Diss. ETH No. 9382

Design and Implementation of a Three-Dimensional, General Purpose Semiconductor Device Simulator

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
GERNOT HEISER
Master of Science, Brock University
born 7 July 1957
citizen of Germany

accepted on the recommendation of
Prof. Dr. W. Fichtner, examiner
Prof. Dr. H. Baltes, co-examiner

1991

To my parents

Acknowledgements

I would first like to thank Prof. Wolfgang Fichtner, who not only was my supervisor for this thesis and the manager of the three dimensional device simulation project, but who also very actively participated in the project. Without his pressing (whenever he felt it necessary) and his encouragement (whenever I needed it) this project would have never been successful, and without the stimulating research atmosphere he provided at the lab I would never have started. I am grateful to Prof. Henry Baltes for accepting to co-examine the thesis and for reading it carefully.

I am greatly indebted to Joseph Bürgler of the lab, whose profound understanding of the theory were essential to the success; the many, often heated discussions we had were extremely productive. Many other people at ETH contributed in one way or the other. Paolo Conti, who has been working with me on the project from the beginning, was a good colleague and a reliable fellow in many battles against boredom, alcohol, and bureaucracy. Claude Pommerell was the \compatible" interface to the linear solver software, and a great help in not-so-serious discussions over a beer or two. The graphics software written by Stephan Paschedag and Marc Westermann was an indispensable help for debugging the simulator, and I am very grateful for their speedy and efficient cooperation. Nancy Hitschfeld, Kevin Kells and Doelf Aemmer contributed through many discussions. Special thanks are due to Peter Lamb, who not only kept the computers running, but whose expertise in may fields, including various non-technical ones, made it always exciting to talk to him.

My work was financially supported by Siemens AG, Berlin and München, and Cray Research Inc., Minneapolis. Roland Kircher from Siemens supplied the first examples for us to play with, and it was always a pleasure to talk to him. Other sample data came from Mark Birrittella from Cray, Richard Fair from MCNC, and Prof. Don Rose from Duke University. Marius Orlowski from Motorola Inc. invited me for a five weeks stay in Austin, where I had an exciting time on and off the job with Matt Noell, Sang-Kyu Park, Ravi Subrahmanyan, and James Berry. Armin Friedli delayed the conclusion of the thesis by giving me the chance to benchmark the world's fastest computers, an experience I do not want to miss.

My very special thanks go to Trudy Weibel. Not only has she always been supportive and encouraging, she also constantly reminded me that there were other things than just work for which it is worth living. Encouragements also came from other dear friends, in particular Hansruedi Heeb, Wes Petersen, Klaus Hinrichs and Edo Biagioni.

Abstract

Since the early work by Gummel in the 1960s, numerical simulation of semiconductor devices has developed into an indispensable tool for device engineers. So far, most device simulations have been one or two dimensional. With continuously shrinking device features truly three-dimensional (3d) treatment of the semiconductor becomes necessary.

A few 3d device simulation programs exist since the early 1980s, but their applicability is limited by the fact that they cannot simulate really general device geometries. They all use grids that are tensor-products of one- and two-dimensional meshes, which leaves little flexibility in modelling the third dimension.

This thesis describes the design and implementation of **Second**, a general-purpose, 3d semiconductor device simulator. **Second** solves the traditional drift-diffusion equations of the semiconductor. The partial differential equations are discretized with the box method on a general 3d mesh consisting of a mixture of tetrahedra, quadrilateral pyramids, triangular prisms, and parallel epipeds. The one dimensional Scharfetter-Gummel scheme is used for integrating the current relations along grid edges. Decoupled (Gummel) and coupled (Newton) methods are implementeded for linearizing the discrete equations. Iterative methods (preconditioned conjugate-gradient type algorithms) are used for the solution of the linear systems. A time discretization with automatic time step control, based on an estimate of the local truncation error, is used for transient simulations. Physical models implemented include doping and field dependent carrier mobilities, surface scattering, band gap narrowing, and generation and recombination models with doping dependent carrier life times.

The flexibility of **Second** is demonstrated on a few case studies. One is an investigation of parasitic MOSFETs in a trench isolated sub-micron n-MOS device. This study demonstrates how design rules may be drawn up based on the results of 3d device simulations. A second example investigates latchup in CMOS devices and contains a comparison between two- and three-dimensional simulation results. A third case is a study of the switching behaviour of a bipolar transistor.

Zusammenfassung

Seit den frühen Arbeiten von Gummel in den sechziger Jahren hat sich die numerische Simulation von Halbleiterbauelementen zu einem unverzichtbaren Werkzeug für den Entwurf neuer Bauelemente entwickelt. Bisher waren die meisten Bauelementsimulationen ein- oder zweidimensional. Mit zunehmender Reduktion der Größe der Bauelemente wird jedoch eine echt dreidimensionale (3d) Behandlung der Halbleiterstrukturen notwendig.

Einige wenige 3d Bauelementsimulationsprogramme existieren seit den frühen achziger Jahren, ihr Anwendungsbereich ist jedoch durch die Tatsache beschränkt, daß sie keine Behandlung wirklich allgemeiner Geometrien erlauben. Dies ist vor allem darauf zurückzuführen, daß die von diesen Programme verwendeten Gitter Tensorprodukte ein- und zweidimensionaler Gitter sind, was nur eine wenig flexibile Modellierung der dritten Raumdimension erlaubt.

Diese Dissertation beschreibt den Entwurf und die Implementierung von **Second**, einem dreidimensionalen Bauelementsimulator mit breitem Anwendungsspektrum. **Second** basiert auf der numerischen Lösung der traditionellen Drift-Diffusionsgleichungen für Halbleiter. Diese partiellen Differentialgleichungen werden mittels der Box-Methode auf einem allgemeinen dreidimensionalen Gitter, bestehend aus Tetraedern, Vierecks-pyramiden, Dreiecksprismen und Parallelepipeden, diskretisiert. Für die Integration der Kantenströme wird das eindimensionale Scharfetter-Gummel-Verfahren benutzt. Zur Linearisierung der diskreten Gleichungen wurden entkoppelte (Gummel-) und gekoppelte (Newton-) Verfahren implementiert. Die resultierenden linearen Gleichungssysteme werden mit iterativen Verfahren, basierend auf der Methode der konjugierten Gradienten, gelöst. Die Zeitintegration verwendet eine automatische Schrittweitenkontrolle basierend auf einer Abschätzung des lokalen Diskretisierungsfehlers. Die implementierten physikalischen Modelle beinhalten dotierungs- und feldabhängige Beweglichkeiten der Ladungsträger, Oberflächenstreuung, Bandlückenverengung, sowie Erzeugungs- und Rekombinationsmodelle mit dotierungsabhängigen Lebensdauern.

