EINDHOVEN UNIVERSITY OF TECHNOLOGY

From circuit to signal : development of a piecewise linear simulator

Citation for published version (APA):
Buurman, H. W. (1993). *From circuit to signal : development of a piecewise linear simulator*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR392231

DOI:
10.6100/IR392231

Document status and date:
Published: 01/01/1993

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

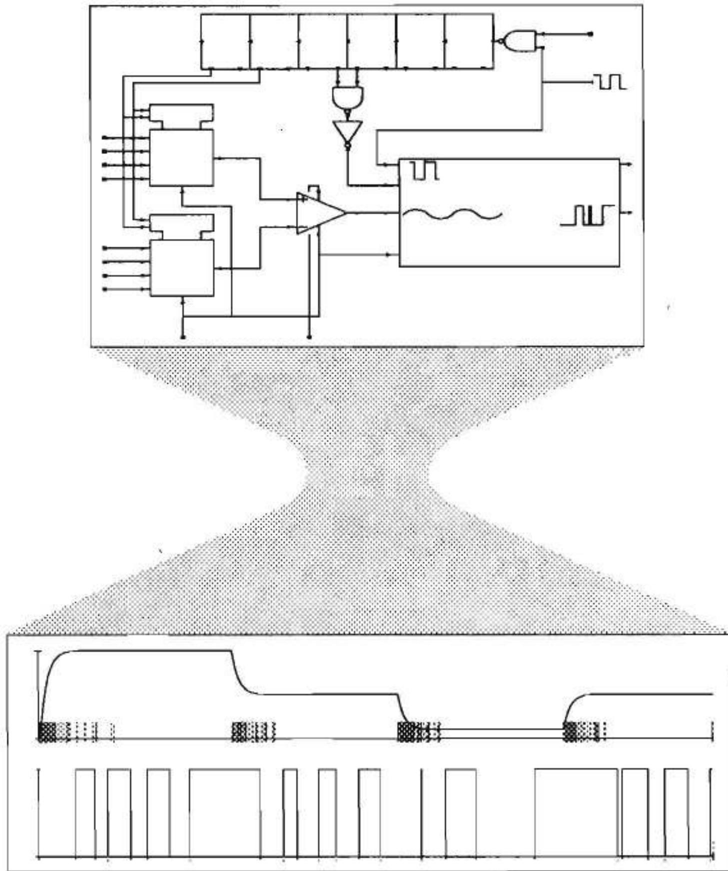
openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

From Circuit to Signal

development of a piecewise linear simulator



H.W. Buurman

From Circuit to Signal

development of a piecewise linear simulator

From Circuit to Signal

development of a piecewise linear simulator

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof. dr. J.H. van Lint,
voor een commissie aangewezen door het College
van Dekanen in het openbaar te verdedigen op
woensdag 20 januari 1993 om 16.00 uur

door

Hendrik Willem Buurman

geboren te Wageningen

Dit proefschrift is goedgekeurd
door de promotoren

prof. Dr.–Ing. J.A.G. Jess
prof. dr. ir. W.M.G. van Bokhoven

en de copromotor

dr. ir. J.T.J. van Eijndhoven

© Copyright 1993 H.W. Buurman
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from the copyright owner.

Druk: Wibro dissertatiedrukkerij, Helmond
Oplage: 200 exemplaren

CIP–GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Buurman, Hendrik Willem

From circuit to signal : development of a piecewise linear simulator /
Hendrik Willem Buurman – [S.l. : s.n.]. – Fig., tab.
Proefschrift Eindhoven. – Met lit. opg. – Met samenvatting in het Nederlands.
ISBN 90–9005713–7
NUGI 832
Trefw.: electronische schakelingen ; simulatie

Contents

Abstract .... **vii**

Samenvatting .... **ix**

Preface .... **xi**

1 Introduction **1**
1.1 Simulation 1
1.2 Modeling 2
1.3 PLATO 5
1.4 Overview 6

2 Piecewise Linear Modeling **7**
2.1 Introduction 7
2.2 The matrix model 8
2.3 Properties of the modeling 15
2.4 Existence of solutions 17
2.4.1 The linear equations 17
2.4.2 The piecewise linear equations 18
2.4.3 The dynamic problem 21
2.5 Creating piecewise linear models 22

3 The Linear Equations **31**
3.1 The data structure 31
3.2 Nodal analysis 35
3.3 The LU decomposition 38
3.4 The rank m update 40
3.5 Parallel capacitors 42

4 The Piecewise Linear Equations **45**
4.1 Introduction 45
4.2 Path–following algorithms 46
4.3 Implementation of the Van de Panne algorithm 53
4.4 The DC solution 55

5 **The Dynamic Equations** **59**
5.1 Introduction 59
5.2 Properties of integration methods 60
5.3 Implementation of the integration methods 62
5.4 Multirate integration 65
5.5 Exponential integration 70
5.6 The integration method and the Van de Panne algorithm. .. 73

6 **Aspects of the Implementation** **77**
6.1 Introduction 77
6.2 User environment 77
6.3 The event driven simulator 78
6.4 Sparse vectors by divided differences 81
6.5 Output processing 83
6.6 Numerical accuracy 84
6.7 Parallelism 86
6.8 Multilevel simulation and functional modeling 87

7 **Examples** **89**
7.1 Introduction 89
7.2 An analog–digital converter 89
7.3 A neural network simulator 93
7.3.1 Modeling a Hopfield neuron 95
7.3.2 PLANNET, a new neural net simulator 96
7.3.3 Results 97
7.4 A switch capacitor filter 99
7.5 Program statistics 102

8 **Conclusions** **105**

References **107**

Appendices **113**
A Mapping a four–segment function on a 2x2 matrix 113
B Notation 117

Biography **119**

Abstract

In this thesis the development of PLATO, a special circuit simulator, is sketched, with an accent on efficiency and applicability. The simulator is a mixed–level simulator, based on a general piecewise linear modeling technique, which allows uniform modeling for a broad class of components. A piecewise linear dynamic model consists of a matrix, relating three types of variables: linear, dynamic and complementary. The piecewise linear behavior is determined by the complementary variables and equations. Several properties of this modeling are given, and some examples of models are introduced.

The internal data structure of the simulator reflects the structure of electric and logic circuits, that are the basic circuits to be solved with PLATO. The linear equations are stored in a sparse matrix structure, while the dynamic and complementary equations are stored in the components of the circuit. The hierarchy of components is replaced by a list, connecting the global values with the components. This allows a considerable reduction of the size of a problem.

Each type of equation is solved with its own method: the linear equations with an LU decomposition, the complementarity problem by an algorithm devised by Van de Panne, and the dynamic equations by a numerical integration method. The interaction between these three basic methods, especially between the Van de Panne algorithm and the integration method, is explained.

To keep the LU decomposition up to date an efficient update is performed, which visits only those elements in the matrix that change value. The sparsity guarantees that this involves only few elements.

The achievement of the Van de Panne algorithm is excellent, as always a valid solution is found for the nonlinear equations. In contrary, traditional algorithms have convergence problems, or are not so broadly applicable on a wide range of components. The Van de Panne algorithm is also efficient, because only those components are visited that are related to the calculations.

Of the large number of known integration methods, several implicit methods with a low order are applied. To improve the efficiency, a multirate integra-

tion method is used. In this method, variables are grouped in several clusters based on their activity. Less active clusters are recalculated less often, so most effort is put in calculating the active components. These clusters are determined dynamically, so variables may shift to another cluster if their activity changes. An explicit integration method, based on the exponential behavior in time of the variables, is shown to be too instable to be applicable.

For reasons of efficiency and accuracy, only the updates on the variables are determined. To improve the efficiency further, these updates are not calculated directly, but again only their update is determined. This second update is a sparse vector whose calculation is easy. Because the actual values of the variables solved by the linear equations are not used any more, the output of the simulator is based on the update vectors also.

Several examples show the applicability of the simulator. An analog–digital converter shows the behavior of the simulator on a typical mixed–level circuit. A special vector–parallel version of the simulator is developed for neural networks, used in the solution of the Traveling Salesman Problem. A switch capacitor filter shows the behavior of the simulator on a circuit with strong dynamic properties.

Samenvatting

In deze dissertatie is de ontwikkeling geschetst van PLATO, een speciale circuit simulator, met de nadruk op de efficiëntie en toepasbaarheid van het programma. PLATO is een mixed–level simulator, gebaseerd op een algemene stuksgewijs–lineaire modelleringstechniek, waarmee een brede klasse van componenten beschreven kan worden. Een stuksgewijs–lineair dynamisch model bestaat uit een matrix die drie typen variabelen met elkaar relateert: lineaire, dynamische en complementaire variabelen. Het stuksgewijs–lineaire gedrag wordt bepaald door de complementaire variabelen en vergelijkingen. Verschillende eigenschappen van de modellering worden besproken, en enkele voorbeelden van modellen worden geïntroduceerd.

De interne data–structuur van de simulator is afgeleid van de structuur van electrische en logische circuits, de belangrijkste circuits die met PLATO gesimuleerd worden. De lineaire vergelijkingen zijn in een ijle matrix–structuur opgeslagen, terwijl de dynamische en complementaire vergelijkingen van de componenten in het circuit worden gebruikt. De hiërarchische opbouw van het circuit is vervangen door het gebruiken van een lijst, die de globale waarden koppelt aan de componenten. Dit maakt het mogelijk om de grootte van een probleem aanzienlijk te reduceren.

Ieder type vergelijking wordt opgelost met zijn eigen methodiek: de lineaire vergelijkingen via een LU–decompositie, het complementariteitsprobleem met een algoritme bedacht door Van de Panne, en de dynamische vergelijkingen met een numerieke integratiemethode. De interactie tussen deze drie basismethoden is beschreven, in het bijzonder die tussen het Van de Panne algoritme en de integratiemethode.

De LU–decompositie moet vaak veranderd worden. Dit gebeurt efficiënt door een algoritme waarmee alleen de te veranderen elementen bezocht worden. Dit zijn er slechts weinig door de ijlheid van het systeem.

Het Van de Panne algoritme geeft uitstekende resultaten, en vindt altijd een oplossing voor de niet–lineaire vergelijkingen. Op dit punt schieten veel traditionele algoritmen tekort en hebben problemen met de convergentie voor

zo'n grote klasse van componenten en circuits. Dit algoritme is ook efficiënt, aangezien slechts in die componenten wordt gerekend waarin iets verandert.

Van de vele mogelijke numerieke integratiemethoden worden enkele impliciete methoden met een lage orde toegepast. Om de efficiëntie te verbeteren wordt een zogenaamde multirate integratiemethode gebruikt. Hierin worden de variabelen gegroepeerd aan de hand van hun aktiviteit. Minder aktieve groepen worden minder vaak doorgerekend, dus de meeste moeite wordt gestoken in de berekening van aktieve componenten. De groepen worden dynamisch bepaald, zodat variabelen van groep veranderen als hun aktiviteit verandert. Een expliciete integratiemethode, gebaseerd op het exponentiële tijdsgedrag van de variabelen, is te instabiel om toe te passen.

De efficiëntie en nauwkeurigheid worden beter door alleen de veranderingen van de variabelen uit te rekenen. Om de efficiëntie verder te verhogen, worden die veranderingen niet direkt bepaald, maar worden de veranderingen van de veranderingen uitgerekend. Dit geeft een ijle vector die makkelijk te bepalen is. Omdat de eigenlijke waarden van de variabelen niet meer gebruikt worden, is de uitvoer van de simulator ook gebaseerd op de veranderingen in waarden.

Verschillende voorbeelden laten het toepassingsgebied van PLATO zien. Een analoog–digitaal omzetter illustreert het gedrag van de simulator voor een karakteristiek mixed–level circuit. Een speciale vector–parallel versie van de simulator voor neurale netwerken is ontwikkeld en toegepast om het handelsreizigersprobleem op te lossen. Een switch–capacitor filter maakt duidelijk dat de simulator ook een circuit met een sterk dynamisch gedrag goed kan uitrekenen.

Preface

This thesis describes the development of a piecewise linear simulator at the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology. This simulator has been developed for more than a decade, based on the work of Wim van Bokhoven, by Jos van Eijndhoven, Mart van Stiphout, myself, and our students. Because the different aspects of the problem, and their solutions, can not be understood without a description of the principles and basic algorithms used in the development of the simulator, this thesis contains a description of both these principles and algorithms, and the enhancements made by myself.

Acknowledgements

I would like to thank professor Jochen Jess for the opportunity he has given me to work in his group, and the foundations FOM (Foundation for Fundamental Research on Matter) and STW (Technology Foundation) for sponsoring my research under grant no. EEL 66.1206. I owe much to Jos van Eijndhoven and Mart van Stiphout, who have introduced me to the piecewise linear simulation and with whom I have co-operated with pleasure. Ruud, Kees, Arianne, Maxime, and Rudy have helped me with the implementation of the graphic programs. John has assisted in creating the neural network simulator PLANNET. The careful proofreading of Gjalt de Jong, prof. Mattheij, and prof. Talman has improved the clearness and readability of this thesis.

I would like to thank Hans Fleurkens for being a good–tempered roommate and companion. I would like to thank the other members of the group, in particular Michel Berkelaar, for explaining and helping me with various problems, and for creating a good atmosphere.

I would like to thank my family for their everlasting interest, and in particular Peter for supplying a picture of a spike pattern. Last of all, I owe much to my parents who have supported me throughout my study.

„Ik moet nog veel onderzoek doen",
zegt de promovendus.

Kees Fens, *De Volkskrant*, 29 feb. 1992

Aan mijn ouders

1 Introduction

1.1 Simulation

Simulation is the use of one or more models, whose behavior can be determined easily, in order to analyze the behavior of a system. This behavior is the reaction of the system on external and internal stimuli. Simulation can therefore be used to solve three types of problems: to check the correctness of newly designed systems, to predict the behavior of complex systems, or to compare several different systems.

After the design of a new system, it should be checked on design errors. Possible errors in the design can usually be found by examining the results of a simulation. Simulators (tools that perform simulations) are regularly used for this purpose by architects, electronic chips designers, aircraft builders, and hydraulic engineers. Of course, simulation is only interesting if (nearly) all errors can be found at a reasonable price.

A simulator can also be used to forecast the behavior of an existing system that is too complicated or too large for an accurate enough prediction of its behavior. Using a simulator, a better prediction may be found. The standard example is the weather forecast, of which the quality has clearly been improved since the meteorologists use supercomputers to calculate the weather.

A third use for a simulator is the exploration of the design space, to try to find a system that better suits the specifications of the desired behavior. If the system is complicated, a first design may not be satisfactory, and it may not be clear how to improve this design. Using variations on the first design may indicate whether it can be improved or not. Simulating these variations will be much easier and usually cheaper, and the best variant can be chosen.

Simulation can have several important disadvantages: it may be expensive, some errors in the design may not be found, and the results may be totally wrong. Errors may occur due to inaccurate modeling or to simulator specific errors. Therefore a user of a simulator should always keep in mind that the results may be of inferior quality. We like to advocate here that a simulator

(at least in the beginning) should be used only by experienced people, who must examine the results of the simulation carefully and who can judge whether these results are good or bad. Only when the simulator is tested for a range of well–known problems, a less experienced user could use the simulator confidently *on this type of problems*. In all other cases the user should have control over some global parameters to direct the simulator to a good approximation.

A simulator approximates the characteristics of one or more systems. It can be mechanical, e.g. a wind–tunnel with small models of airplanes, or it can be a program running on a computer. We are only interested in computer programs. The kind of systems that we simulate are in the first place VLSI circuits (chips). These are electronic circuits, either digital or mixed analog/digital of nature. Other kinds of systems are less suitable for our simulator, because it is tuned to these circuits, and will therefore not be discussed.

1.2 Modeling

A system is defined by its inputs and outputs, i.e. the system is a function from the space of inputs to the space of outputs, which may depend on the state of the system. We are almost always interested in time–dependent systems, for a finite time interval. The system itself is also finite (finite–dimensional input, output and internal state). The inputs and outputs are functions of the time. They usually represent some measurable physical entities, but we will not restrict ourselves to this. The values of the inputs are prescribed and do not depend on the system. They may be chosen by the designer, but they can also be the output of (a simulation of) an other system.

The first step in simulation is the creation of a mathematical description that approximates the system's functionality. This description is called the *model* of the system. Usually there are many descriptions possible, ranging from rather simple models to very complex ones. Which model is chosen depends on the following requirements:

1. Accuracy: a simple model of a system is usually less accurate than a complex model. This may also be applied in one system, where one (sub)function may be accurately enough modeled in one part of the system, but the same model may be too inaccurate in an other part.
2. Numerical stability: it may be necessary to replace a simple model by a complex model, *or vice versa*, to ensure that the simulator approximates the behavior of the system accurate enough.
3. Simulation time and computer resources: a simple model needs usually less time and less other resources than a more complicated model.

Complex systems can be built from (simpler) subsystems. The models taken for the subsystems form, together with the *topological* relations (connections) of the complex system, the model of the complex system. These subsystems may again contain subsystems, etc. Recursive relations between the systems

are forbidden, so a system may not contain itself as a subsystem. Such a hierarchical description of a system is usually shorter than a description with its hierarchy expanded. Many of its subsystems are identical or are characterized with a few parameters, so they are described only once.

The model of the basic components determines the characteristics of a simulator. Therefore a simulator is usually classified by the type of models it can handle. For electronic circuits there are several classes of simulators. They are ordered by the level of complexity and abstraction of the models, from low (physical and accurate) to high (logical and behavioral but inaccurate). In general, a higher level of modeling implies a faster simulation, and therefore allows a larger circuit to be handled, but yields less accurate results.

The lowest level of modeling is the *device* modeling. With this method each single component (transistor, MOSFET, etc.) is modeled by several equations that approximate the physical laws very accurately. At most a few components can be simulated together, because otherwise the simulation takes too much time. The behavior of a single component is calculated very accurately, but a circuit with hundreds of transistors is too large to simulate (Selberherr 1984).

The *electrical* level is the second level of modeling. Each component is modeled with several continuous nonlinear equations. These simulators calculate the behavior accurately, but for many circuits their results are poor. For digital or mixed digital/analog circuits the numerical stability is not guaranteed, and the simulator can easily fail. Also for too large circuits (more than 1000 components) they use too much computer resources to be of practical use. These simulators already exist a long time (the first ones appeared around 1965) and are sometimes called traditional simulators. The most famous ones of these simulators are SPICE (Nagel 1975) and ASTAP (Weeks *et al.* 1973).

Because these simulators are used intensively, new programs have been developed to improve the performance by applying modern techniques in modeling and simulation. Two of these programs, optimized for MOS simulation, are MOTIS (Chawla *et al.* 1975) and SPLICE (Newton 1979). A large class of simulators using the Waveform Relaxation technique have been developed in recent years, for example RELAX (Lelarasmee *et al.* 1982), SWAN (Dumlugol *et al.* 1987), and TOGGLE (Hsieh *et al.* 1985).

The third class of simulators are the *logic* simulators. They model the circuit with logical units, and are exclusively designed to simulate large digital circuits. Attempts to use multi–valued logic to simulate analog components too seem interesting, but lack a thorough numerical foundation. Also strongly coupled systems are not handled well. Examples of logic simulators are COSMOS (Bryant *et al.* 1987) and LDSIM (Krodel and Antreich 1990).

A subclass are the *switch level* simulators, that model transistor circuits as RC networks with ideal switches. Also for digital systems they behave well, but with analog components the simulation result depends on the type of cir-

cuit. Four of these simulators are BRASIL (Warmers *et al.* 1990), CHAMP (Saab *et al.* 1988), MOSSIM (Bryant 1984), and SLS (van Genderen and de Graaf 1986).

The *behavioral* simulators form the last class of VLSI simulators. They model the circuit on a high and abstract level. This is useful for exploring the design space at a high level. Because they can model large pieces of hardware in one component, it is possible to simulate multi–chip designs in one run. They are inherently unable to simulate analog circuits. Examples of these simulators are ESCHER$^+$ (Janssen 1989) and VHDL simulators (IEEE 1988).

In recent years, a new branch of simulators has been developed, the *mixed–level* simulators, specially suited for both analog and digital circuits. The reason for creating these simulators is that the traditional analog and digital simulators do not meet the requirements for state–of–the–art VLSI design. The increasing number of mixed analog/digital chips and the rapidly growing number of components on the chips imply that new techniques are necessary in simulation. The creation of new simulators able to simulate these chips is usually based on an existing simulator. Until now three different techniques have been developed to design a mixed–level simulator:

- Incorporating digital components in an analog simulator, without basically changing the solution methods.
- Incorporating analog components in a digital simulator. This can lead to a multi–valued logic simulator.
- Splitting the circuit in an analog and a digital part and using separate simulation algorithms for the different parts. This is called a mixed–mode simulator. The different algorithms may be implemented in two different simulators.

A basic problem in mixed–level simulation is the interface between digital and analog components. Whether both types are simulated apart or together does not really matter, because their interaction always introduces instabilities in the simulation process. In particular, when a discontinuity, originating from a digital component, is transferred to an analog component, the analog simulation algorithm must handle this carefully, to avoid incorrect simulation results. In general, the results will be better for problems that suit better the original simulator, i.e. a mostly analog problem solved by an originally analog simulator, a mostly digital problem by an originally digital simulator, and a clearly separated analog/digital problem by a mixed–mode simulator.

An extension of a mixed–level simulator is a multi–level simulator, which can also simulate parts of a circuit on the behavioral level. With this extension it is possible to use one simulator at each stage of the design process. It is then possible to predict the behavior of a system accurately, even before large parts are designed at the hardware level.

1.3 PLATO

In this thesis, we discuss the electronic circuit simulator called PLATO, an acronym for Piecewise Linear Analysis TOol. This simulator has been developed at the Eindhoven University of Technology during the last decade (van Eijndhoven 1984; van Stiphout *et al.* 1990; van Stiphout 1990). It is a genuine mixed–level simulator, based on a uniform modeling technique for all components: the piecewise linear modeling introduced in (van Bokhoven 1981). A broad class of models is available, ranging from models at the device level up to the register transfer level. On the lower bound the accuracy of the equations is the limiting factor, while on the upper bound the complexity of the components limits their modeling.

Piecewise linear modeling is not widely used in circuit simulation. Other research on piecewise linear models has been performed by Chua and others (Chua and Deng 1986; Kahlert and Chua 1990). However, because their set of models is a subset of our set of models, their techniques and algorithms are not directly suitable in the design of PLATO.
Recently, piecewise linear models have been used in the process of designing analog circuits, in the simulator PLANET (Leenaerts 1992). These are the same models as used in PLATO, but the algorithms used in PLANET do not allow employing all possible models.

The precision of a piecewise linear model is inherently lower than a model with nonlinear functions. Although it is possible to create accurate models, these are not used much, because they are cumbersome to define with piecewise linear functions. But the accuracy of the basic analog component models is sufficient to calculate the behavior of most analog system with nearly the same accuracy as an analog simulator.

As said before, the accuracy and the effort spent in the simulation process are directly related. So a fast simulation will usually be inaccurate, and an accurate simulation will take a long time. Because our simulator is a mixed–level simulator, its run times will be somewhere between those of an electrical level simulator and a digital simulator. A disadvantage is that it will be slower than the electrical level simulators on electrically modeled circuits and also slower than digital simulators on digitally modeled circuits. This is inevitable because our simulator is not optimized for these kinds of models.

Although the accuracy is not too high, the algorithms and techniques used internally in the simulator are accurate enough for all purposes. Many of these algorithms and techniques are also used in traditional or more recently developed analog circuit simulators. The important difference with these simulators is the piecewise linear modeling technique. The modeling and the algorithms allow a natural mixing of digital and analog components. Discontinuities do not pose problems, so our attitude towards them is: “We love them !”.

1.4 Overview

In this thesis the development of PLATO will be discussed. The intent is to give insight in the choices of the algorithms used in the creation of an efficient simulator. The basics of the piecewise modeling technique are introduced in Chapter 2. Several of its basic properties are derived and a theorem on the existence of solutions is proved. Some examples give an idea of the capabilities of the modeling. The models have three different types of variables and equations: linear, dynamic, and piecewise linear.

In Chapter 3, the basic data structures are introduced and the solution algorithm for the linear equations is discussed. The Van de Panne algorithm is presented in Chapter 4. This algorithm solves the piecewise linear equations. Also a method to find a DC solution is given. In Chapter 5, the solution of the dynamic equations is discussed and the multirate integration method is investigated. The combination of the Van de Panne algorithm and the integration method is also discussed here.

Several aspects of the implementation, to improve the efficiency and accuracy of the simulator, are considered in Chapter 6. Chapter 7 contains several examples of systems that have been simulated with PLATO. In Chapter 8 some conclusions are formulated and future extensions are discussed. A 'small' model for certain functions is derived in Appendix A. Some non standard notations are explained in Appendix B.

2 Piecewise Linear Modeling

2.1 Introduction

In this chapter the heart of our simulator is described, the piecewise linear modeling technique. We restrict ourselves to systems that can be modeled by an implicit nonlinear differential–algebraic equation $f(s(t), r(t)) = T\dot{s}(t)$, in which f is a function from $\mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$, $s(t)$ is the state vector of the system at time t with dimension n, $r(t)$ is a vector of inputs (external stimuli) at time t of dimension m, and the time t is in the interval $[t_0, t_e]$ with $t_0 < t_e$. $\dot{s}(t) = \frac{ds(t)}{dt}$, and the $n \times n$ matrix T, of the form $\begin{bmatrix} 0 & 0 \\ 0 & I \end{bmatrix}$ with I the identity matrix, describes the partition of the state vector s into an algebraic vector x and a dynamic vector u, with length n_x and n_u respectively. The algebraic and differential equations are split similarly to create the system:

$$\begin{cases} f_1(x(t), u(t), r(t)) & = & 0 \\ f_2(x(t), u(t), r(t)) & = & \dot{u}(t) \end{cases} . \tag{2.1}$$

The differential equation $f_2(x(t), u(t), r(t)) = \dot{u}(t)$ must have boundary or initial conditions before it can be solved, otherwise it is ill posed. Several conditions are possible, of which we generally choose the initial condition $u(t_0) = u_0$. For now, the inputvector $r(t)$ will be ignored; in Chapter 3 it is shown that adding this vector is straightforward. Furthermore, in the next chapters the explicit time dependency is in most places not shown, i.e. x, u, etc. is written instead of $x(t)$, $u(t)$.

Our models are restricted to piecewise linear mappings, defined by a finite set of pairs (R_i, L_i). Each R_i is a closed polygonal convex region, and each L_i is an implicit linear differential–algebraic equation defined on this region. The polygonal convex regions can be written in the form

$$R_i = \{ (x, u) \mid N_i \begin{pmatrix} x \\ u \end{pmatrix} + a_i \geq 0 \} , \tag{2.2}$$

with N_i a matrix of size $k_i \times n$, defining the k_i boundary hyperplanes of the region R_i. The intersection of a boundary plane with the region is called a *facet*. In general, all R_i have full dimension, and their union covers $\mathbb{R}^n$. In Section 2.3, exceptions to these assumptions are given. The linear equations L_i have the form

$$\begin{bmatrix} A_{xx}^{(i)} & A_{xu}^{(i)} \\ A_{ux}^{(i)} & A_{uu}^{(i)} \end{bmatrix} \begin{pmatrix} x \\ u \end{pmatrix} + \begin{bmatrix} b_x^{(i)} \\ b_u^{(i)} \end{bmatrix} = \begin{pmatrix} 0 \\ \dot{u} \end{pmatrix} . \tag{2.3}$$

Each submatrix, $A_{xx}^{(i)}$ etc., and each subvector, $b_x^{(i)}$ and $b_u^{(i)}$, have an appropriate fixed size, independent of the current region. All $A_{uu}^{(i)}$ are square ($n_u \times n_u$), and there are $p \le n_x$ linear equations ($A_{xx}^{(i)}$ is $p \times n_x$). If $p < n_x$, $n_x - p$ extra (input) equations must be added to the system to create a unique and solvable system.

Because most systems we are interested in have continuity properties, we consider only continuous mappings. Later it is shown that discontinuous relations are still possible in our modeling. Continuity is defined by the following properties:

- Each facet B of a region R_i is the intersection of this region with an adjacent region R_j, and B is also a facet of this adjacent region.
- The right hand side of the linear equations is continuous on common facets, i.e. $\begin{bmatrix} A_{xx}^{(i)} & A_{xu}^{(i)} \\ A_{ux}^{(i)} & A_{uu}^{(i)} \end{bmatrix} \begin{pmatrix} x \\ u \end{pmatrix} + \begin{bmatrix} b_x^{(i)} \\ b_u^{(i)} \end{bmatrix} = \begin{bmatrix} A_{xx}^{(j)} & A_{xu}^{(j)} \\ A_{ux}^{(j)} & A_{uu}^{(j)} \end{bmatrix} \begin{pmatrix} x \\ u \end{pmatrix} + \begin{bmatrix} b_x^{(j)} \\ b_u^{(j)} \end{bmatrix}$ for each $\begin{pmatrix} x \\ u \end{pmatrix}$ in $R_i \cap R_j$.

The continuity does not require that the intersection of two regions is restricted to their common boundary face, so two regions may (partially) overlap. Also the linear equations may differ in such an intersection.

2.2 The matrix model

The implementation of piecewise linear mappings must be considered carefully. Simply tabulating each region with its linear function will give many problems, because the number of regions increases exponentially when new functions and variables are added to the system. This can be deduced from the following example: consider a system with n variables. Suppose that one variable x_{n+1} with the following equation is added to the system: $x_{n+1} = |x_n|$. The state vector has one extra element and one global boundary ($x_n = 0$) is added to the system. The number of regions doubles, because each original region is split in two regions, in the nth dimension. So adding one nonlinear equation to the system at least doubles the number of regions. This exploding number of regions will give insurmountable problems even for moderately small systems.

To avoid this exploding number of regions and linearizations, a compact method must be found for storing all relevant information. The continuity of each mapping strongly reduces the degrees of freedom, as will be shown in this section for a special case. This low degree of freedom can be captured in a matrix model, that will be used in the simulator. Some properties of this modeling are discussed in the next section.
To simplify the equations, consider temporarily only linear equations with no u variables. In the next paragraphs, the relevant parameters are enumerated with indices indicating the region or their facets, e.g. n_{12} is the normal of the facet between adjacent regions R_1 and R_2, pointing into region R_1. The facet itself is denoted by B_{12}. All regions are full dimensional.

First, consider the linear equations of a mapping in two adjacent regions R_1 and R_2: $A_1\ x + a_1 = 0$ in R_1 and $A_2\ x + a_2 = 0$ in R_2. Let their common facet B_{12} be (part of) the hyperplane $n_{12}^T\ x + c_{12} = 0$. The continuity requires that $A_1\ x + a_1 = A_2\ x + a_2$ for each x with $n_{12}^T\ x + c_{12} = 0$.
This is only possible if there is a vector v_{12} with $A_1\ x + a_1 = A_2\ x + a_2 + v_{12}(n_{12}^T\ x + c_{12})$, i.e. $A_1 - A_2 = v_{12}\ n_{12}^T$ and $a_1 - a_2 = v_{12}\ c_{12}$. So the continuity implies that there is a vector v_{12} that, together with the boundary equation, determines the update of the linear equations from region R_1 to region R_2. This well–known fact has been described earlier, see for example (Chien and Kuh 1976; van Eijndhoven 1988). The update is a rank 1 update, i.e. the difference between the two matrices A_1 and A_2 is the product of two rank 1 matrices (vectors).

When R_1 has more than one facet, the continuity also restricts the degrees of freedom of a mapping. First consider a third region R_3, also adjacent to R_1, which has a non–empty intersection with R_2. Let the boundary hyperplane between R_1 and R_3 be $n_{13}^T\ x + c_{13} = 0$, with n_{13} linearly independent of n_{12}, and let the update of the linear equation on this plane be determined by a vector v_{13}. Assume that R_2 and R_3 do not overlap. See Figure 2.1 for an illustration of this topology.

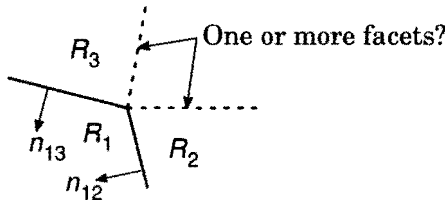


Figure 2.1. Topology with more than one facet

First notice that either R_2 and R_3 have different linearizations, or the linear equations are identical in all three regions. If the linearizations in R_2 and R_3 would be equal, the linear updates $v_{12}\ n_{12}^T$ and $v_{13}\ n_{13}^T$ are equal. Because n_{12} and n_{13} are linearly independent, this implies $v_{12} = v_{13} = 0$, so the updates vanish and the linear equations are equal in $R_1 \cup R_2 \cup R_3$.

The continuity of the modeling implies that R_2 and R_3 do not have a common facet, unless their updates are dependent, i.e. $v_{12} = \lambda v_{13}$ for a certain scalar λ. Then also their intersection is determined by λ. The following proof shows this: suppose $R_2 \cap R_3$ is part of the hyperplane $n_{23}^T\ x + c_{23} = 0$. The continuity of the linear equations implies that there exists a vector v_{23} with $v_{23}\ n_{23}^T = v_{13}\ n_{13}^T - v_{12}\ n_{12}^T$. Because n_{12} and n_{13} are linearly independent, this is only possible if $v_{12} = \lambda v_{13}$ for a certain λ. The conclusion can be drawn that, if v_{12} and v_{13} are linearly independent, R_2 and R_3 do not have a common facet. If v_{12} and v_{13} are linearly dependent, the mapping is fully determined by the region R_1 and its linear equations, the update v_{12}, and the factor λ.

Now suppose that two facets, B_{24} and B_{34}, are boundaries of a region R_4 separating R_2 and R_3, as depicted in Figure 2.2. Because $R_2 \cap R_3 \subset B_{34}$, the normal n_{34} is in the plane spanned by n_{12} and n_{13}, satisfying $n_{34} = \mu_{11} n_{12} + \mu_{12} n_{13}$ for certain scalars μ_{11} and μ_{12}. Similarly the normal n_{24} satisfies $n_{24} = \mu_{21} n_{12} + \mu_{22} n_{13}$ for certain μ_{21} and μ_{22}, and the scalars c_{24} and c_{34} must satisfy $c_{24} = \mu_{21} c_{12} + \mu_{22} c_{13}$ and $c_{34} = \mu_{11} c_{12} + \mu_{12} c_{13}$.

