= A_{12}\left\{\delta^{(n+1)}(D^{-1})^{(n+1)} - \delta^{(n)}(D^{-1})^{(n)}\right\}A_{21} \\ &= A_{12}\Delta D^{-1}A_{21}\,.\end{aligned} \tag{5.22}$$

Since the matrix D^{-1} is required quite often, it is stored explicitly by the simulator. Therefore the matrix ΔD^{-1} can be determined very fast and the row and column update vectors required for the dyadic update can be provided. Generally only a small subset of leafcells will have to change stepsize at the same time point. So the system matrix will be subject to relatively small deviations and the LU decomposition can be updated very efficiently with algorithm 2.2.

After the changes $\Delta\bar{x}$ have been determined, the impact on the related leafcells must be investigated. Since the circuit variable derivatives have changed for those leafcells, it may very well be possible that the stepsizes substituted are no longer valid. Therefore we have to update the x-, $\dot{u}$-, p- and $\bar{x}$-variables, recompute $\bar{u}$ and $\bar{p}$, and check if the applied stepsizes are still appropriate. If necessary, the event times must be adjusted. It appears that changing stepsizes is a costly operation which should be avoided whenever possible.

A pwl event occurs if the solution vector reaches a boundary hyperplane of the current segment. This happens if for some leafcell a component of its p vector becomes zero. Pwl events can be determined explicitly using p and the time derivative estimates $\bar{p}$ at the current time t_{ev}. Suppose

$$\exists_k \forall_{i\neq k}\,[\,\bar{p}_k < 0 \wedge \bar{p}_i < 0 \wedge ((p_k \;/\; \bar{p}_k) \geq (p_i \;/\; \bar{p}_i))\,]\,,$$

then the next pwl event for this leafcell is at time $t_{ev} - (p_k \;/\; \bar{p}_k)$. At that time point the integration must be stopped until the piecewise linear equations are solved and the solution curve can proceed in another segment. A complication arises if simple pivots are not sufficient to solve the pwl problem and the van de Panne algorithm has to take a step θ in the direction of λ or a q variable. Now $\dot{u}$ may change

discontinuously in which case the integration has to be restarted for all leafcells involved. Such a restart is not necessary if only simple pivots are performed. More important however is that the step θ cannot be taken as long as related leafcells are still represented in the system matrix by their companion models. For those leafcells the stepsize has to be reset to zero to prevent undesired side effects such as step functions in capacitor voltages instead of exponential characteristics. Obviously, since changing the system matrix can have implications on $\tilde{x}$, we have to recompute the step θ in the direction of q if stepsizes were actually reset.

Apart from discontinuities in $\dot{u}$ due to steps taken by van de Panne, note that pivoting causes an upperbound on the number of continuous state variable derivatives since these directly depend on on $\bar{x}$. Unfortunately this restriction implies that nothing can be gained from the application of integration formulas with order larger than two [71]. After the piecewise linear equations have been solved, the leafcells affected by changes in $\bar{x}$ have to be examined. If necessary their events and stepsizes have to be corrected.

Since it is very well possible that several events coincide at the same time point, we deal with the corresponding leafcells simultaneously. This is especially easy for dynamical events by applying the superposition principle. E.g. all new contributions to $\bar{b}$ are merged together before determining $\Delta\bar{x}$. So at any event time we process a cluster of leafcells with similar events instead of single events.

Definition 5.1
A cluster is defined as a 4-tuple $C = (E, t_{ev}, ET, IF)$ in which E is the set of leafcells with events of type ET, simultaneously occurring at time t_{ev}. The integration formula being applied for the current time step is defined by IF. Proper values for ET are $\{pl\_event, dynamic\_event\}$. For simplicity we will assume cardinality one for clusters of type *pl_event*.

The transient analysis and the modifications to van de Panne's algorithm are listed in algorithms 5.2 and 5.3

It appears that there are two situations in which leafcell events have to be rescheduled:

- while the pwl equations are solved, the algorithm may very well reach and change leafcells with events scheduled in the future.
- changes in one particular leafcell may influence another one causing a new (dynamical or pwl) event prior to its present scheduling.

Here we encounter a basic limitation of the piecewise linear simulator. At the moment of rescheduling, we find that the original stepsize is no longer valid and should be replaced by another, smaller, one. In practice, assuming the difference is not too large, we simply reschedule the event without changing the stepsize. In some cases events have to be forced to the current time. This implies that since the integration step was interrupted, the truncation error becomes larger than estimated. It can be shown that the order of a second order method such as the trapezoidal rule falls back to one [13]. This appears to be a major limitation since in order to preserve integration accuracy, stepsizes have to be estimated by a first order criterion even if second order integration methods are applied. Clearly this has a negative influence on simulation run times.

The problem becomes even more disturbing if two-step backward differentiation or A-contractive methods are selected. These methods use two previous values for u and $\dot{u}$ which must be administered carefully. Step size truncation has a disastrous effect on these previous values: they become invalid. The integration must now switch back to a one-step formula such as the trapezoidal rule or backward Euler, after which the two-step scheme can continue.

Let us introduce some additional variables before listing the transient analysis algorithm:

B	set of leafcells that require a recomputation of $\bar{b}$.
$\Delta\bar{b}$	sparse vector containing updates for $\bar{b}$ contributed by one or more leafcells.
$\Delta\bar{x}$	sparse vector containing updates for $\bar{x}$.
$L_{\Delta\bar{x}}$	set of leafcells related to nonzero entries in $\Delta\bar{x}$:

Algorithm 5.2. *Transient analysis.*

```
transient():
        C = next_event_cluster();
        while ( t_ev < T ) do
                process_event_cluster(C);
                C = next_event_cluster();
        od;

next_event_cluster():
        /* Determine the set of leafcells E belonging to the same
         * class that require servicing at the nearest event time t_ev.
         */
        assemble cluster C;
        return C;

process_event_cluster(C):
        update_module_variables(C);
        Δb̄ = ∅;
        if ET ≡ pl_event then
                /* Set I contains only a single leafcell i and p_r^i = 0.
                 * SetBcontainsleafcellsthatrequirearecomputationofbdot.
                 */
                initialize sets I, B = E, ∅;
                initialize vectors e_1^i, e_2^i, e_3^i = 0.0, -ū, -p̄;
                λ = 0.0;
                push(i, "λ column", down);
                push(i, r, up);
                vdpanne(λ);
                for all l∈B do compute update Δb̄^l for b̄ od;
                solve LDU·Δx̄ = Δb̄;
                b̄ += Δb̄;
        elseif IF ≡ forward_euler then /* Restart integration */
                for all e∈E do
                        compute_update Δb̄^e for b̄;
                        Δb̄ += Δb̄^e;
                od;
                solve LDU·Δx̄ = Δb̄;
                b̄ += Δb̄;
                for all e∈E do "increment" integration method od;
        else /* Regular integration step */
                recompute = false;
                for all e∈E do
```

```
                    "increment" integration method;
                    new_stepsize(e);
                    if e→stepsize ≠ e→previous_stepsize then
                            recompute = true;
                            update LU decomposition;
                    fi;
                    compute contribution Δb̄^e for b̄;
                    Δb̄ += Δb̄^e;
            od;
            if recompute ≡ true then
                    solve LDU·x̄ = b̄;
                    generate Δx̄;
            else
                    solve LDU·Δx̄ = Δb̄;
                    b̄ += Δb̄;
            fi;
    fi;
    process_leafcells(C, Δx̄);

update_module_variables(S):
    for all s∈S do
            update x^s with x̄^s till the current time point t;
            update u^s with ū^s till the current time point t;
            update p^s with p̄^s till the current time point t;
    od;

process_leafcells(C, Δx̄):
    eliminate non relevant entries from Δx̄;
    for all entries i in Δx̄ do update x[i] and x̄[i] od;
    assemble L_Δx̄; /* Note: E∩L_Δx̄ ≡ ∅ */
    update_module_variables(L_Δx̄);
    for all e∈E do
            compute ū^e;
            compute ẇ^e and p̄^e;
            schedule(e);
    od;
    for all e∈L_Δx̄ do
            compute ū^e;
            compute ẇ^e and p̄^e;
            reschedule(e);
    od;
```