Die vielfältige Verwendbarkeit von **Second** wird anhand einiger Fallstudien demonstriert: Eine Untersuchung parasitärer MOSFET-Elemente in einem n-MOS-Transistor zeigt, wie aufgrund von Simulationsergebnissen Designregeln für integrierte Schaltungen aufgestellt werden können. Als weiteres Beispiel dient eine Studie von Latchup-Effekten in CMOS-Strukturen, die auch einen Vergleich zwischen zwei- und dreidimensionalen Simulationsergebnissen präsentiert. Als letzter Fall wird das Schaltverhalten eines Bipolartransistors untersucht.

Contents

Abstract vii

Zusammenfassung ix

0 Introduction 1

1 The Semiconductor Modelling Problem 5

- 1.1 The Semiconductor Equations 5
- 1.2 Boundary Conditions 8
  - 1.2.1 External Boundaries 8
  - 1.2.2 Internal Boundaries 11
- 1.3 Scaling 11
- 1.4 Physical Models 14
  - 1.4.1 Intrinsic Concentration and Effective Intrinsic Concentration 14
  - 1.4.2 Mobility 14
  - 1.4.3 Recombination and Generation 15

2 Numerical Solution of the Semiconductor Equations 19

2.1 Spatial Discretization of the Differential Equations . . . 19

2.1.1 The Box Discretization Method 20

2.1.2 Box Discretization of the Semiconductor Equations 21

2.1.3 Limitations of the box method 23

2.1.4 Other Spatial Discretization Methods 24

2.2 Time Discretization 26

2.3 Non-linear Equation Solution 28

2.3.1 Damped Newton Iteration 28

2.3.2 Coupled and Decoupled Solution 30

2.3.3 Choice of Variables 31

2.4 Sparse Linear Systems 32

3 Semiconductor Device Modelling in Three Dimensions 35

3.1 Previous Work 35

3.2 What Makes 3D Harder Than 2D? 37

3.2.1 Computational Complexity 37

3.2.2 Numerical Aspects 38

3.2.3 Geometry Definition 39

3.2.4 Grid Generation 39

3.2.5 Visualization of Results 40

3.3 Our Approach 40

4 Methods 43

4.1 Assembly 43

4.1.1 Poisson's Equation 43

4.1.2 Continuity Equations 44

4.2 Treatment of Boundary Conditions 46

4.2.1 Dirichlet Boundaries 46

4.2.2 Neumann Boundaries 48

4.2.3 Internal Boundaries 48

4.3 Terminal Currents 49

4.4 Electric Field 50

4.5 Initial Solution 53

4.6 Stopping Criteria 54

4.6.1 Non-linear Iterations 54

4.6.2 Linear Iterations 55

4.6.3 Transient Simulations 56

5 Implementation **57**

5.1 Software Engineering Aspects 57

5.1.1 Hardware and Software Environment 57

5.1.2 Drawbacks of FORTRAN 58

5.1.3 Further Complications 64

5.1.4 Preprocessing 64

5.2 Description of the Implementation 66

5.2.1 Modules and Files 66

5.2.2 Macros for Portability and C/C++ Interface . . . 68

5.2.3 Libraries and Tools 73

5.2.4 Program Structure 80

5.2.5 Data Structures 84
5.2.6 Algorithms 89
5.2.7 Availability and Portability 95
5.3 Integration Into a Simulation System 96

6 Results 99

6.1 Parasitic MOSFETs 99
6.2 CMOS Latchup 103
6.3 Transistor Switching 106

7 Conclusions and Future Work 111

List of Figures 115

List of Programs 117

List of Tables 119

Bibliography 121

Index 129

Curriculum Vitae 135

0

Introduction

Since the invention of the transistor forty-four years ago, solid state electronics has developed with a breathtaking pace, and has irreversibly transformed technology. The computer revolution, only possible with VLSI, is still at its beginning and has the potential to even more significant changes. While it may certainly be argued whether the social impacts of these changes is generally for the better or worse, many problems in contemporary society can only be solved with the use of microelectronics, even some of the problems created by this progress. Examples include the use of sophisticated control logic to improve energy efficiency of such diverse objects as cars, trains and wind turbines, or better understanding of environmental processes due to more realistic numerical modelling with faster computers.

The first integrated circuits, which became commercially available in the early 1960s, contained only a few devices. In the year 1990, DRAM chips containing more than four million devices could be bought in the store, and chips with sixteen times that number have already been fabricated in the laboratory. The numbers go up by a factor of four every two to three years. Quantitatively speaking, this rate of progress is unrivaled in the history of mankind.[1]

The increasing packing density of VLSI chips implies shrinking device

---

[1] The increase in computer power, which occurs at roughly the same rate, is but a result of the advances in microelectronics.

dimensions. Reduced feature size, on the other hand, requires more complicated, and time consuming, manufacturing processes. This means that a pure \trial-and-error" approach to device optimization becomes impossible since it is both too time consuming and too expensive. Simulation has therefore become an indispensable tool for the device engineer. Besides offering the possibility to \test" hypothetical devices that have not (or could not) yet been manufactured, simulation offers unique insight into device behaviour by allowing the \observation" of entities that cannot be measured on real devices.

The first one dimensional (1d) device simulations were performed by Gummel [37] in 1964, based on the partial differential equations (PDEs) of the semiconductor proposed by van Roosbroeck [61]. Soon after, two dimensional (2d) simulations were performed, and during the 1970s, 2d simulation developed into a standard tool for device design.