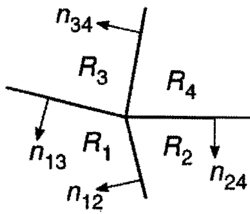


Figure 2.2. Topology with four facets

This topology imposes several restrictions on the scalars μ_{ij}. Consider the facet $B_{12} = \{ x \mid n_{12}^T\ x + c_{12} = 0 \wedge n_{13}^T\ x + c_{13} \geq 0 \}$. Let x be an interior point of B_{12} (i.e. $x \notin B_{13}$); x also satisfies $n_{24}^T\ x + c_{24} > 0$.
Now $n_{24}^T\ x + c_{24} = \mu_{21}(n_{12}^T\ x + c_{12}) + \mu_{22}(n_{13}^T\ x + c_{13}) = \mu_{22}(n_{13}^T\ x + c_{13})$.
So $n_{24}^T\ x + c_{24} > 0$ and $n_{13}^T\ x + c_{13} \geq 0$ together imply that $\mu_{22} > 0$.

Analogously one can prove that $\mu_{11} > 0$. Applying this reasoning on each of the facets B_{24} and B_{34} gives only one extra relation: $\mu_{11}\mu_{22} - \mu_{12}\mu_{21} > 0$. Because the magnitude of the vectors n_{24} and n_{34} is not important, we restrict the values of μ_{11} and μ_{22} to 1: $\mu_{11} = \mu_{22} = 1$. A geometric interpretation is that the facets B_{12} and B_{34}, when taken together, form one continuous boundary surface, which may bend on the crossing with another boundary surface. We call B_{34} an *extension* of B_{12} *over* B_{13}. Notice that B_{34} and B_{12} form one hyperplane if $\mu_{12} = 0$. The case $\mu_{11}\mu_{22} - \mu_{12}\mu_{21} = 0$ corresponds with a topology with only three facets, as described previously.

The linear equations on R_4 depend entirely on previously defined values. This is shown by comparing the update in R_4 over two different paths, one crossing B_{12} and B_{24}, the other crossing B_{13} and B_{34}. This gives the equation $(v_{12} + \mu_{21}v_{24})n_{12}^T + v_{24}\ n_{13}^T = v_{34}\ n_{12}^T + (v_{13} + \mu_{12}v_{34})n_{13}^T$. Because n_{12} and n_{13} are linearly independent, the vectors v_{24} and v_{34} can both be expressed in previously defined values: $v_{24} = (\mu_{12}v_{12} + v_{13})/d$ and $v_{34} = (v_{12} + \mu_{21}v_{13})/d$, where $d = 1 - \mu_{12}\mu_{21}$. As shown above, $d > 0$ in this topology. The conclusion is that a continuous piecewise linear function in four regions depends on the linearization and boundaries of one region plus the update vectors v_{12} and v_{13} and the factors μ_{ij}.

These properties of a piecewise linear continuous mapping can be extended if more facets of R_1 meet in one corner of the region. The following theorem describes the degree of freedom in such a situation, with k facets with linearly independent normals and k linearly independent updates on the linearization. First a *principal submatrix* and a *principal minor* are defined: a principal submatrix with index set α is $M_{\alpha\alpha} = \left(M_{ij}\right)$ for $i,j \in \alpha$, and a principal minor is the determinant of a principal submatrix, $\det(M_{\alpha\alpha})$. Notice that the positiveness of the factors μ_{11}, μ_{22} and d in the case of four regions is equivalent to: all principal minors of the matrix $\begin{bmatrix}\mu_{11} & \mu_{12} \\ \mu_{21} & \mu_{22}\end{bmatrix}$ are positive.

Theorem 2.1:

Let a region R_1 be given, with k facets B_i, each part of a hyperplane $\{\ x \mid n_i^T\ x + c_i = 0\ \}$, with linearly independent normals n_i. Let also a linearization on R_1 be given, and k linearly independent vectors v_i, that define the rank 1 updates across the B_i. Let k^2 scalars μ_{ij} be given, with all $\mu_{ii} = 1$, defining the $k^2 - k$ facets that are found by combining two

facets, as described previously. Suppose that each principal minor of the matrix (μ_{ij}) is positive. Then a continuous piecewise linear mapping with 2^k regions and a linearization on these regions is uniquely defined by these vectors and scalars.

Proof:

For $k = 1$ and $k = 2$ this theorem has already been proven in the previous paragraphs. The proof will be given with complete induction, so suppose that the theorem is valid for given $k - 1,\ k \geq 3$.

First, create from the first $k - 1$ facets and their $(k - 1)^2$ parameters μ_{ij} the topology with 2^{k-1} regions. Add the last facet hyperplane to the system, $n_k^T x + c_k = 0$. This facet determines, together with a given update vector v_k, the linearization on both of its sides. It can be extended over each of the other facet of R_1, because its normal is independent of the other facet normals. On each of these intersections, two parameters determine the extensions of the related facets and the linearizations in the newly defined regions. These parameters are the scalars μ_{ik} and μ_{ki}, $i = 1,\ldots,k - 1$ with the special element $\mu_{kk} = 1$.

Because the normal of B_k is independent of the other normals, and the principal minors of the matrix (μ_{ij}) are positive, the normals of the extensions of B_k are also independent of all other normals of facets. Therefore these extensions intersect the other facets, and each region of the $k - 1$ topology is split in two regions by an extension of B_k. So the new topology contains 2^k regions.

On each intersection of one of the original facets and an extension of B_k, two parameters determine the extensions of these two facets. But these parameters depend on the already given parameters μ_{ij}, except for the facets of R_1, as will be shown now.

Consider one of these intersections, say of the extension of B_2 over B_1 with the extension of B_k over B_1. Because the new extension of B_k has already been determined, one of the parameters can not be chosen freely. Now this extension is also found by another path, crossing first B_2 and then B_1. In both ways, an update on the linearizations over this facet is determined, depending on one parameter each. Both updates must be identical, which implies that the other parameters are also dependent on the parameters μ_{ij}. Concluding, the linearizations in the regions that can be found by crossing no more than two facets have already been determined. It is clear that the extension of B_k over the other facets is also determined by the given parameters and vectors.

□

Corollary:

If the vectors v_i are not linearly independent of each other, Theorem 2.1 is still valid.

Proof:

The topology of the regions is determined only by the normals n_i and the parameters μ_{ij}. □

This theorem shows that the degrees of freedom in a simple topology are only dependent on the facet equations of one region, the linearization in that region, the updates on the linearization after crossing a facet, and the bends of the facets. For more complex topologies, in which the facet normals are not all linearly independent or with overlapping regions, the freedom in the mapping can not yet be determined. Also the freedom in a topology with a facet that has not a common point with all other previously given facet is not known.

With this theorem in mind, a matrix model with four submatrices is introduced. The first submatrix is the linearization in a region R_1. A second submatrix contains all facet normals of the mapping, i.e. n_{12} and n_{13} in the previous topology of four facets. Not only the facets of R_1, but also normals of other facets, trivially satisfied in R_1, are in this matrix. The update vectors v_i related to the facets, form the third submatrix of the model. The last part of the matrix contains the mutual influence factors μ_{ij}. However, the values of these factors are not restricted in any way. The implications of this freedom will be given in the next section.

This modeling is shown with the previously discussed topology with four facets. The composite matrix/vector system is described by:

$$\begin{bmatrix} A_1 & v_{12} & v_{13} \\ n_{12}^T & 1 & \mu_{12} \\ n_{13}^T & \mu_{21} & 1 \end{bmatrix} \begin{bmatrix} x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} a_1 \\ c_{12} \\ c_{13} \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.4}$$

The four variables z_1, z_2 and w_1, w_2 are introduced to simplify the next reasoning, and play an important role in the subsequent sections and chapters. First notice that the choice $z_1 = z_2 = 0$ gives the linearization of region R_1. Then the boundary inequalities of R_1 are equivalent to $w_1 \geq 0$ and $w_2 \geq 0$.

Now z_1 can be expressed in terms of w_1, z_2 and x:

$$-n_{12}^T x + w_1 - \mu_{12} z_2 - c_{12} = z_1 , \tag{2.5}$$

and z_1 can be eliminated in the linear equation and in the last row of (2.4):

$$\begin{aligned} (A_1 - v_{12}n_{12}^T)\, x + v_{12}w_1 + (v_{13} - v_{12}\mu_{12})\, z_2 + a_1 - v_{12}c_{12} &= 0 \\ (n_{13}^T - \mu_{21}n_{12}^T)\, x + \mu_{21}w_1 + (1 - \mu_{21}\mu_{12})\, z_2 + c_{13} - \mu_{21}c_{12} &= w_2 . \end{aligned} \tag{2.6}$$

Both equations can be combined in:

$$\begin{bmatrix} A_2 & v_{12} & \lambda v_{24} \\ n_{21}^T & 1 & -\mu_{12} \\ n_{24}^T & \mu_{21} & \lambda \end{bmatrix} \begin{bmatrix} x \\ w_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} a_2 \\ c_{21} \\ c_{24} \end{bmatrix} = \begin{bmatrix} 0 \\ z_1 \\ w_2 \end{bmatrix} . \tag{2.7}$$

This is the same system as (2.4), with z_1 and w_1 swapped. Now the choice $w_1 = z_2 = 0$ together with $z_1 \geq 0$ and $w_2 \geq 0$ describes the linearization and facets of R_2. Notice that the last equation in (2.4) denotes B_{13}, and the last equation in (2.7) denotes B_{24}, the extension of B_{13}.

Analogously, the two other regions can be derived with their choice of z_1, z_2, w_1 and w_2: R_3 has $z_1 = w_2 = 0$ and $w_1 \geq 0$, $z_2 \geq 0$, and R_4 has $w_1 = w_2 = 0$ and $z_1 \geq 0$, $z_2 \geq 0$. The process of swapping entries of the z and w vectors is called *pivoting* or *performing a pivot*. It is in fact calculating a partial inverse of the matrix. The diagonal entry of the matrix related to pivoting is called the *pivot*. Each region can be found by pivoting on a number of diagonal entries. Earlier in this section, it has been proven that in this configuration of four different non–overlapping regions all possible pivots are *positive*. Notice that each region satisfies

$$\begin{cases} w \geq 0 \\ z \geq 0 \\ w \cdot z = 0 \end{cases} , \tag{2.8}$$

where $w \geq 0$ means $\forall_i : w_i \geq 0$. So w and z are *complementary*: if $z_i = 0$ then $w_i \geq 0$ and vice versa. Each region is fully determined by the set $\alpha = \{ i \mid w_i = 0 \}$ and the piecewise linear mapping is then fully described by equation (2.4) and the complementarity relations (2.8).

The general form of our piecewise linear model contains also the vector u to model the dynamics of the system. Both x and z depend on the dynamic variable u, and $\dot{u}$ depends on x and the piecewise linear vector z. So a piecewise linear differential equation for u is introduced this way. The general model has the following form:

$$\begin{cases} \begin{bmatrix} A_{xx} & A_{xu} & A_{xz} \\ A_{ux} & A_{uu} & A_{ux} \\ A_{zx} & A_{zu} & A_{zz} \end{bmatrix} \begin{bmatrix} x(t) \\ u(t) \\ z(t) \end{bmatrix} + \begin{bmatrix} a_x \\ a_u \\ a_z \end{bmatrix} = \begin{bmatrix} 0 \\ \dot{u}(t) \\ w(t) \end{bmatrix} \\ w(t) \geq 0, \; z(t) \geq 0, \; w(t) \cdot z(t) = 0 \\ u(t_0) = u_0 \end{cases} . \tag{2.9}$$

Both the w and the z variables (equations) are called the piecewise linear or *pl* variables (equations). The x variables (equations) are called the *linear* variables (equations), and the u variables (equations) are called the *dynamic* variables (equations).

2.3 Properties of the modeling

In equation (2.9) a system is introduced, of which in this section some general properties are given.

First, this modeling can be associated with an already known problem. Consider only the pl equations and the complementary relations. Introducing $M = A_{zz}$ and $q = A_{zx}\,x + A_{zu}\,u + a_z$ gives the following relations:

$$\begin{cases} w = Mz + q \\ w \geq 0,\ z \geq 0,\ w \cdot z = 0 \end{cases} . \tag{2.10}$$

This problem is known in the area of Operations Research as the Linear Complementarity Problem (LCP). It is related to the linearly constrained quadratic optimization problem: that problem transforms to an LCP by applying the Kuhn–Tucker conditions. For the LCP several existence theorems are known, as well as many algorithms to solve the problem. One of these algorithms will be applied to solve equation (2.9).

It is easy to see that instead of performing several different pivots one at a time, one pivot with the submatrix $M_{\alpha\alpha}$ can be performed, where $M_{\alpha\alpha} = \left(M_{ij}\right)$, $i, j \in \alpha$, and α is the set of pivots. This is called a block pivot, contrary to a single pivot on one diagonal element. The sign of such a block pivot is the sign of the determinant of $M_{\alpha\alpha}$. If all single pivots of α are nonzero, this sign is the product of the signs of the single pivots. Notice that this sign needs not to be the product of the diagonal entries of $M_{\alpha\alpha}$.

Some other obvious properties are that the number of facets of a region is maximally n_z, the length of the vectors w and z, and that the maximal number of regions is 2^{n_z}. There can be less regions, in particular when regions have less than n_z facets. The continuity of the mapping is simply deduced from the continuity of the z and w vectors across the facets.

An other interesting property is that the boundary inequalities are not defined uniquely. An inequality relation can be updated with a linear equation:

$$(A_{zx} + L\,A_{xx})\,x + (A_{zz} + L\,A_{xz})\,z + a_z + La_x = w\ , \tag{2.11}$$

in which L is a arbitrary matrix. This means that the facet can change, but that then also the LCP changes. With this method one or more x variables can be eliminated from A_{zx}. These variables can be considered as output variables, depending on the other x variables. This dependency can then be investigated more directly.

A second interesting possibility is the diagonalization of A_{zz}. If L can be chosen so that the new A_{zz} is a diagonal matrix, the new facets, described by $((A_{zx} + L\,A_{xx})\,x + a_z + La_x)_i = 0$, have an independent topology, i.e. the facets do not bend over each other. In this way, the influence of two parallel facets on each other can sometimes be explained.

In the previous section, the pivot was introduced in the derivation of the matrix model. It is related to the relative position of the two involved neighboring regions, as is shown now. Introduce $\alpha = \{i\}$ and $\bar{\alpha}$ as the complement of α, i.e. $\bar{\alpha} = \{1,\ldots,n_z\}\backslash\alpha$. Consider again the basic problem with no dynamic variables. The pl equations after a nonzero pivot on M_{ii} are:

$$\begin{pmatrix} z_i \\ w_{\bar{\alpha}} \end{pmatrix} = \begin{bmatrix} M'_{ii} & M'_{i\bar{\alpha}} \\ M'_{\bar{\alpha}i} & M'_{\bar{\alpha}\bar{\alpha}} \end{bmatrix} \begin{pmatrix} w_i \\ z_{\bar{\alpha}} \end{pmatrix} + \begin{bmatrix} q'_i \\ q'_{\bar{\alpha}} \end{bmatrix}, \tag{2.12}$$

with

$$\begin{cases} M'_{ii} = M_{ii}^{-1} \\ M'_{\bar{\alpha}i} = M_{\bar{\alpha}i} M_{ii}^{-1} \\ M'_{i\bar{\alpha}} = -M_{ii}^{-1} M_{i\bar{\alpha}} \\ M'_{\bar{\alpha}\bar{\alpha}} = M_{\bar{\alpha}\bar{\alpha}} - M_{\bar{\alpha}i} M_{ii}^{-1} M_{i\bar{\alpha}} \end{cases}$$

and

$$\begin{cases} q'_i = -M_{ii}^{-1} q_i \\ q'_{\bar{\alpha}} = q_{\bar{\alpha}} - M_{\bar{\alpha}i} M_{ii}^{-1} q_i \end{cases},$$

where $q_p = (A_{zx}\, x + A_{zu}\, u + a_z)_p$ for $q = i,\ \bar{\alpha}$.

There are three possibilities: the pivot is positive, negative, or it is zero and can not be performed. For a positive pivot, q'_i is positive if q_i is negative, i.e. x is on the other side of the facet. So both regions (in the x–space) are on different sides of their common facet. But if the pivot is negative, q'_i is positive when q_i is positive. This means that both x–regions are on the same side of the facet and overlap. There are two linearizations defined in the same (sub–) region. Which linearization will be chosen depends on criteria used to solve the problem. As will be explained in a later section, two negative pivots can be used to describe a model with a hysteresis behavior (see Figure 2.6.c).

The third possibility is different: the pivot, M_{ii}, is zero and can not be performed. This is the situation:

$$\begin{cases} \begin{bmatrix} 0 & M_{i\bar{\alpha}} \\ M_{\bar{\alpha}i} & M_{\bar{\alpha}\bar{\alpha}} \end{bmatrix} \begin{pmatrix} z_i \\ z_{\bar{\alpha}} \end{pmatrix} + \begin{pmatrix} q_i \\ q_{\bar{\alpha}} \end{pmatrix} = \begin{pmatrix} w_i \\ w_{\bar{\alpha}} \end{pmatrix} \\ w_i = 0,\ w_{\bar{\alpha}} \geq 0,\ z_i \geq 0,\ z_{\bar{\alpha}} = 0 \end{cases}. \tag{2.13}$$

The region described by (2.13) is:

$$\begin{cases} q_i = 0 \\ z_i \geq 0 \\ q_{\bar{\alpha}} + M_{\bar{\alpha}i}\, z_i \geq 0 \end{cases}. \tag{2.14}$$

This region is part of the facet hyperplane $q_i = 0$. The variable z_i is not directly solvable from the system. The number of inequalities remains the

same, while an extra linear equation is added to the system. In this case the region is not full dimensional. When a larger set $\beta \supset \alpha$ can be found so that $M_{\beta\beta}$ is regular, a block pivot with this matrix can be performed. The newly found region is no longer a part of this facet hyperplane, and is full dimensional. The sign of the block pivot once again determines on which side of the facet hyperplane the next region is situated.

Considering this problem in the full system (with x– and u–variables), equation (2.14) can be interpreted with stating that one inequality is transformed in an equality ($q_i = 0$), and that (when there is a j with $(A_{xz})_{ji} \neq 0$) one equality can be chosen that is transformed in an inequality: an inequality is exchanged with an equality. The nett effect of a zero pivot is a discontinuity with respect to this hyperplane. So with a continuous piecewise linear *mapping* a discontinuous piecewise linear *function* can be modeled. The conclusion is that, because the pivots are not limited to be positive, discontinuities can be modeled with a zero pivot.

2.4 Existence of solutions

Consider the full system

$$\left\{\begin{array}{l} \begin{bmatrix} A_{xx} & A_{xu} & A_{xz} \\ A_{ux} & A_{uu} & A_{uz} \\ A_{zx} & A_{zu} & A_{zz} \end{bmatrix} \begin{pmatrix} x \\ u \\ z \end{pmatrix} + \begin{pmatrix} a_x \\ a_u \\ a_z \end{pmatrix} = \begin{pmatrix} 0 \\ \dot{u} \\ w \end{pmatrix} \\ w \geq 0, \;\; z \geq 0, \;\; w \cdot z = 0 \\ u(t_0) = u_0 \end{array}\right. . \qquad (2.15)$$

The problem of determining criteria for the existence of a solution is split in three parts, related to interpretation of the system at three different levels. Firstly it is a linear equation in x, depending on u and z. Secondly it is a differential equation in u, depending on x and z. Thirdly it is an LCP in w and z, depending on x and u. The existence problem is also considered on these three levels.

2.4.1 The linear equations

Notice that x can be solved only when A_{xx} is regular. This is not a strong condition, because A_{xx} represents the current linearization of the x–part of the system. In general, a singular A_{xx} means that the system specification is either incomplete or overdetermined. If however a solution exists when A_{xx} is singular, some elements of x are determined by the LCP or dynamic equations and some elements of z or u by the linear equations. This problem is discussed in Section 3.5 about parallel capacitors, where a heuristic algorithm is described to find a solution for a special case.

With A_{xx} regular, x can be eliminated from the other equations in (2.15):

$$\begin{cases} \begin{bmatrix} A'_{uu} & A'_{uz} \\ A'_{zu} & A'_{zz} \end{bmatrix} \begin{pmatrix} u \\ z \end{pmatrix} + \begin{pmatrix} a'_u \\ a'_z \end{pmatrix} = \begin{pmatrix} \dot{u} \\ w \end{pmatrix} \\ w \geq 0, \ z \geq 0, \ w \cdot z = 0 \\ u(t_0) = u_0 \end{cases}, \tag{2.16}$$

where $A'_{pq} = A_{pq} - A_{px} A_{xx}^{-1} A_{xq}$ and $a'_p = a_p - A_{px} A_{xx}^{-1} a_x$ for $p, q = u, z$. These matrices are independent of any update on the LCP or dynamic equations with a linear equation, as described in the previous section. Therefore several representations, with different pl or dynamic equations, may describe the same problem.

Equation (2.16) can also be viewed on two levels. It is a differential equation in u depending on z (a piecewise linear differential equation), or an LCP in w and z depending on u (a dynamic LCP).

2.4.2 The piecewise linear equations

A criterion must be found that establishes the solvability of the LCP for all reached values of u. First, some remarks are made about the more general requirement that there is a solution for all q, with q defined as in (2.10). There is a vast amount of articles written about this problem, see for example (Cottle and Stone 1983; Doverspike and Lemke 1982; van Eijndhoven 1984). The conditions, under which a solution exists for all q, all depend on properties of the matrix $A'_{zz} = A_{zz} - A_{zx} A_{xx}^{-1} A_{xz}$. The matrices A_{zz}, A_{zx}, and A_{xz} are usually composed from a given set (see the next chapter), but the matrix A_{xx} varies unpredictable for different problems. Therefore the properties of A'_{zz} can not be (easily) validated. In (van Bokhoven 1981), some conditions on electrical systems are given under which a unique solution for a static problem is found. However, many complex and interesting systems do not satisfy these conditions.

In many cases, the term $A_{xz}\ z$ reaches only a subspace of all possible vectors. The existence of a solution can sometimes be derived from considering this property. With the following theorem, a criterion is presented for linking the solvability of a system to the mapping of this subspace. The idea is that the updates on the linearizations are in a way bounded. Only the linear part must be bounded, the dynamic variables have more freedom. See Figure 2.3 for a graphical interpretation of this theorem.

Theorem 2.2:

If there exist a matrix B and a number c, such that each x, z and w that satisfy for each u the LCP

$$\begin{cases} A_{zz}\ z + A_{zx}\ x + A_{zu}\ u + a_z = w \\ w \geq 0, \ z \geq 0, \ w \cdot z = 0 \end{cases} \tag{i}$$

also satisfy $\| A_{xz}\ z - B\ x \| \leq c$, then, for any regular A_{xx} with $A_{xx} + B$

also regular, and for any A_{xu}, any a_x, and all u, the system

$$\begin{cases} \begin{bmatrix} A_{xx} & A_{xu} & A_{xz} \\ A_{zx} & A_{zu} & A_{zz} \end{bmatrix} \begin{bmatrix} x \\ u \\ z \end{bmatrix} + \begin{pmatrix} a_x \\ a_z \end{pmatrix} = \begin{pmatrix} 0 \\ w \end{pmatrix} \\ w \geq 0, \quad z \geq 0, \quad w \cdot z = 0 \end{cases}$$

has a solution.

Proof:

Let $d = \| A_{xx}^{-1} \| \; c$. Let u be fixed, and let $a_x(u) = A_{xu}\, u + a_x$ and $a_z(u) = A_{zu}\, u + a_z$. Define three surfaces in $\mathbb{R}^{2n}$:
$V = \{(x, y) \mid \exists_z \exists_w : y = -A_{xx}^{-1}\,(A_{xz}\, z + a_x(u)) \wedge (x, w, z)$ satisfies (i)$\}$,
$V' = \{ (x, y) \mid y = -A_{xx}^{-1}\,(B\, x + a_x(u)) \}$, and $W = \{ (x, y) \mid y = x \}$.
For each point p in V there exists a point q in V' with $\| p - q \| \leq d$ (take the same x–coordinate for p and q, then automatically $\| p - q \| \leq d$).
$W \cap V'$ has one point, because $(x, y) \in W \cap V'$ implies $y = x = -A_{xx}^{-1}\,(B\, x + a_x(u))$, or $(A_{xx} + B)\; x = -a_x(u)$, which has exactly one solution.
Because V is a continuous surface, there must be at least one point (x,y) in $W \cap V$. This point satisfies $y = x = -A_{xx}^{-1}\,(A_{xz}\, z + a_x(u))$. The x–coordinate of this point is a solution of the problem.

□

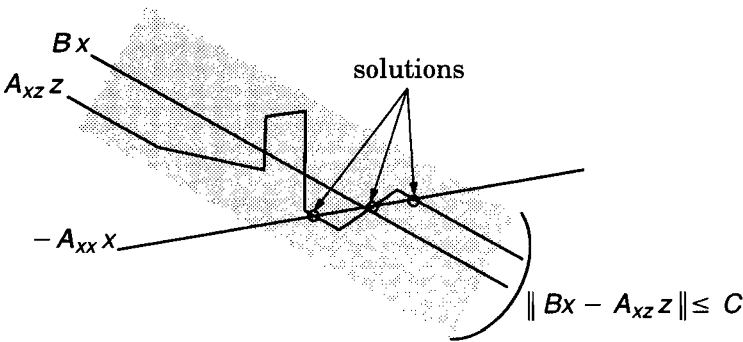


Figure 2.3. Graphical interpretation of Theorem 2.2

Corollary:

If the matrix A_{zz} is block–diagonal, i.e. $A_{zz} = \mathrm{diag}((A_{zz})_1, \ldots, (A_{zz})_r)$, and each triple $\left((A_{zz})_i, (A_{zx})_i, (A_{xz})_i\right)$, $i = 1, \ldots, r$ satisfies the conditions of Theorem 2.2, then A_{zz} satisfies the conditions of Theorem 2.2.

As will be explained in Chapter 3, the A_{zz} matrix is block–diagonal, with each block originating from one (electrical) component. So the corollary states that it is enough to test the theorem for each component or block of components.

There are several types of components and circuits that satisfy Theorem 2.2. First, logic components have their output bounded between 0 and 1, so for all logic components the theorem is satisfied, independent of the chosen model. A second class of components, satisfying the conditions, contains all linear components, and all components with $A_{zx} = 0$ or $A_{xz} = 0$. These are components like resistors, linear sources, and capacitors. Especially nonlinear capacitors, of which the voltage is a function of dynamic variables only, fall into this category. So circuits containing only these components will always have a solution, provided the system's matrix A_{xx} is regular.

A third class of circuits that satisfy the conditions of the theorem are systems where only certain combinations of components are allowed. For example transistor circuits, where each node has a capacitor connected to ground and consequently for each transistor current an explicit equation is defined in the transistor model, have always a solution. The applicability of the theorem is in these cases based on the *dynamics* of the system: at the current time point a solution is found, and at the next time point a new solution is also found. This is observed several times: if a system is modeled with many static components (components with no u variables), and the system contains a feedback loop, it is possible that no solution exists or that no solution can be found. When some critical components are remodeled so their behavior is dynamic, a solution can be found more easily.

Some components do clearly not satisfy the conditions of the theorem, and so may introduce a non–solvable problem. For example, the ideal diode, a major electronic device, can be in two states: $v = 0 \wedge i \geq 0$, or $v \leq 0 \wedge i = 0$, where v and i are elements of the x vector. This model contains only one z variable, and does not satisfy the conditions of the theorem. A circuit that has no solution is simply found (Figure 2.4). The circuit with an ideal voltage source and an ideal diode is solvable if V_0, the output of the voltage source, is negative. If it is positive, there is no solution. So the existence of a solution depends on the state of the circuit.

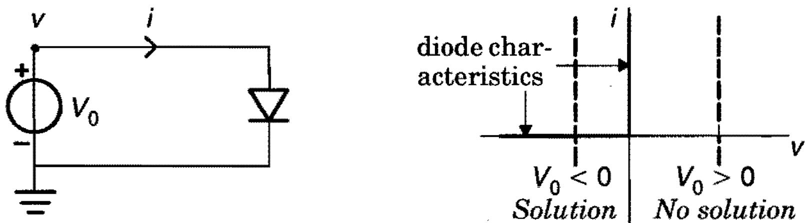


Figure 2.4. A simple circuit and two possible situations

This example shows that ideal components may create a non–solvable circuit. When a circuit is created physically with these two components, the parasitic effects in the circuit determine a solution. Here, the internal resistances of

diode and voltage source determine it. However, this does not imply that a circuit may not contain an ideal diode. The theorem gives only sufficient conditions for finding a solution. Specially combining two diodes can create a function that satisfies the conditions of the theorem. Because our piecewise linear modeling can also be derived from circuits containing only linear dynamic and resistive elements and ideal diodes (van Bokhoven 1981), this is not surprising.

2.4.3 The dynamic problem

The dynamic problem is trivial. If the linear equations and the LCP have a solution, a u–region is defined in which the following linear differential equation must be solved:

$$\begin{cases} A''_{uu}\ u(t) + a''_u = \dot{u}(t) \\ u(t_0) = u_0 \end{cases} , \qquad (2.17)$$

with $A''_{uu} = A'_{uu} - (A'_{uz})_{*\alpha}\ (A'_{zz})^{-1}_{\alpha\alpha}\ (A_{zu})_{\alpha *}$ for α the set of pl pivots. Here the star notation is introduced: A_{i*} is the ith row of A, and A_{*j} is the jth column of A.

This linear differential equation has always a unique solution. Problems can be encountered only at the borders of a region. A continuation of the current solution must then be found in another region. This occurs when the time derivative of w, $\dot{w}$, points to the exterior of the current region, so if there is an i with $w_i = 0$ and $\dot{w}_i < 0$. A solution that can not be dynamically continued in the current region is called *dynamically invalid*.

If a single nonzero pivot can be performed, the next region is entered. If this pivot is positive, the next region is on the other side of the facet. Also the time derivative of w in this region, $\dot{w}'_i$, is positive, so the solution can be continued into this region, and this dynamically valid solution can be chosen. Because all functions are continuous, $\dot{u}$ is continuous over the facet and the solution is continuously differentiable on the facet.

If the pivot is negative, a continuation in the new region is not found, because $\dot{w}'_i < 0$ also points out of the new region. So a third region must be found. This exists under the conditions of Theorem 2.2 (there is a solution for a point infinitesimally beyond the boundary) and a dynamically valid solution in this region is found, because it overlaps the facet. Only if also a zero pivot is encountered, this is not true, as is explained in the next paragraph. Concluding, there is a discontinuity here, but when the theorem is valid, a new region exists from which the solution continues. This discontinuity has a hysteresis–like behavior because the new region and the original region overlap. In the next section, a model exhibiting hysteresis behavior is derived.

The third possibility is a zero pivot. This also implies discontinuous behavior on the facet, but it may give more serious problems. If a block pivot is performed and the piecewise linear problem is solved, this solution may still be

not dynamically valid. Contrary to the situation with a negative pivot, this solution still can be on a facet. This can happen for instance if the block pivot changes the value of an entry of $\dot{u}$ (see Figure 2.5). So a third region must be found that is dynamically valid. But this region may not exist, because that region is part of the facet itself.

This problem can not easily be solved. Instead of the general implicit description of the boundaries in terms of x and u, two explicit inequalities for $\dot{u}$ must be used in the description of the region. Also the functionality between u and $\dot{u}$ is inverted, i.e. u is a function of $\dot{u}$. These properties will give difficulties in calculating a solution. This problem will emerge in the next section in an example with ideal logic components Notice that this problem does not exist in the x domain, as long as $\dot{u}$ does not change at a facet.

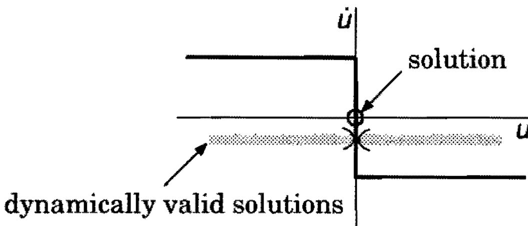


Figure 2.5. An example of a dynamically invalid solution

The conclusion is that a solution exists under some not too severe restrictions. A_{xx} must be regular, and the LCP must have a solution in the neighborhood of the starting point (which may be hard to prove, but is valid in most practical situations). A direct feedback of u and $\dot{u}$ with a zero pivot may give problems.

2.5 Creating piecewise linear models

With our modeling technique many different components can be modeled. The used component models range from simple devices to register transfer level. In this range the modeling is best suited to the complexity of the components. For register transfer level and higher levels the internal complexity of the components and the transported values makes the modeling too difficult. For accurate device simulation the model will be too large. To give some insight in the possibilities of the modeling, in this section the creation of models is discussed.

One important aspect, which is sometimes forgotten, is that a model should function under all circumstances, i.e. a solution is found whenever the input of the component is within its domain. This depends also on the algorithm being used to find the solution. There is an interaction between modeling and solving: if a model is in principle right, but an otherwise good algorithm fails in special cases to find the right region, usually not the algorithm is replaced, but another model is developed.

The creation of piecewise linear models is a non–trivial task. There are a few general rules that can be applied to simplify the problem, but the creation of a specific function can be difficult.

First some general rules are given to simplify the task of creating models. Then the basic modeling of functions from $\mathbb{R}$ to $\mathbb{R}$ is given. These basics are applied to create three different models of logic gates, that are also interesting in itself.