Algorithm 5.3. *Transient analysis: Van de Panne.*

```
vdpanne (λ):
    carry_on = true;
    while carry_on do
        /* Compute derivatives of x */
        generate source vector b;
        solve LDU · x̃ = b;
        assemble L_x̃;

        /* Determine maximal number of pivots */
        Si = 0;
        for i = tos downto 1 do
            s = sign (stack [i ].leaf, stack [i ].column);
            if s ≠ 0 then
                S = s;
                Si = i;
            fi;
        od;
        check_sign();
        if Si = 0 then /* Cannot pivot */
            determine_theta ();
            if θ ≡ ∞ then abort; /* Cannot solve */
            if θ > 0 then
                check_related_leafcells();
                x = x + dir·θ·x̃;
                for all l∈L_x̃ do update u̇^l and p^l od;
                update λ or q [column ];
            fi;
            if column ≡ λ and λ ≡ 0 then
                carry_on = false /* Solution found */
            elseif column ≠ λ and q [column ] ≡ 0 then
                /* Current active column became 0 */
                pop ();
                stack [tos ].dir = − stack [tos ].dir ;
            else /* New blocking row */
                push (leaf, row, up)
            fi;
        else
            perform_pivots ();
            for i = tos downto Si do
                B = B ∪ leaf;
                pop ();
            od;
```

```
                    dir = dir·S;
                fi;
        od;

check_sign():
        m = stack[Si].column;
        n = stack[tos].column;
        if sgn(A33^mn) < 0 then
                for all l∈Lx̃ do
                        reset stepsize h^l of leafcell l to 0;
                od;
                set S = sgn(A33^mn);
                recompute x̄;
                assemble new Lx̃;
        fi;

check_related_leafcells():
        for all l∈Lx̃ do
                reset stepsize to 0;
                B = B ∪ l;
                update LU decomposition;
                update x with x̄ till the current time point;
        od;
        recompute x̃;
        assemble new Lx̃;
        recompute θ;
```

5.5 Some optimizations

Let us consider the consequences of small perturbations in jacobian J_{11} as well as the source vector $\bar{b}_1$. This situation arises quite often in the simulator: due to pivoting during transient analysis and secondly when changing the integration stepsize for a leafcell. Since these operations cause only minor changes to the system matrix as well as source vector, we suspect that $\bar{x}$ can be easily updated instead of performing a full recomputation.

We first examine the general case in which the new situation is given by

$$J_{11}^{*} \cdot \bar{x}^{*} + \bar{b}_1^{*} = 0 \tag{5.23}$$

where

$$J_{11}\bar{x} + \bar{b}_1 = 0$$

$$J_{11}^* \triangleq J_{11} + \Delta J_{11}$$

$$\bar{b}_1^* \triangleq \bar{b}_1 + \Delta\bar{b}_1$$

$$\bar{x}^* \triangleq \bar{x} + \Delta\bar{x}\,.$$

Substitution of the above relations and some rewriting yields:

$$(J_{11} + \Delta J_{11})\Delta\bar{x} + \Delta J_{11}\cdot\bar{x} + \Delta\bar{b}_1 = 0\,. \tag{5.24}$$

Now reconsider the two special cases mentioned before.

The application of equation (5.24) in the case of pivoting was already documented in [72]. Here the modification of the system matrix is simply a rank one update where the update vectors must be taken from submatrices J_{12} and J_{21} as presented in equation (5.10). So assuming matrix element J_{22}^{kl} is taken as a pivot, the updates ΔJ_{11} and $\Delta\bar{b}_1$ are simply given by:

$$\Delta J_{11} = c\cdot r^t = -\frac{J_{12}^{*l}}{J_{22}^{kl}}\cdot J_{21}^{k*}$$

$$\Delta\bar{b}_1 = c\cdot a = -\frac{J_{12}^{*l}}{J_{22}^{kl}}\cdot\bar{b}_2^k\,.$$

The update for $\bar{x}$ is readily computed from

$$(J_{11} + c\cdot r^t)\cdot\Delta\bar{x} + c\cdot(r^t\cdot\bar{x} + a) = 0\,.$$

The same optimization can be applied in case of stepsize changes. An expression for the update of the system matrix was already derived in section 5.4 equation (5.22). An update for vector $\bar{b}$ is computed easily by using the same temporary matrix as was used for the update of J_{11}: e.g. $\Delta\bar{b}_1 = \Delta D^{-1}\dot{u}_n$ for BE or TR.

Another optimization is found by simply reducing the sets $L_{\Delta\bar{x}}$ and $L_{\Delta\tilde{x}}$. During the forward-backward process we inevitably introduce small insignificant errors in the solution vectors $\Delta\bar{x}$ and $\Delta\tilde{x}$. These almost zero entries in the sparse solution vectors may cause the unnecessary

evaluation of the related leafcells. This overhead can be reduced by carefully ignoring the leafcells related to very small sparse vector entries. The danger lies in the undesired introduction of errors in the leafcells $\dot{u}$ vectors. Since the circuit variables x are computed by updating with the $\dot{x}$ vector, errors in $\dot{x}$ are directly reflected in x. Next the errors in x are fed in the $\dot{u}$ vector, but multiplied by matrix A_{21} which may drastically increase the errors. Finally, $\dot{b}$ depends on $\dot{u}$ yielding an incorrect source vector and resulting in even larger errors in $\dot{x}$. Therefore elimination is guided by a very conservative heuristic and can be turned off if desired.

5.6 Event clustering

As already pointed out by C.W. Gear [66] there is a disadvantage to the approach sketched in section 5.4. Since it is impossible to predict the circuit behavior while determining a new stepsize for a specific leafcell, we may be forced to change the stepsize before the leafcell reaches the time point at which its next event is scheduled due to activities in other parts of the circuit. This causes a lot of computational overhead, involving the update of circuit variables and the recomputation and effectuation of stepsizes. Gear also showed that no merit can be expected from the application of a multirate integration technique if the stepsizes of the subproblems lie close together. So the rigorous application of the multirate principle described above may not be very efficient.

The transient analysis algorithm shown in algorithm 5.2 already implements the handling of multiple events at the same time point. Multiple events can be processed by simply merging their contributions in the $\bar{b}$ vector before computing the new $\bar{x}$. Since it is often the case that related components have events at about the same time, it is clear that a lot of cpu time can be saved by clustering dynamical events. As long as every component is assigned its own optimal stepsize however, it will rarely be possible to deal with more than one event at a time.

A way to achieve improvements for both drawbacks is the discretization of event times. Forcing nearby events to a grid increases the average cluster size. A further reduction of the overhead can be achieved by forcing groups of related leafcells to use the same (minimum) stepsize. This way the recomputation of stepsizes is

minimized and the need for reducing them will diminish. Two heuristic approaches to event clustering have been tested. The first and most effective procedure determines, given the nearest event time, a set of leafcells which have almost reached that time (i.e. some fraction of its stepsize). Unfortunately this implies a systematic preemptive treatment of events. Since this has a negative effect on the integration order, the second, less effective approach was selected. Dynamic event times are discretized as follows. Assume the length of the integration interval is given by T, then events are forced to one of the following grids:

$$T \cdot 2^{-k} \text{ for } k = 1,2,\cdots,N. \tag{5.25}$$

Every time a new event has to be determined, a value of k is chosen such that the desired stepsize can be mapped onto the discrete time axis without violating the time integration accuracy requirements reflected by the stepsize h. After the next event has been determined, the leafcells stepsize is set accordingly.

The speed-up achieved by event clustering is circuit dependent. Generally, the required cpu time is reduced by 10-20%. Although the number of events can decrease with about 50%, the larger number of leafcells involved causes more fill-ins in source and solution vectors. Therefore the amount of time needed for the sparse matrix operations increases, limiting the final gain to "only" 20%. A comparison of clustering versus no clustering is shown in table 5.2 for a number of inverter chains (see chapter 6). In this example the effect of clustering is minimal due to the nature of the circuit: initially many leafcells already obtain about the same events. The increase in cpu time occurring at 20 stages is caused mainly by the ripple effect, which causes extra circuit activity and, consequently, extra simulation time. A more successful example is the counter (see chapter 6) where clustering reduces the cpu time from 405 to 270 seconds. Note that the applied heuristic may turn out expensive in a situation with a relative large number of pwl events. In this case we require lots of stepsize changes to force the leafcells involved to the discrete time grid.

Some gain can be expected from a reduction of the number of clusters allowed. Assume all leafcells are forced into a suitable cluster. Switching from one cluster to another would be restricted to overlapping time points on the discrete time axis. Restriction of the

TABLE 5.2. *Inverter chain clustering results.*

#stages	cpu time no clustering	cpu time clustering
5	1.28	1.15
10	6.27	4.78
15	11.38	8.57
20	17.90	15.37
25	25.68	27.67

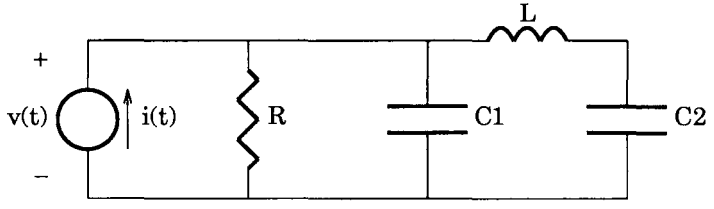
number of clusters will then prevent leafcells from running "loose".