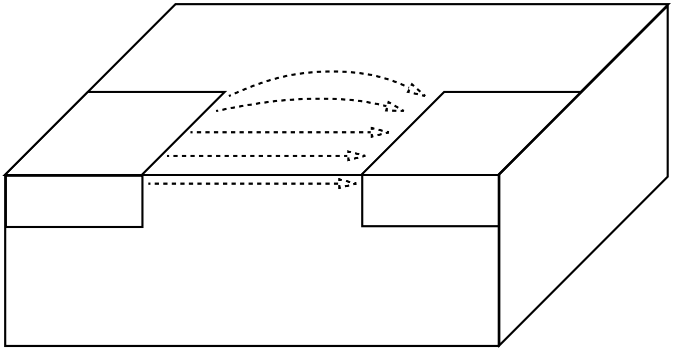

Figure 0.1: *Spreading of current at device edges causes 3d effects.*

2D treatment of semiconductor devices becomes unrealistic once current flow is no longer predominantly limited to a plane. The first source of such non-2d behaviour are edge effects. In a MOSFET, for example, carrier flow is two dimensional in the interiour. Near the sides of the channel, however, the current spreads outside the region between source and drain (Figure 0.1). This effect can be neglected if the transistor is wide enough so that the edge currents do not matter; with shrinking device dimensions this is no longer the case and 2d modelling can no longer be accurate.

A second class of 3d effects incorporates various kinds of device cross talk. A MOSFET that is isolated by an oxide trench may suffer from leakage currents due to parasitic devices that can be partially turned on under certain conditions. Operation of such devices is usually inherently 3d and cannot be modelled in two dimensions. Latchup effects in CMOS structures are impossible to model two dimensionally if the tubs are not arranged in line. Leakage currents in DRAM trench cells are 3d effects since the the trenches as they are used in 4 Mbit and 16 Mbit chips are too small to be reasonably modelled in 2d [11].

Finally, CMOS latchup or DRAM upset caused by ionizing radiation (e.g. natural α activity) can only be modelled in 3d due to the small diameter of the ionized channel [72].

The necessity to model such effects lead to the first 3d device simulators in 1980 [14, 84]. However, while general-purpose simulators are available for 2d problems, most of the currently available 3d device simulators can only model a small class of unrealistic devices, and more general ones [17] are still significantly limited in the generality of device geometries they can handle.

The aim of this thesis is the construction of a 3d device simulation program that is general enough to simulate arbitrary device structures under general operating conditions, including transient analysis. In order to be generally useful, the program must also be fast, \reasonable" in its memory requirements, and user friendly. On the other side we restrict ourselves to conventional physical models as they have been used in most device simulators so far. Provided a sufficiently general design, more sophisticated models can be added at a later stage.

This work is structured as follows: Chapter 1 will present the basic physical problem that must be solved by a device simulator. We restrict ourselves to the traditional drift-diffusion formulation of the semiconductor equations. Chapter 2 discusses how this problem can be solved numerically on a digital computer. We introduce methods for discretizing the PDEs, focusing on the box method which has turned out to be the most successful discretization scheme in device simulation.

In Chapter 3 the state of the art in 3d device simulation is examined and problems that are particular to 3d are discussed. Our approach to model general 3d device geometries is outlined. Chapter 4 describes the basic mathematical and computational methods we use in our simulator in more detail. Chapter 5 describes the actual software implementation of the device simulator **Second**, starting with an assessment of software engineering problems in the \real world" of scientific computing, and presenting solutions to these problems. The basic algorithms and data structures used in **Second** are then described.

The usability and flexibility of **Second** is demonstrated in Chapter 6 by means of actual simulations performed on a set of very different problems. Chapter 7 concludes the thesis with an outlook on further work that can be done to enhance **Second**'s usefulness.

1

The Semiconductor Modelling Problem

In this chapter we present the basic problem to be solved in device simulation. The first section introduces the partial differential equations (PDEs) used to describe the behaviour of a semiconductor. Section 1.2 discusses the boundary conditions for which the equations are to be solved. Section 1.3 shows how the PDEs are scaled for numerical treatment and Section 1.4 contains a discussion of the physical models used.

1.1 The Semiconductor Equations

A semiconductor is usually modelled as a medium with two kinds of mobile carriers of charge: *electrons* carrying a charge $-q$ and *holes* carrying a positive charge of the same magnitude. In addition there are impurities, positively charged donors and negatively charged acceptors. These are immobile, however a donor can recombine with an electron, or an acceptor can recombine with a hole, to form an electrically neutral impurity. We assume in the following that all impurities are ionized in the temperature ranges of interest to us (i.e. around room temperature).

Classical electrodynamics (see e.g. [45]) relates the electric field $\boldsymbol{E}$ to the charge density ϱ by *Poisson's equation*

$$\nabla \cdot \varepsilon\varepsilon_0 \boldsymbol{E} = \varrho, \tag{1.1}$$

where ε is the dielectric constant of the material and ε_0 the permittivity of vacuum. Note that ε may vary throughout the device but is assumed to be independent of time or bias conditions. In the case of a semiconductor we have

$$\varrho = q(p - n + N_d^+ - N_a^-), \tag{1.2}$$

where n and p denote the concentrations (*densities*) of electrons and holes respectively, and N_d^+ and N_a^- are the concentrations of ionized donors and acceptors. If we express the electric field by the electrostatic potential $\boldsymbol{E} = -\nabla\psi$ and write $N := N_d^+ - N_a^-$ for the net impurity concentration, Poisson's equation for the semiconductor becomes

$$-\nabla \cdot \varepsilon\varepsilon_0 \nabla\psi \; - q(p - n + N) = 0. \tag{1.3}$$

Conservation of charge is expressed by the *continuity equation*

$$\nabla \cdot \boldsymbol{J} + \frac{\partial\varrho}{\partial t} = 0 \tag{1.4}$$

for the current density $\boldsymbol{J}$, where t is the time. In our two carrier model of the semiconductor, charge conservation applies individually to the two carrier types except for recombination processes. Therefore we obtain separate continuity equations for both carriers:

$$\begin{aligned} -\nabla \cdot \boldsymbol{J}^n + q(R + \frac{\partial n}{\partial t}) &= 0, && (1.5) \\ \nabla \cdot \boldsymbol{J}^p + q(R + \frac{\partial p}{\partial t}) &= 0. && (1.6) \end{aligned}$$

Here $\boldsymbol{J}^n$ and $\boldsymbol{J}^p$ are the electron and hole current density, and R is the net recombination rate, i.e. the rate at which carriers vanish due to recombination processes. Pair generation of carriers gives a negative contribution to R.