The general rules are:

- The dimension of the input and output vectors should be as small as possible. This can be accomplished in many cases by taking a suitable linear combination of variables. The reason is that lower dimensional modeling is much simpler.
- The function should be partitioned into simpler functions. Like in system design, it is easier to build simple subsystems and connect them than to build a large system out of one piece. The models can be combined with a linear transformation of their inputs and outputs.
- Symmetry should be used if possible. Notice that even functions can be created by inserting an absolute function (only one extra z entry) in front of the input. For an odd function five extra entries in the w and z vectors need to be used (using an absolute and a sign function, and a switch).
- Already known models can be investigated and adjusted in order to create new models. By using linear transformations a different model is created. Many models have matrix entries that can be modified to change the behavior of the model. This can be accomplished by parameterizing these entries, instead of creating a separate model for every application.

By applying these rules, part of the system might be known already, and other parts should be as simple as possible. Now the second part of the modeling must be performed. If the resulting functions are from $\mathbb{R}$ to $\mathbb{R}$ and continuous, it is possible to approximate them by applying the following algorithm.

Algorithm 2.1: Creation of a model of a continuous function $f : \mathbb{R} \rightarrow \mathbb{R}$.

{ *The function is approximated with a continuous piecewise linear function with* n *breakpoints. Let the breakpoints be at* $x_1, \ldots, x_n$. *Let the linearization on the interval* $[x_i, x_{i+1}]$ *be* $y = d_i\,(x - x_i) + b_i$, *on* $(-\infty, x_1]$ *be* $y = d_0\,x + b_0$, *and on* $[x_n, \infty)$ *be* $y = d_n\,x + b_n$.
Notice that only $2n+2$ *of these parameters are sufficient due to the continuity.*
}

{ A_{xx} : matrix[1x2]; A_{xz} : matrix[1xn];
A_{zx} : matrix[nx2]; A_{zz} : matrix[nxn];
Only the nonzero elements are given.
}

```
Axx := [d0, -1]; ax := b0;          { Axx x + ax ≡ d0 x - y + b0 }
for i := 1 to n do
    (Azx)i1 := -1; (az)i := xi;      { x ≤ xi }
    (Axz)1i := di - di-1;            { Update with difference }
    (Azz)ii := 1;                    { Azz = I };
od;
```

With this algorithm n hyperplanes (lines) are created in the (x,y)–space at the breakpoints, all parallel to the y–axis. For each plane the update to the linear function is simply calculated. Notice that the number of defined regions is n_z+1 instead of 2^{n_z}, the theoretical maximum. This is a general property of this modeling: if A_{zz} is the identity matrix and the rank of A_{zx} is 1, there are (at most) n_z+1 regions.

The problem of creating a minimal–sized model, i.e. a model with minimal n_z, has not yet been solved. To model a function $f : \mathbb{R} \rightarrow \mathbb{R}$ with n breakpoints, in general more than $\log_2 n$ pl variables are necessary. Only in special cases of symmetry (directly visible or not) less pl variables need to be used, as in the square function (van Eijndhoven 1984). In several cases it is possible to use a 2x2 submatrix A_{zz} to map a function with four segments. The derivation of these 2x2 submatrices and the conditions, under which this mapping is possible, are given in Appendix A.

Algorithm 2.1 can be automated if a rule is given to determine the breakpoints of a function. Possible rules are, for instance, placing the breakpoints equally spaced, or situating them based on the approximation error. This can lead to large models, but internally only A_{xz}, a_x, A_{zx} and a_z need to be stored.

These rules and Algorithm 2.1 are clarified by the example of logic gates. Models for logic gates are necessary for any mixed–level simulator, so some of these models have been defined for our simulator. There are several models possible, depending on their use in the system. Both continuous/discontinuous and static/dynamic models can be created and simulated in our system. In some detail will be described how three static models (a continuous, a discontinuous, and a hysteresis–like model) are derived. Afterwards a dynamic model will be given. The inputs of the gate are numbered $x_1, x_2, \ldots$, the output is y (1–dimensional).

The second rule gives a starting point: the model should be divided in simple submodels. For logic gates this is accomplished by defining an intermediate variable s, representing the wanted output value. Its value has the property that it is ≥ 1 if the output of the gate should be 1 and ≤ 0 if the output should be 0. The first submodel creates this variable, the second submodel creates the desired output behavior.

Now the first rule, diminishing the dimension of the vectors, can be applied to the input vector. For most basic logic gates, a linear function can be used

to define s. For example, an inverter has $s = 1 - x_1$ and a 3–input nand–gate has $s = 3 - x_1 - x_2 - x_3$. Only the xor–gate has a nonlinear input part. This can be modeled by $s = |x_1 - x_2|$, which uses one pl variable pair. More complex logic function models can be generated by the algorithm described in (Kevenaar 1990).

The first and simplest output part of the gates is the following piecewise linear function:

$$\begin{cases} y = 0 & \text{for } s \leq 0 \\ y = s & \text{for } 0 \leq s \leq 1 \\ y = 1 & \text{for } s \geq 1 \end{cases} . \tag{2.18}$$

With Algorithm 2.1, the system is directly created:

$$\begin{bmatrix} -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ s \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.19}$$

This model is depicted in Figure 2.6.a.

The second model of the output part is a discontinuous function. It is modeled with:

$$\begin{cases} y = 0 & \text{for } s < 0.5 \\ y = 1 & \text{for } s > 0.5 \end{cases} . \tag{2.20}$$

To create a continuous mapping of this discontinuous function, the vertical line segment at $s = 0.5$ is added to equation (2.20):

$$s = 0.5 \quad \text{for } 0 \leq y \leq 1 . \tag{2.21}$$

Instead of considering y as a function of s, consider y as a function of $s + y$. This is a continuous function and can be modeled by applying Algorithm 2.1:

$$\begin{bmatrix} -1 & 0 & 1 & -1 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ s \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.22}$$

This mapping has facets that depend on the output of the mapping. The y variable can be eliminated from the pl equations of (2.22) to remove that dependency and show clearly the zero pivot:

$$\begin{bmatrix} -1 & 0 & 1 & -1 \\ 0 & -1 & 0 & 1 \\ 0 & -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} y \\ s \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.23}$$

This model is shown in Figure 2.6.b. Notice that this has become an example of a mapping in which two parallel facets have influence on each other.

The third static model of the output of a logic gate is a model with hysteresis. This model has $y = 0$ for $s \leq 0.75$ and $y = 1$ for $s \geq 0.25$. Because both regions overlap, there is the possibility to change the output of the gate at an arbitrary value of s between 0.25 and 0.75. This model is made continuous by adding the linearization $y = -2s + 1.5$ in the region $0.25 \leq s \leq 0.75$. Like in the previous model, consider y as a function of $s+y$ and apply Algorithm 2.1:

$$\begin{bmatrix} -1 & 0 & 2 & -2 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ s \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.75 \\ 1.25 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.24}$$

Eliminating y yields a negative pivot:

$$\begin{bmatrix} -1 & 0 & 2 & -2 \\ 0 & -1 & -1 & 2 \\ 0 & -1 & -2 & 3 \end{bmatrix} \begin{bmatrix} y \\ s \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.75 \\ 1.25 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.25}$$

This model is pictured in Figure 2.6.c. If the negative pivot is performed, the second pivot becomes also negative. This example shows that two consecutive negative pivots create a hysteresis.

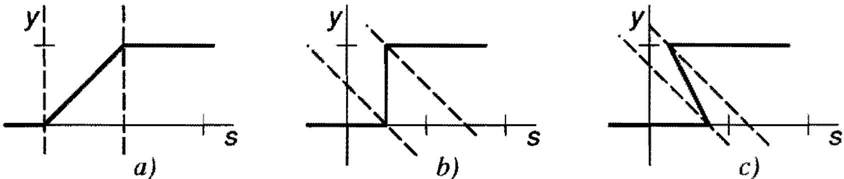


Figure 2.6. The logic gate output mapping: *a)* with a continuous model *b)* with a discontinuity *c)* with a hysteresis

These three examples show the behavior of simple models. A dynamic logic gate is a more difficult model. The output of this gate is analog (the dynamic variable u), but it is easy to switch to a digital or hysteresis–like output by transforming the output with one of the above presented models.

This dynamic model is based on a split in four regions of the (s,u)–plane, with in each region a constant $\dot{u}$. This has the advantage, that the function is also piecewise linear in time, instead of having an exponential behavior. The idea is that the output is dynamic and will reach the desired output value after a certain time. This is implemented with two boundaries: on one boundary a switch of s sets $\dot{u}$ to a fixed nonzero value. On the second boundary u switches $\dot{u}$ back to 0. This is shown in Figure 2.7. The behavior of this model is that the output increases to 1 if $s > 0.5$ and it decreases to 0 if $s < 0.5$. As soon as the

output reaches its maximum or minimum, it remains constant. The model uses only four pl variables to implement this behavior:

$$\begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_1 & 0 & d_2 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & r & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} y \\ s \\ u \\ z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0.5 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \dot{u} \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}, \tag{2.26}$$

where $r = -\dfrac{d_1}{d_2} - 1$.

This model is difficult to understand at once, so some explanation is given. The model is based on two discontinuities, each modeled with the 2x2 pl matrix of equation (2.27).

$$\begin{bmatrix} -1 & 0 & p & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} y \\ x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ x_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix}. \tag{2.27}$$

The matrix of (2.27) models a discontinuity at $x = x_0$: $y = 0$ for $x < x_0$ and $y = p$ for $x > x_0$.

The first discontinuity in (2.26), related to z_1 and z_2, implements a step function on the input s. This functional unit sets $\dot{u}$ high if $s > 0.5$. Through the entry $(A_{zz})_{31}$ it shifts the second boundary, so there is a multiple boundary on the line segment $(s = 0.5, 0 < u < 1)$. Through the entry $(A_{zz})_{41} (= r)$ the influence of the second boundary on $\dot{u}$ is changed. This is only possible if $r > -1$ (otherwise $w_4 < 0$), which is equivalent to $d_1 d_2 < 0$, exactly what the model requires. The second discontinuity is related to z_3 and z_4 and implements the dependency of $\dot{u}$ on u. This model only behaves as described if $0 \leq u \leq 1$, which the model ensures after the initialization. An initialization outside this region has an unpredictable effect on the other components.

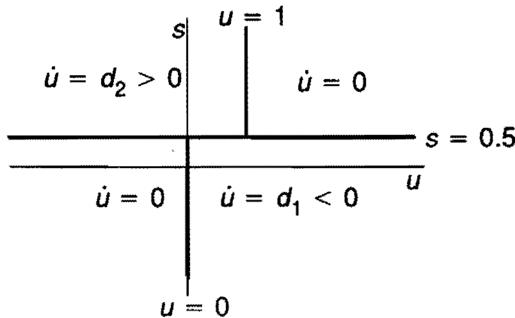


Figure 2.7. The functionality of a dynamic logic gate

The influence of the circuit

With these four output parts of logic gates some characteristics of the influence of the circuit are shown. One of the simplest possible circuits is considered: one inverter with its output connected to its input. This adds to the equation for y two equations for s and x respectively: $s = 1 - x$ and $x = y$. Because their values are trivial, x and s are eliminated from the matrices. The graphic solution is shown in Figures 2.8 and 2.9. The following four systems are found:

1) Static with continuous output:

$$\begin{bmatrix} -1 & 1 & -1 \\ 0 & 2 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.28}$$

Pivoting on $(A_{zz})_{11}$ yields the only possible solution. Notice that after this pivot the A_{zz} matrix has become singular. A graphic representation of this system is shown in Figure 2.8.a.

2) Static with discontinuous output:

$$\begin{bmatrix} -1 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.29}$$

Here the zero pivot is changed in a positive pivot and the A_{zz} matrix is changed back to the identity matrix, because the circuit equation is parallel to the original facets. After performing a pivot on $(A_{zz})_{11}$ the unique solution is found (see Figure 2.8.b).

3) Static with hysteresis output.

$$\begin{bmatrix} -1 & 2 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -0.25 \\ 0.25 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} . \tag{2.30}$$

The influence of the circuit is that again the A_{zz} matrix is changed to the identity matrix. The model is not in a valid state, but it is simple to find the same solution as above. See Figure 2.8.c for a sketch of the situation.

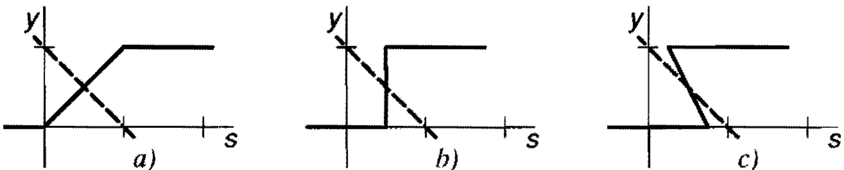


Figure 2.8. Finding a solution for an inverter with its input connected to its output: *a)* with a continuous model *b)* with a discontinuity *c)* with a hysteresis

4) Dynamic with analog output:

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_1 & 0 & d_2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 \\ 0 & 0 & r & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} y \\ u \\ z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -0.5 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \dot{u} \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}, \tag{2.31}$$

where $r = -\dfrac{d_1}{d_2} - 1$.

This model still has its two discontinuities. But from Figures 2.9.a and 2.9.b follows that both discontinuities are crossed for only one u($u = 0.5$) and that $\dot{u}$ can have only the values $\dot{u} = d_1$ and $\dot{u} = d_2$. This is the situation as described in Section 2.4.3: the dynamic solution can be found only by performing a zero pivot. As this is impossible in our simulator, no dynamically valid solution is obtained and the simulation will always abort in this point. So this idealized model may cause problems. In practice, however, no problems are encountered with this model. Only once we have observed this behavior: connecting an odd number of inverters with this continuous model, all modeled exactly the same and with the same starting value, crashed the simulation algorithm, because the calculations were too exact. This problem is solved by changing one inverter, either its delays d_1 and d_2 or its initial starting point.

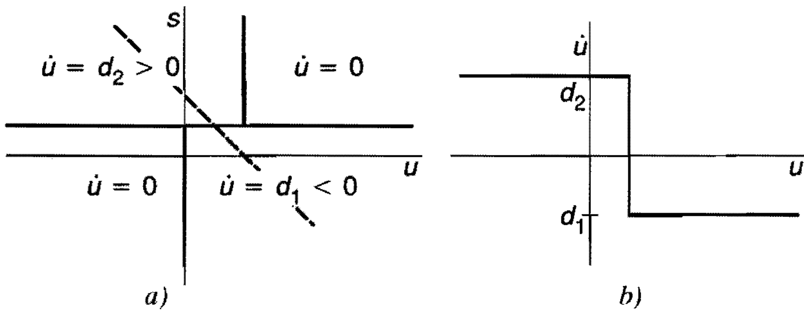


Figure 2.9. Finding a solution for a dynamic inverter with its input connected to its output:
a) the linear equation in the (u,s)–space
b) $\dot{u}$ as function of u

3 The Linear Equations

3.1 The data structure

In Chapter 2 the general model of systems has been introduced. In this and subsequent chapters the simulator PLATO, which uses these models, is described. The modeling is suited best for circuits containing relatively simple components. Its primary use is in simulating mixed analog/digital electrical circuits. Therefore the simulator has been optimized for these types of circuits. In this chapter the choice of the data storage is discussed, as well as the solution of the linear equations of the form $A_{xx}\ x = b$.

The input to the simulator is described as a circuit containing electrical and logical components. Whether in reality the system is an electrical circuit or something else like a neural network or a dynamic LCP is not important. Each circuit can be built up hierarchically from several subcircuits, which each may contain other subcircuits, etc. A circuit may also be a component with a piecewise linear model, which of course has no subcircuits. These components are called *leaf cells*, whereas higher level circuits are called *compound cells*.

Each circuit has several contacts to the outer world called *terminals*, which are used to connect it to an other circuit. There are two types of terminals: electric ones and signal ones. Electric terminals have two circuit variables related to them, called *voltage* and *current*. These variables have different topological relations: connected terminals have the same voltage value, while the sum of the values of the related currents is zero. For digital components, terminals are available with only one circuit variable related to them, called *signal*, which behaves like a voltage. To simplify the modeling, the terminal variables are restricted (without loss of generality) to be x variables, so the u, w and z variables are local to a leaf cell.

In this hierarchy of leaf cells and compound cells, for each entry in the input vector $r(t)$ an *input cell* can be associated with one entry in the x vector. This cell can be viewed as a special type of leaf cell, not with a piecewise linear model, but with a functional model $x_i = r_i(t)$. This cell can emulate easily the actions performed by a leaf cell.

The system of equations that must be solved, contains the different types of equations from the model descriptions, combined with the topological relations (connections) and the input relations on the terminals of the top–level circuit. The general matrix of such a system of equations is given in Figure 3.1. Only the nonzero elements are shown, the other elements are inherently zero.

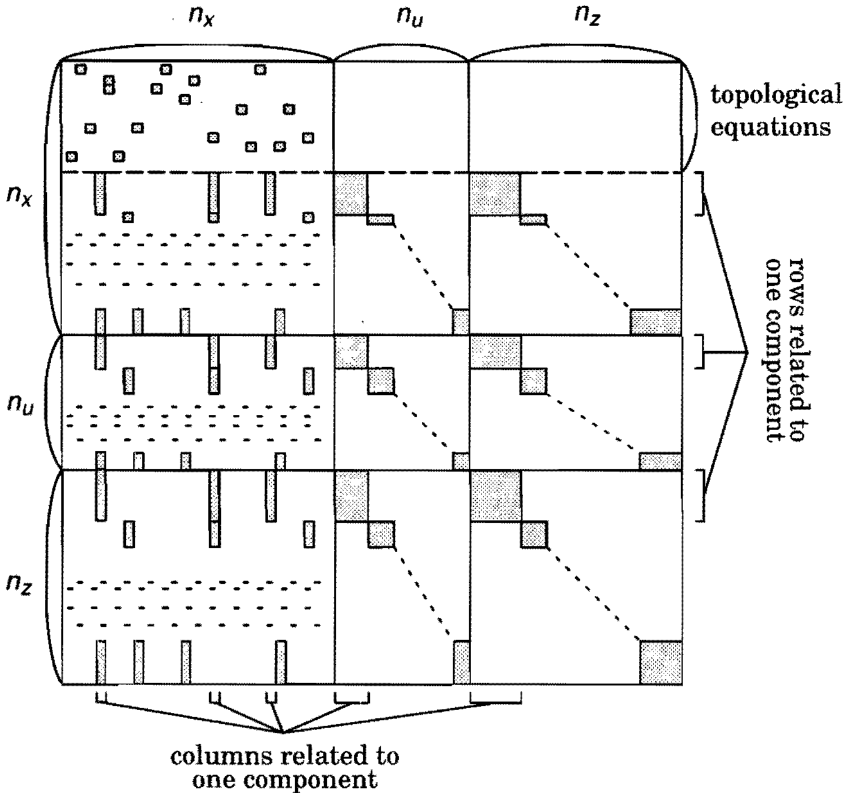


Figure 3.1. General structure of the system matrix A

A main objective in the design of a simulator is to minimize the number of calculations that are to be performed by the program to decrease the execution time. Therefore the internal data structures must be optimized with respect to the expected data, and the algorithms must take advantage of previously acquired knowledge. The internal data structure in PLATO has been optimized for electric circuits.

In an electric or logic system the number of terminals of a component ranges usually from 2 to 4. Furthermore most connections are local, connecting only a few components. Exceptions to this are some basic connections like the power supply and the clock. This implies that the matrices A_{*x} are sparse and

have no special structure, i.e. the matrices are not banded, symmetric or block–diagonal.

Other types of systems do not need to have this sparsity. An example is a Hopfield neural network, a fully connected network containing neurons (simple components). In this network A_{xx} is a full matrix and A_{zx} can be a matrix with zeros, as is explained in Section 7.3. Because PLATO is primarily intended for circuit simulation, its data structure is based on the sparsity of the matrices. For other types of systems the next considerations are not valid. A non sparse implementation is considered in Section 7.3 describing a Hopfield neural network simulator.

The A_{*u} and A_{*z} matrices are always block diagonal. To preserve the sparse structure of the problem, especially this block diagonal structure of the A_{*u} and A_{*z} matrices, the matrix of Figure 3.1 is not created explicitly in one data structure. The local matrices of the leaf cells are used whenever the global matrix is referenced. This implies that there is a separation between the implementation of the algorithms at the global level and the processes at the leaf cell level. An advantage is that the implementation at the leaf cell level can change depending on the particular type of cell. For instance, the models created with Algorithm 2.1 can be stored sparsely, differing from general models. Because all changes in a component are only visible through changes in x and in A_{xx}, these are (at the global level) the only interesting parts of the leaf cells. There are two different methods to handle the x variables and the A_{xx} matrix.

The first possibility is to keep all linear equations locally within the components. This means that in each component the variables are split into input and output variables. With the values of the input variables, the component can determine the value of the output variables. These values are propagated to its children and its parent, so the hierarchy of the network description is preserved. In Figure 3.2, the data flow and the calling sequence are shown, as well as the changed variables (in the compound cells of both their children and their parent). Suppose in one leaf cell some x variables change value. This change is passed to its parent, which subsequently calculates the change in the x variables of his parent, etc., until either the root of the tree is reached or (as in Figure 3.2) its parent's x variables do not change. Then all calculated changes are propagated to the children of the current cell, that calculate the changes for their children, etc.

The advantage of this method is that all calculations on x variables are performed locally. Because the influence of a variable is usually limited, only a few components are reached. There is one important drawback of this method: the number of variables and equations is large. If one x variable is output of a leaf cell and input to an other leaf cell, at least three variables are used for one value: in both components as well as in their common parent.

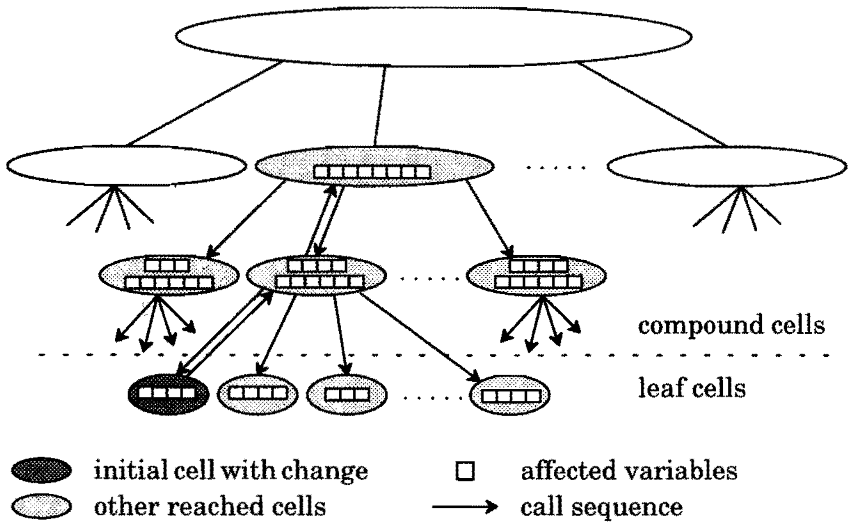


Figure 3.2. Hierarchy of cells and the calling sequence after a variable change

Because there is one equation per variable, the number of equations is also large.

The second method is to use a global x vector. Each component accesses this global vector when it needs its value. The advantage over the hierarchical method is that many variables and equations can be eliminated. The voltage (and signal) variables that are related to connected terminals have the same value, so they are replaced by one global variable, called the *node voltage*. If exactly two electric terminals are connected, their related current variables differ only in sign, so one is eliminated. This greatly reduces the number of x variables and linear equations.

If a global x vector is used, another choice must be made: the A_{xx} matrix can be stored locally or globally. The advantage of storing the matrix globally is that further eliminations can be performed (which often are possible) and that the compound cells and the hierarchy are not used any more. Because the hierarchy is mainly used to limit the effect of one leaf cell action to the leaf cells that are affected by this action, an alternative method will be necessary for an efficient implementation of the actions between leaf cells.

Such a method is found by noticing that the A_{ux} and A_{zx} matrices of a leaf cell have only nonzero entries in the columns of terminal variables. Because the number of terminal variables is in general low, only these columns are stored in the component's matrix. During the calculations, many times an update on the component's variables is performed, initiated by a (usually sparse) update

on the x vector. If this update is sparse, many leaf cell values will not change. Therefore a list is maintained with for each x variable the leaf cells that are related to it. This list is, in a way, equivalent to the use of a hierarchical system. Local effects are handled efficiently in a hierarchical system, because only a subtree will be traversed. With such a list structure the same effect is accomplished: only the affected leaf cells are visited. See Figure 3.3 for a sketch of this list. There is one drawback on the use of such a list: for each entry in the x vector a list of leaf cells is maintained. The list that is used is created by merging the lists of each entry in the vector, which process can be inefficient. If two entries in x are coupled, i.e. they are always both zero or both nonzero, one of those lists can be emptied. This coupling can not be detected, because it is dynamic in behavior, and may emerge or vanish at certain time points. In our implementation, the overhead of this list structure is small (see Chapter 7 for experimental results).

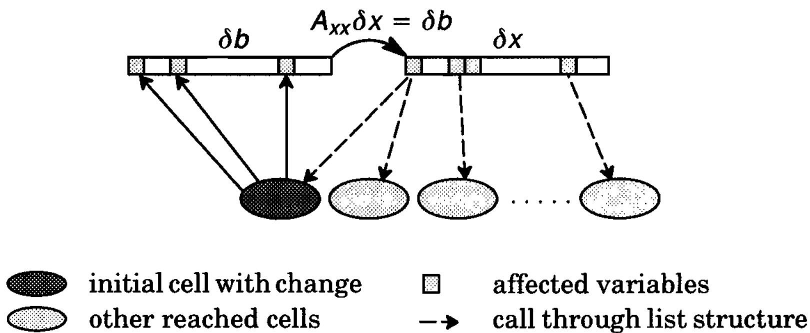


Figure 3.3. List structure with calculations

In PLATO the second approach has been implemented, using a global x vector and a global A_{xx} matrix. The choice of mixing local and global matrices implies some far–reaching consequences. The obvious consequence is that the x variables are not eliminated from the leaf cell models. Elimination does usually not result in a smaller system, because the other matrices would lose their block structure and be filled totally. The local structure of these matrices would disappear and they had to be stored globally. This will certainly use much more memory and the calculations would be much more expensive than without elimination, taking into account that the number of x variables is usually much smaller than the total number of u and z variables.

3.2 Nodal analysis

In general, too many variables and equations are abstracted from the circuit topology and component descriptions. This is because it is not known a priori which variables are interesting for the user or necessary for the simulation

process. The models may contain any dynamic piecewise linear equation, but it is possible that some components define only trivial linear equations. Decreasing the number of equations and variables will increase the efficiency in some parts of the simulator, and therefore as much as possible variables are eliminated. This elimination is called *nodal analysis* and is performed after the initial equations are created. Instead of applying the standard techniques for nodal analysis, some adaptations are made to the method, in order to maintain the sparsity of the total system, and the low density of the A_{xx} matrix in particular.

There are theoretically two basic methods to create the network equations. The first is the sparse tableau method, the second is the (modified) nodal analysis. In the sparse tableau method, all terminal currents, terminal voltages and node voltages are used. All topological relations are put in the global matrix A_{xx} together with the linearizations of the leaf cell equations, the local A_{xx} matrices. This has the advantage that the matrix will be very sparse, mostly because many entries are of the form $x_i = x_j$. The disadvantage is the large number of variables, because this method is the same as using local x variables.

With the nodal analysis method, only the node voltages are calculated. These variables are solved with the nodal current equations, i.e. the topological equations related to the currents. All other equations are used to eliminate the remaining variables. This creates a small but rather dense system of equations. Therefore this method is used in most popular circuit simulators. Variables with type signal, which do not exist in other simulators, are also calculated. Because there is no current related to these variables, they are calculated from their leaf cell equations.

With the piecewise linear models, both methods can be used, except that the terminal voltages have already been eliminated during the expansion of the network hierarchy. Because an explicit equation is available for each x variable, it is possible to eliminate any x variable. There are however three reasons to keep some x variables:

- Elimination is sometimes possible only by merging two (or more) leaf cells. Consider a variable x_i, depending directly on some nonlinear variables, which is used in a component, other than where its equation is defined. Eliminating this variable implies that these two components must be merged. Because this will finally lead to one large component with non sparse internal matrices (as discussed in Section 3.1) this type of variables will not be eliminated.
- The density of the A_{*x} matrices may increase. When a variable x_i is referenced a few times in the A_{xx} matrix, elimination of x_i can yield more elements in this matrix. Eliminating this variable may also increase the number of terminal variables in the local A_{ux} and A_{zx} matrices, thereby increasing the number of local calculations and increasing the connectivity of components.

- The output of the simulator are the values of several x variables over the time interval $[t_0, t_e]$. Which x variables are printed depends on the user. There are two reasons to keep the output x variables in the system: the associated values must be calculated anyhow, and elimination implies that the printing of some global variables must be performed by the local leaf cells. The partial elimination of variables also implies that the size of the system which is solved can vary between different simulations, depending on the output (see also Figure 7.3).

The following algorithm is used to remove as many variables and equations as possible from the system, taking the three points above into account. For a good result, also the leaf cell data are manipulated, discarding linear equations and eliminated or unused variables.

Algorithm 3.1: Nodal analysis

```
{   A candidate is a pair (variable, equation) where the variable can be
    solved from this equation, and the equation is linear or the
    variable is not used in other components
}
C := set_of_candidates( );
while C ≠ ∅ do
    c := take_el_from( C );
    eliminate_from_system( c );
    if nr_new_elements > element_grow_limit or
      nr_new_leaf_cell_variables > variable_grow_limit then
        undo_elimination( );
    else
        if not outputvar( c ) then
            remove_equation( c );
        fi;
        C := C ∪ new_candidates( ) \ no_longer_candidates( );
    fi;
od;
clean_leafs( );
```

The equations and variables that are eliminated with this algorithm are usually in some restricted classes. First, variables with constant value are eliminated from the system. These are usually the power supply voltage and the ground voltage (both connected to many components), and constant inputs. Then variables with simple relations are eliminated, like the Q and $\overline{Q}$ outputs of a flip–flop that satisfy $\overline{Q} = 1 - Q$. Third, many current variables (those that are not interesting) are removed.

However, most voltage and signal variables are not eliminated, because these variables are interesting to a user and most of them are non–local nonlinear variables. This is also the reason that most capacitors and capacitor–like components keep their current variable(s), because these currents depend on several other variables.

Theoretically a subsystem that has no output to the rest of the system and that is also not interesting for the user can be eliminated completely. But this situation is not detectable by the algorithm. For example, if two not interesting components are connected to each other through two nonlinear variables, then these variables can not be eliminated. This flaw in the algorithm is not important in practical cases.

Algorithm 3.1 can be refined when a better output processor is available. Then also interesting variables can be removed, depending linearly on some other output variables. The output processor can recreate the variable at the moment it is needed.

3.3 The LU decomposition

During the simulation process two actions are performed many times with the matrix A_{xx}:

- Equations of the form $A_{xx}\ x = b$ are solved.
- The matrix A_{xx} is updated, either due to the changes in the linearizations when a new region is entered, or due to the application of an implicit integration method to solve the dynamic equations, as will be explained in later chapters.

Because the execution time in these algorithms dominates the execution time of the simulator for large circuits, both actions must be performed efficiently.

The equation $A_{xx}\ x = b$ is solved with an LU decomposition, a direct method. An iterative method is not applied, because for our type of problems the LU decomposition is efficient enough. An other direct method, for example the QR decomposition, is not used, because possible advantages in efficiency or accuracy, compared with the LU decomposition, are not outweighed by the simplicity of the LU decomposition.

The LU decomposition is named after the matrices L and U, which are determined so that $PA_{xx}Q^T = LU$. These matrices have the following properties: P and Q are permutation matrices, L is a lower triangular matrix and U is an upper triangular matrix. With these matrices, the solution of the matrix equation $A_{xx}\ x = b$ is determined by solving $L\ v = P\ b$ and $U\ Q\ x = v$. These two steps are called the forward and backward substitution.

The LU decomposition will maintain the sparse structure of the original matrix, which property is essential to solve large problems. The new matrices have a nonzero entry for each original nonzero entry, and a number of new nonzero entries called *fill–ins*. As is shown earlier (Chua and Lin 1975; van Stiphout 1990), it is possible to order the rows and columns of the matrix in

such a way, that for an electric circuit the number of fill–ins in the L and U matrices is in general less than the number of original elements.

Algorithm 3.2: The LU decomposition.

```
for i := 1 to n do
    <p, q> := select_pivot ( );
    perform_permutation ( p, q, i );
    U_ii := A_ii;
    for j := i + 1 to n do
        L_ji := A_ji / U_ii;
        U_ij := A_ij;
        for k := i + 1 to n do
            A_jk := A_jk - L_ji * U_ik;
        od;
    od;
od;
```

The most important part of this algorithm is the selection of the pivot. This selection serves two purposes: the pivot should be as large as possible, and the structure of the matrices L and U should be as sparse as possible. These objects are in general conflicting. The value of the pivot must be large in order to make the decomposition numerically stable. There are two different selection methods in use to find a large pivot: partial and full pivoting. With full pivoting the largest entry in the remaining submatrix is chosen. With partial pivoting the largest entry in the current column is chosen ($q = i$, $Q = I$). Partial pivoting is theoretically not as numerically stable as full pivoting, but in practice it is always stable; see Section 6.6 for more problem specific considerations. Therefore partial pivoting is always chosen and has been implemented in PLATO.

To fulfill the second purpose, the selection of the pivot is changed. If more than one candidate is numerically acceptable, i.e. not more than 10 times smaller than the largest candidate, the one which keeps the structure more sparse is chosen. This can be estimated by a Markowitz count (Markowitz 1957). This method is used in PLATO and has the desired properties (van Stiphout 1990).

The costs of the LU decomposition are for full matrices and full vectors:
- decomposition: $n_x^3/3$
- forward and backward substitution: $n_x^2/2$ each.

For sparse matrices these costs are considerably lower. Consider a structure with on average p elements per row or column in the L and U matrices, as usually happens in electric and logic circuits:

– decomposition (including Markowitx count): $\mathcal{O}(p^2 n_x)$
– forward and backward substitution: $\mathcal{O}(pn_x)$.

If the vector x contains afterwards only q elements, the forward and backward substitution costs are even lower: $\mathcal{O}(pq)$.