Part of the problem in optimizing the event scheduling is the generic character of the simulator. A much more efficient clustering could be implemented if the simulator had some knowledge about components like e.g. MOS transistors. In that case the circuit could be partitioned into subcircuits each being operated with the same stepsize. Crossing a segment boundary would trigger a reevaluation of the partitioning. This way the simulation of logic circuitry at the transistor level could become more efficient. A similar approach but combined with waveform relaxation was shown to be very effective [73]. Ironically, another problem is caused by the piecewise linear modeling itself. Hitting a boundary always interrupts the discretized event scheme, causing expensive matrix updates which sometimes cancel the gain obtained by clustering. We already pointed out in the beginning of this chapter that integrating across segment boundarys is not a viable option.

6. Simulation results

6.1 Landman circuit

The limitations of numerical integration methods can be illustrated using the nasty little circuit shown in figure 6.1 constructed by B. Landman. The circuit appears in [59] and is used to examine some properties of integration methods such as numerical damping and time errors. The plots in [59] p. 257, although created with a fixed stepsize that is much too large to expect a reasonable result, compare nicely to the signals obtained by PLATO as shown in figures 6.3, 6.4, 6.5 and 6.6, in which both the exact (dashed line) and computed (solid line) are plotted. The multirate methods show the same damping and time error tendencies as their fixed time step counterparts.



$R = 1/102$ $\quad C1 = 1$
$L = 102/20602$ $\quad C2 = 20602/10100$

Figure 6.1. *Landman circuit.*

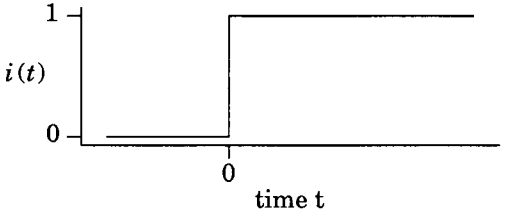


Figure 6.2. *Excitation for the Landman circuit.*

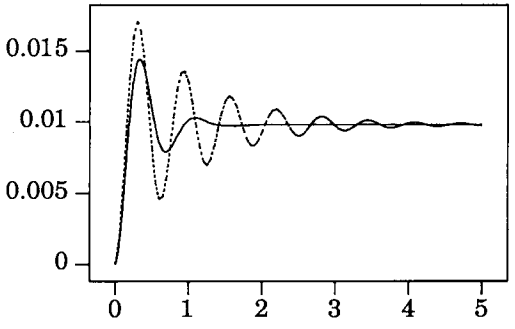


Figure 6.3. *Transient analysis of the Landman circuit using PLATO with BE.*

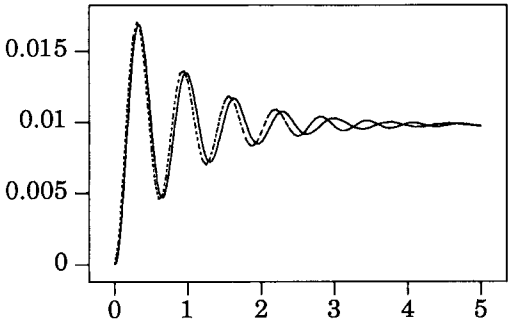


Figure 6.4. *Transient analysis of the Landman circuit using PLATO with TR.*

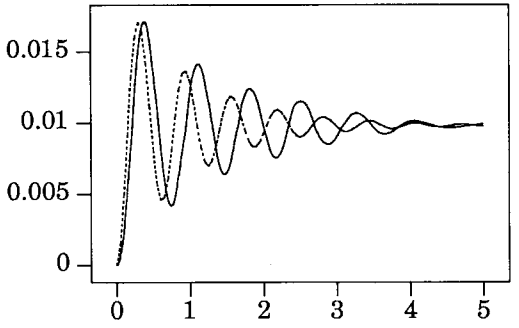


Figure 6.5. *Transient analysis of the Landman circuit using PLATO with BDF2.*

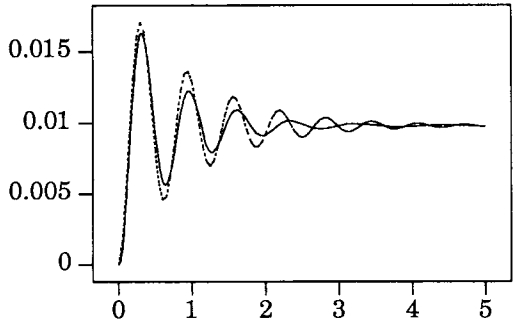


Figure 6.6. *Transient analysis of the Landman circuit using PLATO with ACT2.*

The exact solution is given by:

$$v_{C2} = c_1 e^{\lambda_1 t} + c_2 e^{\lambda_2 t} \cos(\omega t) + c_3$$

$c_1 = 1/(99{\cdot}102)$ $\quad\quad$ $\lambda_1 = -100$

$c_2 = -50{\cdot}c_1$ $\quad\quad$ $\lambda_2 = -1$

$c_3 = 1/102$ $\quad\quad$ $\omega = 10$

6.2 Inverter chains

A useful circuit for the illustration of latency behavior is a chain of nmos (cmos) inverters as plotted in figure 6.7. The circuit activity will be a reflection of the amount of change in the input signal. When a constant input is applied the chain will be in rest and computational effort will be at a minimum. If the input changes, the change will propagate through the chain but will be restricted to a small amount of inverters.

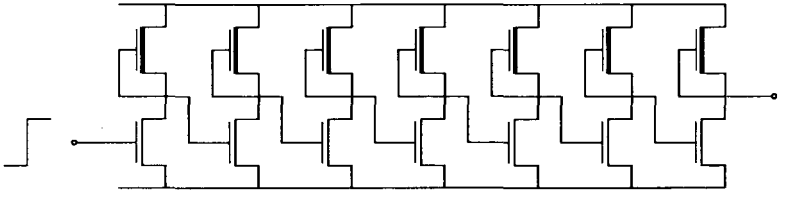


Figure 6.7. *7-Stage inverter chain.*

6.3 Ring inverters

Another notorious test circuit for circuit simulators is the ring inverter in figure 6.8. An oscillation will occur if the number of inverters is large enough and the oscillation frequency will diminish if the number of stages increases. So large parts of the circuit will be in rest. Only those inverters that are currently involved in the propagation of the wave front are active and require treatment.

6.4 A/D converter

A circuit typically suited for simulation by the piecewise linear simulator is the A/D converter circuit depicted in figure 6.9. It contains control logic modeled at the gate level as well as an analog multiplexer composed out of MOS transistors. The actual conversion from analog to digital is implemented by a macromodel. An inhomogeneous collection of simulation results is presented below.