In addition to the conduction currents there is the displacement current $\boldsymbol{J}^d = \varepsilon\varepsilon_0\dot{\boldsymbol{E}}$. These currents add up to the total current:

$$\boldsymbol{J}^t = \boldsymbol{J}^d + \boldsymbol{J}^n + \boldsymbol{J}^p. \tag{1.7}$$

Taking the time derivative of Poisson's equation, we find the relation

$$\nabla \cdot \boldsymbol{J}^d = \nabla \cdot \frac{\partial}{\partial t}\varepsilon\varepsilon_0\boldsymbol{E} = \frac{\partial}{\partial t}q(p - n + N) = q(\dot{p} - \dot{n}). \tag{1.8}$$

Together with (1.5), (1.6) this yields

$$\boldsymbol{\nabla} \cdot \boldsymbol{J}^t = 0 \tag{1.9}$$

in accordance with Maxwell's second equation [45].

In the drift-diffusion approximation usually employed in device simulation, the current is assumed to be composed of a drift part, driven by the electric field, and a diffusion part, driven by the concentration gradient:

$$\begin{aligned} \boldsymbol{J}^n &= -q\mu^n n \boldsymbol{\nabla}\psi + qD^n \boldsymbol{\nabla} n, && (1.10) \\ \boldsymbol{J}^p &= -q\mu^p p \boldsymbol{\nabla}\psi - qD^p \boldsymbol{\nabla} p. && (1.11) \end{aligned}$$

Here μ^n, μ^p are the mobilities while D^n, D^p are the diffusion coefficients for electrons and holes respectively. These are related by the Einstein relation

$$D = \mu \frac{kT}{q}, \tag{1.12}$$

where k denotes the Boltzmann constant and T the temperature.

If Boltzmann statistics is applicable and the semiconductor is in thermal equilibrium, the densities can be described by the Fermi level E_F as

$$\begin{aligned} n &= n_i \exp \frac{q\psi - E_F}{kT}, && (1.13) \\ p &= n_i \exp \frac{E_F - q\psi}{kT}, && (1.14) \end{aligned}$$

where n_i is the intrinsic concentration, which has the property

$$n_i^2 = np. \tag{1.15}$$

Away from equilibrium the above equations no longer hold, but we can still write the densities as

$$\begin{aligned} n &= n_i \exp \frac{q(\psi - \phi^n)}{kT}, && (1.16) \\ p &= n_i \exp \frac{q(\phi^p - \psi)}{kT}, && (1.17) \end{aligned}$$

where ϕ^n and ϕ^p are the *quasi-Fermi potentials* (also called *imrefs*). These are the driving forces of the particle currents, as can be seen by using Eqs. (1.16) and (1.17) to rewrite (1.10), yielding

$$\begin{aligned} \boldsymbol{J}^n &= -q\mu^n n \boldsymbol{\nabla}\phi^n, && (1.18) \\ \boldsymbol{J}^p &= -q\mu^p p \boldsymbol{\nabla}\phi^p, . && (1.19) \end{aligned}$$

In thermal equilibrium the quasi-Fermi potentials become equal to the Fermi potential $\phi^F := E_F/q$ (cf. [31]).

Deviations from Boltzmann statistics due to degeneracies are usually treated by replacing the intrinsic concentration by an *effective intrinsic concentration*

$$n_{ie} = n_i \exp \frac{\Delta E_g}{2kT}, \tag{1.20}$$

where ΔE_g represents the *bandgap narrowing*. Eqs (1.16) and (1.17) now read

$$\begin{aligned} n &= n_{ie} \exp \frac{q(\psi - \phi^n)}{kT}, & (1.21)\\ p &= n_{ie} \exp \frac{q(\phi^p - \psi)}{kT}, & (1.22) \end{aligned}$$

and in equilibrium, where $\phi^n = \phi^p = E_F$,

$$n_{ie}^2 = np \tag{1.23}$$

holds. With the introduction of the effective intrinsic concentration Eqs (1.18) and (1.19) remain the same, while (1.10) and (1.11) must be replaced by

$$\begin{aligned} \boldsymbol{J}^n &= -q\mu^n n \boldsymbol{\nabla}\psi + qD^n \boldsymbol{\nabla} n - kT\mu^n \boldsymbol{\nabla} \ln n_{ie} \\ &= -q\mu^n n \boldsymbol{\nabla}(\psi + \frac{\Delta E_g}{2q}) + kT\mu^n \boldsymbol{\nabla} n, & (1.24)\\ \boldsymbol{J}^p &= -q\mu^p p \boldsymbol{\nabla}\psi - qD^p \boldsymbol{\nabla} p + kT\mu^n \boldsymbol{\nabla} \ln n_{ie} \\ &= -q\mu^p p \boldsymbol{\nabla}(\psi - \frac{\Delta E_g}{2q}) - kT\mu^p \boldsymbol{\nabla} p. & (1.25) \end{aligned}$$

These equations have precisely the same form as (1.10) and (1.11) if we replace the electrostatic potential by an *effective potential* $\psi^n = \psi + \Delta E_g/2q$ for electrons and $\psi^p = \psi - \Delta E_g/2q$ for holes.