These costs are only reachable for an efficient implementation, i.e. the costs of accessing the nonzero entries of A_{xx}, L and U are small compared to the costs of multiplication. With an orthogonally linked list structure and three arrays, in nearly all cases the entries are directly accessible in the employed algorithms (van Stiphout 1990). Only the backward substitution with a sparse vector x is slightly more complicated.

3.4 The rank m update

There is a second operation performed on the matrix during the simulation: the rank m update. The matrix is updated as follows: $A'_{xx} = A_{xx} + c\ r^T$, with c and r matrices of size $n_x \times m$, and $m \ll n_x$. In most cases it is only a rank 1 update. The update has already been executed in the component's matrix, and to keep the LU decomposition valid, this update must also be performed on L and U. The following direct algorithm, that updates the decomposition efficiently, has been introduced in (Bennett 1965).

Algorithm 3.3: Rank m update (Bennett)

```
{   preconditions: A_xx = L U, A'_xx = A_xx + c r^T;
    A_xx: matrix[nxn]; c, r: matrix[nxm];
}
var d: matrix[mxm], p, q: vector[m], oldU, fact: scalar;
d := I;
for i := 1 to n do
    p := d^T * (c_{i*})^T;
    q := d * (r_{i*})^T;
    oldU := U_{ii};
    U_{ii} := U_{ii} + r_{i*} * p;
    fact := U_{ii}/oldU;
    d := d - q * p^T/U_{ii};
    for j := i + 1 to n do
        c_{j*} := c_{j*} - L_{ji} * c_{i*};
        r_{j*} := r_{j*} - r_{i*} * U_{ij}/oldU;
        L_{ji} := L_{ji} + c_{j*} * q/U_{ii};
        U_{ij} := fact * U_{ij} + p * r_{j*};
    od;
od;
```

This algorithm efficiently updates only those elements of the decomposition that really change. In our simulator, the matrix c consists of a few columns of A_{xu} or A_{xz} and r of a few rows of A_{ux} or A_{zx}. In many cases, c has only one or two nonzero entries. When, as is often the case, the update is only rank 1 on one row of the matrix, say the kth row, then the costs of this update are for a full matrix: $n_x^2/2 + 1.5k^2$. For a sparse matrix structure in which a sparse update is performed, the costs are lower. Consider an update where c and r are sparse vectors with at most p entries after the update, and the L and U matrices have on average p elements per row or column (the number of entries in c and L will be about the same). Then the costs of the update are $\mathcal{O}(p^2)$. So only when a very large part of the matrix changes it is cheaper to perform a full LU decomposition instead of a rank m update.

The rank m update is still one of the more expensive actions in the simulator. When two rank 1 updates must be performed simultaneously, they can be combined in one rank 2 update. But this is only cheaper if the updates have an overlap in the elements they change, else it is more expensive. Because usually many rank 1 updates are performed with little or no overlap, this algorithm is implemented in two versions: once as a rank 1 update and once as a general rank m update. Usually the rank 1 update is chosen. The rank m update is chosen in only two situations:

1. If a block pivot is performed.
2. If the rank 1 update gives an invalid decomposition.

In the first case, a block pivot is performed during the solution of the piecewise linear equations, if a diagonal element of $A_{zz} - A_{zx} A_{xx}^{-1} A_{xz}$ is zero. This is equivalent to a discontinuity in the piecewise linear function, and it means that if the block pivot is performed as a number of single pivots, the matrix $\begin{bmatrix} A_{xx} & A_{xz} \\ A_{zx} & A_{zz} \end{bmatrix}$ is not regular between the first and the last update. Therefore the submatrix A_{xx} may become singular after the first update. Because an LU decomposition is not possible then, the rank 1 update algorithm will abort with a failure. But with a rank m update the block pivot is performed in one step, so the matrix A_{xx} will remain regular.

The second case occurs when during the rank 1 update a diagonal element of U becomes small. Because a column of L, and later an entry of the solution vector, is divided by this value, this is numerically not acceptable. If the matrix itself remains regular, a better decomposition is usually possible. A full LU decomposition can be performed, but usually it suffices to swap two or three rows. Because swapping two rows is also a rank 1 update, a rank m update is performed instead of a full decomposition. A full LU decomposition is performed when the rank m update fails to find a valid decomposition.

In PLATO, the data structure contains a sparse A_{xx} matrix, and sparse vectors are used. A detailed description of the LU decomposition and the rank 1 up-

date with these sparse structures can be found in (van Eijndhoven and van Stiphout 1988). The forward and backward substitution is implemented twice: one routine using a nonsparse vector, the other one using only sparse vectors. The first routine is used for solving vectors that may become not sparse, while the latter one is employed if the vector will remain sparse. These two cases are heuristically determined: the sparse version is applied during the Van de Panne algorithm (see next chapter), the nonsparse version is used in the other cases.

3.5 Parallel capacitors

As mentioned in Section 2.4, on the existence of solutions, it may happen that the A_{xx} matrix is not regular. This will happen when the circuit specification is wrong, for instance if not all inputs are specified, or if loose wires exist. Other errors occur when the component's models are invalid. These errors are fatal and can not be repaired. However, sometimes the circuit is valid, but has a singular A_{xx} matrix, so the system can not be solved. This happens most frequently in the case of parallel capacitors, although it can happen in other cases too. Therefore the case of parallel capacitors is studied first. An example, a part of a network with two linear parallel capacitors, is shown in Figure 3.4.

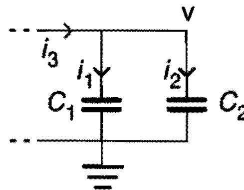


Figure 3.4. Two parallel capacitors

Two linear capacitors, denoted with their capacitances C_1 and C_2, are set parallel. Their local matrices are (after the nodal analysis):

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1/C_1 & 0 \end{bmatrix} \begin{bmatrix} v \\ i_1 \\ u_1 \end{bmatrix} = \begin{pmatrix} 0 \\ \dot{u}_1 \end{pmatrix}, \text{ and } \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1/C_2 & 0 \end{bmatrix} \begin{bmatrix} v \\ i_2 \\ u_2 \end{bmatrix} = \begin{pmatrix} 0 \\ \dot{u}_2 \end{pmatrix} . \tag{3.1}$$

The topology relation is:

$$i_1 + i_2 - i_3 = 0 . \tag{3.2}$$

From the linear part of the equations (3.1) and (3.2), the variables v, i_1 and i_2 must be calculated. This is a singular system and can not be solved directly, although a solution exists. One of the equations used to solve v must be replaced by an equation for solving i_1 or i_2. This equation can be derived in the following way: $v = u_1 = u_2$ implies $\dot{u}_1 = \dot{u}_2$ and $u_1(t_0) = u_2(t_0)$, and this implies $C_2\ i_1 = C_1\ i_2$. Replacing the equation $v = u_2$ with this equation will

give a valid system. The dynamic behavior of capacitor C_2 then fully depends on capacitor C_1: the dynamic variable u_2 is not used any more and can be removed from the system.

The automatization of this procedure is straightforward. The LU decomposition does not find an equation for i_2 (the diagonal element is zero). The relevant components are gathered, which gives only capacitor C_2. There is no possibility to change the internal state of the component in such a way that a linear equation can be used to determine i_2. Therefore all other components, related to the other variable(s) of C_2 must be checked. Amongst others capacitor C_1 is found. Because its linear equation defines v, this component is checked and it has the same type as C_2. Both initial values are checked and if they are equal the equation $C_2\ i_1 = C_1\ i_2$ can be used to replace the original equation of C_2.

The general problem of a singular matrix A_{xx} is however in practice unsolvable. If not two parallel linear capacitors are found, an exhaustive search over all possible regions for a regular linearization is the only possible algorithm to solve this problem. Because this is very expensive, as these kinds of problems usually only occur in large problems, and because a regular linearization can in general not be found, a singular matrix induces in PLATO a failure to solve the given problem.

4 The Piecewise Linear Equations

4.1 Introduction

This chapter describes the methods employed to solve the piecewise linear equations. As described in Section 2.4, on existence of solutions, these equations can be written as the following LCP problem:

$$\begin{cases} w = M\,z + q \\ w \geq 0,\ \ z \geq 0,\ \ w \cdot z = 0 \end{cases}, \tag{4.1}$$

with

$$M = A_{zz} - A_{zx}\,A_{xx}^{-1}\,A_{xz}$$
$$q = a_z - A_{zx}\,A_{xx}^{-1}\,a_x + A_{zu}\,u$$

Several methods have been developed to solve an LCP. They can be divided into three classes: pivoting algorithms, integer labeling algorithms, and relaxation algorithms. For all known algorithms it has been proven that a solution will be found with certainty only if M belongs to a restricted class. The most important matrix classes are enumerated in Table 4.1. The classes *Positive Definite*, *Positive* and *Strict SemiMonotone* are usually abbreviated to PD, P and SSM respectively, while the class $\mathcal{L}$ has no other name. It is straightforward to prove that PD ⊂ P ⊂ SSM ⊂ $\mathcal{L}$.

Table 4.1. Matrix classes

class	class definition
PD	$\forall_{x\in\mathbb{R}^n, x\neq 0}\ [\ x^TMx > 0\ ]$
P	$\forall_{x\in\mathbb{R}^n, x\neq 0}\ \exists_k\ [\ x_k \cdot (Mx)_k > 0\ ]$
SSM	$\forall_{x\in\mathbb{R}^n_+, x\neq 0}\ \exists_k\ [\ x_k \cdot (Mx)_k > 0\ ]$
$\mathcal{L}$	$\forall_{x\in\mathbb{R}^n_+, x\neq 0}\ \exists_k\ [\ x_k > 0 \wedge (Mx)_k \geq 0\ ]\ \wedge$ $\forall_{x\in\mathbb{R}^n_+, x\neq 0, Mx\geq 0, x^TMx=0}\ \exists_{\text{diagonal matrix}\ \Lambda,\Omega\geq 0}$ $[\ \Omega x \neq 0 \wedge (\Lambda M + M^T\Omega)x = 0\ ]$

These classes are related to the number of solutions of the LCP (van Eijndhoven 1988):

- If the matrix M is in class P, the problem has one solution.
- If the matrix is in class SSM, the problem has an odd number of solutions.
- If the matrix is in class $\mathcal{L}$, the problem has either no solution, an odd number of solutions, or a bounded subspace of solutions.

The class P is also characterized by the fact that each pivot is positive, and it is therefore closed with respect to the pivot operation. Unfortunately, the class SSM is not closed under pivoting. All the following algorithms have global convergence properties, i.e. a solution is found independent of the starting value, if the problem is in the given class.

As stated before, the properties of the matrix M are not known in general, and in our simulator several models are used of which the matrix is not in one of the above mentioned classes. For example, the matrix of the discontinuity model of equation (2.27) is in $\mathcal{L}$, but not in SSM, and the hysteresis model of equation (2.25) has a matrix not in $\mathcal{L}$. To be able to solve as many systems as possible, an algorithm must be chosen which will find a solution for the largest matrix class. Because relaxation methods will only converge with certainty if the matrix is in PD (van Bokhoven 1981), these methods are not considered further. An integer labeling method finds, with the simplex method, an approximate solution of the LCP, and converges to a solution if the matrix is in the class SSM. Some path–following algorithms find an exact solution for a class larger than the class of $\mathcal{L}$–matrices (Jones 1986).

Because we do not want to restrict the modeling of the systems, one of the path–following algorithm is used in our simulator. These algorithms have the nice property that they either find a solution or fail quickly. Using a path–following algorithm has as a second advantage that the pivots are performed explicitly. If a solution is found, the vector z is zero, the vector w gives the current region, and the current linearization is available. A third advantage is that the algorithms finish in a limited number of steps. Although the path–following algorithms will, in worst case, perform all possible pivots, they tend to find a solution in a few steps, many times in the minimal number of steps.

The path–following algorithms are not specially suitable to find *all* possible solutions. The algorithms will find a region that is in a sense closest to the starting region. In contrary, the integer labeling algorithms are more suitable in finding more solutions. For the types of problems that we solve, it is usually enough to find one solution. Only if no solution is found, which does not happen often, finding all solutions might be an appropriate method to continue the simulation.

4.2 Path–following algorithms

There are several path–following algorithms. All these algorithms find a solution by tracing a continuous path through the (w, z) space. Notice that the con-

tinuity of the mapping is related to the continuity in the (w, z) space, and that a continuous path in the (w, z) space is also a continuous path in the x space. The two best–known algorithms are described in (Katzenelson 1965) and (Lemke 1968). Katzenelson's algorithm is a simple algorithm, which has as main drawback that it can only handle positive pivots: it converges only if the matrix is in class P. Lemke's algorithm performs better: it can handle every possible pivot, and will be successful if the matrix is in a class described in (Jones 1986). Instead of using Lemke's algorithm directly, a variant of this algorithm is used, described in (Van de Panne 1974). The advantage of using Van de Panne's algorithm above Lemke's is its behavior on zero pivots, and the possibility of a feasibility check.

The Katzenelson, Lemke and Van de Panne algorithms start as follows: a vector e (always non–negative) and a positive number λ are chosen such that $q + \lambda e \geq 0$. This choice gives the feasible point $(q + \lambda e,\ 0)$ in the (w, z)–space. λ is a measure of the error made by choosing $z = 0$. Therefore the objective is to decrease λ until 0 is reached, under the conditions $w \geq 0$, $z \geq 0$ and the complementarity condition $w \cdot z = 0$. Then a path in the (w, z)–space is found, a mapping of the line segment $\{\lambda e \mid 0 \leq \lambda \leq \lambda_{max}\}$. This mapping needs not to be a function, as more than one point in the (w, z)–space may be related to one value of λ.

All three algorithms trace the path in the following way: one free positive parameter, currently describing a piece of the path, is changed in one direction (increasing or decreasing) until a blocking row is found or the free parameter becomes zero. A blocking row (boundary) is the index i for which $w_i = 0$ and decreasing under influence of the free parameter. Then a pivot is performed and/or a new free parameter is chosen, after which the free parameter is changed, and the iteration is continued.

The Katzenelson algorithm, described in Algorithm 4.1, is the simplest of the three. Its free parameter is λ. The first step is to decrease λ until a blocking row k is found or λ reaches zero. If λ reaches zero a solution is found, otherwise a pivot on M_{kk} must be performed to continue. When this pivot is positive the path can continue in the same direction entering a new region. But for a negative or zero pivot the new region can not be entered by decreasing λ. Because the Katzenelson algorithm can only handle a decreasing λ it aborts when the pivot is not positive. An extension on this algorithm can be given by noticing that, after a negative pivot, the new region is entered by increasing λ. This only slightly increases the complexity of the algorithm.

To simplify the notation, in the Lemke and the Van de Panne algorithms the vectors w and z will denote different vectors before and after a pivot operation. One entry of w is swapped with one entry of z, and the resulting vectors are again denoted by w and z (e.g. after a pivot on M_{11} is $w = (z_1, w_2, \dots, w_{n_z})^T$ and $z = (w_1, z_2, \dots, z_{n_z})^T$.

Algorithm 4.1: The Katzenelson algorithm

```
initialize( );
while true do
    <step, k> := determine_blocking_row( );
    if step = ∞ then
        exit unsuccessful;  { no blocking row }
    fi;
    make_step( step );
    if λ = 0 then
        exit successful;
    fi;
    if M_kk > 0 then
        perform_pivot( k );
    else
        exit unsuccessful;
    fi;
od;
```

The Lemke algorithm, Algorithm 4.2, is more complicated than the Katzenelson algorithm. It uses λ as an extra (zeroth) element of z and e as an extra (zeroth) column of the matrix M. One aspect is that only one entry of z, denoted by j, is positive during the algorithm. In the starting position therefore $j = 0$. The free parameter is z_j. The algorithm starts by decreasing z_j until a blocking row, say $k \neq j$, is found. $M_{kj} \neq 0$, because w_k depends on z_j. Now z_k can be increased while w_k remains zero. To satisfy the condition that only one entry of z is positive, z_j and w_k are swapped: a pivot on M_{kj} is performed. The next variable, z_k, starts increasing, so set $j = k$. The algorithm terminates if $k = 0$ is found, i.e. λ reaches 0. To finish with λ in the z vector the last pivot on M_{0j} must be performed, where row 0 is the row currently related with λ. The algorithm fails when no blocking row is found.

The Van de Panne algorithm, Algorithm 4.3, is the most complicated of the three path–following algorithms described here. Its description uses the term *active variable*, which is either λ or a z_i and a *direction* (increase or decrease) which together determine the direction in which the path is pursued. A stack is used containing blocked variables, i.e. those active variables that are stopped by a blocking row. This stack, S, is transparent, i.e. all elements of the stack can be inspected, but the only available operations are *push* and *pop*. Each stack element is a record containing one variable (λ or a z_i) and its direction, *up* (+1) or *down* (−1). The algorithm starts with an empty stack and the active variable is the decreasing λ.

Algorithm 4.2: The Lemke algorithm

```
initialize( );
M*0 := e; z0 := λ; j := 0;
while true do
    <step, k> := determine_blocking_row( j );
    if step = ∞ then
        exit unsuccessful; { no blocking row }
    fi;
    make_step( step );
    perform_pivot( k, j );
    if k = 0 then
        exit successful;
    fi;
    j := k;
od;
```

The algorithm finds the path by tracing the active variable in its direction until a blocking row j is found or the active variable becomes zero. If no blocking row is found the algorithm fails. If M_{jj} is positive, a pivot operation is performed on it, and the path is continued in the new region by tracing the active variable in the same direction, like in the Katzenelson algorithm. If M_{jj} is negative, the pivot is also performed, but now the active variable can not maintain the same direction, because the new region is on the same side of the boundary. Therefore its direction is reversed (decreasing to increasing and vice versa) and the new region is entered.

The third possibility is a zero pivot. As explained earlier, the blocking row j is changed temporarily into an equality and one extra inequality $z_j \geq 0$ is added to the system. Now the path stays in the blocking boundary. This is equivalent with keeping the active variable fixed and following the path in the new direction of an increasing z_j. So the old active variable is pushed onto the stack and z_j becomes the new active variable.

After a new blocking row k is found, a block pivot is sought that releases as many variables from the stack as possible, i.e. it exchanges as many entries of w and z variables as possible. If a regular block or single pivot is found, it is performed and the related variables are popped from the stack. The active variable is set to the new top element of the stack or λ if the stack is empty, and its direction is reversed if the sign of the block pivot was negative. If no regular pivot can be found, the active variable is pushed on the stack and z_k becomes the new active variable.

There is one other possibility: the active variable becomes zero. If this is λ, the algorithm will terminate successfully. If this variable is not λ, it is popped from the stack, and the new active variable will be the one on top of the stack,

or λ if the stack is empty. To follow the path in the right direction, the direction must be reversed, since there was one negative pivot performed during the blocking of this variable.

Algorithm 4.3: The Van de Panne algorithm

```
initialize( );
act_var := λ; act_dir := down;
while true do
    <step, k> := determine_blocking_row( act_var );
    if step = ∞ then
        exit unsuccessful;  { no blocking row }
    fi;
    make_step( step, act_var, act_dir );
    if k = 0 then                          { λ became 0 }
        exit successful;
    elsif act_var = z_k then         { act_var became 0 }
        pop( 1 ); act_var := get_act_var( ); act_dir := – act_dir;
    else
        push( k, up );
        d := 0;   { size of the block pivot }
        i := 1;
        while i ≤ top_of_stack ∧ d = 0 do
            Sig := sign( M_{S[i].var, k} );
            if Sig ≠ 0 then
                d := top_of_stack – i + 1;
            fi;
            i := i + 1;
        od;
        if d > 0 then  { regular block pivot found }
            perform_block_pivot( d ); pop( d );
            act_var := get_act_var( );
            act_dir := act_dir * Sig;  { down <-> up }
        else
            act_var := z_k;
        fi;
    fi;
od;
```

It has been shown that both Lemke's algorithm and the Van de Panne algorithm follow the same path and have the same convergence properties (Van de Panne 1974). If only positive pivots are performed, this is the same path as followed by the Katzenelson algorithm. There is one important reason why the more complicated Van de Panne algorithm suits our purposes better than the Lemke algorithm does: the Van de Panne algorithm performs only block pivots. As has been proven (van Stiphout 1990), either a block pivot is performed with all pivots locally in some leaf cell, or a block pivot can be performed with pivots on the diagonal of A_{zz} in separate leaf cells. This will keep the block–matrix structure of the A_{zz} matrix intact. In contrary, the Lemke algorithm performs off–diagonal pivots only. If a blocking row is found in a cell differing from the starting one, the related pivot in the A_{zz} matrix is zero. This pivot can be performed by adding a row A_{x*} to the w row, pivoting and subtracting the same row, but this will create entries outside the current block structure. This is cumbersome in our implementation, because only the blocks are stored. The feasibility check of the Van de Panne algorithm is not used, and is therefore not described in Algorithm 4.3.

From the description of the Van de Panne algorithm it follows directly that it can handle any number of positive pivots. But a negative pivot must be followed by an other negative pivot later in the process, otherwise λ will not be bounded. Because λ will not change while the variables on the stack are manipulated, the stack is empty if a solution is found. So no solution is found *in* a facet of a region, i.e. on a discontinuous relation in the (x,u)–space.

The success of the algorithms depends on the choice of the vector e. The path which is traced depends on the initial direction in which a solution is sought. A choice for e is good if it will give a converging and short path (minimal number of pivots), while with another choice a path diverging from the solution may be found. The minimal requirement for the vector is that for all negative entries in w a positive entry in e is chosen, and that negative entries of e have a bounded value so that $q + \lambda e$ is initially positive. However, the termination of the algorithms is only guaranteed if e contains initially only positive entries (Lemke 1968).

The choice of e indicates which pivot is performed first. Therefore an indecisive choice for e is to set $e_i = -w_i$ if $w_i < 0$ and $e_i > 0$ if $w_i \geq 0$, because then the first pivot is chosen at random, depending on numerical rounding errors. This may lead to an unpredictable performance: if more solutions exist, the one that is found may differ in different situations, even if the initial situation is (nearly) the same.

In most applications, the vector e is chosen as $e_i = 1$ for all i, which gives good convergence of the algorithms. In our problems, however, the matrix M is block–diagonal and combined with linear equations. To use the sparseness of this system, a sparse vector e is chosen. Then as few as possible entries of w

have to be checked during the algorithm. Our experience is that the Van de Panne algorithm nearly always finds a solution with the following choice:

$$e_i = \begin{cases} 1 \text{ if } w_i < 0 \\ 0 \text{ if } w_i \geq 0 \end{cases} . \tag{4.2}$$

Then the rows with a large negative w_i will be in general pivoted earlier than the rows with a small negative w_i. This is logical, when the problem is considered in the x– domain. It is usually sensible to cross a facet closer to the starting point before crossing a facet further away in the direction of the solution.

To show that this choice is not always the best choice, consider the following problem, based on a model of a square function (van Eijndhoven 1984):

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + \begin{pmatrix} -2.5 \\ -1.5 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} . \tag{4.3}$$

The solution of this problem with the Van de Panne algorithm can be described with a few tableaux. First, notice that the solution can be found with one pivot on M_{22}. However, the algorithm uses three pivots, denoted in the next tableaux with a circle. The algorithm can be described by the different tableaux: first tableau 1 is created, containing the current matrix, the current e vector and the vector w for the indicated value of λ. It is inspected for a blocking row, which yields the pivot (on the (1,1) entry) to transform this tableau to tableau 2. In this tableau first λ is decreased until a blocking row is detected, in this case from 2.5 to 1.5. A pivot on entry (2,2) gives tableau 3, etc. In tableau 4 the final situation is encountered, in which λ reaches 0 and the final solution is found.

Table 4.2. The steps of the Van de Panne algorithm

tableau 1					tableau 2						tableau 3						tableau 4				
		e	$\lambda=$ 2.5				e	$\lambda=$ 2.5	$\lambda=$ 1.5				e	$\lambda=$ 1.5	$\lambda=$ 0.5				e	$\lambda=$ 0.5	$\lambda=$ 0
①	2	1	0	→	1	−2	−1	0	1	→	①	−2	1	1	0	→	1	2	−1	0	0.5
0	1	1	1		0	①	1	1	0		0	1	−1	0	1		0	1	−1	1	1.5

Although this problem suggests that the Van de Panne algorithm uses too many pivots, it must be noted that in this example this number of pivots is nearly inevitable. The four regions of this example are given in Figure 4.1.a as function of the vector q. Also the line $q = q_0 + \lambda e$, $\lambda \geq 0$, with q_0 the initial value of q, is shown. This line crosses three facets, so three pivots are performed. The related path in the x–domain is given in Figure 4.1.b. So in the x–domain an obvious path is followed, although a non–trivial path with less pivots can be found. Because the path in the x–domain is the interesting one, it does not matter that a longer path is followed.

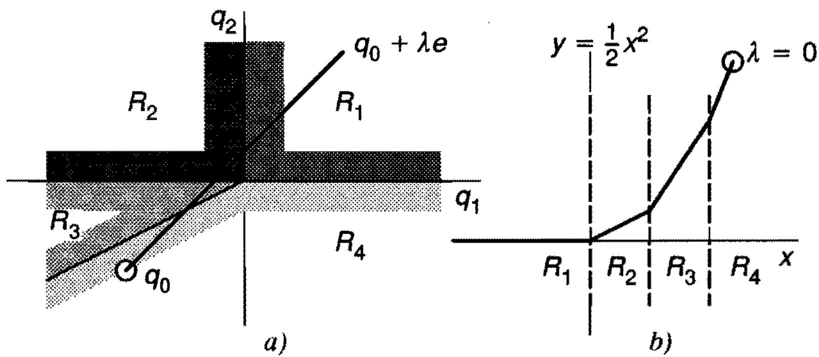


Figure 4.1. Topology (a) and linearization (b) of the square function in equation (4.3)

4.3 Implementation of the Van de Panne algorithm

The detailed algorithm that is implemented in our simulator is discussed here. The problems are shown that are introduced by the use of a global x vector and the distribution of the matrices over the components. As explained in the previous section, the algorithm uses some temporary variables and structures. A positive variable λ is used which will hopefully reach zero. A vector e is set with entries +1 or 0. Each element of the stack S is a record containing a variable (λ or z_i) and its direction, up (+1) or down (−1). The stack contains *top_of_stack* elements, which number is affected by the *push*() and *pop*() operations. To simplify the implementation, the free or active variable is always the top element of the stack, so λ is always $S[1]$. Furthermore, because the first blocking row is known, the start of the algorithm is shifted to halfway the loop of description 4.3.

Algorithm 4.4: Implementation of the Van de Panne algorithm

$z := 0$; $w := A_{xx}\ x + A_{zu}\ u + a_z$;
$\lambda := 0$; $e := 0$;
for $i := 1$ **to** n_z **whenever** $w_i < 0$ **do**
  $e_i := 1$;
  if $\lambda < -w_i$ **then**
    $\lambda := -w_i$; $row := i$;
  fi;
od;
if $\lambda = 0$ **then** { *the quadruple* (x, u, z, w) *is a solution* }
  exit successful;
fi;

```
w := w + λ * e; push( λ, down );
push( row, up );        { the first blocking row }
{ main loop }
while λ > 0 do
     d := 0;            { size of the block pivot }
     i := 2;            { skip λ }
     while i ≤ top_of_stack ∧ d = 0 do
          Sig := matrix_sign( S[i], S[top_of_stack] );
          if Sig ≠ 0 then
               d := top_of_stack − i + 1;
          fi;
          i := i + 1;
     od;
     if d > 0 then  { regular block found }
          perform_block_pivot ( d ); pop( d );
          S[top_of_stack].dir := S[top_of_stack].dir * Sig;  { down ↔ up }
     fi;
     <step, row> := determine_blocking_row( S[top_of_stack] );
     if ( step = ∞ ) then
          exit unsuccessful;  { no blocking row }
     fi;
     make_step( step );
     if row ≠ λ then
          if ( row = S[top_of_stack].var ) then
               pop( 1 ); S[top_of_stack].dir := − S[top_of_stack].dir;
          else
               push( row, up );
          fi;
     fi;
od;
```

There are several points in this algorithm where simple routines must be replaced by more elaborate ones.
Denote $p = S[i].var$ and $q = S[top_of_stack].var$, then the first value that must be calculated (in *matrix_sign*()) is the sign of M_{pq}, or $(A_{zz} - A_{zx} A_{xx}^{-1} A_{xz})_{pq}$, for several p's. To accomplish this, first the temporary variable $\tilde{b} = (A_{xz})_{*q}$ is created. The vector $\tilde{x} = A_{xx}^{-1} \tilde{b}$ is calculated from it, with a forward and backward substitution. This vector is then used for each p, because $M_{pq} = (A_{zz})_{pq} - (A_{zx})_{p*} \tilde{x}$. Notice that only one leaf cell generates $\tilde{b}$, due to the

block structure of A_{xz}. The calculation of each M_{pq} involves only those leaf cells, that already have a blocking row on the stack.

Secondly the pivot(s) must be performed if the block is regular. The routine *perform_block_pivot*() performs an update of the local matrix as well as an update of the global matrix. The update of the local matrix is performed easily, with one exception. It may happen that the algorithm finds a nonzero pivot M_{pq}, while the corresponding A_{zz} entry is zero. That is only possible if there exists a nonzero entry $(A_{xz})_{kq}$ in the local A_{xz}, according to (van Stiphout 1990), so then a multiple of the row k can be added to the pivot row p to create a nonzero A_{zz} entry. As noted before, a block pivot in more than one component can be performed with diagonal pivots, i.e. with pivots on the diagonal of A_{zz}. The update of the global matrix is easily found by a rank m update (Algorithm 3.3) with the pivot row(s) of A_{zx} and the pivot column(s) of A_{xz}. As already described, this rank m update is usually performed as a rank 1 update, but in case of a block pivot it is performed as a full rank m update.

The third action is the determination of a new blocking row and a step which can be taken. This depends on the free variable. There are two possibilities: if this variable is λ, the step is made in the direction of the e vector, otherwise in the direction of the A_{*z} column of the active variable. In both cases this has a local and a global effect. The local effect is easily calculated, but the global effect once again needs solving the global equations. A change in the free parameter will also change the x vector. This effect is taken into account by calculating again $\tilde{x} = A_{xx}^{-1}\tilde{b}$, with $\tilde{b} = e$ or $\tilde{b} = (A_{xz})_{*q}$. Note that if the free variable is λ, every leaf cell is visited during the calculation of $\tilde{b}$. Otherwise only one leaf cell generates nonzero entries in $\tilde{b}$. With this vector $\tilde{x}$, the global effect of changing the free parameter can be calculated in every leaf cell.

The last routine with global and local influence is *make_step*(). Here the free variable is changed, and all other variables are updated. With the help of the previously determined value of $\tilde{x}$, the local update is performed. The global x vector also changes, but fortunately with an update vector $step \cdot \tilde{x}$, and the previously calculated vector can be used again.

4.4 The DC solution

The Van de Panne algorithm is applied in two situations: to find the initial solution and to find a new linearization during the integration. Both problems are solved with Algorithm 4.4, but the latter of these two needs a different initialization, because the current region is not yet left.

During the integration, first the time point t is determined at which a facet is actually crossed. This depends of course on the dynamic equations and will be discussed in the next chapter. Then the values are updated up to this time point. A useful initialization can be found by examining the situation at the

time $t+\delta$ for some positive δ: $w(t)$ is a valid solution, but $w(t+\delta) \cong w(t) + \delta\dot{w}(t)$ is not valid for any δ. This suggests that the choice $\lambda = \delta$ and $e = -\dot{w}(t)$ is a valid initialization. Taking the limit for $\delta \downarrow 0$ gives the exact initialization for λ: $\lambda = 0$, direction down. Also the criterion for exiting the loop, $\lambda > 0$, is adjusted to ensure that the loop is executed at least once. In PLATO, the search for a new linearization is initialized this way. Because in nearly all cases only one boundary is crossed, or, in the case of a discontinuity, two, a few pivots are enough to find the new region and linearization, and this application of the algorithm nearly never gives problems.

The initial solution problem in general is also simply solved by the simulator. Although the initial region and the initial value do not match, the Van de Panne algorithm has such strong convergence properties that nearly always a solution is found. The initial solution problem is one of the more difficult problems for traditional circuit simulators, because finding a valid initial solution with a Newton–Raphson iteration is often unsuccessful, due to its local convergence property.

As described in Chapter 2, the differential equation needs an additional equation to define a unique solution. In circuit simulation, except the condition $u(t_0) = u_0$, also the condition $\dot{u}(t_0) = 0$ is used. A solution satisfying this condition is called a DC solution. More than one DC solution may exist, but finding one solution is currently satisfying. In general, a circuit may contain several components whose local u vector should not be set to zero, for example a clock or an oscillator. Therefore we restrict the DC problem to: find a solution (x, u, w, z) so that $\forall_{i \in I} [\ \dot{u}_i(t_0) = 0\ ]$ for a given set I. Because it is nearly impossible to determine whether a component should belong to the set I, the user of the simulator must supply this set. The continuity of the modeling implies that a solution exists in which for example a clock is in a stable state. Because the Van de Panne algorithm finds this solution without problems, it is necessary to specify the set I accurately.

To determine a DC solution, the equations must be transformed. In the original equations $\dot{u}$ is a function of x, u and z. To describe a DC problem, this is combined with $\dot{u} = 0$. Together with these extra linear equations a linear system is formed to solve both the x and the u variables. This is only possible if the resulting system is regular, which is not always the case. Consider for example a system with the following type of equations:

$$\begin{cases} A_{xx}\,x + \qquad\qquad A_{xz}\,z + a_x = 0 \\ A_{ux}\,x + \qquad\qquad A_{uz}\,z + a_u = \dot{u} = 0 \\ A_{zx}\,x + A_{zu}\,u + A_{zz}\,z + a_z = w \\ w \ge 0,\ \ z \ge 0,\ \ w \cdot z = 0 \end{cases} \quad . \tag{4.4}$$

In this case the linear system will not be regular, because there is no direct equation for u. Therefore a different formulation of the problem is used, which always will create a regular system, if that exists. Another advantage of this

new formulation is that the global equations are not extended. Instead of adding the equation $\dot{u}(t_0) = 0$, the differential equation is changed in an equation defining u. This means that, instead of $\dot{u}$ being a function of u, x and z, now u is a function of $\dot{u}$, x and z, and that instead of the initial condition $u(t_0) = u_0$ now $\dot{u}(t_0) = 0$ is used. So u and $\dot{u}$ are swapped in the differential equation. This is the same situation as with the w and z vectors, of which also entries may swap sides. So pivoting on A_{uu} will yield this new formulation.