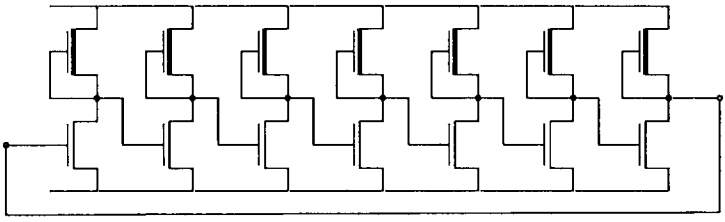


Figure 6.8. *7-Stage ring inverter.*

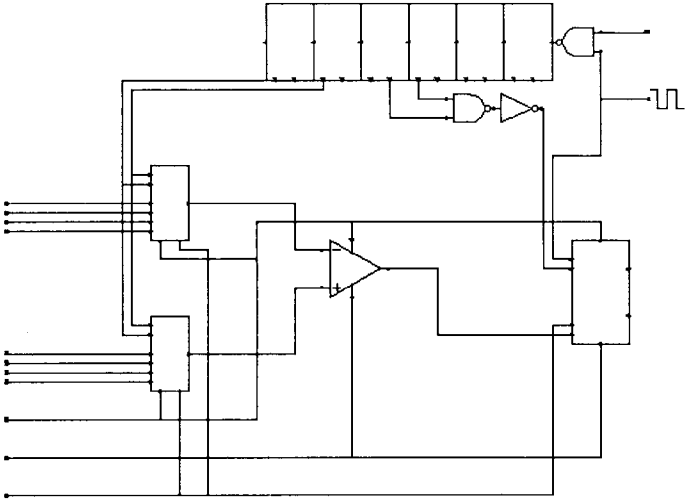
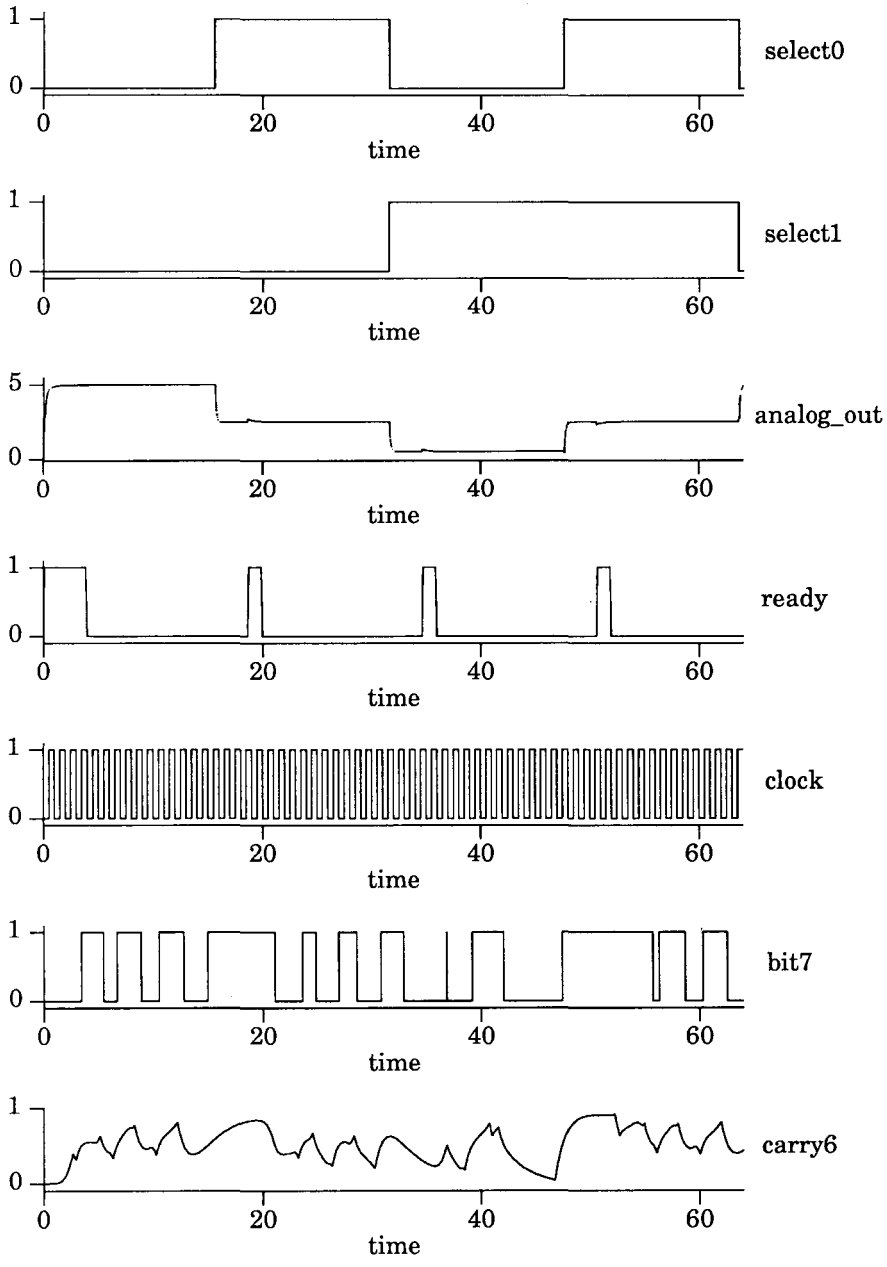
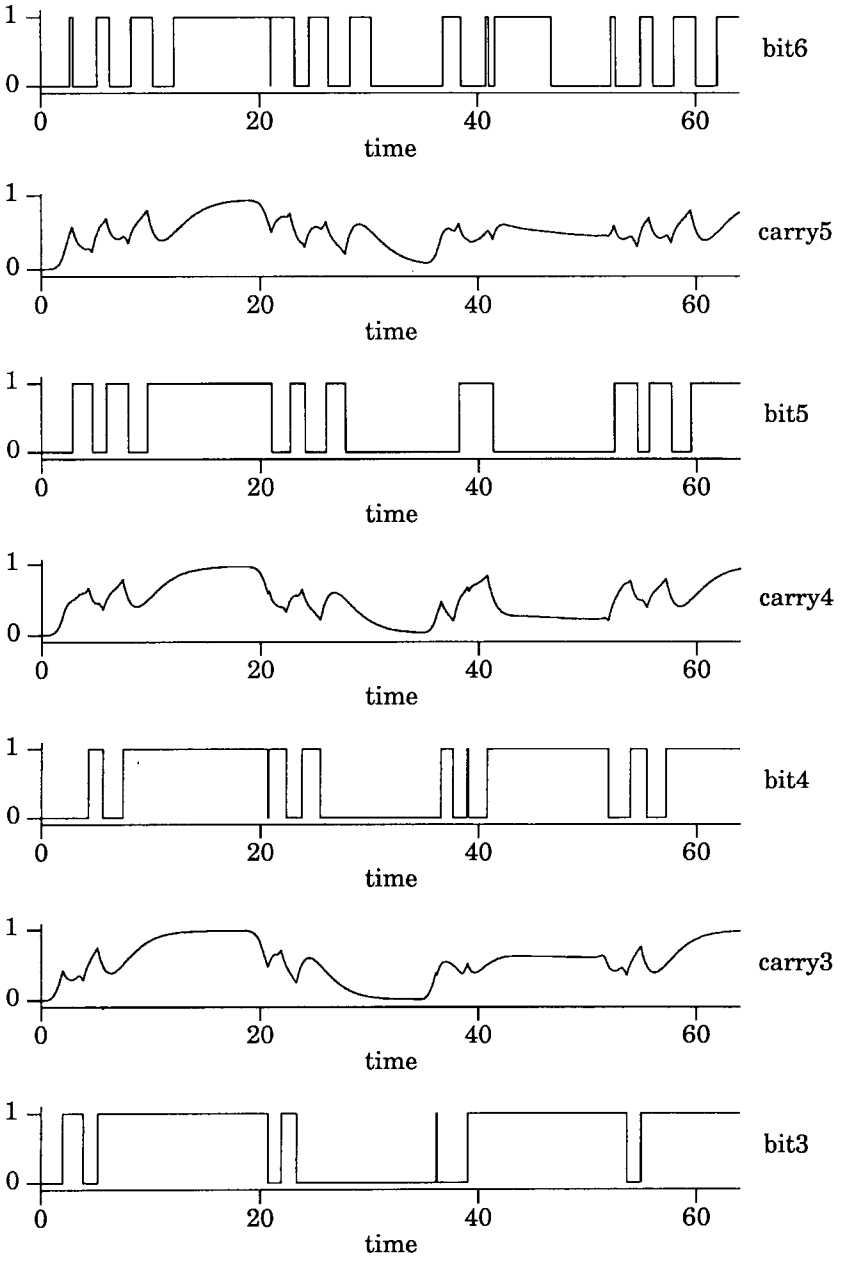


Figure 6.9. *A/D converter circuit.*


1
0
0
20
40
60
time
select0
select1
5
analog_out
ready
clock
bit7
carry6


1
0
0
20
40
60
time
bit6
carry5
bit5
carry4
bit4
carry3
bit3

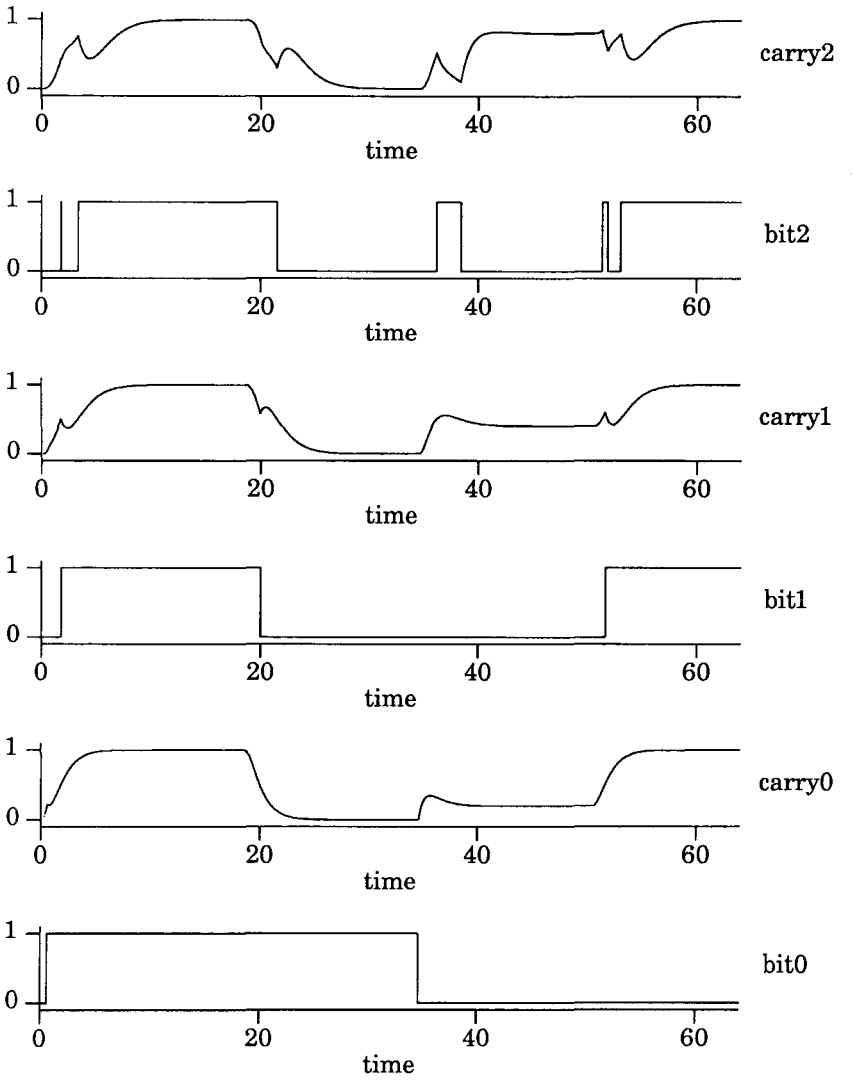


Figure 6.10. *A/D converter signals.*

6.5 Counter

The counter circuit is a simple four bit counter built out of flip-flops as shown in figure 6.11 and some logic gates. All transistors applied are modeled using the simplest MOST model, comparable to SPICE level 1. Output signals are presented in figure 6.12.