1.2 Boundary Conditions

1.2.1 External Boundaries

In order to solve the device equations presented in the previous section, we have to specify appropriate boundary conditions. The boundary for a

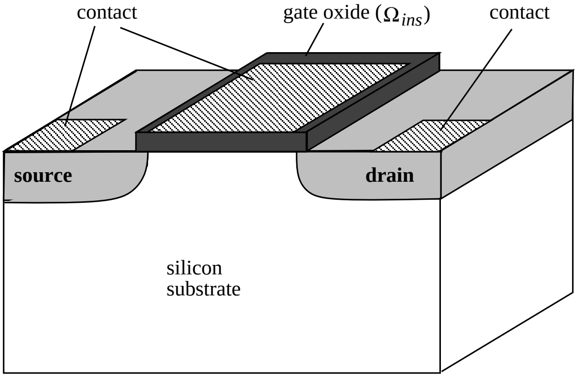

Figure 1.1: *Simple MOSFET structure showing silicon,* Ω_{semi}*, oxide,* Ω_{ins}*, and contacts,* Γ_0

device to be simulated (a simple example is shown in Fig. 1.1) consists of two parts: *contacts* and *free* boundary. We denote the whole domain as Ω, the i-th contact as Γ_i and the remaining boundary as Γ_h:

$$\partial\Omega = \Gamma_h \cup \Gamma_0, \tag{1.26}$$

where

$$\Gamma_0 := \bigcup_{i=1}^{N_C} \Gamma_i, \tag{1.27}$$

N_C being the number of contacts. Contacts are sources and sinks of carriers while no carriers are allowed to cross the free boundaries. This latter condition means that the current densities normal to the boundary must be zero,

$$\boldsymbol{n} \cdot \boldsymbol{J}^n = \boldsymbol{n} \cdot \boldsymbol{J}^p = 0 \text{ on } \Gamma_h, \tag{1.28}$$

where $\boldsymbol{n}$ is the outward unit normal vector of the boundary. Because of (1.18) and (1.19) this implies that the gradients of the quasi-Fermi levels must vanish in the direction normal to the boundary. Under the condition of no surface charge, we impose the same condition on the electrostatic potential [45], so that we obtain on Γ_h a set of *Neumann* boundary conditions

$$\boldsymbol{n} \cdot \boldsymbol{\nabla}\phi^n = \boldsymbol{n} \cdot \boldsymbol{\nabla}\phi^p = \boldsymbol{n} \cdot \boldsymbol{\nabla}\psi = 0 \text{ on } \Gamma_h. \tag{1.29}$$

Because of (1.21) and (1.22) this implies

$$\boldsymbol{n} \cdot \boldsymbol{\nabla}n = \boldsymbol{n} \cdot \boldsymbol{\nabla}p = 0 \text{ on } \Gamma_h, \tag{1.30}$$

provided that $\boldsymbol{n} \cdot \boldsymbol{\nabla} n_{ie}$ is also zero on Γ_h.

At contacts we require charge neutrality

$$\varrho = q(p - n + N) = 0 \tag{1.31}$$

and local thermal equilibrium (Eq. 1.23). The latter condition, because of (1.21), (1.22), (1.13) and (1.14), means that the quasi-Fermi potentials become equal to the Fermi potential

$$\phi^n = \phi^p = \frac{E_F}{q} = \phi^F \text{ on } \Gamma_0. \tag{1.32}$$

Eq. (1.31), together with (1.21), (1.22) and (1.32), determines the electrostatic potential as a function of the Fermi potential

$$\psi = \phi^F + \sinh \frac{N}{2n_{ie}} \text{ on } \Gamma_0. \tag{1.33}$$

The Fermi potential, however, must be equal to the applied voltage in an ohmic contact [71], so that we get a *Dirichlet* condition on contacts:

$$\psi - U_{bi} = \phi^n = \phi^p = U_i \text{ on } \Gamma_i. \tag{1.34}$$

Here U_i (*applied voltage"*) is the potential applied to contact i. The voltage

$$U_{bi} := \sinh \frac{N}{2n_{ie}} \tag{1.35}$$

is called the *built-in voltage*.

It must be noted that these boundary conditions are sensible only if they do not influence the physical behaviour of the device. This normally means that the boundaries must be sufficiently far away from physically active parts of the device, such as space charge regions or regions where the impurity concentration changes appreciately in the direction normal to the boundary.

There is one exception to that rule: The nature of our boundary conditions forces boundaries to behave like a symmetry plane between the simulated device and a \virtual" device whose geometry and physical composition is a mirror image of the \real" device. One can take advantage of this fact when simulating devices with a symmetry plane: Only one half of the device must be simulated and the simulated current densities will be exactly the same as if the whole device had been simulated.

1.2.2 Internal Boundaries

Besides the external boundaries of the simulation domain Ω, there also exist internal boundaries (or *interfaces*) between different materials. In the case of silicon devices the only kind of interface of interest is between insulator (oxide) and semiconductor (silicon).

In the insulator, Ω_{ins}, we assume that there are no charges, neither mobile nor immobile. This means that Poisson's equation is reduced to Laplace's equation

$$-\boldsymbol{\nabla}\cdot\varepsilon_{ins}\varepsilon_0\boldsymbol{\nabla}\psi = 0 \text{ in } \Omega_{ins}, \tag{1.36}$$

where ε_{ins} is the dielectric constant of the insulator. In the absence of surface charges, the interface condition is simply the continuity of the electric displacement $\varepsilon\varepsilon_0\boldsymbol{\nabla}\psi$ [45]. Hence the normal component of the electric field is discontinuous:

$$\boldsymbol{n}\cdot\varepsilon_{semi}\boldsymbol{\nabla}\psi|_{\Omega_{semi}} = \boldsymbol{n}\cdot\varepsilon_{ins}\boldsymbol{\nabla}\psi|_{\Omega_{ins}} \text{ on } \partial\Omega_{semi}\cap\partial\Omega_{ins}, \tag{1.37}$$

where ε_{semi} is the dielectric constant in the semiconductor region Ω_{semi}.

Since we do not allow carriers in the insulator, there is no current across the interface and the same Neumann boundary conditions (1.29) apply for the quasi-Fermi levels (or the densities) as in the case of external boundaries.

Sometimes we want to apply Dirichlet boundary conditions in the interior of the device, e.g. if we want to simulate a device with a metallic contact (of zero thickness) that lies between semiconductor and oxide. This is really a limiting case of a simulation domain with a hole, where the hole is part of the Dirichlet boundary. Hence we treat such \internal" contacts as part of the external boundary Γ_0.

1.3 Scaling

When numerically simulating physical phenomena it is customary to scale the physical entities. This has several reasons. The most important one is to shift the order of magnitude of the variables as closely to unity as possible, to avoid problems with the finite numeric range of digital computers. Other reasons include the \scaling away" of constants to simplify formulae. Usually the scaled entities become dimensionless.