Pivoting on A_{uu} can have the same kind of problems as pivoting on A_{zz}. First notice that this pivoting can be performed locally, so the global matrix structure (in particular the block–diagonal u and z parts of the system) will not change. But it may happen that the diagonal of A_{uu} has zero entries. This might be solved by adding a (local) linear equation to the row. If x depends on u, the matrix becomes regular, and the nett effect of the pivot is the replacement of one or more linear equation by one or more equation of the form $\dot{u}_i = (A_{ux}\ x + A_{uz}\ z + a_u)_i = 0$. However, as was shown in the previous paragraph, such a linear equation may not exist. Another pitfall is that possibly $A_{ux} = 0$, in which case such an addition will lead to a singular A_{xx} matrix, with a row containing only zero entries.

To overcome these two problems it is sometimes not enough to swap only the u and $\dot{u}$ vectors, but it may be necessary to swap also some entries of w and z, in order to find a system of equations that can be solved. This means that the system is put into another starting region, where the linearizations differ from the original ones. There is again the possibility, that swapping entries of w and z will not solve the problems, because either w depends not on u or $\dot{u}$ depends not on z. In both cases a linear row must be available, that is added to the relevant $\dot{u}$ or w row. This linear row is available in general, because otherwise a u variable is spurious in the equations. Now a block pivot is found to swap the entries of u or $\dot{u}$, as well as one or more entries of w and z. This block pivot exists, assuming that u, w, and z have all influence, directly or indirectly, on x.

This procedure leads to Algorithm 4.5, after which the equations are suitably changed to solve a DC problem. In this algorithm a simple optimization is performed, in order to minimize the number of pivots needed. If it is possible to find a j with $(A_{uz})_{ij} \neq 0$ and $(A_{zu})_{ji} \neq 0$, only on these two entries a pivot is performed. If such a j can not be found, an additional subblock must be found in the A_{zz} part of the (local) matrix.

Algorithm 4.5: Swapping u and $\dot{u}$.

```
for i := 1 to n_u whenever i ∈ I do
    if (A_uu)_ii ≠ 0 then
        perform_u_pivot( i );
    elsif (A_ux)_i* ≠ 0 ∧ (A_xu)_*i ≠ 0 then
        j := add_row_to_u( i );   { j not used }
        perform_u_pivot( i );
    else
        if (A_uz)_i* = 0 then          { add linear equation }
            j := add_row_to_u( i );
        else
            j := get_index( i );   { (A_uz)_ij ≠ 0 }
        fi;
        if (A_zu)_ji = 0 then          { add linear equation }
            add_row_to_pl( j, i );
        fi;
        pivot_u_and_pl( i, j );
    fi;
od;
```

To return to the original equations after a DC solution is found, Algorithm 4.5 is executed a second time. In practical applications this algorithm has never failed, although not always a DC solution has been found. This is caused by the difficulty of the problem, because in many cases the initial region and linearization are "too far" from a solution to be found. This might in some cases be based on our choice for the vector e. For some circuits the DC solution can be found by manually putting some components in the right state, but that is contrary to the policy that a user may enter any problem and the simulator will solve it.

5 The Dynamic Equations

5.1 Introduction

In the previous chapters, algorithms were discussed to solve linear equations or piecewise linear equations. In this chapter the numerical solution of the ordinary differential equation is investigated. In Section 2.4, on the existence of solutions, it has already been shown that the basic problem to solve is a non-linear ordinary differential equation in u:

$$\begin{cases} \dot{u}(t) = A(u(t))\ u(t) + a(u(t), t) \\ u(t_0) = u_0 \end{cases} \tag{5.1}$$

for $t_0 \leq t \leq t_e$.

The matrix $A(u(t))$ is piecewise constant, and its value is determined by the linear complementarity problem. The vector $a(u(t), t)$ is the sum of a piecewise constant vector depending on $u(t)$ (like the matrix), and a vector depending only on the time, through the inputs of the system. The solution of equation (5.1) can be given in an exact but implicit formula:

$$u(t) = e^{A_i (t-t_i)}\ u(t_i) + \int_{t_i}^{t} e^{A_i (t-\tau)}\ a(\tau)\ d\tau \tag{5.2}$$

for $t_i \leq t \leq t_{i+1}$. Here e^A is the matrix exponent, defined for a square matrix A by $e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}$, A_i is the current linearization matrix, and t_i is the time point at which a new region is entered, i.e. $A(u(t))$ changes from A_{i-1} to A_i.

One of the main problems is the determination of the time points t_i, at which a new region must be found. But even if the time points t_i are known a priori, formula (5.2) is not explicitly used in general, because the calculation of e^{A_i} is expensive and numerically unstable (Moler and Van Loan 1978). Instead, numerical methods are employed (*integration methods*) to find an approximate solution of equation (5.1). Which integration method must be chosen is

discussed in the next section. It depends on the main properties of the methods: accuracy and efficiency, and on the type of problems that are solved. In Section 5.3 the implementation of the chosen integration methods is discussed, as well as the determination of the grid of time points, in Section 5.4 the multirate integration is introduced to employ further efficiency. In Section 5.5 an explicit exponential integration method is investigated. The combination of the integration method with the piecewise linear mapping is discussed in Section 5.6.

5.2 Properties of integration methods

Numerous integration methods have been proposed to solve the initial value problem (5.1). A good overview of integration methods can be found in (Hairer *et al.* 1987). Each method has its own advantages and disadvantages, so the choice for a particular method is guided by the characteristics of the problem that must be solved. In this section, those properties of integration methods are introduced that are important on our type of problems.

An integration method chooses a grid of time points, and on each grid point an approximation of the solution is calculated. The determination of this grid is explained in the next section. The grid consists of the set of points t_i, $0 \le i \le N$, with $t_0 < t_1 < \dots < t_N = t_e$. The time steps are denoted by $h_i = t_{i+1} - t_i$. If the set of time points is equidistant, the uniform time step is denoted by h. For now, the time points are assumed to be equidistant. In this section, the matrix $A(u(t))$ and the vector $a(u(t), t)$ are taken to be independent of $u(t)$, and $y(t)$ denotes the calculated solution of problem (5.1). $u_{(i)}$ denotes $u(t_i)$. In principle, $y(t)$ is known only in the points t_i.

A first aspect of an integration method is the accuracy, i.e. the difference between the calculated solution $y(t)$ and the exact solution $u(t)$. This difference should be less than a user–specified tolerance ε, i.e.

$$\| y(t_i) - u(t_i) \| < \varepsilon \tag{5.3}$$

for all i, with $\| \cdot \|$ a suitable vector norm. However, this criterion is difficult to check, because the exact solution is not known for most problems. Instead, a less strict criterion is used, based on the local behavior. Let $u'_i(t)$, in our case, be defined on the interval $[t_i, t_{i+1}]$ as the exact solution of:

$$\begin{cases} \dot{u}'_i(t) = A\ u'_i(t) + a(t) \\ u'_i(t_i) = y(t_i) \end{cases} \tag{5.4}$$

The *Local Truncation Error* (LTE) in time point t_i, LTE_i, is then defined as:

$$LTE_i = \| y(t_{i+1}) - u'_i(t_{i+1}) \| \ . \tag{5.5}$$

For each integration method, the LTE can be determined. Its form is in general:

$$LTE_i = Ch^{q+1} \| u'^{(q+1)}_i(\xi_i) \| \ , \tag{5.6}$$

with q an integer called the *order* of the method, C a real constant, $u'^{(q+1)}_i$ the $(q+1)^{\text{th}}$ derivate of u'_i, and $t_{(i)} < \xi_i < t_{(i+1)}$. Equation (5.6) implies that the method is exact if the solution is a polynomial of order q.

An integration method is called *convergent* if the calculated solution converges to the exact solution if $h \to 0$. If $q > 0$, and if the method satisfies an extra stability condition, it is convergent (Hairer *et al.* 1987). For convergent methods of order q, the global error is $O(h^q)$. In the next section is explained, how the LTE is used to determine the time steps, so that a small step is taken if the LTE is large, and vice versa.

A second aspect of an integration method is its efficiency, which is expressed by the number of calculations of $\dot{u}$ performed by the integration method. Efficiency and accuracy are in general conflicting. Each calculation of $\dot{u}$ involves the calculation of x, so this operation is expensive. From equation (5.6) follows that for most differential equations the LTE decreases for higher order. There are two different schemes to achieve a high order, by employing a *one–step* method or by employing a *linear multistep* method. A one–step method uses the values of u and $\dot{u}$ at the current grid point and possibly at intermediate time points not on the grid. A linear multistep method uses the values of u and $\dot{u}$ at the current and previous grid points. But, for a one–step method the number of calculations of $\dot{u}$ is equal to its order, while a linear multistep only needs one calculation. Therefore, a linear multistep method is more efficient than a one–step method.

A third aspect of a method is the global error made if large time steps h are taken. This only happens if a *stiff* problem is solved, i.e. if the time constants of the problem have different orders of magnitude, so u has a component converging fast and a component converging slow to its final value. As the length of the time interval is related to the slow component, the fast component may introduce instability in the later part of the time interval. Integration methods that do not show this behavior are called *A–stable*. For these methods, the time step may be chosen as large as the LTE_i indicates. In circuit simulation, most problems are stiff.

A–stability of a method is determined with the one–dimensional linear differential equation $\dot{u} = \lambda u$, with $\lambda \in \mathbb{C}$. A region in the complex plane, called the *region of absolute stability*, is defined as the set of values $h\lambda$ for which the calculated solution of this differential equation is bounded. If this region contains the left half of the complex plane, $\{ z \in \mathbb{C} \mid \text{Re}(z) < 0 \}$, the method is A–stable.

Because A–stable methods have an order of at most 2 (Dahlquist 1959), a less strong criterion has been developed, called *A(α)–stability*. A method is A(α)–stable if the region $\{ z \in \mathbb{C} \mid -\alpha < \pi - \arg(z) < \alpha \}$ is contained in the region of absolute stability. To achieve A(α)–stability, it is necessary to use in the method the value of $\dot{u}$ at the next grid point. So only some implicit methods are A(α)–stable.

The only methods that are convergent, use a minimal number of calculations and are A(α)–stable, are several implicit linear multistep methods. Their general form is:

$$u_{(i+1)} = \sum_{j=0}^{p} a_j \, u_{(i-j)} + h \sum_{j=-1}^{p} \beta_j \, \dot{u}_{(i-j)} \ , \tag{5.7}$$

with p the number of previously calculated values, and a_j and β_j real coefficients. If the time steps are not uniform, these coefficients can depend on the time steps.

Of these methods, the most important ones are the Trapezoidal Rule (TR), and the BDF methods, popularized by Gear (Gear 1971). There are two reasons to choose a method with a low order:

- If a discontinuity occurs, the high order method must be restarted with a number of steps by a lower order method. Because discontinuities may occur often, this means that then the average order is relatively low with high overhead costs.
- The multirate method, as will be discussed in the next section and in Section 6.3, is much more efficient if the order of the integration method does not exceed 2.

For these reasons only a few relatively simple integration methods have been implemented in PLATO: the one–step BDF (BE, the Backward Euler method), the one–step Trapezoidal Rule, the two–step BDF, and a less used two–step A–contractive method (ACF). These formulae are summarized in Table 5.1 with their LTE.

Table 5.1. Integration methods in PLATO with their LTE

name	formula	LTE
BE	$u_{(i+1)} = u_{(i)} + h\dot{u}_{(i+1)}$	$-\frac{1}{2}h^2y^{(2)}(\xi)$
TR	$u_{(i+1)} = u_{(i)} + \frac{1}{2}h(\dot{u}_{(i+1)} + \dot{u}_{(i)})$	$-\frac{1}{12}h^3y^{(3)}(\xi)$
BDF	$u_{(i+1)} = \frac{4}{3}u_{(i)} - \frac{1}{3}u_{(i-1)} + \frac{2}{3}h\dot{u}_{(i+1)}$	$-\frac{2}{9}h^3y^{(3)}(\xi)$
ACF	$u_{(i+1)} = \frac{4}{5}u_{(i)} + \frac{1}{5}u_{(i-1)} + \frac{2}{15}h(5\dot{u}_{(i+1)} + 2\dot{u}_{(i)} + 2\dot{u}_{(i-1)})$	$-\frac{2}{9}h^3y^{(3)}(\xi)$

5.3 Implementation of the integration methods

Applying formula (5.7) on a general differential equation $\dot{u}(t) = f(u(t)) + g(t)$ gives, after splitting $\dot{u}_{(i+1)}$:

$$\begin{aligned} u_{(i+1)} - h\,\beta_{-1}\, f(u_{(i+1)}) = \\ \sum_{j=0}^{p}(a_j\, u_{(i-j)} + h\,\beta_j\, \dot{u}_{(i-j)}) + h\,\beta_{-1}\, g_{(i+1)} \ , \end{aligned} \tag{5.8}$$

where $g_{(i+1)}$ is the value of $g(t_{i+1})$.

The right hand side of this equation is known, but the left hand side is an implicit relation in $u_{(i+1)}$, because $\beta_{-1} \neq 0$. In a traditional simulator like SPICE, equation (5.8) is solved by a Newton–Raphson iteration, which is expensive and converges only locally. Using piecewise linear models avoids such an algorithm, because the equation is transformed into an implicit matrix equation:

$$(I - h\beta_{-1} A) u_{(i+1)} = \sum_{j=0}^{p} (\alpha_j \, u_{(i-j)} + h \, \beta_j \, \dot{u}_{(i-j)}) + h \, \beta_{-1} \, a \; . \tag{5.9}$$

Equation (5.9) can be solved directly, yielding the value of u at the next time point.

The linear multistep formula is not applied to a simple linear differential equation, but to the linear differential–algebraic system

$$\begin{bmatrix} A_{xx} & A_{xu} \\ A_{ux} & A_{uu} \end{bmatrix} \begin{pmatrix} x \\ u \end{pmatrix} + \begin{pmatrix} a_x \\ a_u \end{pmatrix} = \begin{pmatrix} 0 \\ \dot{u} \end{pmatrix} \; . \tag{5.10}$$

Combining equations (5.9) and (5.10), the resulting system that must be solved is

$$\begin{bmatrix} A_{xx} & A_{xu} \\ A'_{ux} & A'_{uu} \end{bmatrix} \begin{bmatrix} x_{(i+1)} \\ u_{(i+1)} \end{bmatrix} + \begin{pmatrix} a_x \\ a'_u \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \; , \tag{5.11}$$

with

$$\begin{cases} A'_{ux} = -\, h \, b_{-1} \, A_{ux} \\ A'_{uu} = I - h \, b_{-1} \, A_{uu} \\ a'_u \;\; = h \, b_{-1} \, a_u + \displaystyle\sum_{j=0}^{p} (\alpha_j \, u_{(i-j)} + h \, \beta_j \, \dot{u}_{(i-j)}) \; . \end{cases}$$

Notice that the matrices depend on the time step h. A disadvantage of a variable step two–step method is that, although the formulae are not given here, the matrices depend not only on the next step, but also on the previously applied step. Therefore the matrices change *twice* if the step size changes.

The linear system (5.11) can only be solved if the previous values $x_{(i)}$ and $u_{(i)}$ are known. The initial value is simple, because $u_{(0)} = u_0$ is given. The value $x_{(0)}$ is calculated with the explicit equations (5.10). Now a one–step method (BE or TR) can be applied to find the values $x_{(1)}$ and $u_{(1)}$, after which a two–step method can be used.

The linear system (5.11) can not be solved directly, because the x and u equations are stored in different places: the A_{xx} matrix is global, the other matrices are scattered over the leaf cells. Therefore the $u_{(i)}$ variables are eliminated from the other equations. This is simple, because the matrix A'_{uu} is block di-

agonal with small blocks (usually only 1x1 or 2x2), always regular, and its inverse can be calculated block by block, leading to:

$$u_{(i+1)} = -A'^{-1}_{uu} \left(A'_{ux} \, x_{(i+1)} + a'_u \right) \; . \tag{5.12}$$

Equation (5.12) is substituted in the linear equations, and since the A_{*u} matrices are block diagonal, it has only limited influence on A_{xx} and a_x:

$$\begin{aligned} A'_{xx} &= A_{xx} - A_{xu} \, A'^{-1}_{uu} \, A'_{ux} \\ a'_x &= a_x - A_{xu} \, A'^{-1}_{uu} \, a'_u \end{aligned} \; . \tag{5.13}$$

If the A'_{uu} matrix of a leaf cell has dimension m, the update on the A_{xx} matrix is for each leaf cell a sparse rank m update, which is performed efficiently with Algorithm 3.3. With the updated matrices of (5.13), the new value of $x_{(i+1)}$ is calculated, which is used to calculate $u_{(i+1)}$ with (5.12).

The method sketched above can be simplified, if it is allowed to apply a simpler integration method. In general, an implicit method must be applied, but for a problem with $A_{uu} = A_{ux} = 0$, i.e. $\dot{u}$ is (piecewise) constant, the explicit Forward Euler method solves the problem exactly. Then equation (5.11) has the following different terms:

$$\begin{aligned} A'_{ux} &= 0 \\ A'_{uu} &= I \\ a'_u &= u_{(i)} + h \, \dot{u}_{(i)} \; . \end{aligned}$$

Therefore the A_{xx} matrix does not need to be updated, and the linear system with the vector a'_x of formula (5.13) can be solved directly. The possible application of the Forward Euler method can be detected directly, because then $\ddot{u} = 0$.

Step size control

The time grid of an integration method is always determined dynamically. It must be chosen to satisfy the specified tolerance and to minimize the number of time points. The specified tolerance contains two terms to control the error, the absolute tolerance ε_{abs} and the relative tolerance ε_{rel}. Each LTE_i must satisfy $\| LTE_i \| \le \varepsilon_{abs} + \varepsilon_{rel} \| u'_{(i)} \|$. The maximal time step $h_{max}(t_i)$ that is allowed is determined by using equation (5.6):

$$h_{max}(t_i) = \left[\frac{\varepsilon_{abs} + \varepsilon_{rel} \| u'_{(i)} \|}{C \| u'^{(q+1)}_i(\xi_i) \|} \right]^{\frac{1}{q+1}} \tag{5.14}$$

From this relation, a good indication for the maximal time step can be determined. Instead of using $u'_i(t)$, the calculated value $y(t)$ is used. The value of $u'^{(q+1)}_i(\xi_i)$ is approximated by calculating divided differences on the $y_{(i)}$. Using $\dot{u}'_i(t_{i+1}) \doteq \dfrac{u'_i(t_{i+1}) - u'_i(t_i)}{h_i}$, $\ddot{u}'_i(t_{i+1}) \doteq \dfrac{\dot{u}'_i(t_{i+1}) - \dot{u}'_i(t_i)}{h_i}$, etc., and by

approximating $u'_i(t)$ with $y(t)$, etc., an estimation for $h_{max}(t_i)$ is found. Because this is only an approximation, usually a step $h_i \approx 0.9h_{max}(t_i)$ is chosen.

Notice that this scheme is implicit, and is therefore only useful to check afterwards whether the step was not too large. An explicit scheme is found in the following way:

- Start the integration by calculating the size of one step of the Forward Euler method. For this method, $\ddot{u}'_i(\xi_0)$ is approximated by $\dot{y}(t_0)$. Because the step has no influence on the matrices, this value is calculate directly from $\dot{u}(t_0)$ and $\dot{x}(t_0)$.
- Instead of taking this step, a one–step implicit integration method is applied (BE or TR), with the step size of the Forward Euler method. Before the step is actually made, it is checked if it is not too large. This is possible, because $y(t_1)$, etc. can be calculated exactly without actually performing a step. If the step is too large, the newly calculated maximal time step is used as the new step.
- In the next steps, the estimation of $h_{max}(t_i)$ of the previous step is used as a new step.

One modification to this method is made for efficiency reasons: if the new step and the old step are about the same, the old step is used. In that case the matrices do not need to be updated, which operation is expensive. Some other modifications of the choice of the step size are made for the application of the multirate integration, as explained in the next section.

5.4 Multirate integration

If the system that is simulated is large, it is expected that most parts of the circuit are at rest at many time intervals. These parts at rest can be integrated with a much larger time step than the active components, without loss of accuracy. It saves a considerable amount of computer time, if the implementation can use this property. Using different integration steps at the same time point is called *multirate integration*.

Splitting a system is applied with much success in simulators based on Waveform Relaxation, where usually the system is statically divided into parts, and each part is solved independently of the others. The Waveform Relaxation does not perform well on strong feedback systems, because these can not be partitioned in a profitable way. On these problems the method performs at best as a multirate method.

Multirate integration can also be applied in direct integration methods, but this is not a common practice. Only in recent years multirate integration has been investigated theoretically and practically (Gear and Wells 1984; de Almeido *et al.* 1989; van Eijndhoven *et al.* 1990). In this section it is shown that multirate integration fits well to the piecewise linear models with a linear multistep integration method. The necessary partitioning is performed dynamically, which is more attractive than a static partitioning.

To explain the properties of multirate integration, consider a linear dynamic system, whose solution can be split in two parts, a fast u_1 and a slow u_2 part:

$$\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} + \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} . \tag{5.15}$$

For simplicity the Backward Euler integration rule will be applied, but with two different step sizes for the two parts. To simplify the equations, $t_0 = 0$ is chosen. Let $h_2 = Kh_1$, with K integer, so the grid points of the integration of u_2 form a subset of those of u_1. Applying the BE rule on (5.15) gives:

$$\begin{cases} (I - h_1 A_{11})\, u_1(h_1) - h_1 A_{12}\, u_2(h_1) = u_1(0) + h_1 a_1 \\ - h_2 A_{21}\, u_1(h_2) + (I - h_2 A_{22})\, u_2(h_2) = u_2(0) + h_2 a_2 \end{cases} . \tag{5.16}$$

This can not be solved directly, because only the values at the time point $t = 0$ are known. It is, however, possible to find a good approximate solution. Two methods can be chosen: extrapolate u_2, or extrapolate u_1. In the first case u_2 is extrapolated in the time points $t = kh_1$ for $k = 1, 2, \ldots, K-1$, and these values are used to calculate u_1 in these time points. Then u_1 and u_2 are calculated at $t = h_2$. The alternative method is to extrapolate u_1 to the time point $t = h_2$, calculate $u_2(h_2)$ and interpolate u_2 on the interval $[0, h_2]$ to calculate u_1. The first of these methods is called fastest–first, the second slowest–first (Gear and Wells 1984).

Both alternatives differ theoretically not, because the coupling between the two parts is low, i.e. both A_{21} and A_{12} are small, so no large error is made. If the coupling is strong, only A_{12} can be large; if A_{21} is large, u_2 will change if u_1 changes, and it is therefore not slow. So the terms $h_2 A_{21}$ and $h_1 A_{12}$ can have comparable norm and the error made by the interpolation and extrapolation will be of the same size.

There is, however, an important practical difference. First, in practical situations the fast parts of the system are usually created by parasitic elements. Their effect on the slower parts of the circuit can mostly be ignored, i.e. A_{21} is very small. The second difference shows up in the fastest–first scheme when the integration of the slow part is erroneous at the next time point h_2. Then the extrapolation of u_2 was also wrong, and (part of) the integration of u_1 can be wrong. Therefore the integration should restart at the time point 0 with a smaller h_2. So the situation at time 0 had to be saved, which is very expensive.

It is much easier to use a slowest–first scheme: if the integration of u_2 becomes erroneous halfway the interval $[0, h_2]$, its integration can restart from the current time point with an error equal to the interpolation error. Therefore only a slowest–first scheme has been implemented in PLATO. This scheme is more compact than the scheme of Gear and Wells, because the first steps of both rates are calculated together.

The most important properties of a multirate integration are the same as those of traditional integration methods: the precision and the efficiency. These properties are compared to those of its basic integration method with a uniform step.

The precision of a multirate integration method depends not only on the precision of the basic integration method, but also on the precision of the interpolation and the extrapolation. It is shown (Gear and Wells 1984) that the LTE for the fast part contains two terms: the LTE of the basic integration method, $-\frac{1}{2}h_1^2\ u^{(2)}(\xi)$ for BE, and the expression $(I - h_1A_{11})^{-1}h_1A_{12}\varepsilon_2$, with ε_2 the interpolation error. For other integration methods this second expression has terms with comparable norm. Due to the factor h_1 in this expression, the error in the interpolation scheme may be one order lower than the one in the basic integration method.

The error in the slow method can be written as the LTE of the basic integration method plus the expression $(I - h_2A_{22})^{-1}h_2A_{21}\varepsilon_1$, with ε_1 the extrapolation error. This shows that the extrapolating polynomial can also be chosen one order lower than is expected from the integration method.

In uniform–step integration methods the LTE is used to select a step size that maintains the error within user–specified bounds. In multirate integration this is not directly possible, because the LTE consists of two or three terms: the LTE of the basic integration method, the interpolation error, and the extrapolation error.

If the interpolating and extrapolating polynomials have the same order as the basic integration method, for example linear interpolation with the first order BE method, the interpolation and extrapolation errors in the LTE may be neglected with respect to the integration error. Then the maximal step size can be derived from the error specification.

But if a linear interpolation and extrapolation are combined with a second order integration method (TR, BDF, or ACF), the interpolation and extrapolation introduce errors of the same order as the integration error. To ensure that these errors are within the user–specified bounds, the time steps are then chosen with a lower order accuracy method: in formula (5.14), the order q is taken 1, the order of the interpolation.

The stability of a multirate integration method is related to the stability of its uniform–step variant. If the matrix in (5.15) is block triangular, the method is A(α)–stable if the basic integration method is A(α)–stable. If both A_{12} and A_{21} are nonzero, the error in the extrapolation will dominate the LTE for a large step, so the step may not increase freely. The following proposition can be derived in that case (Gear and Wells 1984): there exists a K such that the method is A–stable if $h_2 \| A_{21} \| < K$ and if the basic integration method applied on the matrices A_{11} and A_{22} is strictly A–stable, i.e. the terms $h_1\lambda_1$ and $h_2\lambda_2$, with λ_1 and λ_2 eigenvalues of A_{11} and A_{22}, are in the interior of the

region of absolute stability. A multirate method is not A–stable if the basic integration method is not A–stable.

Clustering

The efficiency of a multirate method depends on the ratio of the steps that are taken, the costs of interpolation and extrapolation, and the overhead in the program. In the next chapter it is shown that the chosen implementation of the interpolation and extrapolation is cheap with respect to the full calculation of each variable at each time point. The overhead in the program consists of three parts: the check in the slow components, the selection of the time steps, and the selection of the variables at each time point.

The slow components must be checked during their interpolation, to ensure that the error remains within the user–specified bounds. Usually the extrapolation of the fast components is accurate enough, so the integration of the slow components is accurate enough. But if this extrapolation is questionable, the error that is made should be kept bounded. Therefore the estimation of $\| u^{(q+1)}(\xi) \|$ is monitored during the interpolation. If it grows during the interpolation interval, the time step was chosen too large and should be decreased. As explained above, this is not easy, so the interpolation is stopped instead. This gives an error that is bounded by the interpolation error, because until this time point the interpolation was sufficiently accurate. The interruption of the slow integration happens not too often, and occurs in many cases if a component at rest becomes active.

It is possible to give each u variable its own optimized time step, so the quotient of two time steps is not integer. This is the ultimate multirate method, but it is not optimal. Because there exist in general clusters of variables, all coupled and with nearly the same time step, giving each variable its own step introduces a waste of effort. Each variable in a cluster must be checked each time point, which leads to a large overhead. Also the solution of the linear equations at each time point gives the same nonzero entries in the x vector. Therefore the efficiency increases if variables with roughly the same step size are synchronized with each other to identical time points: they are clustered.

The clustering of variables can be performed statically or dynamically. Because a static clustering is based on the situation before the integration starts, it is coarse and might not be suitable after several time steps. A dynamic clustering is more interesting, because it can be fine–tuned and will put a variable in a different cluster if necessary, so that the clusters contain at each time point variables with equal step sizes.

The clustering of variables in our simulator is based on restricting and ordering the points of the time grid. The grid points are restricted to the values $\tau_i = r\,2^{-k}T$ with integers $k \geq 0$ and $1 \leq r < 2^k$, and $T = t_e - t_0$. The time step that is allowed is the distance between the current time point and a next grid point so that the specified error is not violated. This next grid point is the

one with the lowest k, so that in general the time step has also a value $h_i = r\,2^{-k}T$, usually with $r = 1$. This clustering creates an integer grid on the time interval. Because the number of used grid points is unknown in advance, the grid is recalculated for each time point and each time step. This recalculation is efficient, so no special implementation techniques are applied here. The calculations are given in Algorithm 5.1.

With this algorithm the variables are not only implicitly dispatched over different clusters, it also ensures that the time step of the slow variables is a multiple of the fast variables. Also, if two subsequent time steps are about the same, the same step is calculated. This increases the efficiency in the other parts of the simulator, as described earlier. By not choosing the maximal allowed time step, the number of interruptions of the integration is also decreased.

Algorithm 5.1: Clustering of variables

```
for i := 1 to n_u do
    h_max := maximal_step( u_i^(q+1) );
    max_time := current_time + h_max;
    first_time := t_0; last_time := t_e;
    while last_time − first_time > h_max do
        mid_time := (last_time − first_time)/2;
        if mid_time < max_time then
            first_time := mid_time;
        else
            last_time := mid_time;
        fi;
    od;
    h_i := last_time − current_time;
od;
```

A final efficiency problem can show up when the variables must be selected for the next time point. A straightforward method is a linear scan over all variables and selecting those variables with the earliest time point. This is currently implemented in PLATO, and the overhead in this routine is reasonable, as is shown in Section 7.5. Using a partially ordered list is more efficient. An other possibility for efficient clustering is the application of an integer grid of time points, i.e. the time is scaled with a factor $2^N T$, with 2^N the maximal size of an integer, and only allowing integer values.

The conclusion is that multirate integration is attractive, if the system is large and the connectivity is low. If the connectivity is high, multirate integration is not a good alternative. But, as has been explained in Chapter 3, an elec-

trical or digital circuit usually has a low connectivity. In Chapter 7, the efficiency of the multirate method is shown through some examples.

5.5 Exponential integration

From equation (5.2) it can be deduced that the solutions of problem (5.1) have usually exponential characteristics. The general solution allows other types of solutions, for example sine functions or linear functions. Because these types of functions are not stiff, the number of time steps used to solve these functions is not large. But to determine the exponential–like solutions sufficiently accurate, the number of time steps used by the implicit integration methods is large. Therefore, the computational effort spent in these methods is large. This is not only due to the large number of steps, but also to the implicit behavior. Because the system matrix depends on the step size, the matrix must be updated with each change of the time step. This also implies that the system matrix used in the Van de Panne algorithm depends on the time steps, which sometimes has negative consequences, as described in the next section. Therefore an explicit method, although it can not be very stable, still can be attractive to apply. In this section again $t_0 = 0$ is chosen.

Because exponential behavior exists in most problems, it may be profitable to apply an explicit exponential integration method. This will be most advantageous on problems in which the solutions are truly exponential. These problems are of the form:

$$\begin{cases} \dot{u}(t) = A\ u(t) + b \\ u(0) = u_0 \end{cases} , \tag{5.17}$$

with A and b constant, and with all eigenvalues of A, λ_i, negative real and simple. This problem has the solution

$$u(t) = e^{At}u(0) + \int_0^t e^{A(t-\tau)}b\ d\tau = u(0) + (e^{At} - I)\dot{u}(0) , \tag{5.18}$$

which can also be written as

$$u_i(t) = \sum_{j=1}^{n} \beta_{ij}\ e^{\lambda_j t} + \gamma_i , \tag{5.19}$$

for some real coefficients β_{ij}, λ_i, and γ_i.

If the matrix A is diagonal, (5.19) can be simplified to

$$u_i(t) = \beta_{ii}\ (\ e^{\lambda_i t} - 1\ ) + \gamma_i\ . \tag{5.20}$$

This equation is the basis of the exponential integration method that is investigated. A simpler expression can not give better results, because it will not give valid results on a one–dimensional problem.

First consider a one–dimensional problem. The values β_{11}, λ_1 and γ_1 can be calculated from the equations:

$$\beta_{11} = \frac{(\dot{u}(0))^2}{\ddot{u}(0)} \ , \ \lambda_1 = \frac{\ddot{u}(0)}{\dot{u}(0)} \ , \ \gamma_1 = u(0) \ . \tag{5.21}$$

With these values the one–dimensional explicit exponential method is introduced by:

$$u(h) = u(0) + (e^{h\lambda_1} - 1)\beta_{11} \ . \tag{5.22}$$

The n–dimensional explicit exponential method is now defined by:

$$u(h) = u(0) + (e^{h\Lambda} - I)B \ , \tag{5.23}$$

where B and Λ are diagonal matrices with

$$\begin{cases} B_{ii} = \dfrac{(\dot{u}_i(0))^2}{\ddot{u}_i(0)} \\ \Lambda_{ii} = \dfrac{\ddot{u}_i(0)}{\dot{u}_i(0)} \end{cases} . \tag{5.24}$$

If necessary, B and Λ are indexed with the time point at which they are calculated. Notice that the values of $\dot{u}(0)$ and $\ddot{u}(0)$ need to be calculated by solving a matrix equation.