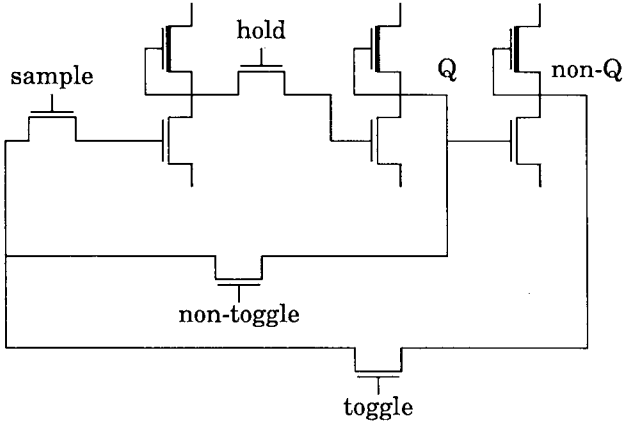
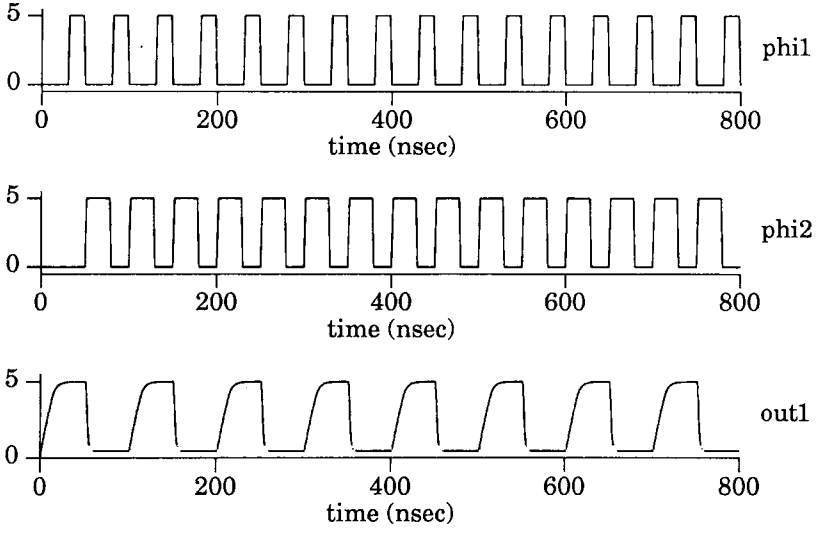


Figure 6.11. *Flip-flop.*



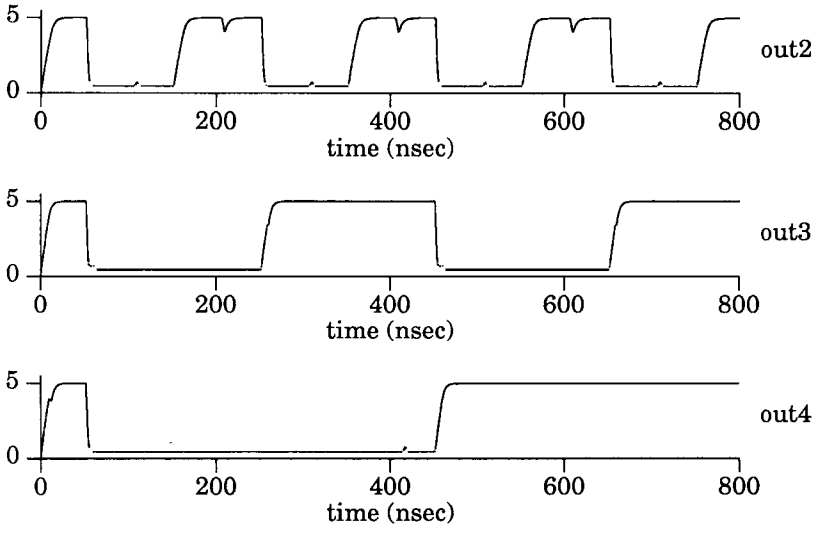


Figure 6.12. *Counter signals.*

6.6 Two complement accumulator

A circuit containing logic components only is a counter as used in the proportional tracking A/D converter described in [74]. In default mode the counter just counts in a normal up or down fashion. Every bit of the counter is equipped with a range enable input. If such an input becomes active, the corresponding bit effectively becomes the least significant bit. All less significant bits are blocked, the more significant bits count as if the enabled bit is the least significant one.

6.7 Simple phase locked loop

A simple little circuit typically suited for the piecewise linear simulator is a phase locked loop composed out of some comparators, logic gates and integrators. It consists of a mixture of analog, digital and abstract components.

6.8 Neural networks

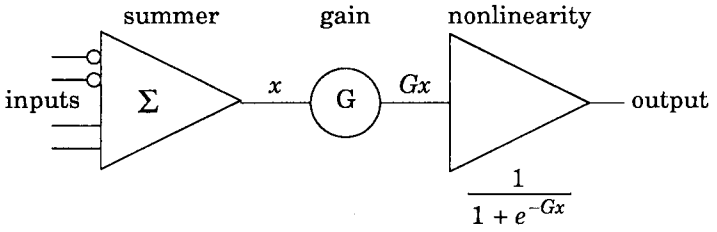


Figure 6.13. *Circuit model for a Hopfield neuron. Inputs can be either inhibitory (circles) or excitatory. The result of the input summation x is multiplied by a constant gain factor G and compressed between 0 and 1 by a nonlinear function.*

A nice example of the flexibility of the piecewise linear simulator is its ability to simulate rather abstract systems such as neural networks. Figure 6.13 shows a circuit model for a Hopfield neuron [75, 76]. A simple piecewise linear model using only a few segments to implement the nonlinear function already suffices to simulate the neural circuits in [77]. A typical characteristic of neural networks is the high connectivity of the circuit. This causes the system matrix to be rather full instead of sparse. Since the simulator is built for operation on sparse matrices, the simulation of large neural networks becomes rather inefficient. At this moment a special neural version of the simulator is developed, using full matrix techniques and the parallel and vector capabilities of an Alliant FX/8 mini-supercomputer.

6.9 Program statistics

This section presents an overview of the program statictics that are gathered during simulation for the circuit examples mentioned before. They form a selection of the benchmark and test circuits I used to test and optimize the simulator. The statistics printed are supposed to give an impression of the nature of the circuits. Furthermore they intend to illustrate the algorithmic concepts explained in previous chapters. The simulations were done on a Hewlett-Packard workstation (hp9000s835) and are measured in seconds. Results are scattered over tables 6.1, 6.2 and 6.3.

TABLE 6.1. *Program statistics: inverter chains. The "#" character is a shortcut for "number".*

circuit	chain5	chain10	chain15	chain20	chain25
#components	11	21	31	41	51
matrix size	25	50	75	100	125
matrix density	15.0	7.8	5.2	3.9	3.2
#pivots (initial)	5	10	15	20	25
#pivots (transient)	11	23	36	48	61
#full LU decompositions	1	1	1	1	1
#dyadic updates	165	357	546	721	920
average #changed elements	17.2	34.1	50.2	65.5	81.5
#forward/backward subst.	602	1303	2028	2634	3316
#pwl events	11	23	36	48	61
#dynamic events	558	1213	1889	2449	3082
average size $\Delta\dot{x}$	13.2	23.5	32.3	39.0	43.4
average #leafs reached (pwl)	6.8	12.5	17.7	22.8	27.0
average #leafs reached (dyn)	6.3	10.5	14.3	17.4	19.4
cpu time	1.28	6.27	11.38	17.90	25.68