For our simulations we use the scaling proposed by de Mari [50]. As scaling factors we define: the *intrinsic Debye length* l_i, the intrinsic concentration n_i, the *thermal voltage* U_T and a *unit diffusivity* D_0 for lengths, voltages, concentrations and diffusion coefficients respectively. These are defined as

$$
\begin{aligned}
l_i &:= \left(\frac{\varepsilon_{Si}\varepsilon_0 kT}{q^2 n_i}\right)^{\frac{1}{2}}, && (1.38)\\
U_T &:= \frac{kT}{q}, && (1.39)\\
D_0 &:= 1\,\mathrm{m}^2\mathrm{s}^{-1}, && (1.40)
\end{aligned}
$$

while for n_i a phenomenological formula for the temperature dependence is used (see Section 1.4.1).

Quantity		**Scaling factor**	
Name	**Symbol**	**Symbol**	**Value**
Displacement	$\boldsymbol{x}$	l_i	3.3865×10^{-5} m
Concentration	n, p, N	n_i	1.4824×10^{16} m^{-3}
Current density	$\boldsymbol{J}$	$qn_iD_0l_i^{-1}$	7.0135×10^{1} Am^{-2}
Voltage	U, ψ, ϕ	U_T	2.5852×10^{-2} V
Electric field	$\boldsymbol{E}$	$U_Tl_i^{-1}$	7.6339×10^{2} Vm^{-1}
Time	t	s	1.1468×10^{-9} s
Current	I	$qn_il_iD_0$	8.0434×10^{-8} A
Mobility	μ	$D_0U_T^{-1}$	3.8681×10^{1} $\mathrm{m}^2\mathrm{V}^{-1}\mathrm{s}^{-1}$
Recombination rate	R	$n_iD_0l_i^{-2}$	1.2926×10^{25} $\mathrm{m}^{-3}\mathrm{s}^{-1}$

Table 1.1: *De Mari scaling factors for* $T = 300\,K$.

From these definitions the scaling factors for all other relevant entities can be derived. The various scaling factors and their values are summarized in Table 1.1. Table 1.2 summarizes the fundamental and material constants used for determining the normalization factors.

We use the symbols u, v, and w for the scaled potentials ψ, ϕ^n and ϕ^p. For all other quantities we use the same symbols irrespective on whether or not they are scaled. Usually the potential variables are sufficient to indicate that an equation assumes scaled quantities.

Quantity	Symbol	Value
Universal constants		
Elementary charge	q	$1.602\,177\,33 \times 10^{-19}$ C
Boltzmann constant	k	$1.380\,658 \times 10^{-23}$ JK^{-1}
Permittivity of vacuum	ε_0	$8.854\,187\,818 \times 10^{-12}$ Fm^{-1}
Material constants		
Dielectricity of *Si*	ε_{Si}	11.9
Dielectricity of SiO_2	ε_{SiO_2}	3.9

Table 1.2: *Fundamental constants (after Cohen and Taylor [20]) and material constants (after Sze [71])*

The scaled semiconductor equations for a silicon device now read

$$\begin{aligned} -\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} u - (p - n + N) &= 0, & (1.41)\\ -\boldsymbol{\nabla} \cdot \boldsymbol{J}^n + R + \frac{\partial n}{\partial t} &= 0, & (1.42)\\ \boldsymbol{\nabla} \cdot \boldsymbol{J}^p + R + \frac{\partial p}{\partial t} &= 0, & (1.43) \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{J}^n &= -\mu^n(n\boldsymbol{\nabla} u + \boldsymbol{\nabla} n) = -\mu^n n \boldsymbol{\nabla} v, & (1.44)\\ \boldsymbol{J}^p &= -\mu^p(p\boldsymbol{\nabla} u - \boldsymbol{\nabla} p) = -\mu^p p \boldsymbol{\nabla} w, & (1.45)\\ n &= n_{ie} e^{u-v}, & (1.46)\\ p &= n_{ie} e^{w-u} & (1.47) \end{aligned}$$

in the semiconductor, and

$$-\boldsymbol{\nabla} \cdot \frac{\varepsilon_{ox}}{\varepsilon_{semi}} \boldsymbol{\nabla} u = 0 \qquad (1.48)$$

in the oxide. The boundary conditions are

$$u - U_{bi} = v = w = U_i, \qquad (1.49)$$

on Dirichlet, and

$$\boldsymbol{n} \cdot \boldsymbol{\nabla} u = \boldsymbol{n} \cdot \boldsymbol{\nabla} v = \boldsymbol{n} \cdot \boldsymbol{\nabla} w = \boldsymbol{n} \cdot \boldsymbol{\nabla} n = \boldsymbol{n} \cdot \boldsymbol{\nabla} p = 0 \qquad (1.50)$$

on Neumann boundaries.

1.4 Physical Models

In this section we present the models used for the quantities n_i, n_{ie}, μ, and R.

1.4.1 Intrinsic Concentration and Effective Intrinsic Concentration

The intrinsic concentration in silicon is given in [10], based on measurements over a temperature range of 250{500 K, as

$$n_i = 3.87 \times 10^{22} (T/\mathrm{K})^{1.5} \exp(-7000\,\mathrm{K}/T)\,\mathrm{m}^{-3}. \qquad (1.51)$$

For a temperature of 300 K this gives the value of $1.4824 \times 10^{16}\,\mathrm{m}^{-3}$ in Table 1.1.

The effective intrinsic concentration is given by Eq. (1.20) as a function of the bandgap narrowing. Bandgap narrowing is a phenomenological way to incorporate deviations from Boltzmann statistics due to heavy doping and quantum effects. Therefore bandgap narrowing is an approximate correction that lumps several different physical phenomena together into a single parameter. Since this is only a coarse approximation of the actual device physics, it is not surprising that more than one bandgap narrowing model exists.