The two main properties of this method that must be determined are its order and its stability. For this analysis, let $y(t)$ be the calculated solution of the differential equation (5.17). The order of the method is determined by calculating the Taylor series of the exact solution and comparing it with the solution of the integration method:

$$\begin{cases} u(t+h) = u(t) + h\dot{u}(t) + \frac{1}{2}h^2\ddot{u}(t) + \frac{1}{6}h^3u^{(3)}(t) + O(h^4) \\ y(t+h) = y(t) + h\dot{y}(t) + \frac{1}{2}h^2\ddot{y}(t) + \frac{1}{6}h^3\Lambda^2\dot{y}(t) + O(h^4) \end{cases} . \tag{5.25}$$

So if $u(0) = y(0)$, $\dot{u}(0) = \dot{y}(0)$ and $\ddot{u}(0) = \ddot{y}(0)$, the LTE at time point h, $u(h) - y(h)$, is:

$$LTE = \frac{1}{6}h^3(\Lambda^2\dot{y}(0) - u^{(3)}) + O(h^4) \ . \tag{5.26}$$

Therefore the order of the method is 2. Notice that the order of the method does not show its behavior on exponential solutions. It is simple to show that the LTE vanishes for these solutions.

The stability of an integration method is usually tested with the differential equation $\dot{u} = \lambda u$, with $\mathrm{Re}(\lambda) < 0$. Because the explicit integration method solves this equation exactly for all time steps h, it seems that the method is A–stable. To analyze the stability more precisely, consider an n–dimensional problem with $n > 1$. Denote by $\delta(t+h)$ the error made in the step from time t to $t+h$, i.e. the LTE. The exact error made in a step, $\delta(t+h)$, assuming $u(t) = y(t)$, $\dot{u}(t) = \dot{y}(t)$, and $\ddot{u}(t) = \ddot{y}(t)$, is:

$$
\begin{aligned}
\delta(t+h) &= y(t+h) - u(t+h) \\
&= y(t) - u(t) + (e^{h\Lambda(t)} - I)\Lambda(t)^{-1}\dot{y}(t) - (e^{hA} - I)A^{-1}\dot{u}(t) \\
&= \left((e^{h\Lambda(t)} - I)\Lambda(t)^{-1} - (e^{hA} - I)A^{-1}\right)\dot{u}(t) \; .
\end{aligned}
\tag{5.27}
$$

Therefore the formula for $\delta(h)$ is:

$$
\delta(h) = \left((e^{h\Lambda(0)} - I)\Lambda(0)^{-1} - (e^{hA} - I)A^{-1}\right)\dot{u}(0) \; . \tag{5.28}
$$

With some slightly more difficult calculations the exact error after two steps can be determined, using $\dot{y}(h) = \dot{u}(h) + A\delta(h)$, leading to:

$$
y(2h) - u(2h) = \phi_1\delta(h) + \delta(2h) \tag{5.29}
$$

with $\phi_1 = I + (e^{h\Lambda(h)} - I)\Lambda(h)^{-1}A$.

From equation (5.29) the formula is deduced for the global error at the nth time point:

$$
y(nh) = u(nh) + \sum_{i=1}^{n}\left(\prod_{j=i}^{n-1}\phi_j\right)\delta(ih) \; , \tag{5.30}
$$

with $\phi_j = I + (e^{h\Lambda(jh)} - I)\Lambda(jh)^{-1}A$.

By using relation (5.26), the local truncation errors $\delta(ih)$ can be estimated, and the step h can be adjusted so that $\| \delta(ih) \| < \varepsilon$ for given ε. To keep the global error $\sum_{i=1}^{n}(\prod_{j=i}^{n-1}\phi_j)\,\delta(ih)$ bounded, it is therefore necessary that $\| \phi_j \| \le R$ for all j for some $R < 1$. In that case the global error is bounded by $\varepsilon/(1-R)$. The condition $\| \phi_j \| \le R < 1$ depends on the matrix A, the time step h and the estimate $\Lambda(jh)$ of the eigenvalues of A. Because the eigenvalues of A were assumed to have a negative real part, there is a region in which h can be chosen so that $R < 1$. If the matrix A is not diagonal, this region is bounded (as is shown in the next paragraphs), and the explicit exponential method is not A(α)–stable.

A simple example shows that with a non–diagonal matrix the condition $\| \phi_j \| \le R < 1$ can not be fulfilled. The problem, given in equation (5.31), has the exact solution given in equation (5.32):

$$
\begin{cases}
\dot{u} = \begin{bmatrix} -1 & 0 \\ 2 & -2 \end{bmatrix} u + \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0 \\
u(0) = \begin{bmatrix} 0 \\ -\frac{3}{4} \end{bmatrix}
\end{cases}
\tag{5.31}
$$

$$
u = \begin{bmatrix} 1 - e^{-t} \\ 1 - 2e^{-t} + \frac{1}{4}e^{-2t} \end{bmatrix} \tag{5.32}
$$

The method's estimate of the eigenvalues gives the exact result for the first one, while the second eigenvalue is estimated with $\lambda_2 = \frac{\dot{u}_2}{2(\dot{u}_1 - \dot{u}_2)}$. This eigenvalue decreases from −1.5 to −1 for increasing time, if the exact solution is used. A contour map of $\| \phi_j \|$ as function of λ_2 and h is given in figure 5.1 [calculated with *Mathematica*, version 1.2 (Wolfram 1988)]. Now, $\| \phi_j \| > 1$ in the region denoted with H, so it is clear that, even in the case of a good value for λ_2, the time step can not be chosen arbitrarily large.

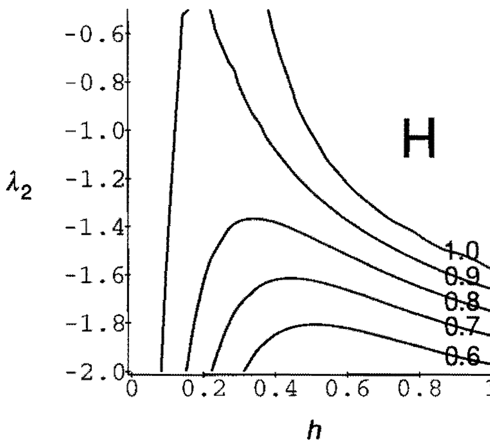


Figure 5.1. Contour map of $\| \phi_j \|$ as function of h and λ

Because the explicit exponential method is not very stable, its application is less attractive than it seemed to be. A second and major disadvantage is the instability of the calculation of A and B. Even with a slightly stiff system, a small error in the calculation of $\dot{u}$ or $\ddot{u}$ will introduce a large error in A and B. This showed up in numerical experiments, where the estimate for the large negative eigenvalue changed from step to step. Other disadvantages are that an other type of approximation needs to be employed if $\dot{u}$ or $\ddot{u}$ is close to zero, and that the method is not multirate, so part of the efficiency of the other integration methods is lost. These reasons led to the conclusion that the method is too unstable to be suitable in the simulator.

5.6 The integration method and the Van de Panne algorithm.

One of the most difficult parts of the simulator is the interaction of the integration method with the Van de Panne algorithm. In traditional analog simulators, the integration method is applied first, and the current linearization is determined during the solution of the resulting nonlinear system. In PLATO, the current linearization is already known in a certain region. If the solution

at the next time point is still in this region, the integration method yields the solution immediately. Problems may occur at the boundary of a region, namely:

- The time point at which the boundary is crossed is not a grid point of the integration method.
- The integration method has to be stopped properly at the boundary.
- The integration method has to be restarted properly after a new region and linearization have been determined.

The first problem, that the time point of crossing is not a grid point of the integration method, can not be ignored. The time point t_b denotes the (unknown) time point at which a facet is crossed. A simple but erroneous method is to integrate with the old linearization over the boundary and calculate afterwards a new linearization. Because the solution will not match the new linearization, it will change discontinuously at the next time point. Ignoring the boundary conditions introduces also an error in the solution that may be unacceptably large. This happens if the time step h is large, in which case timing errors or large integration errors occur.

To avoid these errors, the integration method must be stopped on the boundary. This can be performed by two methods: iteratively determining the time step until the solution at the next time point will be sufficiently close to the boundary, or stopping the integration in the middle of the integration step. The first method is expensive, because the integration step may have to be changed a number of times, and each change will need an update of the A_{xx} matrix and the solution of the new linear equations.

It is less expensive to stop the integration in the middle of the integration step. Instead of performing the full step, the values are interpolated to the time point t_b. The interpolating function can be chosen exactly in the begin and end points of the integration interval, and with an error smaller than the user–defined error bound in the other points of the interval. The time point t_b can be found directly with such an interpolating function: the interpolation of the boundary variable is known explicitly, and only the zero point of this function must be determined. This is much simpler than the method described in the previous paragraph. So, depending on the integration method, an interpolation method is chosen to check if a boundary is crossed, before this step is performed. Notice that the implicit linear integration methods are transformed into an explicit method with equation (5.9), i.e. at the time point $t_{(i)}$ the values at $t_{(i+1)}$ can already be inspected. So the existence of a break point t_b is determined before it is crossed.

The second problem is the proper halting of the integration method and the start of the Van de Panne algorithm. With the use of the interpolating function the values are updated up to the time point t_b. The Van de Panne algorithm is initiated as described in Section 4.4, so that first the boundary must be crossed before the algorithm is finished. The first problem that is encoun-

tered, is the influence of implicit integration method on the A_{xx} matrix. As shown in equation (5.13), this matrix depends on the time step h for an implicit linear multistep method. Because the A_{xx} matrix is also used in the Van de Panne algorithm, this gives some problems.

There are three possible solutions:
- Use two matrices and two LU decompositions.
- Set the time step to zero before the Van de Panne algorithm is started.
- Take into account the influence of the integration method on the Van de Panne algorithm.

The first of these solutions is too expensive, while the second solution is only necessary if in the Van de Panne algorithm a step is taken, i.e. the pivot was negative or zero. The third possibility is simpler, because only locally the integration method must be taken into account. If the pivot is positive after the correction, it is performed; otherwise the time step must be set to zero, because large errors are made, as a Van de Panne step can be considered as an instantaneous integration. Concluding, the Van de Panne algorithm is adapted slightly: the first time the sign of the pivot is calculated with a time–dependent local matrix. If this sign is not positive and the time step is not zero, the time step is set to zero, the matrix is updated, and the sign is recalculated. Because a multirate integration method is applied, a choice must be made whether the time steps of all u variables are set to zero, or only those that might be affected. In order to not disturb the unaffected variables, only the directly influenced u variables are set to zero.

The third problem is the restart of the integration method. First the derivates must be recalculated, because they have possibly been changed during the Van de Panne algorithm. If the pivot was positive, the time step has not changed, so the integration method can continue without too much difficulties. The linear one–step formulae determine possibly a new time step, update the matrices and start the integration. The linear multi–step formulae must also take into account that the last time step was not the time step used in the matrices. If the pivot was not positive, the integration must be restarted as described earlier.

Special care must be given to the possibility that the (implicit) integration and the Van de Panne algorithm interfere with each other, i.e. the integration indicates that a boundary must be crossed, but the Van de Panne algorithm does not find a dynamically valid new region. Because an invalid region is found, the Van de Panne algorithm is started again and is likely to find the original region. This is only possible in two situations: there is no valid dynamic region, as has been described in Section 2.4.3, or the time step has not been set to zero. In the first situation no solution can be found, and the simulator will fail. In the other situation, the time step for all u variables is forced to zero and a new try is performed. If this does not help, the simulator

will fail. The possibility that the Van de Panne algorithm will not find a solution due to a bad choice of the e vector in this case, has never been observed.

6 Aspects of the Implementation

6.1 Introduction

In the previous chapters the basic algorithms have been described that are used in our simulator. In this chapter the main loop is specified, in which these algorithms are used to solve a problem specified by a user. Before this algorithm will be discussed, a short description of the user environment for PLATO is given. The introduction of divided differences, used for creating a more sparse problem, and the output processing are discussed. Also some numerical problems are discussed. A special topic is a possible parallel implementation of PLATO and its effect on the run time. Some future extensions are discussed finally.

6.2 User environment

Although it is of little importance for the numerical aspects of a simulator, the user environment is still an important factor in the design of a simulator. Internally only the matrices and the initial values of the u variables are needed to describe the problem. These values are determined from a description of the network and the components. Furthermore, some parameters specific to the simulation process must be given, like the simulation interval, the integration accuracy, etc. The creation of both the network and component description and the simulation specific parameters should be simple and straightforward.

Nowadays, a graphical user interface is preferred to relay this information to the simulator itself. Because a graphical user interface is not relevant for the simulator, the interface with the simulator is split in a simple and an advanced layer. The simple interface consists of two input files, one containing a description of the circuit and one containing the specific simulation parameters. Several graphic programs, creating these two input files, are available as an advanced user interface for PLATO.

The main graphic interface program is PLTASK, used to control the simulator as well as other graphic and auxiliary programs. The two other graphic pro-

grams are ESCAPE and PLOV. ESCAPE is a newly developed graphic editor and simulator of circuit schematics, based on the older schematic editor ESCHER (Lodder *et al.* 1986). With ESCAPE, a schematic diagram of the circuit can be drawn and checked for consistency. PLOV is a signal viewer, to draw and examine the calculated signal values on the screen. All these graphic programs have the same 'look and feel', based on the OSF/Motif interface (OSF 1990).

The circuit description file that is used by PLATO, is written in a special language called *NDML* (Janssen 1986; Buurman *et al.* 1990). To simplify the (hierarchical) description of the circuit for a user, the language has two levels of description:

- A high level specification with PASCAL–like statements (**if**–statements, **for**–loops, etc.) and data structures (**array**s of nets). Circuits may be parameterized, so properties of a circuit are determined by its parent. This specification may be incomplete, i.e. referred sub–circuits are not in the description.
- A low level specification containing a complete specification of a circuit, in which the high level statements are expanded (loops unrolled, conditional statements executed, etc.) and the parameters are evaluated. This specification is a proper subset of the high level specification.

The language compiler NDML can convert a high level description into a low level description that is understood by the simulator. Such a high level description can be created with a text–editor, but the compiler can also extract the information from the graphical data of ESCAPE. Unspecified components are looked up in a library of previously created circuits and models.

Because the output of the simulator can be very large, the graphic signal interpreter PLOV plays an important role in the interpretation of the results of a simulation. It can be used to compare, both numerically and graphically, output signals from one run, from different runs, or from other input sources, like an output specification. The interpreter can also print a selection of signals on a postscript printer.

Furthermore, some auxiliary programs are available in the environment. To analyze new or complicated models, a simple model interpreter is available, which determines all possible regions and linearizations in a model. Various filters and signal comparators can also be used. This cluster of programs simplifies the use of the simulator in a design environment. Because these programs are available, the simulator program contains a minimum set of necessary features, allowing its use independently of the environment. All programs are written in the language *C* (Kernighan and Ritchie 1988).

6.3 The event driven simulator

In the previous chapters the basic algorithms were explained, that are used in the circuit simulator PLATO. In this section these algorithms are integrated in the main event loop, the main algorithm of PLATO (Algorithm 6.1). This algo-

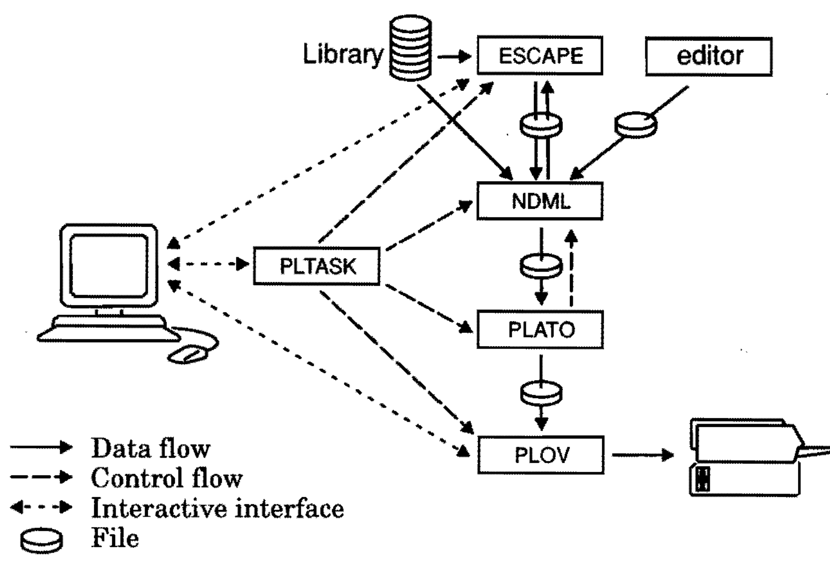


Figure 6.1. User environment with the most important programs

rithm is rather straightforward, compared to some of the earlier described ones. At this point of the description, the practical efficiency of the algorithms is also important. Several methods will be described to increase this efficiency.

The algorithm performs two tasks: finding an initial solution and solving the nonlinear dynamic problem. First the routine *initialize*() is called to perform several tasks:

- Reading the simulation specific parameters.
- Reading the circuit description.
- Creating the data structures.
- Scaling the equations. The purpose and effect of this task are described in Section 6.6.

After the data structures have been created, a nodal analysis of the system is performed. As described in Section 3.2, during this analysis the number of equations and variables may decrease. So only hereafter the LU decomposition is calculated. The initial solution, either the DC solution or a simpler initial value problem, is determined now. Sometimes this is the only part of the simulation that the user is interested in, so then the next loop is skipped. This loop is the main event loop, in which the *transient* solution is calculated, i.e. the dynamic solution of the problem.

Algorithm 6.1: The main algorithm of PLATO

```
var cluster: array of leafcell;
initialize( );
nodal_analysis( );
LU_decomposition( );
DC_analysis( );
time := t0;
while time < te do
    <cluster, new_time, type> := next_event( );
    if oscillation( time, new_time ) then
        exit unsuccessful;
    fi;
    if time < new_time then
        output_variables( );
        time := new_time;
    fi;
    update( cluster, time );
    if type = pl then
        Van_de_Panne( cluster );
    else
        for i := 1 to size( cluster ) do
            new_time_step( cluster[ i ] );
        od;
    fi;
    calculate_new_x_bar( );
    for i := 1 to size( cluster ) do
        new_event( cluster[ i ] );
    od;
    for i := size( cluster ) + 1 to related_leafs( ) do
        check_event( cluster[ i ] );
    od;
od;
```

As has been explained in the previous chapter, the dynamic problem must be solved by a numerical integration method. To solve this problem efficiently, a multirate integration scheme is applied. This scheme introduces for each component a so called *next event*, i.e. a time point at which a new integration step must be taken. It is also possible, that between the current time point and the next integration time point a facet of the piecewise linear mapping is crossed. In this case, the integration must be performed by an interpolation

up to that time point, followed by the determination of a new mapping, after which the integration can be continued.

The usage of the multirate integration scheme and the strict checking of the boundary conditions of the piecewise linear mapping introduce an *event driven* simulation scheme. The simulator repeatedly determines a cluster of components and a time point (*event*), at which the equations in the components of the cluster become invalid. The sequence of time points is processed, and at each time point a number of actions is performed as follows:
- The values are calculated or updated, and printed.
- If necessary, the linear equations are updated by the Van de Panne algorithm or by the integration method.
- The values, used for the interpolation, are calculated.
- The new events are determined, and old events are checked.

Here the advantage of using sparse techniques becomes most obvious. Because during an event only the variables related to the leaf cells in the cluster might change, updating is restricted to this set of variables. This set is in most electric and logic circuits limited, because only a fraction of the leaf cells is member of the cluster, and because the connectivity is low. The interpolation of the other variables remains valid, so no intermediate value at the current time point is calculated. Only in the last part of the loop, a subset of the leaf cells not in the cluster is checked. This subset contains those leaf cells whose next event might become invalid, because one of its variables changed direction at this time point.

Because the applied integration methods are restricted to order 1 or 2, a linear interpolation for the multirate integration and for the computation of the boundary crossings is used. For optimal use of the sparsity, the x vector is calculated only once in the initialization phase, while in the other places, if it is necessary, it is updated with a divided difference, $\bar{x}_i = \frac{x_{(i+1)} - x_{(i)}}{h}$. However, the x vector is not used in the calculations, except in the initialization, because for the other vectors also a divided difference vector is maintained. These difference vectors are directly calculated from the vector $\bar{x}_i$, which is as simple as using the vector x. In the next section it is explained that using these difference vectors is more efficient compared to the application of normal vectors.

6.4 Sparse vectors by divided differences

One of the major features of our simulator is the use of sparsity. This is partially implemented by using a sparse A_{xx} matrix, a list of leaf cells, and a list for connecting the entries of the x vector with the leaf cells. To use this sparsity optimally, the calculations with the x vector should be also as sparse as possible. This can be accomplished in two steps, the first being the introduction of divided differences:

$$\bar{x}_i = \frac{x_{(i+1)} - x_{(i)}}{h}$$
$$\bar{u}_i = \frac{u_{(i+1)} - u_{(i)}}{h} \quad . \tag{6.1}$$

Using these variables, equation (5.11) is transformed into

$$\begin{bmatrix} A_{xx} & A_{xu} \\ A'_{ux} & A'_{uu} \end{bmatrix} \begin{bmatrix} \bar{x}_i \\ \bar{u}_i \end{bmatrix} + \begin{pmatrix} 0 \\ a''_u \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} , \tag{6.2}$$

with

$$\begin{cases} A'_{ux} = -\, h\, b_{-1}\, A_{ux} \\ A'_{uu} = I - h\, b_{-1}\, A_{uu} \\ a''_u = \sum_{j=0}^{p} \Big(\alpha_j\, (u_{(i-j)} - u_{(i-j-1)}) + h\, \beta_j\, (\dot{u}_{(i-j)} - \dot{u}_{(i-j-1)}) \Big) \end{cases} .$$

Because most variables change, $\bar{u}$ and therefore $\bar{x}$ are not sparse vectors. So only the updates on these vectors are calculated:

$$\delta \bar{x}_i = \bar{x}_{i+1} - \bar{x}_i$$
$$\delta \bar{u}_i = \bar{u}_{i+1} - \bar{u}_i \quad . \tag{6.3}$$

These updates are calculated with the matrix of equation (6.2), using only a different a''_u:

$$a'''_u = \sum_{j=0}^{p} \Big(\alpha_j\, (u_{(i-j+1)} - 2u_{(i-j)} + u_{(i-j-1)}) + h\, \beta_j\, (\dot{u}_{(i-j+1)} - 2\dot{u}_{(i-j)} + \dot{u}_{(i-j-1)}) \Big) \quad . \tag{6.4}$$

Notice that to solve $\delta \bar{x}_i$ the same matrices are used as in equation (5.11), so the method outlined in Section 5.3 is also applied in these calculations. The gain in using updates on divided differences is twofold. First, the solution of the equation $A'_{xx}\, x_{(i+1)} = a'_x$ is in general more expensive to calculate than the solution of $A'_{xx}\, \delta \bar{x}_i = a'''_x$, because the vector a'''_x has less entries than a'_x, and as a result the vector $\delta \bar{x}_i$ has less entries than $x_{(i+1)}$. Secondly, because $\delta \bar{x}_i$ has less entries, the number of related leaf cells is only a fraction of the total number of leaf cells. In the case of a multirate integration method, the gain is even larger, because a'''_x and $\delta \bar{x}_i$ contain only a limited set of entries. The divided differences are also employed in the calculation of the time step.

This combination of a multirate integration method with a sparse update method fully exploits the sparsity of the problems. The components that change are used in the calculations, while the other components are only checked when necessary. Because the exploitation of the sparsity is the only

method to simulate efficiently large circuits, the divided difference method is implemented in PLATO. It is straightforward to calculate also the divided differences of w and z with the values of $\bar{u}$ and $\bar{x}$.

To maintain the values of $\bar{x}$ and $\bar{u}$ up to date during the simulation, they must be recalculated when a facet is crossed, i.e. during or after the Van de Panne algorithm. A direct calculation of $\bar{u}$ is expensive and not necessary, because in many situations only a few entries of $\bar{u}$ change when a boundary is crossed. Instead, an update vector $\delta\,\bar{u}$ is calculated during the Van de Panne algorithm. This vector is updated at several points in the algorithm:
- On initialization, the time, and therefore x and u, has been changed, so $\delta\,\bar{u}$ is calculated accordingly.
- If the Van de Panne algorithm finds a negative or zero pivot, the step size is set to zero, and $\delta\,\bar{u}$ is updated.
- If the Van de Panne algorithm takes a step, x and u may change, so $\delta\,\bar{u}$ is updated.
- If a pivot is performed, the matrices change. Then $\bar{x}$ and $\bar{u}$ may change, and this effect on $\delta\,\bar{u}$ is calculated.

The effect of these updates on the vector $\delta\,\bar{u}$ is always restricted to those elements that have been affected in the Van de Panne algorithm. Because this number is small, the vector is as sparse as with a dynamic event. The gain in computer time for these calculations is, compared with the direct calculation of $\bar{u}$, about 15 % of the total computer time.

6.5 Output processing

With the use of divided differences, the value of x is calculated only once, at the beginning of the time interval. A new strategy can be developed to create the output also efficiently. The verb *print* will be used to describe the process of creating the output, although the results are not directly available on a printer, but only in raw form in a file, and must be interpreted graphically with a postprocessor. The choice which of these x variables are printed is made by the user, and this choice has some influence on the actual calculations, as described in Section 3.2.

The actual act of printing is expensive with respect to other calculations and actions, because the storage or printing device can handle much less data per second than the processor. Therefore it is useful to decrease the data output as much as possible, without decreasing the accuracy. This is possible, for example if one output variable is constant for some time, when only the signal values at the begin time and the end time are necessary to describe the signal perfectly. If however a signal has many fluctuations, the signal value must be known at many intermediate points. In this case there is more information, so it is reasonable to use more data to describe the signal. Because the information is not known for future time points, it is not easy to describe a signal other than with a number of pairs (*time*, *value*). Note that then also a mathe-

matical function with an explicit formula must be described with many points. Although the output could be reduced if the numerical results are approximated with a set of mathematical functions, this option is not explored in PLATO.

Traditionally, the time points at which variables are printed are on a regular grid. This has the advantage that the postprocessing can be simple. This method has however two drawbacks: first, to ensure that fast changes in value can be noticed, the distance between the time points must be small. Secondly, many piecewise linear variables (for example the output of simple dynamic logic gates) are printed and calculated too many times. These variables can be printed exactly if the 'corners' of the time function are printed, i.e. the points at which the direction of the signal changes or at which the signal is discontinuous. These time points do not exactly fit the chosen grid, and in general the value of a signal is checked or printed at many intermediate time points. To avoid this numerous checking and printing, the output processing of PLATO is based on changes in the values of the signals.

This is possible, because a user of a simulator requires two seemingly contradicting properties. The simulation must be accurate enough to be sure that the modeled behavior is well approximated, but on the other hand this accuracy does not need to be shown explicitly in the output. The user is more interested in the 'global' behavior of the different signals. Therefore in PLATO an accuracy for printing is introduced: if three successive values of a variable are (nearly) on a straight line, the middle one need not to be printed. In this way, constant signals are printed only at the begin and the end, and piecewise linear functions are printed only at their 'corners'. For analog types of signals, the accuracy of the output is still good, while less points are printed.

The second choice is to create the time grid at which the results are printed dynamically. This grid is chosen to be equal to the events of the event–driven simulator. So a variable is only checked if the integration method indicates that its direction has changed, or if it changes due to a Van de Panne step. Because only a limited set of variables must be checked, this method is efficient. A second advantage is that also fast varying signals will be printed nearly exactly. In the implementation, the signal is checked for printing in two cases: if $\delta \overline{x}_i \neq 0$ after a dynamic or a pl event, or if $\tilde{x}_i \neq 0$ during the Van de Panne algorithm.

6.6 Numerical accuracy

In a large program like a circuit simulator, several numerical problems show up. The main problems are the accuracy and stability of the integration, and the accuracy of the LU decomposition. In our case, also several accuracies of the Van de Panne algorithm are important. The accuracy and stability of the integration were discussed in the previous chapter. Here some additional re-

marks are made on the numerical precautions made in the LU decomposition and the Van de Panne algorithm.

The accuracy and stability of the LU decomposition and the solution of the linear equations depend on the *condition number* of the matrix A_{xx} and the pivoting strategy. The condition number $c(A)$ of a matrix is defined as $c(A) = \| A \| \cdot \| A^{-1} \|$. It is a measure for the error made in the solution of the equations, i.e. the error has a value in the order of $c(A)$ times the initial error. For linear equations originating from circuit equations, the condition number is relatively small for most circuits. Therefore the accuracy and stability of the LU decomposition and the solution of the linear equations suffice in those cases. If the condition number of the matrix is large, and the accuracy degrades, this indicates that the system is difficult to solve. Using a more elaborate solution method will not really solve the problem, because the other algorithms in the simulator will also have difficulties to find a solution, i.e. the numerical problems are inherent to the circuit.

To enlarge the accuracy and stability of the LU decomposition, the condition number of the matrix should be lowered. This can sometimes be accomplished by *scaling* or *equilibrating* the matrix, i.e. dividing the rows and columns by suitably chosen numbers. A reasonable scaling creates a matrix where the largest entry in each row and column is $\mathcal{O}(1)$.

Scaling allows loosening the criterion for choosing the LU pivot. A small pivot now always denotes a nearly singular system. After the scaling, it is also possible to keep the sparsity of the matrix up to date by deleting entries with very small values. Instead of using a situation–dependent norm to determine the meaning of "very small", an absolute value is used to define this notion. This same definition is used to determine if an entry of a vector can be ignored.

The determination of a good scaling depends on the problems that are solved with the simulator. If nothing is known about the matrix, scaling is difficult and may yield worse results (Golub and Van Loan 1989). In PLATO, some knowledge of the problem is available, and the applied scaling is based on it. In an analog or digital system, usually the variables of one kind (voltage, current, signal) have the same magnitude. For example, currents are $\mathcal{O}(10^{-4})$ (ampere), voltages are $\mathcal{O}(1)$ (volt), and signals are $\mathcal{O}(1)$. Therefore a heuristic scaling is performed: each column of the A_{*x} matrices is multiplied by a norm factor, depending on the type of the related variable. Then each variable should be $\mathcal{O}(1)$ during the simulation. Because it is not directly known in which units the problem is given, the user can specify these norm factors. After scaling the columns of the matrices, the rows (linear equations) are equilibrated. The $\| \cdot \|_\infty$ norm of each row is used as scaling factor, to set the entries in the matrix on a reasonable value.

To maximize the stability and accuracy of the remaining algorithms, their related variables and equations are also normalized. The time–dependent u variables are scaled so that they are also $\mathcal{O}(1)$. The time is scaled with the

length of the integration interval, so the determination of fast and slow components can be done on a global base, and the integration methods can be applied with usually a larger time step.

A last scaling is performed on the w and z variables. Any positive number may be chosen as normalization factor. The most important variable is w, determining the region, so it is normalized to be at least $O(1)$. To keep the entries on the diagonal of A_{zz} the same, z is scaled with the inverse of the scale factor of w.

After the last normalization, the accuracy used in the Van de Panne algorithm can be determined. Because this algorithm is combinatorial by nature, an implementation in a floating point program must take care of the precision. The main problem is the decision which row is blocking, i.e. for which $\phi > 0$: $w + \phi \tilde{w} = 0$ for given w and $\tilde{w}$. To guard the computation against rounding errors, w_i is independent of the active variable if $|\tilde{w}_i| < \varepsilon_p$. Therefore, ε_p is also used to determine whether a Van de Panne pivot is zero.

6.7 Parallelism

One of the new techniques to decrease the time spent in programs is the use of more computing devices (processors) to solve one problem. This will decrease the time a user must wait for the solution, but requires more computer resources, used in the communication between the processors. The application of more processors is called *parallelization*, because the different processors work in parallel (concurrently) on one problem. The computing devices can be processors of one specially devised computer (called a *parallel* computer), or can be several loosely coupled computers in a network. Another possibility to increase the performance of a program is the use of *vector* computers, that can handle blocks of data (*vectors*) more efficiently than ordinary (scalar) computers. The application of vector and parallel operations in a program is called *vectorization* respectively *parallelization* of the program. These actions are only possible if the data are used independent in the algorithms.

In PLATO, most algorithms can not be directly parallelized or vectorized. It is clear that both the Van de Panne algorithm and the integration method must be solved step by step. The LU decomposition and the rank m update are well vectorizable for non–sparse matrices, but for sparse problems it is usually not efficient. The sparse data structure can not be used directly in the vector operations, and applying a scalar algorithm with a sparse data structure is more efficient than the application of a vector algorithm on a non–sparse data structure. Parallelization of the algorithms is possible, but the data dependencies in the algorithms imply that in using more than one processor, the cost of cooperation is high and the gain is low. It might easily be possible that only one or two processors are active, while the other processors are waiting for data. Another possibility is that many processors are busy, updating the

data used in the cooperation, instead of performing calculations used for solving the problem. Some experiments led to the conclusion that a straightforward parallel implementation was much slower than the standard sequential implementation, due to these reasons. However, a vector–parallel implementation on a non–sparse data structure gave the expected gain. This implementation is used in the neural network simulator PLANNET, as described in Section 7.3.

Parallelism can be applied with success in one part of the simulator, namely in the leaf cell routines. Both in the integration method and in the Van de Panne algorithm, a fraction of the leaf cells is used to calculate and update several vectors. During these calculations, there is no dependency between the leaf cells, so the calculations in each leaf cell can be done in parallel on different processors. This is not implemented in PLATO, because the time spent in these routines, although large, is not the major part of the run time: it is typically 30 to 50 %. Therefore parallelization will yield a gain of maximally about 45 % on a system with 10 processors, provided the overhead costs are sufficiently low. This gain is in most situations not large enough to be acceptable.

6.8 Multilevel simulation and functional modeling

One drawback of the simulator is its inability for simulating circuits containing complex models, both at the device level and at the behavior level of description. This is explained by the properties of the piecewise linear modeling technique, which is not suitable for these types of models. Two methods can be used to overcome this disadvantage: multi–level simulation and functional modeling. In multi–level simulation, PLATO is coupled to a second simulator that is used to simulate an other part of circuit. With functional modeling, the group of models is extended with functions that behave like standard piecewise linear models.