TABLE 6.2. *Program statistics: ring inverters.*

circuit	ring7	ring15	ring31	ring61	ring121
#components	21	45	93	183	363
matrix size	42	90	186	366	726
matrix density	10.4	4.9	2.4	1.2	0.6
#pivots (initial)	7	15	31	61	121
#pivots (transient)	98	118	158	547	610
#full LU decompositions	1	1	1	1	1
#dyadic updates	2449	3491	5259	16083	19835
average #changed elements	38.6	78.8	165.6	313.9	517.4
#forward/backward subst.	13794	17084	23102	76482	82711
#pwl events	98	118	158	547	610
#dynamic events	13485	16699	22565	74718	80638
average size $\Delta\dot{x}$	33.5	35.9	40.9	38.4	43.1
average #leafs reached (pwl)	20.0	24.4	27.9	29.2	40.3
average #leafs reached (dyn)	18.1	20.9	23.8	22.7	25.2
cpu time	68.82	108.15	198.25	875.42	1505.08

TABLE 6.3. *Program statistics: several other circuits.*

circuit	landman	adc	counter	accumulator	pll
#components	4	74	60	146	10
matrix size	4	98	174	134	10
matrix density	68.8	2.3	3.4	0.7	23.0
#pivots (initial)	0	78	84	116	6
#pivots (transient)	0	9532	1008	5584	238
#full LU decompositions	1	2	12	1	1
#dyadic updates	38	11010	34263	5714	298
average #changed elements	6.2	2.5	88.9	0.5	6.4
#forward/backward subst.	140	23478	30281	10661	925
#pwl events	0	2430	1008	1710	138
#dynamic events	139	6073	26948	46	256
average size $\Delta\dot{x}$	3.9	7.8	23.4	1.1	1.0
average #leafs reached (pwl)	0	1.3	10.8	2.8	2.5
average #leafs reached (dyn)	3.0	7.5	12.4	3.1	2.1
cpu time	0.1	37.3	269.55	12.12	0.53

Conclusion

Let us conclude this thesis with some final remarks. The description in full detail of the techniques implemented in the piecewise linear simulation program was one of the purposes of this text. Nevertheless many details have been omitted in favor of the basic principles. Through the past years the program developed to quite a complicated piece of software. Paradoxically, the simplification of nonlinear mappings to piecewise linear mappings not necessarily yields a simpler solution method.

The program was intended as a tool for "difficult" circuits, i.e. circuits causing convergence problems or consuming large amounts of cpu time when simulated with SPICE-derivates. Clearly the performance results are incomparable to e.g. the ones obtained by SPICE since the latter program is applicable to a more restricted class of circuitry. Nevertheless people are always interested in benchmarks in which PLATO is compared to SPICE. In such benchmarks, SPICE often wins. Several reasons to justify this result can be mentioned. First of all, a system as generic as PLATO can hardly beat a dedicated, carefully tuned program such as SPICE, in which many man-years of development work are accumulated. Every program segment of SPICE has been fully optimized based on years of experience. Secondly, a number of low-level functions in PLATO could be optimized. The sparse matrix datastructure for instance, although quite practical for operations like fill-in, probably increases the time required for matrix operations due to the dispersion of row and column elements over memory. Conventional storage mechanisms usually store row or column elements in contiguous memory parts, causing a less flexible

but probably faster datastructure, especially for large matrices. Finally, the benchmark circuits used in a comparison with SPICE were rather small circuits, which is an advantage for SPICE which is particularly good at small circuits.

As can be seen from the simulation examples and the corresponding statistics, the ideas behind the simulator work well. An analysis of the nature of the cpu time spent during transient analysis, clearly shows that the processing of dynamical events accounts for a large part of the computational effort. In my view, future research should be aimed at a more efficient integration, perhaps combined with a more explicit partition of the circuit into sub-circuits in order to reduce overhead costs even further. Additional speed-ups of an order of magnitude should be feasible. Especially the application of exponential integration methods seems promising. It is quite clear that signals arising in typical electronic circuits often show exponential characteristics. In this respect, the common practice to use polynomials to approximate exponential functions seems rather awkward.

Finally some concise conclusions can be formulated:

- all components are modeled in a uniform way allowing for a natural implementation of mixed-level simulation. The introduction of the piecewise linear modeling technique does not cause any limitations. The simulator reads all component models from an external database, thus facilitating the introduction of new models and macro models.
- circuits can be specified in a high level description language with many sophisticated language features and constructs.
- matrix update techniques and carefully tuned forward-backward substitution are used to eliminate full sparse LU decomposition. The efficiency of these techniques is illustrated by the simulation results.
- latent sub-circuits are exploited by the introduction of multirate integration techniques. Although theoretical analysis is pessimistic about the application of these techniques in general, they show quite satisfying results when applied to the circuit simulation problem.

- the strategy of the simulation program to compute updates where possible instead of full recomputations effectively exploits large differences in sub-circuit activity.
- the simulator can cope with a large class of circuitry that cause problems to conventional simulators. Typical circuits highlighting the piecewise linear simulator capabilities are mixed-level simulation problems with tight feedback loops like analog-to-digital converters, phase-locked loops, sigma-delta modulaters, etc. Although not optimized for analog and MOS circuits, the simulator shows reasonable performance in a number of comparable cases.

I would like to mention a few people who have contributed to this work. First I would like to mention prof. J.A.G. Jess for the opportunity to write this thesis and for patiently proofreading the manuscript. Furthermore I am indebted to Jos van Eijndhoven for numerous discussions and ideas during the preparation of this thesis. Thanks to Pim Buurman for lots of additional optimizations. Finally I would like to thank my parents for continuous support.

Appendix 1: Algorithmic notations

The algorithms in this thesis are expressed in a convenient C-like programming language. This means that most operators conform to the ones defined in C. The well known set operations have been added to allow a more abstract notation of some algorithms using sets. An overview of all operators is shown in table A1.1. Furthermore the language is enriched with some Pascal-inspired loop statements and conditional constructs. The superfluous *begin* - *end* constructs for block statements are deleted in favor of e.g. *do* - *od*.

The simplest loops are borrowed from Pascal and slightly modified:

for *I* **to** *IE* **do** *S* **od**;

for *I* **downto** *IE* **do** *S* **od**;

in which:

I initialize control variable.

S loop body consisting of one or more statements separated by semicolons.

IE integer expression determining the highest (or lowest for the downto loop) value of the loop variable for which the loop body *S* is executed.

The semantics remain unchanged.

The same modification has been applied to the *while* loop.

while *B* **do** *S* **od**;

with:

B a boolean expression in the C sense, e.g. an expression is regarded *true* if non zero and *false* otherwise. The loop body *S* is executed as long as *B* evaluates to *true*.

Conditional statements look like:

if *B* **then** *S* **fi**;

if *B* **then** *S* **else** *S* **fi**;

Keyword **elseif** is equivalent to the sequence **else if**. As in C, the **else**-part always corresponds to the most recent **if** clause. The comment delimiters are also taken from C: /* and */.

The datastructure used to store sparse vectors is simply a linear linked list. In some algorithms however, this list is treated as a set of non zero entries. An empty list is denoted by the ∅-symbol rather than **nil** or **NULL** (PASCAL or C respectively). In some cases a sparse vector is added to a full one. For this purpose the addition operator is redefined: "add all entries in the sparse vector to their full vector counterparts".

TABLE A1.1. *Operator overview.*

operator	description	example
[]	array index	x[3]
.	direct record tag selection	*rec.tag*
→	indirect record tag selection	*rec_ptr→tag* is equivalent to (**rec_ptr*).*tag*
++	increment	$a++$
−−	decrement	$a--$
*	dereference	**ptr*
&	address	*&var*
*	multiplication	
/	division	
+	addition	
−	subtraction	
<	less than	
>	greater than	
≤	less or equal	
≥	greater or equal	
∩	set intersection	
∪	set union	
⊂	set inclusion	
∈	member	
∉	no member	
≡	equality	
≠	non equality	
!	logical not	
∧	logical and	
∨	logical or	
=	assignment	
+=	assignment	$a += b$ is equivalent to $a = a + b$
−=	assignment	$a -= b$ is equivalent to $a = a - b$

References

[1] Leon O. Chua and P.-M. Lin, *Computer Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques,* Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[2] W.J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation,* Kluwer Academic Publishers, Norwell, USA, 1988.

[3] W.T. Weeks, A.J. Jimenez, G.W. Mahoney, D. Mehta, H. Qassemzadeh, and T.R. Scott, "Algorithms for ASTAP - A Network Analysis Program," *IEEE Trans. on Circuit Theory*, vol. CT-20, no. 6, pp. 628-634, Nov. 1973.

[4] L.W. Nagel, "SPICE2: a Computer Program to Simulate Semiconductor Circuits," Memorandum No. ERL-M520, University of California, Berkeley, May 1975.

[5] B.R. Chawla, H.K. Gummel, and P. Kozak, "MOTIS - An MOS Timing Simulator," *IEEE Trans. on Circuits and Systems*, vol. CAS-22, no. 12, pp. 901-910, Dec. 1975.

[6] A.R. Newton, "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 9, pp. 741-749, Sept. 1979.

[7] H. de Man, G. Arnout, and P. Reynaert, "Mixed-Mode Circuit Simulation Techniques and Their Implementation in DIANA," in *Computer Design Aids for VLSI Circuits*, ed. H. de Man, pp. 113-174, Sijthoff & Noordhoff, Groningen, the Netherlands, 1980.