In **Second** we use the model after Slotboom [67]:

$$\Delta E_g = q\,0.009\,V \left[\ln\left(\frac{|N|}{10^{23}\,\mathrm{m}^{-3}}\right) + \sqrt{\ln\left(\frac{|N|}{10^{23}\,\mathrm{m}^{-3}}\right) + \frac{1}{2}} \right], \qquad (1.52)$$

or alternatively the one after Gaur et al. [34]:

$$\Delta E_g = 2kT \begin{cases} 9.248 \cdot 10^{-10} (|N|/\mathrm{cm}^3)^{0.4678} & \text{for } |N| < 5 \cdot 10^{19}/\mathrm{cm}^3 \\ 1.52 & \text{otherwise} \end{cases} . \qquad (1.53)$$

1.4.2 Mobility

The carrier mobilities in doped semiconductors are reduced from their intrinsic values, μ_0, due to scattering at impurities, leading to doping dependent mobilities $\mu(N)$. Caughey and Thomas [16] fitted experimental data to the formula

$$\mu(N) = \mu_0 + \frac{\mu_1}{1 + N/N_r}. \tag{1.54}$$

An electric field does not accelerate the carriers to arbitrary velocities due to *velocity saturation*. We model this after Scharfetter and Gummel [64] with a mobility depending on the electric field as

$$\mu(N, E_{\|}) = \mu(N) \left\{ 1 + \left[\frac{\mu(N) E_{\|}}{v_s} \right]^2 + \frac{\left[\mu(N) E_{\|} / v_c \right]^2}{\mu(N) E_{\|} / v_c + G} \right\}^{-1/2}, \tag{1.55}$$

where $E_{\|}$ is the component of the electric field parallel to the carrier current. However, since the carrier currents are driven by the gradients of the quasi-Fermi potentials (see Eqs. (1.18), (1.19)), it is preferable to use the magnitudes of these gradients as the parallel electric field:

$$E_{\|}^{n} = |\nabla \phi^{n}|, \tag{1.56}$$

$$E_{\|}^{p} = |\nabla \phi^{p}|. \tag{1.57}$$

In MOSFETs, where high current densities flow along insulator interfaces, surface scattering effects become important. Yamaguchi [83] proposed to model these as a function of the transverse electric field as

$$\mu(N, E_{\|}, E_{\perp}) = \mu(N, E_{\|}) \left[1 + \left(\frac{E_{\perp}}{E_c} \right)^c \right]^{-1}, \tag{1.58}$$

where $E_{\perp}$ is the component of the electric field orthogonal to the direction of the current flow. Table 1.3 summarizes the various constants used in Eqs. (1.54), (1.55), and (1.58).

1.4.3 Recombination and Generation

Recombination is a phenomenon that works towards restoring equilibrium (Eq. 1.23) under conditions where an excess of carriers exists. In case of carrier depletion ($np < n_{ie}^2$), the same processes lead to an increase of carrier concentrations, i.e. generation. However, this generation is normally not significant so that the processes involved are generally called \recombination" even though they may actually produce carriers.

quantity	electrons	holes	units
μ_0	0.00880	0.00543	$\mathrm{m^2V^{-1}s^{-1}}$
μ_1	0.125 20	0.040 73	$\mathrm{m^2V^{-1}s^{-1}}$
N_r	143.2	267.0	$10^{21}\mathrm{m^{-3}}$
v_s	100.0	83.7	10^3 $\mathrm{ms^{-1}}$
v_c	49	49	10^3 $\mathrm{ms^{-1}}$
E_c	30.32	15.30	10^6 $\mathrm{Vm^{-1}}$
c	0.657	0.617	
G	8.8	1.6	

Table 1.3: *Mobility parameters for electrons and holes*

The most important recombination processes in silicon are the *Auger process* where an electron-hole pair recombines and the recombination energy is transferred to a third particle, and single level processes where carriers recombine via isolated trap levels in the band gap.

Auger recombination produces the recombination rate

$$R_{Auger} = (np - n_{ie}^2)(nA_{Aug}^n + pA_{Aug}^p), \tag{1.59}$$

where A_{Aug} are the *Auger coefficients*. These are usually considered constant with values of $A_{Aug}^n = 0.5\{2.8\times10^{-43}\mathrm{m^6s^{-1}}$ and $A_{Aug}^p = 0.99\times10^{-43}\mathrm{m^6s^{-1}}$ (according to Pinto [57]).

Single trap level recombination is usually treated according to the *Shockley-Read-Hall model* (cf. [71]) as

$$R_{SRH} = \frac{pn - n_{ie}^2}{\tau^p(n + n_{ie}) + \tau^n(p + n_{ie})}, \tag{1.60}$$

where τ^n and τ^p are the electron and hole *lifetimes* respectively. These are usually modelled using the formula

$$\tau = \frac{\tau_0}{1 + \left(\frac{N}{N_{SRH}}\right)^{G_{SRH}}}. \tag{1.61}$$

Table 1.4 gives typical values. It must be noted, however, that at least the values of τ_0 can vary significantly between different devices. Often

	τ_0 10^{-6} s	N_{SRH} 10^{21} m^{-3}	G_{SRH}
electrons	40	3.0	0.5
holes	8	3.0	0.5

Table 1.4: *Parameters for Shockley-Read-Hall recombination model*

recombination centers are deliberately inserted into a device to control the lifetime of the minority carriers (*lifetime engineering*).

The main generation process is *impact ionization*, also called *avalanche generation*. This phenomenon occurs when electric fields in a device are high enough to accelerate carriers to energies where collision with lattice atoms can ionize the latter. This three-particle process is the inverse of Auger recombination. The effect is modelled after Chynoweth [18] as

$$R_{av} = R_{av}^n + R_{av}^p, \tag{1.62}$$

$$R_{av}^n = -\frac{1}{q}|\boldsymbol{J}^n| A_{av}^n \exp(-E_{crit}^n / E_{\parallel}^n), \tag{1.63}$$

$$R_{av}^p = -\frac{1}{q}|\boldsymbol{J}^p| A_{av}^p \exp(-E_{crit}^p / E_{\parallel}^p). \tag{1.64}$$

where

$$E_{\parallel}^n := \frac{\boldsymbol{E} \cdot \boldsymbol{J}^n}{|\boldsymbol{J}^n|}, \tag{1.65}$$

$$E_{\parallel}^p := \frac{\boldsymbol{E} \cdot \boldsymbol{J}^p}{|\boldsymbol{J}^p|}. \tag{1.66}$$

The minus sign in Eq. (1.63) and (1.64) indicates generation. The values of the ionization coefficients as experimentally determined by Grant [36] are listed in Table 1.5.