To examine multi–level simulation with PLATO, the simulator has been coupled to the behavioral simulator ESCAPE (Fleurkens and Buurman 1992). The resulting compound simulator has the following properties:

- It is a master–slave relation, with ESCAPE the master and PLATO the slave. This is the most practical solution, because the high level simulator ESCAPE has much less events than PLATO.
- ESCAPE sends a circuit description to PLATO, with several input and output terminals that are used for the communication between both parts of the circuit.
- ESCAPE determines a value at the input terminals of the PLATO circuit and a time slot. The circuit is simulated by PLATO, and at the end of the time slot the values at the output terminals are returned to ESCAPE. This ensures that no oscillation between the two simulators occurs.
- The previous step is repeated as many times as necessary. If two subsequent values on an input terminal differ, at the begin of the second time slot

a discontinuity in the input is generated for PLATO. Other variables keep their current value. Notice that discontinuities at the inputs are no problem at all for PLATO, as normal components can also create them.

Several mixed behavioral/pl circuits have been simulated with this simulator with good results on convergence and efficiency. More experiments are currently being prepared.

The other method to incorporate more models into the simulator is by applying functional models. Because the creation of complex models is difficult, a second method to implement a model is sometimes interesting. The internal data structure splits the implementation of the leaf cells from the implementation of the algorithms, so it is simple to use internally in a leaf cell a different model. Such a model should satisfy several basic properties:

- The model should fit into the event driven simulation loop. Therefore it must determine at each event a time interval during which its linearizations will remain valid.
- At each event a linearization based on the internal equations and the integration method is exported to the global equations.
- The model should mimic a standard leaf cell during the Van de Panne algorithm. This is a complicated algorithm, so a good understanding of its behavior and its internals is necessary to accomplish this mimicry.

If these conditions are fulfilled, it should be possible to create several complex models that are not easily modeled with piecewise linear models. These are for example a large block of memory, an analog multiplier, or an accurate model of a MOSFET.

7 Examples

7.1 Introduction

In this chapter three circuits are given that have been simulated with PLATO. These circuits are specially chosen to show the behavior of PLATO. These examples will be discussed in more detail in the next sections, after which some program statistics are given to show the behavior of several algorithms and assumptions. Some other circuits, simulated with an earlier version of PLATO, are described in (van Stiphout 1990).

The analog–digital converter is a prototype of a mixed level circuit. This circuit has many properties that make it suitable as a benchmark to test the simulator. Its behavior can be checked easily, while it is complex enough to catch most errors in modeling, programming and simulation. The effects of small or large changes internally in the simulator can be noticed directly. The size of the circuit is large enough to investigate the behavior of the simulator on larger circuits.

The neural network is a special case: it started as a test to investigate its behavior. However, the special structure of these problems, involving large fully connected networks, is not suitable for PLATO. Because the preliminary results were good, a special version of PLATO has been developed for neural networks.

An important class of simulators for mixed–level circuits are the switch capacitor filters. They are (in the ideal case) linear dynamic circuits, that are switched at regular time intervals to create a complementary linear circuit. This switching involves a strong nonlinear and dynamic behavior. One such switch capacitor filter has been simulated with PLATO.

7.2 An analog–digital converter

An analog–digital converter is really a mixed–level circuit. It contains an analog part to sample and stabilize the analog input (sample–and–hold). A series of 1–bit analog–to–digital converters determine the digital representation of

the sampled value. Furthermore some digital components are used to control the sample–and–hold and create a digital 'ready' signal.

This converter is embedded in a larger circuit, with additional components to create the analog input signals for the converter. This larger circuit contains two analog multiplexers to create an analog signal, and an operational amplifier to subtract both analog values. Furthermore, a number of digital components control the behavior of all subcircuits. The most important parts of the circuit are given in Figure 7.1.

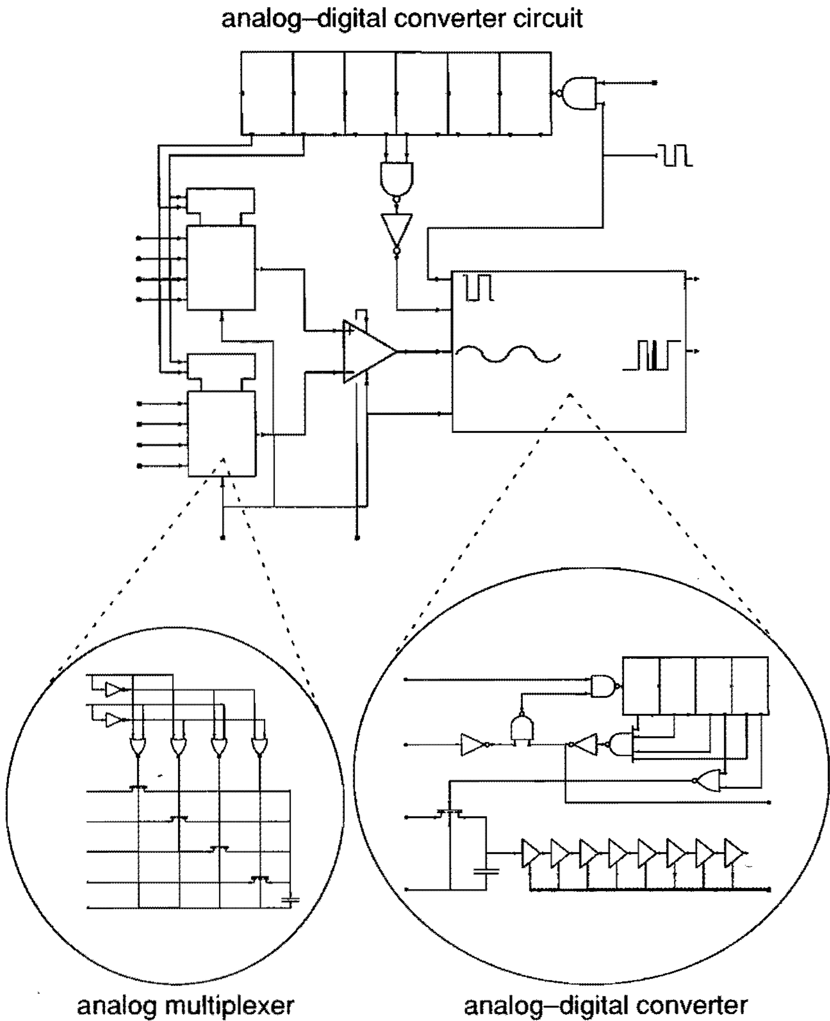


Figure 7.1. An analog–digital converter circuit

The behavior of the circuit is shown from its output, consisting of a number of analog and digital signals. A selection of these signals is given in Figure 7.2.

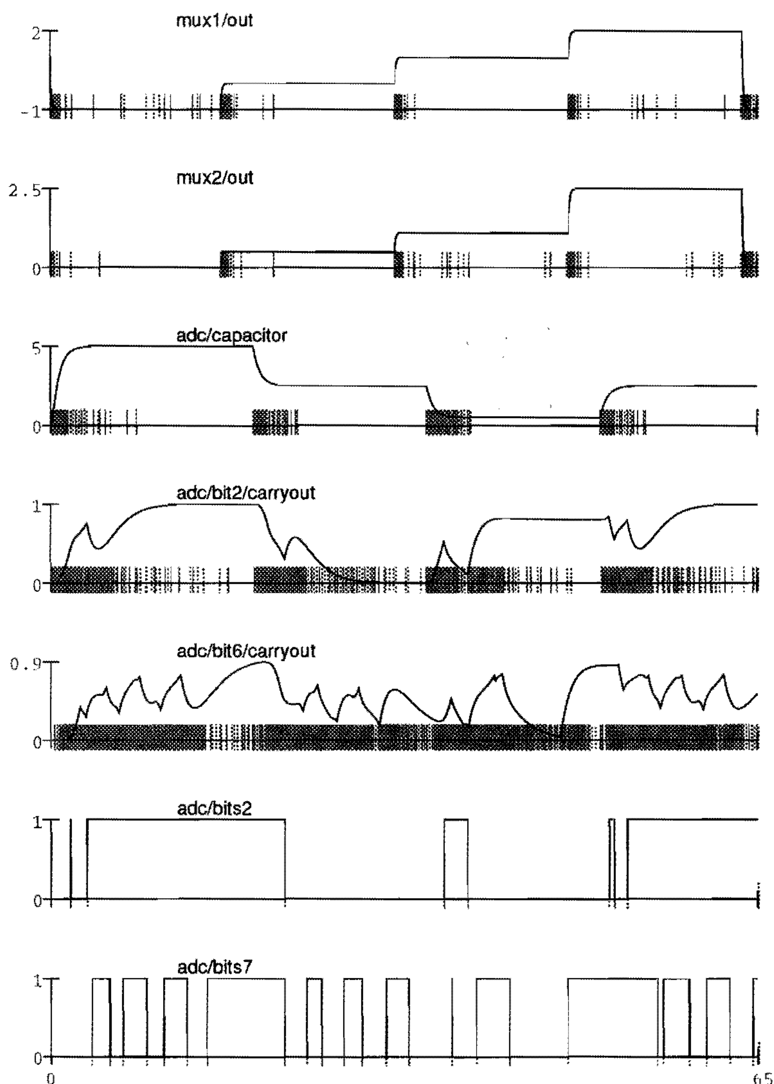


Figure 7.2. Some signals of the analog/digital converter circuit. The grey 'bar code' denotes the points at which a value is printed.

First the two outputs of the analog multiplexers (mux1/out and mux2/out) are shown, followed by the value of the sample–and–hold (adc/capacitor). This value is the input to the 1–bit converters, each processing the analog output (carryout) of its predecessor. Two of these carryouts are given: of the third (adc/

bit2) and the seventh (adc/bit6) converter. It shows that the value of the latter carryout is unstable for a long period of time, so the eighth and last converter will determine a valid value for the last bit only at a late time point. This is also clear from the digital outputs of the converters. The third bit (adc/bits2) has a fixed value shortly after the sample value is fixed by the sample–and–hold. However, the last bit (adc/bits7) is determined just when the next value is sampled. Of course, the circuit is designed with exactly this timing specification.

In the plot of the signals (Figure 7.2), the time points at which a value is printed are denoted with a grey bar. For this figure, the accuracy for printing, as described in Section 6.5, was set to 10^{-4}. The plot clearly shows that some signals are calculated and printed regularly throughout the full simulation interval, while other signals are printed only at a few time points.

To show some of the effects of the previously described algorithms, especially the influence of nodal analysis and output specification, the analog–digital converter circuit has been simulated for five different output specifications. These five specifications range from no output, via specific and more general, to output of all variables. Some relevant parameters of each simulation are shown in Figure 7.3. The column "mult. per fw/bw sub" denotes the number of multiplications per forward and backward substitution.

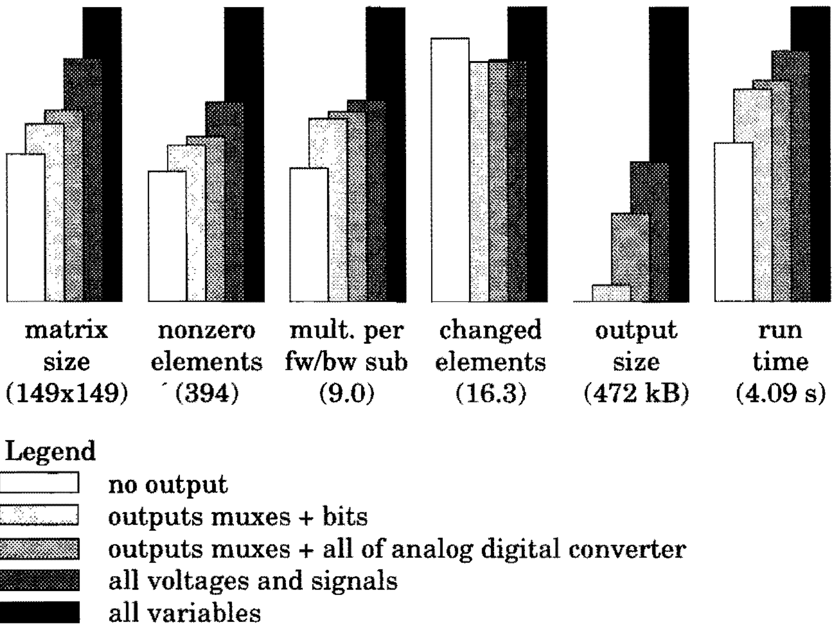


Figure 7.3. Parameter values for five output specifications. The values are relative to the maximal value (between parentheses).

From Figure 7.3 it is clear that specifying to print all variables is expensive. The amount of work done for the trivial case of no output shows that the nodal analysis can not remove a large part of the circuit. Therefore PLATO still simulates in this case half of the circuit.
A comparison of the number of nonzero elements in the matrix with the number of multiplications per forward and backward substitution shows that not all elements are relevant, especially when all variables are printed.
The average number of matrix elements that change in the rank m update of the LU decomposition (Algorithm 3.3) shows that the number of effected elements depends not heavily on the matrix, but mostly on the update vectors. The large value for the trivial case is due to the removal of several simple components, so the average length of the update vectors is larger.

The conclusions of this experiment are:
- It is useful to specify which variables must be printed. A very precise output specification gives only slightly improved results, but of course much less output.
- The rank m update depends not on the size of the matrix, but on the connectivity in the system and the components.
- The output processing is efficient.

7.3 A neural network simulator

One of the advantages of the simulator PLATO is the use of models of non–electronic components. To show this and to investigate the possibilities of parallelizing and vectorizing the program, a Hopfield neural network has been simulated with an adapted version of PLATO, called PLANNET (Buurman *et al.* 1991). First the characteristics and applications of a Hopfield neural net are given, followed by some implementation features and some results.

A neural network is a network consisting of only one type of module. The structure of a neural network is similar to the structure of a (human) brain, with many components and many interconnections between them. The basic modules (cells) are called neurons, which have many inputs and one output. The output of every neuron is connected to the inputs of many cells. Each input has a weight connected to it, to model the influence of the connected cell on this cell. Each weight may change under influence of the inputs and an external force, which process is called learning. The output of a cell is adjusted according to the sum of the weighted inputs, but is always clipped between 0 and 1. This output models the activity of a cell: if its value is 0, the cell is not active; if it is 1, the cell is active. In the human brain and in the neural structures of other animals, this activity is the frequency of the so-called spikes (Figure 7.4) on the output of a cell: more spikes per second mean a higher activity.

Neural networks are used to solve two types of problems: to emulate human decision problems or to solve optimization problems. The human decision

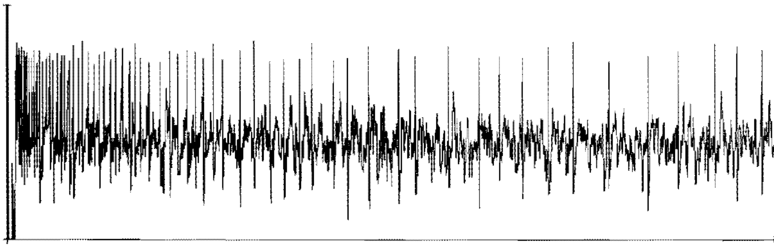


Figure 7.4. Neural activity in a tarsal sensory hair of the cabbage fly (Delia radicum) [Roessingh, Oxford, private communication]

problems are modeled by using a network with a given structure, and to train it. The training consists of offering an input to the network, calculating the output and adjusting the weights on the connections according to the difference between the calculated output and the desired output. In this way, problems like pattern matching and prediction problems can be solved. A trained network is capable of recognizing not only the training patterns, but also noisy variants. Some recent applications are for example the deduction of geologic rock types from seismic data (Cardon *et al.* 1991), and a trouble forecasting system in the steel industry (Tanaka and Endo 1991).

The other main application of neural networks is their use in solving optimization problems. Most optimization problems are hard to solve, and many are NP–hard. Neural networks can be used to find a nearly optimal solution in a limited time. This is done by selecting an appropriate network and setting the weights on the inputs to a predefined fixed value. The neurons are put in a random initial state, and usually converge to a stable state. The complete network will then represent a solution to the problem, that usually is close to the optimal solution. The behavior of the network is explained by the fact that the optimal solution is a stable state of the network. Many optimization problems are solved with a Hopfield neural network as presented in (Hopfield 1984; Hopfield and Tank 1985). This is a fully connected network, where each neuron is connected with all other neurons. Each neuron in a Hopfield neural network is a so called Hopfield neuron, i.e. it is modeled as a dynamic and analog component.

Most optimization problems have the form: find an x satisfying $g(x) \geq 0$ so that $f(x)$ minimal is. To solve such problems, the weights of the neurons depend on both functions f and g: one part of the weights is determined to fulfill the condition $g(x) \geq 0$, while the other part of the weights is determined to force the network to an optimal solution. The ratio between these two parts determines whether it is more important to satisfy the feasibility condition $g(x) \geq 0$, or to find an optimal solution, possibly not in the specified area.

One of the most interesting problems to solve with neural networks is the famous Traveling Salesman problem (TSP) (Garey and Johnson 1979). It is the problem how to determine the shortest tour between n given cities, where each city must be visited once, and the tour finishes in the city where it began. One can map this problem on a Hopfield neural network with n^2 neurons.

This mapping is explained as follows: put the neurons in a matrix of size $n \times n$. A high output in row i and column j means that the ith town will be visited as number j in the tour. The conditions of the problem indicate that each valid tour has exactly one neuron high in each row and in each column, while all other outputs are low. This is equivalent to the condition that the matrix of neurons is a permutation matrix (Figure 7.5).

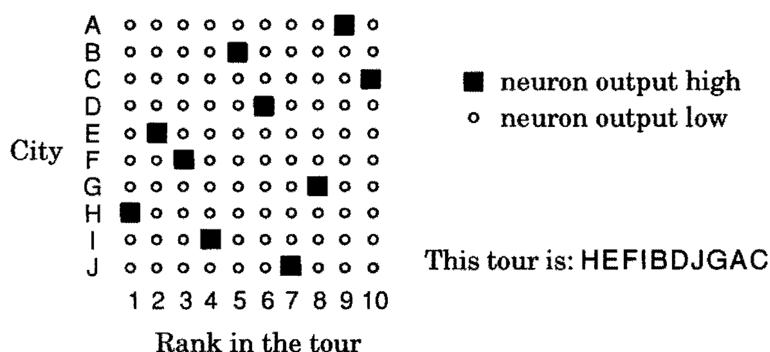


Figure 7.5. Permutation matrix of neurons

The weights of the connections can be determined relatively easy. The conditions of a valid tour are translated into the weights of the connections of a cell with all cells in the same row and in the same column. The optimization problem, to find the shortest tour, has influence on the weights of a cell with the cells in the two adjacent columns. These weights depend on the distance between the related cities. To ensure that the network will converge to a stable state, all weights in all cells are biased to force that the sum of all outputs is equal to the number of cities. Because every weight is nonzero, the resulting network is a fully connected network.

7.3.1 Modeling a Hopfield neuron

To simulate a neural network, first a model of a neuron must be developed. Because in the future different types of neurons might be used, several models are created. The models are built with two or three connected blocks: a summation block, an integration block and a nonlinearity (see Figure 7.6).

The summation block multiplies each input value with its weight, sums these values, and adds an offset to it. This value is the input to the integration block, which implements the dynamic behavior of the neuron. For neurons without

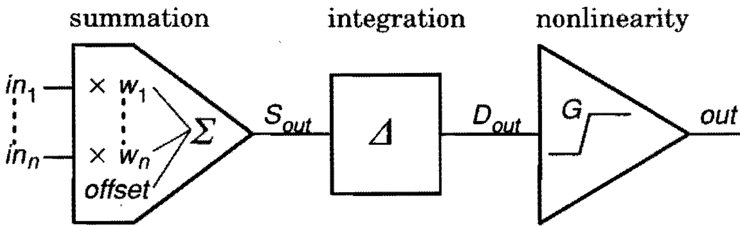


Figure 7.6. Piecewise linear dynamic model of a neuron

delay this block is omitted. The output of this block is magnified and clipped by the nonlinearity. This is the same block as used in the logic gates, and may be either a simple clipping (analog) function, as in equation (2.19), or a discontinuity like in equation (2.23). In a Hopfield neural network used to solve optimization problems, the neuron is dynamic and analog, so the clipping function is continuous. The equations of these three blocks are:

$$\begin{aligned} S_{out} &= \sum_i w_i \cdot in_i + offset \\ \frac{\partial D_{out}}{\partial t} &= \frac{1}{\Delta}(S_{out} - D_{out}) \ , \quad D_{out}(0) = D_0 \\ out &= \begin{cases} 0 & \text{if } D_{out} \le 0 \\ G \cdot D_{out} & \text{if } 0 < D_{out} < 1/G \\ 1 & \text{if } D_{out} \ge 1/G \end{cases} \end{aligned} \tag{7.1}$$

The behavior of a neuron is determined by the parameters w_i, *offset*, Δ, and G. The convergence of the network in optimization problems depends on these parameters. It is found experimentally that the delay time Δ and the gain G only depend on the magnitude of the weights, and can be determined independent of the actual circuit. The weights w_i and the parameter *offset* depend on the problem that is solved.

7.3.2 PLANNET, a new neural net simulator

With the model described in Figure 7.6 and equation (7.1), a number of TSP problems have been simulated with PLATO. In the next subsection, the results indicate that this model is satisfactory for these kind of problems. A disadvantage is the long simulation time. This is to be expected, because PLATO is a simulator whose speed is largely based on the use of the sparse structure of electrical circuits by applying specialized methods. All these techniques have a large overhead when applied to a fully connected network.

Therefore a specialized version of PLATO has been developed, called PLANNET, an acronym for Piecewise Linear Analysis of Neural NETworks. It is a parallelized version, specially adjusted to an Alliant FX/8 computer, a multi–

processor shared memory machine, where each processor has vector calculation capabilities. PLANNET uses the same algorithms as discussed in the previous chapters, with the exception of the multirate integration methods: the standard (uniform–step) integration method is applied instead.

Because in a fully connected network the matrix A_{xx} becomes full if an implicit integration method is applied, the matrix is not stored in a sparse data structure. As a consequence, the sparse versions of the matrix algorithms have been changed to work with this structure. To use the advantages of the sparse implementation, the rank m update (Algorithm 3.3) is optimized for a rank 1 update of one row of the matrix. Also the leaf cell routines are adjusted to the neuron model of Figure 7.6 and equation (7.1).

These basic changes are independent of a specific computer architecture. To use the full power of the Alliant, it is sufficient to modify the simulator and the compilation process on a few points:
- The model evaluation routines are called in parallel. This is possible because the neuron data are independent of each other and do not directly change the global data.
- All vector and matrix routines that are vectorizable or parallelizable are vectorized and parallelized automatically by the compiler.

7.3.3 Results

The behavior of the neural network can be explained by some typical outputs, as are shown in Figure 7.7. The output values of all neurons initially drop to a certain value, depending on the problem at hand. The differences between the various output values are small, and this state of the network is meta stable: applying a DC analysis on the network yields also this state. However, this state is dynamically approached, so the network finally diverges from it. At this point, many neurons have the same behavior as neuron78: their outputs drop to zero, the nonlinearity clips, and they remain in this state. At the same time, a competition starts between those neurons that still have a positive output value. Some of them win easily (neuron9), some win with more difficulties (neuron56), some loose straightforward (neuron66) and some loose after a promising start (neuron93). In the end, a stable state is found and the simulation stops.

The simulation of TSP problems shows that the neural network nearly always converges to a stable and feasible solution. Convergence and feasibility are sometimes a problem in other simulators, most likely because the calculations are performed neuron by neuron. In PLANNET, the neurons are treated all at the same time, which introduces more stability in the calculations. The results of the simulations are also in most cases close to the optimal solution. This solution depends on the initial states of the neurons and on the integration accuracy used in the simulation. Because the neural network has more

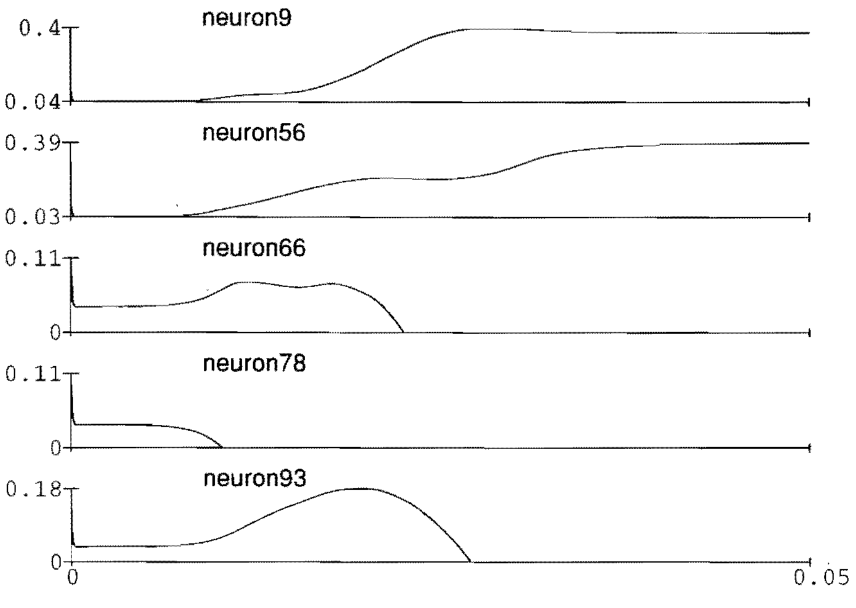


Figure 7.7. Some outputs of neurons

than one optimal solution (the starting city and the direction are not prescribed), this is not surprising.

A second important result is the fact that it is not necessary to apply an implicit integration method to solve the TSP. The explicit Forward Euler method yields the same results as the Trapezoidal Rule. Because the explicit method is much cheaper (the global matrix remains the identity matrix all through the simulation), this method is used to benchmark the simulator. However, it is not sure if the Forward Euler method can be applied in all neural network simulations. The method is applicable here, because the problem is not stiff.

A third important result is the fact that there is no significant difference in the numerical solution calculated by PLATO and by the vector–parallel PLANNET. This is not trivial, because in the vectorizing and parallelizing of loops the rounding errors differ from the straightforward loops. In the signals a difference is found of less than 10^{-5}, which has no significant effect on the results.

The performance of PLANNET is much better, compared to the performance of PLATO. The results are summarized in Table 7.1. The run times are measured on an Alliant FX/8 vector–parallel computer with four processors calculating concurrently on the simulation. On this machine, the vectorization can yield a speed–up factor 3, and with 4 processors a maximal speed–up factor of 12 can be expected. This is comparable with the speed–up factor 7 of the larger problems, which is the best performance one can expect, due to the overhead

when running in vector–parallel mode. Unfortunately, the larger problems could not be run with PLATO, due to the network description size (more than 30 Mbyte) and memory consumption (more than 120 Mbyte).

The quality of the solution can be deduced from the last two columns of Table 7.1. A good solution is obtained with a few runs of the simulator. From inspection by hand the shortest tours are nearly optimal. For the 7 and 10 city problems the optimal solution is found.

Table 7.1. Mean run times for several runs and results for various problem sizes. Between parentheses the number of runs.

number of cities	run time (in seconds)			speed-up factor	mean tour length	shortest tour length found
	PLATO	PLANNET linear	PLANNET vector–parallel			
7	245 (5)	143 (5)	47 (5)	3.0	2.49	2.47
10	480 (5)	271 (10)	106 (10)	2.6	2.77	2.69
20	11400 (1)	3930 (10)	1059 (10)	3.7	3.66	3.40
30	–	26466 (4)	4627 (10)	5.7	4.91	4.25
40	–	128852 (1)	18264 (5)	7.1	6.54	6.22

Some conclusions of these experiments are:

- Neural networks, used in optimization problems, can be modeled and simulated well with PLATO or PLANNET. A good or optimal solution is always found.
- A TSP can be simulated well with the simple Forward Euler integration method.
- Large networks can be simulated with PLANNET, and parallel and vector computing yield a considerable speed up in run time.
- Solving a TSP with a Hopfield neural network is not efficient.

7.4 A switch capacitor filter

A third example is the simulation of a switch capacitor filter described in (Hegt 1988). This is a circuit containing switches, capacitors and operational amplifiers. Its schematic diagram is shown in Figure 7.8. The circuit acts as a filter, i.e. an input signal is transformed to an output signal according to a so called transfer function. The amplitude transfer function of this circuit is given in Figure 7.10 (solid line). The behavior of the circuit is determined not only by its composure, but also by the clock frequency at which the switches are opened and closed. The dynamic behavior of the filter is therefore characterized by two values: the clock frequency and the time to load the capacitors after a switch. The filter will only behave properly, if the former one is much larger than the latter one. So the dynamics of the filter are strong: at the switching points large currents flow through the circuit, and during the time interval between the switching the circuit is nearly at rest.

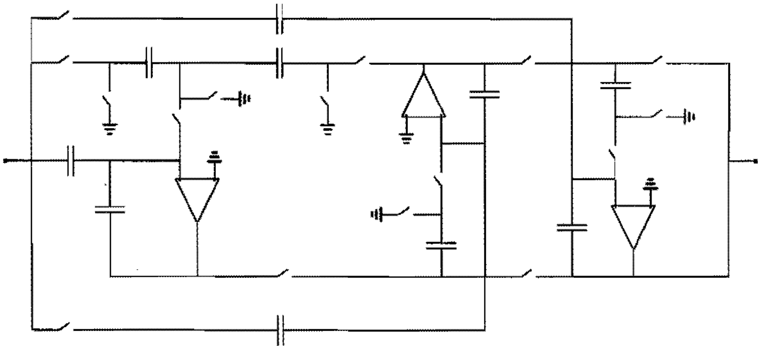
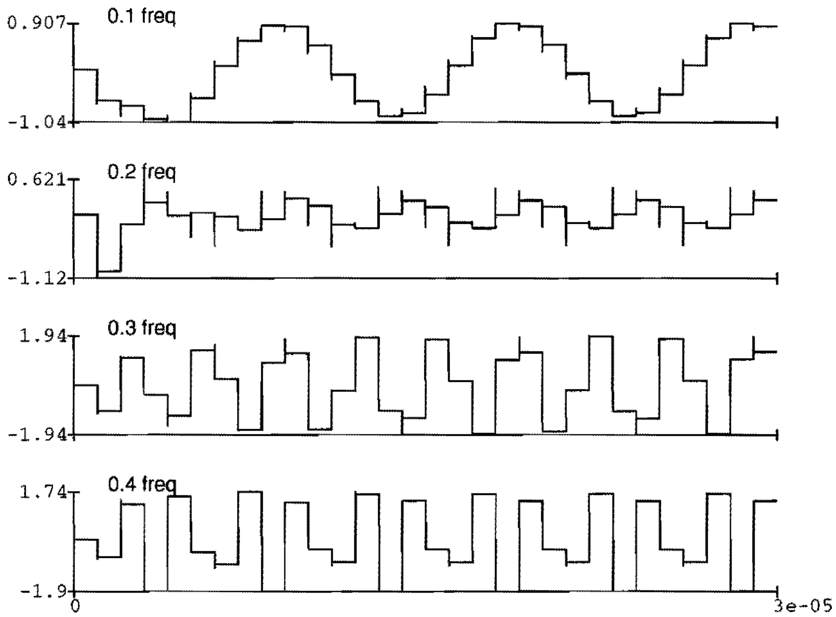


Figure 7.8. A switch capacitor filter

The filter of Figure 7.8 is simulated with idealized components. Because in that case several node voltages in the network are undetermined, and one node voltage is overdetermined, a few parasitic components are added to the circuit to be able to solve the circuit equations. The behavior of the circuit introduces a numerical problem: the characteristic times of the switches and the clock may differ with a factor 10^4. This implies that much effort must be put into the simulation to ensure that all variables are calculated with enough precision. If the error in a derivative can be neglected on the fast time scale, it can not on the slow time scale. Notice that this has nothing to do with a stiff integration problem, because the clock and the internal switch time are not related.

The result of the simulation is an amplitude transfer function. To determine this function, the circuit has been simulated with 10 different input functions. The input functions are sines with different frequencies. A sample–and–hold subcircuit is added to the filter, in order to stabilize the input for the filter. To measure the amplitude of the output of this filter, in practice another filter is used, because the output only slightly resembles a sine (see Figure 7.9). However, we have used a different technique to determine the amplitude. The output signal is sampled, and with a numerical Fourier transform the lowest part of the frequency spectrum is determined. Of this spectrum, the higher components are set to zero. An inverse Fourier transformation yields a sine, whose frequency is the same as the input frequency. The quotient of the amplitudes of both sines is plotted in Figure 7.10 (dashed line with x denoting measured points). These calculations have been performed with *Mathematica* (Wolfram 1988).

The measured points of the transfer function match very good with the exact transfer function. Only for inputs with a high frequency a deviation is found.



The signals are denoted by their relative frequencies. The amplitude of the input sines is 2.

Figure 7.9. Several outputs of the filter for different inputs

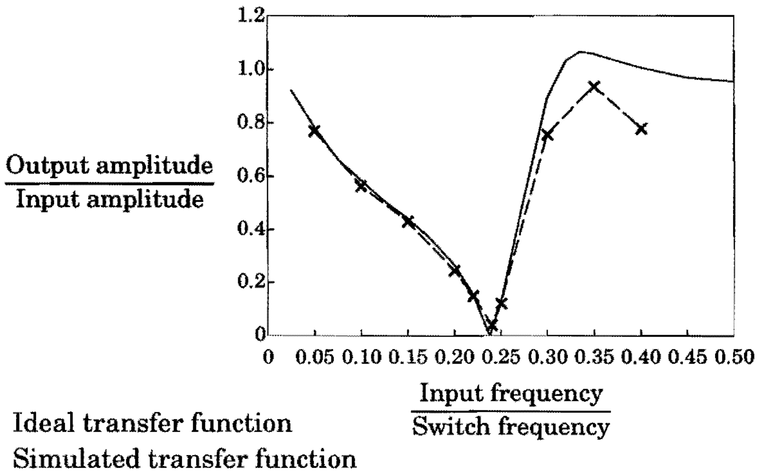


Figure 7.10. Ideal and simulated amplitude transfer function of the filter

This can be explained by the sample–and–hold subcircuit in front of the filter: this acts as a filter with a sin(x)/x transfer function, so the resulting transfer function of the full circuit is a combination of both transfer functions.