[8] E. Lelarasmee, A.E. Ruehli, and A.L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-1, no. 3, pp. 131-145, July 1982.

[9] J.K. White and A. Sangiovanni-Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits,* Kluwer Academic Publishers, Norwell, USA, 1987.

[10] W.M.G. van Bokhoven, "Piecewise-Linear Modelling and Analysis," Ph.D. Thesis, Eindhoven, The Netherlands, May 1981.

[11] Toshio Fujisawa, Ernest S. Kuh, and Tatsuo Ohtsuki, "A Sparse Matrix Method for Analysis of Piecewise-Linear Resistive Networks," *IEEE Trans. on Circuit Theory*, vol. 19, no. 6, pp. 571-584, Nov. 1972.

[12] J.T.J. van Eijndhoven, "Piecewise Linear Analysis," in *Analog Circuits: Computer Aided Analysis and Diagnosis*, ed. T. Ozawa, pp. 65-92, Marcel Dekker inc., New York, March 1988.

[13] J.T.J. van Eijndhoven, "A Piecewise Linear Simulator for Large Scale Integrated Circuits," Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, Dec. 1984.

[14] J.A.G. Jess, "Piecewise Linear Models for Nonlinear Dynamic Systems," *Frequenz*, vol. 42, no. 2/3, pp. 71-78.

[15] 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.

[16] L.O. Chua and R.L.P. Ying, "Canonical Piecewise-Linear Analysis," *IEEE Trans. on Circuits and Systems*, vol. CAS-30, no. 3, pp. 125-140, March 1983.

[17] L.O. Chua and A. Deng, "Canonical Piecewise-Linear Analysis: Part II - Tracing Driving-Point and Transfer Characteristics," *IEEE Trans. on Circuits and Systems*, vol. CAS-32, no. 5, pp. 417-444, May 1985.

[18] L.O. Chua and A. Deng, "Canonical Piecewise-Linear Modeling," *IEEE Trans. on Circuits and Systems*, vol. CAS-33, no. 5, pp. 511-525, May 1986.

[19] L.O. Chua and A. Deng, "Canonical Piecewise-Linear Analysis: Generalized Breakpoint Hopping Algorithm," *Circuit Theory and Applications*, vol. 14, pp. 35-52, 1986.

[20] M.T. van Stiphout, "PLATO Users Guide," Internal communication, 1989.

[21] B.W. Kernighan and D.M. Richie, *The C Programming Language,* Prentice Hall, 1978.

[22] R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986 .

[23] A. Lodder, M.T. van Stiphout, and J.T.J. van Eijndhoven, "The Eindhoven Schematic Editor," in *The Integrated Circuit Design Book*, ed. P. Dewilde, pp. 1.61-1.68, Delft University Press, Delft, The Netherlands, 1986.

[24] A. Lodder, M.T. van Stiphout, and J.T.J. van Eijndhoven, "ESCHER: Eindhoven Schematic EditoR - Reference Manual," EUT Report 86-E-157, Feb. 1986.

[25] G.L.J.M. Janssen, "Circuit Modelling and Animated Interactive Simulation in Escher+," in *Proceedings SCS European Simulation Multiconference, Simulation Applied to Manufactoring, Energy and Environmental Studies and Electronics and Computer Engineering*, ed. S.Tucci, A. Mathis, W. Hahn, R.N. Zobel, pp. 265-270, Rome, Italy, 7-9 June, 1989.

[26] Q.J.A. van Gemert and J.T.J. van Eindhoven, "The NDML Network Desrcription and Modeling Language - Reference manual," Internal communication, 1989.

[27] G.L.J.M. Janssen, "Network Description and Modeling Language - NDML," in *The Integrated Circuit Design Book*, ed. P. Dewilde, pp. 4.60-4.108, Delft University Press, Delft, The Netherlands, 1986.

[28] *OSF/Motif User's Guide*, 1989. Open Software Foundation, Eleven Cambridge Center, Cambridge MA 02142

[29] *OSF/Motif Programmer's Guide*, 1989. Open Software Foundation, Eleven Cambridge Center, Cambridge MA 02142

[30] J. Sherman and W.J. Morrison, "Adjustment of an Inverse Matrix Corresponding to Changes in the Elements of a Given Column or a Given Row of the Original Matrix," *Ann. Math. Stat.*, no. 20, p. 621, 1949.

[31] M. Woodbury, "Inverting Modified Matrices," Memorandum 42, Statistics Research Group Princeton, New Jersey, 1950.

[32] J.M. Bennett, "Triangular Factors of Modified Matrices," *Numerische Mathematik*, no. 7, pp. 217-221, 1965.

[33] A.L. Sangiovanni-Vincentelli, "Circuit Simulation," in *Computer Design Aids for VLSI Circuits: NATO Advanced Study Institute programme*, ed. P. Antognetty, D.O. Pederson and H. de Man, Sijthoff and Noordhoff, 1981.

[34] J.M. Ortega and W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables,* Academic Press, 1970.

[35] I.S. Duff, A.M. Erisman, C.W. Gear, and J.K. Reid, "Sparsity Structure and Gaussian Elimination," *ACM Signum newsletter*, no. 2, pp. 2-8, 1988.

[36] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146-160, June 1972.

[37] G. Kron, "A Set of Principles to Interconnect the Solutions of Physical Systems," *J. Appl. Phys.*, vol. 24, pp. 965-980, 1953.

[38] G. Kron, "A Method of Solving Very Large Physical Systems in Easy Stages," *Proc IRE*, vol. 42, pp. 680-686, 1954.

[39] Alberto Sangiovanni-Vincentelli, Li-Kuan Chen, and Leon O. Chua, "An Efficient Heuristic Cluster Algorithm for Tearing Large-Scale Networks," *IEEE Trans. on Circuits and Systems*, vol. CAS-24, no. 12, pp. 709-717, December 1977.

[40] Chien-Chih Chen and Yu-Hen Hu, "Parallel LU Factorization for Circuit Simulation on a MIMD Computer," in *Proc. of the ICCD88*, pp. 129-132.

[41] M. Vlach, "LU Decompostion Algorithms for Parallel and Vector Computation," in *Analog Methods for Computer-Aided Circuit Analysis and Diagnosis*, ed. Takao Ozawa, pp. 37-64, Ch. 2, Marcel Dekker, Inc., New York and Basel, 1988.

[42] M. Vlach, "LU Decomposition and Forward-Backward Substitution of Recursive Bordered Block Diagonal Matrices," in *Proc. Int. Symp. on Circuits and Systems*, pp. 427-430, Newport Beach, California, May 1983.

[43] M.T. van Stiphout and J.T.J. van Eijndhoven, "Design of a New Piecewise Linear Simulator," in *Proc. of the European Conference on Circuit Theory and Design*, vol. 1, pp. 107-112, Paris, France, Sept. 1-4, 1987.

[44] I.S. Duff, "Parallel Implementation of Multifrontal Schemes," *Parallel Computing*, no. 3, pp. 193-204, North-Holland, 1986.

[45] C.E. Lemke, "On Complementary Pivot Theory," in *Mathematics of the Decision Sciences - part 1*, ed. A.F. Veinott, Jr., Lectures in Applied Mathematics, vol. 11, pp. 95-114, American Mathematical Society, Providence, Rhode Island, 1968.

[46] J. Katzenelson, "An Algorithm for Solving Nonlinear Resistor Networks," *Bell Syst. Tech. J.*, vol. 44, pp. 1605-1620, 1965.

[47] S. Karamardian, "The Complementarity Problem," *Mathematical Programming*, vol. 2, pp. 107-129, 1972.

[48] C. van de Panne, "A Complementary Variant of Lemke's Method for the Linear Complementarity Problem," *Mathematical Programming*, vol. 7, pp. 283-310, North-Holland Publishing Company, 1974.

[49] D.A. Zein, "Solution of a Set of Nonlinear Algebraic Equations for General Purpose CAD Programs," in *Circuit Analysis, Simulation and Design*, ed. A.E. Ruehli, Advances in CAD for VLSI, vol. 3, part 1, pp. 207-234, North-Holland, 1986.

[50] H.R. Yeager and R.W. Dutton, "Improvement in Norm-Reducing Newton Methods for Circuit Simulation," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 5, pp. 538-546, May 1989.

[51] W.T. Weeks, A.J. Jimenez, G.W. Mahoney, H. Qassemzadeh, and T.R. Scott, "Network Analysis Using a Sparse Tableau with Tree Selection to Increase Sparseness," in *Proc. of the IEEE Int. Symp. on Circuit Theory*, pp. 165-168, Toronto, Canada, April 9-11, 1973.

[52] R.L. Bauer, J. Fang, A. P-C Ng, and R.K. Brayton, "XPSim: A MOS VLSI Simulator," in *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD-88)*, pp. 66-69, IEEE Computer Society Press, Washington, USA, Nov. 7-10, 1988.

[53] O. Nevanlinna and W. Liniger, "Contractive Methods for Stiff Differential Equations - Part I," *BIT*, vol. 18, pp. 457-474, 1978.

[54] H. Shichman, "Integration System of a Nonlinear Network Analysis Program," *IEEE Trans. on Circuit Theory*, vol. CT-17, no. 3, pp. 378-386, Aug. 1970.

[55] C.W. Gear, "The Automatic Integration of Ordinary Differential Equations," *Communications of the ACM*, vol. 14, no. 3, pp. 176-179, March 1971.

[56] R.K. Brayton, F.G. Gustavson, and G.D. Hachtel, "A New Efficient Algorithm for Solving Differential-Algebraic Systems Using Implicit Backward Differentiation Formulas," *Proceedings of the IEEE*, vol. 60, no. 1, pp. 98-108, Jan. 1972.

[57] G. Dahlquist, W. Liniger, and O. Nevanlinna, "Stability of Two-Step Methods for Variable Integration Steps," *SIAM Journal on Numerical Analysis*, vol. 20, no. 5, pp. 1071-1085, Oct.1983.

[58] F. Odeh and W. Liniger, "On A-Stability of Second-Order Two-Step Methods for Uniform and Variable Steps," in *Proc. of the IEEE Int. Conf. on Circuits and Computers (ICCC80)*, ed. G. Rabbat, vol. 1, pp. 123-126, Port Chester, New York, Oct. 1-3, 1980.

[59] W. Liniger, F. Odeh, and A. Ruehli, "Integration Methods for the Solution of Circuit Equations," in *Circuit Analysis, Simulation and Design*, ed. A.E. Ruehli, Advances in CAD for VLSI, vol. 2, pp. 235-279 (Ch.5), Elsevier Science Publishers B.V.(North-Holland), 1986.

[60] K.A. Sakallah and S.W. Director, "SAMSON: An Event Driven VLSI Circuit Simulator," in *Proc. of the IEEE 1984 Custom Integrated Circuits Conference*, pp. 226-231, Rochester, N.Y., May 21-23, 1984.

[61] K.A. Sakallah and S.W. Director, "SAMSON2: An Event Driven VLSI Circuit Simulator," *IEEE Trans. on Computer-Aided Design*, vol. CAD-4, no. 4, pp. 668-684, Oct. 1985.

[62] W.M.G. van Bokhoven, "Linear Implicit Differentiation Formulas of Variable Step and Order," *IEEE Trans. on Circuits and Systems*, vol. CAS-22, no. 2, pp. 109-115, Feb. 1975.

[63] J.F. Andrus, "Numerical Solution of Systems of Ordinary Differential Equations Separated into Subsystems," *SIAM Journal on Numerical Analysis*, vol. 16, no. 4, pp. 605-611, 1979.

[64] O.A. Palusinski, "Simulation of Dynamic Systems Using Partitioning and Multirate Integration Techniques," in *Proc. of the 1983 Summer Computer Simulation Conference*, pp. 54-59, Vancouver, B.C., Canada, 1983.

[65] J.F. Andrus, "Automatic Integration of Systems of Second-Order ODE's Separated into Subsystems," *SIAM Journal on Numerical Analysis*, vol. 20, no. 4, pp. 815-827, 1983.

[66] C.W. Gear, "Automatic Multirate Methods for Ordinary Differential Equations," in *Information Processing 80*, pp. 717-722, North-Holland Publishing Company, 1980.

[67] C.W. Gear and D.R. Wells, "Multirate Linear Multistep Methods," *BIT*, vol. 24, pp. 484-502, 1984.

[68] S. Skelboe, "Stability Properties of Linear Multirate Formulas," DIKU Report 86/13, 1986. Institute of Datalogy, University of Copenhagen

[69] S. Skelboe, "Stability Properties of Implicit Linear Multirate Formulas," in *Proc. of the European Conference on Circuit Theory and Design*, vol. 2, pp. 795-800, Paris, France, Sept. 1-4, 1987.

[70] S. Skelboe, "Stability Properties of Backward Differentiation Multirate Formulas," *Applied Numerical Mathematics*, no. 5, pp. 151-160, 1989.

[71] I.N. Hajj and S. Skelboe, "Time-Domain Analysis of Nonlinear Systems with Finite Number of Continuous Derivatives," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 5, pp. 297-303, May 1979.

[72] J.T.J. van Eijndhoven and M.T. van Stiphout, "Latency Exploitation in Circuit Simulation by Sparse Matrix Techniques," in *Proc. of the Int. Symp. on Circuits and Systems*, pp. 623-626, University of Helsinki, Espoo, Finland, June 7-9, 1988.

[73] O. Tejayadi and I.N. Hajj, "Dynamic Partitioning Method for Piecewise-Linear VLSI Circuit Simulation," *International Journal of Circuit Theory and Applications*, vol. 16, pp. 457-472, 1988.

[74] T.A. Last, "Proportional step size tracking analog-to-digital converter," *Rev. Sci. Instrum.*, vol. 51, no. 3, pp. 369-374, March 1980.

[75] J.J. Hopfield and D.W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, vol. 52, pp. 141-152, 1985.

[76] J.J. Hopfield and D.W. Tank, "Computing with Neural Circuits: A Model," *Science*, vol. 233, no. 4764, pp. 625-633, August 8 1986.

[77] J.B. Shackleford, "Neural Data Structures: Programming with Neurons," *Hewlett-Packard Journal*, vol. 40, no. 3, pp. 69-78, June 1989.

STELLINGEN

1. Aan de kwaliteit van de random procedure waarmee voorbeeld 2 in figuur 5 uit onderstaand artikel van Alturaigi en Bickart werd geconstrueerd dient ernstig te worden getwijfeld.

   Reordering a Sparse Matrix to a Sparse Blocked Form, M.A. Alturaigi en T.A. Bickart, Circuit Theory and Applications, vol 13, pp. 173-194, 1985.

2. Het bewijs van I.S. Duff voor stelling 3.1 uit dit proefschrift is fout.

   Direct methods for sparse matrices, I.S. Duff, A.M. Erisman and J.K. Reid, p. 272, Oxford, Clarendon Press.

3. In het hieronder vermelde artikel wordt aangetoond dat bij toepassing van een k-staps bdf integratieformule voor de numerieke integratie van een functie met q continue afgeleiden, $k \geq q$, de locale afbreekfout van de orde $q+1$ is. Voor pwl functies zou dit neerkomen op orde 2. Helaas is voor praktische pwl netwerken het aantal continue afgeleiden vaak 0 zodat bovenstaande conclusie geen praktische waarde heeft.

   I.N. Hajj en S. Skelboe, Time-domain analysis of nonlinear systems with finite number of continuous derivatives, IEEE Trans. on Circuits and Systems, Vol. CAS-26, No. 5, May 1979.

4. Na moeizaam overleg over de definitieve standaard en langdurige onderhandelingen over de interface naar diverse programmeertalen (de zogenaamde *language bindings*) moet toch worden geconstateerd dat de recente grafische standaard GKS (The Graphical Kernel System) als ouderwets en absoluut ontoereikend voor de moderne user interface technologie dient te worden gekwalificeerd (dit proefschrift).

5. Het X windows systeem voor window management is onoverzichtelijk, te groot, weinig orthogonaal, gulzig in geheugengebruik en cpu tijd. Het is dan ook in hoge mate onzeker of het zich als de facto windowing standaard voor unix systemen kan handhaven.

6. Het gebruik van lineaire lijsten in moderne programmeertalen is bijzonder onelegant en foutgevoelig.

7. Het is zo goed als zeker dat Hofstadters intuitie hem wat betreft artificial intelligence voor de tweede maal in de steek zal laten. *Gödel, Escher, Bach: an eternal golden braid, D.R. Hofstadter.*

8. De in Nederland bestaande scheiding tussen "gewone" en technische universiteiten is onnodig en leidt tot een verarming van de keuzemogelijkheden van de student en een eenzijdige geestelijke ontwikkeling.

9. Het zogenaamde "broodje gezond" dat in de meeste snackbars verkrijgbaar is geeft een indicatie omtrent de kwaliteit van het overige aldaar verkrijgbare voedsel.

10. Geheel ten onrechte treffen wij in de bijlagen van dag- en weekbladen een overvloed aan literatuur besprekingen naast een volstrekte verwaarlozing van nieuwe muzikale uitingen en opnamen.

11. In tijden waarin studenten worden geconfronteerd met rentedragende leningen getuigt het van een zeker gebrek aan compassie om maatregelen als die van de 267e faculteitsraad van de Faculteit der Electrotechniek d.d. 5-6-1989 besluit nr. 267.3 onderdeel 2.e voor te stellen en goed te keuren. (Het besluit behelst dat in het bedrijf afstuderende studenten alleen een vergoeding voor extra onkosten mogen krijgen en geen geldelijke beloning).