Concluding, a system which a complex dynamic behavior can be simulated with PLATO without too many difficulties. The application of idealized components gives both advantages (less computer time) and disadvantages (some parasitic components must be added).

7.5 Program statistics

To show the effect of the methods and techniques described in the earlier chapters, some statistical data have been gathered (Table 7.2). These data are divided into four area's: the size of the problem, the number of calls to the main algorithms, the connectivity in the problem, and the time spent in some (possibly overlapping) parts of the simulator. The timing data are measured with the profiling technique available on the computer (Graham *et al.* 1982). Profiling a program means that each call to a routine is recorded, and that also at regular time intervals the currently executed routine is recorded. However, because this time interval is (for the used computer) much longer than the time spent in a routine, quantization errors occur. Therefore, the time percentages are only indications of the real time spent in the indicated parts.

The circuits of which the data are shown, were described earlier: adc is the analog–digital converter, filter is the switch–capacitor filter, and nn20 is a neural network for solving the 20 city TSP. Two versions of the analog–digital converter are used: adc is the circuit described in Figure 7.1, and adc2 is the same circuit, in which the digital components in the main circuit are replaced by some inputs. Also two versions of nn20 are simulated: nn20a is the circuit with neurons as described in Figure 7.6, while the neurons in nn20b are split in a linear summation component and a dynamic piecewise linear component with one input and one output. The expectation is that the second version simulates faster, because the neurons in the first version are (too) large, and the leaf cell routines will become inefficient. The neural network is not simulated with PLANNET, but with PLATO. This implies that the sparse implementation for the vectors and matrices is used, and that no uniform integration step is applied.

Some data in Table 7.2 need an explanation. The unit of sparsity is the number of elements of the LU decomposition per row, which is limited between 1 (a diagonal matrix) and n_x (a full matrix). This number of elements also includes spurious elements (both the matrix entry and the L or U entry are negligible), because these elements are not removed. This is the reason for the high number of elements per row for the filter. About 25 % of the entries changes from zero to nonzero or vice versa at the switching time point. It is also clear, that nn20a uses a diagonal matrix (each x variable depends only

on u and z variables), but that in nn20b a full block is created with a size of 400x400.

The number of rank m updates (nearly all are rank 1 updates) is zero for the neural network, because the applied explicit Forward Euler integration does not change the linear equations.
The high number of leaf cells reached in an event for the filter is explained by the fact that all cells are connected through the currents in the circuit
The run times are measured on a HP 9000/750 computer, a fast workstation running 76 MIPS (million instructions per second) and 22 MFLOPS (million floating point operations per second). The run time for nn20a includes 38.9 seconds to process the 5.5 Mbyte circuit description file and 39.3 seconds to simplify (in vain) the large neurons. The same circuit simulated with PLANNET used only 162 seconds total run time, because this simulator knows the model of the neurons. The run time for nn20b includes 38.8 seconds to process the 5.6 Mbyte circuit description file and 149.6 seconds to swap the (sparsely stored) rows during the LU decomposition. This circuit was not simulated with PLANNET, because the split neurons could not be created.

The relative time spent in several routines is the last part of Table 7.2. Some results can be noticed. The rank m update is efficient with respect to the other routines. The matrix is solved many times, which accounts for the time spent in this routine. In the leaf cell routines most of the calculations are performed, which is the reason for them using a large part of the time. For the neural networks, it is clear that with the very large neurons of nn20a nearly all time is spent in the leaf cell calculations. For nn20b, the large neurons are split, and the large summation block causing the difficulties is handled more efficiently in the global matrix. The efficiency of the output processing is shown by the small amount of time spent in it. The list processing, i.e. the construction of a list of leaf cells related to a sparse vector takes some time, but saves much more time. Finally, the time spent in selecting the leaf cells for the next event is not negligible, but it is not large enough to take action.

Table 7.2. Program statistics for several circuits.
is an abbreviation for 'number of'.

	adc	adc2	filter	nn20a	nn20b
# leaf cells	74	58	39	400	800
n_x	97	85	46	400	800
n_u	61	45	14	400	400
n_z	280	170	34	800	800
sparsity [elements/row]	2.3	2.4	6.5	1	201
# rank m updates	3381	2421	69485	0	0
# matrix solutions	20370	11788	57586	5743	5741
# pl events	2456	1393	119	384	384
# dynamic events	3216	2722	52815	4590	4588
av. # leaf cells / event	3.4	3.0	2.4	166.1	166.1
av. # leaf cells reached	7.6	5.4	22.5	284.4	284.3
run time [s]	2.7	1.8	39.6	787.8	502.6
% rank m update	3.6	2.2	24.8	0.0	0.0
% matrix solution	10.1	18.1	14.6	0.2	35.4
% leaf cell routines	47.7	43.4	33.9	84.5	17.7
% output processing	5.2	6.0	0.5	0.3	0.4
% list processing	2.6	1.7	4.9	3.6	0.9
% clustering	4.2	8.8	2.1	0.2	0.3
% Van de Panne	36.1	28.7	7.5	0.6	0.8

8 Conclusions

In this thesis, the development of PLATO, a piecewise linear circuit simulator, has been sketched. The following topics have been discussed:

- The piecewise linear modeling technique, introducing linear, piecewise linear and dynamic equations.
- The solution of the linear equations with a sparse LU decomposition, and the rank m update on these equations.
- The Van de Panne algorithm, applied in solving the piecewise linear equations, with particular attention to the DC solution.
- The integration methods, for solving the dynamic equations.
- The multirate integration and clustering methods, exploiting latency.
- Several implementation techniques to exploit the efficiency in these algorithms.

The combination of all these methods and algorithms has created a simulator with the following properties:

- It has strong convergence properties, based on the strong convergence of the Van de Panne algorithm. It is therefore robust, and can handle mixed–level circuits that other simulators can not simulate.
- It is efficient, due to the sparse matrix techniques combined with the multirate integration method and the application of divided differences.
- It is flexible, because the modeling allows mixing different types of components in one circuit that can be simulated immediately. Also the connection with other programs is straightforward.

Several circuits are given as examples that have been simulated. These circuits are specially chosen to show the above mentioned properties of the simulator. One of these examples, a neural network, has led to a simulator specially suited to solve fully connected neural networks.

Future developments for improving the performance, stability and flexibility of the simulator will most likely be concentrated on the following fields:

- The concept of leaf cells will be abandoned. This concept is natural in a hierarchical simulator, but it has some serious disadvantages in our simulator: the sparse vectors must be linked with the leaf cells, and in the leaf cells a full matrix is used. Because this matrix is in many cases rather sparse,

much work is done in vain. Using sparsely implemented leaf cells, the cells itself become useless.

- New higher order integration methods, e.g. the Backward Difference formulae, could be employed in dynamic problems, i.e. in cases where the current integration method uses many steps, while no nonlinearities are encountered. Because the efficiency of multirate integration might easily get lost in the combination with a high order implicit integration method, the gain in employing such integration methods is uncertain.
- The connection with ESCAPE will be intensified. Because in the near future ESCAPE will have the possibilities to connect to a range of different simulators, to simulate functionally or behaviorally described components and to use a network of computers to solve a large problem in parallel, those circuits that can not simulated with PLATO can be simulated with a combined ESCAPE–PLATO simulator. The first tests show good results.
- The simulator will be used to solve problems that are not directly electronic circuits. As a problem can also be seen as a dynamic LCP, it is trivial to solve a linear programming problem with PLATO. By adding a time relation to the system, it is possible to determine the relation between two variables of the original problem in one simulation. Some tests on known problems show that such a relation can be determined fast and accurate.
- On the lowest level of the implementation, the data structure for the matrices may become more flexible, in order to solve several problems more efficiently. If it is known that (part of) the system matrix has some specific properties, a specially tuned data structure is chosen. This could imply that the neural network simulator PLANNET will be incorporated in PLATO.

References

ALMEIDO, A.R.C. DE, I.M. MACLEOD, and T.J. YPMA, 1989, "Distributed–Multirate Methods for Large Weakly–Coupled Differential Systems", *Applied Mathematics and Computation*, vol. 31 (Proc. 1986 ODE Conf.), pp. 18–39.

BENNETT, J.M., 1965, "Triangular Factors of Modified Matrices", *Numerische Mathematik*, vol. 7, pp. 217–221.

BOKHOVEN, W.M.G. VAN, 1981, *Piecewise-Linear Modelling and Analysis*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

BRYANT, R.E., 1984, "A Switch–Level Model and Simulator for MOS Digital Systems", *IEEE Trans. on Computers*, vol. C–33, no. 2, pp. 160–177.

BRYANT, R.E., D. BEATTY, K. BRACE, K. CHO, and T. SHEFFLER, 1987, "COSMOS: a Compiled Simulator for MOS Circuits", in *Proc. 24th ACM / IEEE Design Automation Conf.*, June 28–July 1, 1987, Miami, pp. 9–16, IEEE Computer Society Press, Piscataway, N.J.

BUURMAN, H.W., J.T.J. VAN EIJNDHOVEN, and Q.J.A. VAN GEMERT, 1990, *The NDML++ Network Description and Modeling Language*, internal paper.

BUURMAN, H.W., W.J.M. PHILIPSEN, and J. VAN SPAANDONK, 1991, "PLANNET: a New Neural Net Simulator", in *Artificial Neural Networks (Proc. ICANN–91)*, June 24–28, 1991, Espoo, Finland, vol. 2, pp. 1481–1484, North–Holland, Amsterdam.

CARDON, H., HOOGSTRATEN, R. VAN, and DAVIES, P., 1991, "A Neural Network Application in Geology: Identification of Genetic Facies", in *Artificial Neural Networks (Proc. ICANN–91)*, June 24–28, 1991, Espoo, Finland, vol. 1, pp. 809–813, North–Holland, Amsterdam.

CHAWLA, B.R., H.K. GUMMEL, and P. KOZAK, 1975, "MOTIS – An MOS Timing Simulator", *IEEE Trans. on Circuits and Systems*, vol. CAS–22, no. 12, pp. 901–910.

CHIEN, M.J., and E.S. KUH, 1976, "Solving Piecewise Linear Equations for Resistive Networks", *Int. J. of Circuit Theory and Applications*, vol. 4, no. 1, pp. 3–24.

CHUA, L.O., and A. DENG, 1986, "Canonical Piecewise–Linear Modeling", *IEEE Trans. on Circuits and Systems*, vol. CAS–33, no. 5, pp. 511–525.

CHUA, L.O., and P.M. LIN, 1975, *Computer–Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques*, Prentice–Hall, Englewood Cliffs, N.J.

COTTLE, R.W., and R.E. STONE, 1983, "On the Uniqueness of Solutions to Linear Complementarity Problems", *Mathematical Programming*, vol. 27, pp. 191–213.

DAHLQUIST, G., 1959, *Stability and Error Bounds in the Numerical Integration of Ordinary Differential Equations*, Trans. Royal Inst. Tech., no. 130, Stockholm.

DOVERSPIKE, R.D., and C.E. LEMKE, 1982, "A Partial Characterization of a Class of Matrices Defined by Solutions to the Linear Complementarity Problem", *Mathematics of Operations Research*, vol. 7, no. 2, pp. 272–294.

DUMLUGOL, D., P. ODENT, J. COCKX, and H. DE MAN, 1987, "Switch–Electrical Segmented Waveform Relaxation for Digital MOS VLSI and its Acceleration on Parallel Computers", *IEEE Trans. on Computer Aided Design*, vol. CAD–6, no. 6, pp. 992–1005.

EIJNDHOVEN, J.T.J. VAN, 1984, *A Piecewise Linear Simulator for Large Scale Integrated Circuits*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

EIJNDHOVEN, J.T.J. VAN, 1988, "Piecewise Linear Analysis", in *Analog Circuits: Computer Aided Analysis and Diagnosis*, ed. T. Ozawa, pp. 65–92, Marcel Dekker, New York.

EIJNDHOVEN, J.T.J. VAN, and M.T. VAN STIPHOUT, 1988, "Latency Exploitation in Circuit Simulation by Sparse Matrix Techniques", in *Proc. Int. Symp. on Circuits And Systems*, June 7–9, 1988, Espoo, Finland, pp. 623–626, IEEE, Piscataway, N.J.

EIJNDHOVEN, J.T.J. VAN, M.T. VAN STIPHOUT, and H.W. BUURMAN, 1990, "Multirate Integration in a Direct Simulation Method", in *Proc. European Design Automation Conf.*, March 12–15, 1990, Glasgow, Scotland, pp. 306–309, IEEE Computer Society Press, Washington.

FLEURKENS, J.W.G., and H.W. BUURMAN, 1992, "Flexible Mixed–Mode and Mixed–Level Simulation", submitted to *1993 IEEE Int. Symp. on Circuits And Systems.*

GAREY, M.R., and D.S. JOHNSON, 1979, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, Freeman, New York.

GEAR, C.W., 1971, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice–Hall, Englewood Cliffs, N.J.

GEAR, C.W., and D.R. WELLS, 1984, "Multirate Linear Multistep Methods", *BIT*, vol. 24, pp. 484–502.

GENDEREN, A.J. VAN, and A.C. DE GRAAF, 1986, "SLS: A Switch–Level Timing Simulator", in *The Integrated Circuit Design Book – Papers on VLSI Design Methodology from the ICD–NELSIS Project*, ed. P. Dewilde, pp. 2.93–2.146, Delft University Press, Delft, The Netherlands.

GOLUB, G.H., and C.F. VAN LOAN, 1989, *Matrix Computations,* 2nd ed., John Hopkins University Press, Baltimore.

GRAHAM, S.L., P.B. KESSSLER, and M.K. MCKUSICK, 1982, "gprof: A Call Graph Execution Profiler", *SIGPLAN Notices*, vol. 17, no. 6 (Proc. SIGPLAN '82 Symp. on Compiler Construction), pp. 120–126.

HAIRER, E., S.P. NOERSETT, and G. WANNER, 1987, *Solving Ordinary Differential Equations I*, Springer–Verlag, Berlin.

HEGT, J.A., 1988, *Contributions to Switched Capacitor Filter Synthesis*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

HOPFIELD, J.J., 1984, "Neurons with Graded Response Have Collective Computational Properties like Those of Two–state Neurons", *Proc. Nat. Academy of Sciences USA*, vol. 81, pp. 3088–3092.

HOPFIELD, J.J., and D.W. TANK, 1985, "Neural Computation of Decisions in Optimization Problems", *Biological Cybernetics*, vol. 52, no. 4, pp. 141–152.

HSIEH, H.Y., A.E. RUEHLI, and P. LEDAK, 1985, "Progress on Toggle: A Waveform Relaxation VLSI–MOSFET CAD Program", in *Proc. 1985 Int. Symp. on Circuits and Systems*, June 5–7, 1985, Kyoto, Japan, pp. 213–216, IEEE, Piscataway, N.J.

IEEE, 1988, *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076–1987, IEEE, New York.

JANSSEN, G.L.J.M., 1986, "Network Description and Modeling Language – NDML", in *The Integrated Circuit Design Book – Papers on VLSI Design Methodology from the ICD–NELSIS Project*, ed. P. Dewilde, pp. 4.60–4.108, Delft University Press, Delft, The Netherlands.

JANSSEN, G.L.J.M., 1989, "Circuit Modelling and Animated Interactive Simulation in Escher+", in *Simulation Applied to Manufacturing, Energy and Environmental Studies and Electronics and Computer Engineering (Proc. SCS European Simulation Multiconference)*, June 7–9, 1989, Rome, Italy, ed. S. Tucci, A. Mathis, W. Hahn, and R.N. Zobel, pp. 265–270, SCS, Ghent, Belgium.

JONES, P.C., 1986, “Even More with the Lemke Complementarity Algorithm”, *Mathematical Programming*, vol. 25, pp. 239–242.

KAHLERT, C., and L.O. CHUA, 1990, “A Generalized Canonical Piecewise–Linear Representation”, *IEEE Trans. on Circuits and Systems*, vol. CAS–37, no. 3, pp. 373–383.

KATZENELSON, J., 1965, “An Algorithm for Solving Nonlinear Resistor Networks”, *Bell Systems Technical J.*, vol. 44, no. 8, pp. 1605–1620.

KERNIGHAN, B.W., and D.M. RITCHIE, 1988, *The C Programming Language*, 2nd ed., Prentice–Hall, Englewood Cliffs, N.J.

KEVENAAR, T.A.M., 1990, “Generating Macromodels of Combinatorial Logic for Piecewise Linear Mixed–Level Simulation”, in *Proc. 2nd Symp. on Design Methodology*, April 1990, Dalfsen, The Netherlands, ed. J.P. Veen, pp. 111–114, STW, Utrecht, The Netherlands.

KRODEL, T.H., and K.J. ANTREICH, 1990, “An Accurate Model for Ambiguity Delay Simulation”, in *Proc. European Design Automation Conf.*, March 12–15, 1990, Glasgow, Scotland, pp. 563–567, IEEE Computer Society Press, Washington.

LEENAERTS, D.M.W., 1992, *TOPICS: A Contribution to Analog Design Automation*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

LELARASMEE, E., A.E. RUEHLI, and A.L. SANGIOVANNI-VINCENTELLI, 1982, “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.

LEMKE, C.E., 1968, “On Complementary Pivot Theory”, in *Mathematics of the Decision Sciences – part 1*, ed. G.B. Dantzig, and A.F. Veinott, pp. 95–114, American Mathematical Society, Providence, Rhode Island.

LODDER, A., M.T. VAN STIPHOUT, and J.T.J. VAN EIJNDHOVEN, 1986, *ESCHER: Eindhoven SCHematic EditoR – Reference Manual*, EUT Report 86–E–157, Eindhoven University of Technology, Eindhoven, The Netherlands.

MARKOWITZ, H.W., 1957, “The Elimination Form of the Inverse and its Application to Linear Programming”, *Management Sci.*, vol. 3, pp. 255–269.

MOLER, C., and C. VAN LOAN, 1978, “Nineteen Dubious Ways to Compute the Exponential of a Matrix”, *SIAM Review*, vol. 20, no. 4, pp. 801–836.

NAGEL, L.W., 1975, *SPICE2: a Computer Program to Simulate Semiconductor Circuits*, Memorandum No. ERL–M520, University of California, Berkeley.

NEWTON, A.R., 1979, "Techniques for the Simulation of Large–Scale Integrated Circuits", *IEEE Trans. on Circuits and Systems*, vol. CAS–26, no. 9, pp. 741–749.

OSF, 1990, *OSF/Motif Style Guide*, Open Software Foundation, Cambridge, Mass.

SAAB, D.G., R.B. MUELLER–THUNS, D.T. BLAAUW, J.A. ABRAHAM, and J.T. RAHMEH, 1988, "CHAMP: Concurrent Hierarchical And Multilevel Program for Simulation of VLSI Circuits", in *IEEE Int. Conf. on Computer–Aided Design – Digest of Technical Papers*, Nov. 7–10, 1988, Santa Clara, Calif., pp. 246–249.

SELBERHERR, S., 1984, *Analysis and Simulation of Semiconductor Devices*, Springer–Verlag, New York.

STIPHOUT, M.T. VAN, 1990, *PLATO – A Piecewise Linear Analysis Tool for Mixed Level Circuit Simulation*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

STIPHOUT, M.T. VAN, J.T.J. VAN EIJNDHOVEN, and H.W. BUURMAN, 1990, "PLATO: a New Piecewise Linear Simulation Tool", in *Proc. European Design Automation Conf.*, March 12–15, 1990, Glasgow, Scotland, pp. 235–239, IEEE Computer Society Press, Washington.

TANAKA, T., and H. ENDO, 1991, "Trouble Forecasting System by Multi–Neural Network on Continuous Casting Process of Steel Production", in *Artificial Neural Networks (Proc. ICANN–91)*, June 24–28, 1991, Espoo, Finland, vol. 1, pp. 835–840, North–Holland, Amsterdam.

VAN DE PANNE, C., 1974, "A Complementary Variant of Lemke's Method for the Linear Complementarity Problem", *Mathematical Programming*, vol. 7, pp. 283–310.

WARMERS, H., D. SASS, and E.-H. HORNEBER, 1990, "Switch–Level Timing Models in the MOS Simulator BRASIL", in *Proc. European Design Automation Conf.*, March 12–15, 1990, Glasgow, Scotland, pp. 568–572, IEEE Computer Society Press, Washington.

WEEKS, W.T., A.J. JIMENEZ, G.W. MAHONEY, D. MEHTA, H. QASSEMZADEH, and T.R. SCOTT, 1973, "Algorithms for ASTAP – A Network Analysis Program", *IEEE Trans. on Circuit Theory*, vol. CT–20, no. 6, pp. 628–634.

WOLFRAM, S., 1988, *Mathematica™ – A System for Doing Mathematics by Computer*, Addison–Wesley, Redwood City, Calif.

Appendix

A Mapping a four-segment function on a 2x2 matrix

In this appendix, some results are presented about the mapping on a 2x2 matrix of an arbitrary four-segment piecewise linear function $f: \mathbb{R} \rightarrow \mathbb{R}$. Although this seems theoretically trivial, because 2^{n_z} regions can be modeled, the matrix has only a limited freedom. There are in principle only $(n_z+1) \times (n_z+2)$ free parameters (matrix entries) to model a continuous function from $\mathbb{R}$ to $\mathbb{R}$. Because the w and z variables may be divided by any positive value, there are $2n_z$ free parameters less, resulting in $n_z^2 + n_z + 2$ free parameters. This implies that a four-segment function, which has 8 parameters, might be mapped on a 2x2 matrix, having 8 free parameters. In general, the number of free parameters of the matrix limits the number of segments of the function. Only if dependency between the regions exists, more segments might be possible.

The function f is described by

$$y = f(x) = \begin{cases} d_0\, x + b_0, & x \leq x_0 \\ d_1\, x + b_1, & x_0 \leq x \leq x_1 \\ d_2\, x + b_2, & x_1 \leq x \leq x_2 \\ d_3\, x + b_3, & x_2 \leq x \end{cases}, \tag{A.1}$$

with $x_0 < x_1 < x_2$.

The idea is to map this function with the matrix

$$\begin{bmatrix} -1 & a_{00} & a_{01} & a_{02} \\ 0 & a_{10} & a_{11} & a_{12} \\ 0 & a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} y \\ x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix}. \tag{A.2}$$

Because w and z may be scaled arbitrarily, the rows are scaled such that a_{10} and a_{20} are +1 or −1. These values can not be 0, because valid regions must be defined. The columns of the matrix are scaled such that a_{11} and a_{22} are +1 or −1. Then 8 parameters are left, which might be determined by the 8 parameters of the function.

The initial state of (A.2) will map the second segment of the function. A positive pivot on a_{11} yields the first segment, a positive pivot on a_{22} yields the third segment. A positive block pivot on a_{11} and a_{22} yields the last segment.

The construction starts in the second segment of the function ($y = d_1\ x + b_1$ for $x_0 \le x \le x_1$). This implies

$$a_{00} = d_1 \text{ and } a_0 = b_1 \ . \tag{A.3}$$

Furthermore the boundaries must be defined:

$$\begin{aligned} a_{10} &= 1 \quad \text{and } a_1 = -x_0 \ ; \\ a_{20} &= -1 \text{ and } a_1 = x_1 \ . \end{aligned} \tag{A.4}$$

Two positive pivots must be performed, which implies

$$a_{11} = 1 \text{ and } a_{22} = 1 \ . \tag{A.5}$$

The matrix has become:

$$\begin{bmatrix} -1 & d_1 & a_{01} & a_{02} \\ 0 & 1 & 1 & a_{12} \\ 0 & -1 & a_{21} & 1 \end{bmatrix} \begin{bmatrix} y \\ x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ -x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} \ . \tag{A.6}$$

Perform the pivot on a_{11} to find the first segment:

$$\begin{bmatrix} -1 & d_1 - a_{01} & a_{01} & a_{02} - a_{01}a_{12} \\ 0 & -1 & 1 & -a_{12} \\ 0 & -1 - a_{21} & a_{21} & 1 - a_{12}a_{21} \end{bmatrix} \begin{bmatrix} y \\ x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} b_1 + x_0 a_{01} \\ x_0 \\ x_1 + x_0 a_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} \ . \tag{A.7}$$

The linearization is $y = d_0\ x + b_0$, so

$$a_{01} = d_1 - d_0 \ . \tag{A.8}$$

In order to eliminate the possibility at this point that w_2 becomes negative for $x \to -\infty$, the condition $-1 - a_{21} < 0$ must be fulfilled.

The third line segment is found with a pivot on a_{22} (from the initial position):

$$\begin{bmatrix} -1 & d_1 + a_{02} & a_{01} - a_{02}a_{21} & a_{02} \\ 0 & 1 + a_{12} & 1 - a_{12}a_{21} & a_{12} \\ 0 & 1 & -a_{21} & 1 \end{bmatrix} \begin{bmatrix} y \\ x \\ z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} b_1 - x_1 a_{02} \\ -x_0 - x_1 a_{12} \\ -x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ w_1 \\ w_2 \end{bmatrix} \ . \tag{A.9}$$

Using the linearization $y = d_2\ x + b_2$, a_{02} is found:

$$a_{02} = d_2 - d_1 \ . \tag{A.10}$$

Next a positive pivot on a'_{11} must be performed, equivalent with crossing the boundary $x = x_2$. This is only possible if $1 - a_{12}a_{21} > 0$ and

$$a_{12} = -\frac{x_2 - x_0}{x_2 - x_1} \ . \tag{A.11}$$

Now $a_{12} < -1$, so the pivot on a_{11} can be performed. Introduce $d = \det(A_{zz}) = 1 - a_{12}a_{21}$, then $d > 0$ (the last positive pivot) also implies $-1 - a_{21} < 0$.

At last, perform the pivot on a'_{11} and match the linearization. This gives the following expression for a_{21}:

$$a_{21} = \frac{(x_1 - x_0)(d_1 - d_0) - (x_2 - x_1)(d_3 - d_2)}{(x_2 - x_0)(d_3 - d_1) - (x_2 - x_1)(d_2 - d_1)} \ . \tag{A.12}$$

This expression is used to calculate d:

$$d = \frac{x_1 - x_0}{x_2 - x_1} \cdot \frac{y_2 - y'_2}{y_0 - y'_0} \ , \tag{A.13}$$

where

$$\begin{cases} y_0 = f(x_0) \\ y_2 = f(x_2) \\ y'_0 = y_2 - d_3\ (x_2 - x_0) \\ y'_2 = y_0 - d_0\ (x_2 - x_0) \end{cases} .$$

So the inequality $d > 0$ implies $y_2 > y'_2 \wedge y_0 > y'_0$ or $y_2 < y'_2 \wedge y_0 < y'_0$. This is equivalent to the fact that the extrapolations of the first and the last line segment intersect each other in the interval $[x_0, x_2]$. See Figure A.1 for a graphic interpretation of the situation. The conclusion is that not all four–segment functions can be mapped with a 2x2 matrix.

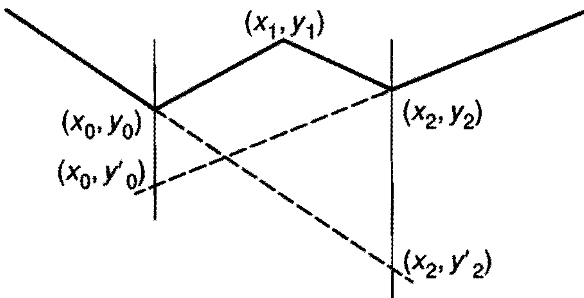


Figure A.1. Graphic implication of $d > 0$

Appendix B Notation

Symbols

x	vector of linear variables
u	vector of dynamic variables
w, z	vectors of pl variables
n_x, n_u, n_z	length of vectors x, u and z; w and z have the same length
A	system matrix
A_{pq}	(p,q) part of matrix A, with $p, q \in \{x, u, z\}$
a_p	p part of the vector a, with $p \in \{x, u, z\}$
A_{i*}, A_{*i}	row, column with index i of matrix A
$\left(A_{ij}\right)$	matrix, composed from all elements of matrix A with row index and column index in a given set
$\dot{u}$	time derivative of u
$x_{(i)}$	value of x at time point $t_0 + i\,h$
$\bar{x}$	divided difference: $\bar{x}_i = \dfrac{x_{(i+1)} - x_{(i)}}{h}$
$w \geq 0$	$w_i \geq 0$ for all indices i
e^A	$\sum_0^\infty \frac{A^n}{n!}$, the exponential of a matrix

Abbreviations

LCP	Linear Complementarity Problem
TSP	Traveling Salesman Problem
LTE	Local Truncation Error
ACF	A–Contractive Formula
BDF	Backward Difference Formula
BE	Backward Euler
TR	Trapezoidal Rule

Algorithms

The algorithms are described in a language, which resembles some features of ALGOL. However, no function headers are given, and only array variables are specified. To differentiate between function calls, variables, constants and keywords, they are printed in different fonts:

Van_de_Panne( )	function call
xvec	variable
successful	constant
while	keyword

Text between braces ({ }) is comment. To denote a field in a compound variable (record), the PASCAL dot notation is used.

A few non–standard statements are the following:

<*var1*, *var2*> := *func*( );	Both *var1* and *var2* are assigned a value by the function *func*.
for *var* := *val1* **to** *val2* ⟦ **whenever** *cond* ⟧ **do** *S* **od**;	Assign each (integer) value from *val1* upto and including *val2* to *var*, and perform the statements *S* if the condition *cond* is true. The **whenever** part is usually omitted, and then the statements *S* are always performed.
if *cond1* **then** *S1* ⟦ **elsif** *cond2* **then** *S2* ⟧+ ⟦ **else** *S3* ⟧ **fi**;	If condition *cond1* is true, perform statements *S1*. Else, if condition *cond2* is true, perform statements *S2*. Else, perform statements *S3*. The **elsif** part may be omitted or repeated, the **else** part may be omitted.

Biography

Hendrik Willem Buurman, better known as Pim, was born on June 4, 1961, in Wageningen. In this city he received his diploma Gymnasium–B in 1979. He studied Mathematics at the Eindhoven University of Technology, where he graduated in December, 1987, on the topic of Numerical Simulation of Photolithography.

From 1988, he has been working towards his Ph.D. degree in the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology. He expects to receive this degree based on the work in this thesis on January 20, 1993.

Stellingen

behorende bij het proefschrift
"From Circuit to Signal – development of a piecewise linear simulator"
van H.W. Buurman

1. Het gebruik in een simulator van een modellering die veel vrijheid geeft, moet vergezeld gaan van een algoritme dat ook alle modellen kan evalueren.

2. De opmerking dat de circuit–simulator PLATO een fastest–first multirate integratie schema gebruikt (p. 53), wordt in een volgend hoofdstuk gelogenstraft. [STIPHOUT, M.T. VAN, 1990, *PLATO – A Piecewise Linear Analysis Tool for Mixed Level Circuit Simulation*, Ph.D. Thesis, Eindhoven, The Netherlands]

3. Het gebruiken van een circuit–hiërarchie om de effecten van ijlheid te exploiteren is niet nodig. Deze effecten kunnen ook op andere manieren geëxploiteerd worden, terwijl een hiërarchie duidelijke nadelen heeft. [Dit proefschrift; KEVENAAR, T.A.M. en D.M.W. LEENAERTS, 1991, "A Flexible Hierarchical Piecewise Linear Simulator", *INTEGRATION, the VLSI Journal*, vol. 12, pp. 211–235]

4. Een symbolisch wiskunde manipulatie programma is zinvol als hulp bij het uitrekenen van moeilijke sommen. De beperkingen die aan de huidige versies van dit soort programma's kleven ondermijnen echter soms wel het vertrouwen in de uitkomst. [WOLFRAM, S., 1988, *Mathematica™ – A System for Doing Mathematics by Computer*, Addison–Wesley, Redwood City, Calif.]

5. Het 'backwards compatible' houden van systemen is niet handig. Het oude systeem is niet voor niets vervangen door een (hopelijk) beter systeem.

6. Een van de handigste mogelijkheden om te onderzoeken of een computerprogramma sneller gemaakt kan worden is het meten van de gebruikte tijd in de diverse routines met behulp van 'profilen'. Aangezien de hierin gebruikte sample–tijd op de nieuwste machines vele malen groter is dan de gemiddelde tijd die een routine gebruikt, verdient het aanbeveling om een oude trage machine te bewaren als 'profile' machine. [GRAHAM, S.L., P.B. KESSLER, and M.K. MCKUSICK, 1982, "gprof: A Call Graph Execution Profiler", *SIGPLAN Notices*, vol. 17, no. 6 (Proc. SIGPLAN '82 Symp. on Compiler Construction), pp. 120–126]

7. Indien een eigenschap van een electronisch circuit is gemeten in een proefopstelling, dan verdient het aanbeveling om bij publikatie van de meetresultaten ook de meetmethode en andere relevante gegevens te vermelden, zoals in andere takken van de wetenschap allang standaard is.

8. Het feit dat veel mensen niet begrijpen dat één computerprogramma op twee verschillende computers verschillende resultaten oplevert, geeft aan dat er meer onderwijs in numerieke wiskunde moet komen. ["comp.lang.c Answers to Frequently Asked Questions (FAQ List)", *newsnet, group comp.lang.c*, ed. S. Summit, monthly, Question 15.1]

9. Vele economische theorieën houden ten onrechte geen rekening met de kosten van communicatie. Aangezien een informele sfeer in een koffiekamer de onderlinge communicatie zeer ten goede komt, is het afschaffen hiervan vaak contra–produktief.

10. Er zijn computerprogramma's die zeer goed kunnen schaken of dammen, maar bridgende computerprogramma's worden makkelijk verslagen. Dit kan verklaard worden uit het feit dat een goed plan, gebaseerd op ervaring en aannames, noodzakelijk is voor een goed resultaat bij het bridgen.

11. Tijdens een wandeling ziet men het meest van de omgeving. Als iemand zijn werk in wandeltempo uitvoert, kan men dus verwachten dat hij goed over eventueel aanwezige problemen heeft nagedacht.

12. Het vangen van een vlieg in het halfdonker heeft een Heisenberg onzekerheidsrelatie met een zeer grote coëfficiënt: als hij stilzit, weet je niet waar; als hij beweegt, kun je hem niet pakken.