	A_{av} $10^6\,\mathrm{m}^{-1}$	E_{crit} $10^6\,\mathrm{Vm}^{-1}$	range for $E_{\parallel}$ $10^6\,\mathrm{Vm}^{-1}$
electrons	260	143	< 24
	62	108	24{42
	50	99	> 42
holes	200	197	< 51
	56	132	> 51

Table 1.5: *Grant's coefficients for the impact ionization model after Chynoweth*

2

Numerical Solution of the Semiconductor Equations

Having presented the equations describing a semiconductor device, we will now discuss methods for their solution. Section 2.1 will present the method used for the spatial discretization, while the time discretization is discussed in Section 2.2. In Section 2.3 methods for the solution of the nonlinear equations arising from discretization are presented, and Section 2.4 finally discusses the solution of linear systems of equations.

2.1 Spatial Discretization of the Differential Equations

In order to solve the boundary value problem (1.41{1.50) on a digital computer, the PDEs must be discretized, i.e. transformed into a system of discrete equations. One method to do this is the *box method* (BM), first presented by Varga [77], which is also known as the *control volume* or *finite volume* method.

2.1.1 The Box Discretization Method

Let us assume a PDE in divergence form, the classical form of conservation laws in physics,

$$\boldsymbol{\nabla}\cdot\boldsymbol{F}(\boldsymbol{x})-S(\boldsymbol{x})=0, \tag{2.1}$$

for some vector field $\boldsymbol{F}$ and a scalar source term S in some domain Ω with a plain faced boundary $\partial\Omega$. Let Ω be covered by a grid consisting of N_v *vertices* $\boldsymbol{x}_i\in\Omega, i=1,\cdots,N_v$, connected by edges (see Figure 2.1). We construct for each vertex, i, a *box*, Ω_i, delimited by the mid-perpendiculars of all the edges terminating in vertex i. If the grid is constructed appropriately, the boxes will form a *partition* of Ω:

$$\Omega = \bigcup_{i=1}^{N_v}\Omega_i, \tag{2.2}$$

$$V = \sum_{i=1}^{N_v} V_i, \tag{2.3}$$

where $V:=\int_\Omega dV$ is the volume of the domain Ω and $V_i:=\int_{\Omega_i} dV$ the volume of Ω_i.

In order to obtain an equation for vertex i, we integrate (2.1) over Ω_i and apply Gauss's theorem, which yields

$$\int\limits_{\Omega_i}[\boldsymbol{\nabla}\cdot\boldsymbol{F}(\boldsymbol{x})-S(\boldsymbol{x})]dV=\int\limits_{\partial\Omega_i}\boldsymbol{F}(\boldsymbol{x})\cdot d\boldsymbol{n}(\boldsymbol{x})-\int\limits_{\Omega_i}S(\boldsymbol{x})dV=0, \tag{2.4}$$

where $d\boldsymbol{n}(\boldsymbol{x})$ denotes the unit vector normal to the box boundary $\partial\Omega_i$ in $\boldsymbol{x}$. We approximate $S(\boldsymbol{x})$ within Ω_i by its value $S_i:=S(\boldsymbol{x}_i)$ at the box center, and $\boldsymbol{F}(\boldsymbol{x})$ within each sector Ω_{ij} of the box by some average value $\boldsymbol{F}_{ij}$. The above equation then becomes

$$\sum_j\int\limits_{\partial\Omega_{ij}}\boldsymbol{F}_{ij}(\boldsymbol{x})\cdot d\boldsymbol{n}(\boldsymbol{x})-\int\limits_{\Omega_i}S_i dV=\sum_j F_{ij}A_{ij}-S_iV_i=0, \tag{2.5}$$

where $F_{ij}:=|\boldsymbol{F}_{ij}|\cos\angle(\boldsymbol{F}_{ij},d\boldsymbol{n})$ is the projection of $\boldsymbol{F}_{ij}$ onto the edge $\overline{ij}$, and the sums run over all *edge neighbours* of vertex i. Edge neighbours of i are all vertices j connected with i by an edge of the grid. A_{ij} is the area of

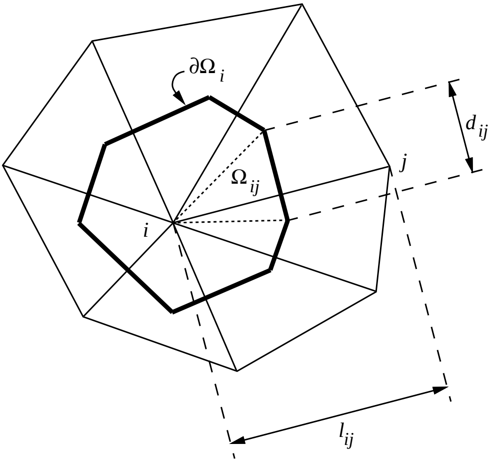

Figure 2.1: *2D example of a box*

the part of the box surface that is normal to $\overline{ij}$. If we define A_{ij} to be zero if vertices i and j are *not* edge neighbours, we can write Eq. (2.5) as

$$\sum_{\substack{j=1\\j\neq i}}^{N_v} F_{ij}A_{ij} - S_iV_i =: \sum_{j\neq i} F_{ij}A_{ij} - S_iV_i = 0. \tag{2.6}$$

This is the discretized form of Eq. (2.1): one discrete equation for each grid point i.

2.1.2 Box Discretization of the Semiconductor Equations

Poisson's equation

To apply the box method (BM) to Poisson's equation (1.41), we have to identify in Eq. (2.1) $\boldsymbol{F}$ with $\boldsymbol{E} = -\boldsymbol{\nabla} u$ and S with $\varrho = p - n + N$. This results in the discretization

$$\sum_{j\neq i} E_{ij}A_{ij} - V_i(p_i - n_i + N_i) = 0, \tag{2.7}$$

with the underlying approximation that $p = p_i := p(\boldsymbol{x}_i)$, $n = n_i := n(\boldsymbol{x}_i)$, $N = N_i := N(\boldsymbol{x}_i)$ are constant in Ω_i, and the projection E_{ij} of $\boldsymbol{E}$ onto the edge $\overline{ij}$ is constant. Under these conditions we obtain as the potential difference along the edge

$$u_{ji} = u_j - u_i = -\boldsymbol{E} \cdot (\boldsymbol{x}_j - \boldsymbol{x}_i), \tag{2.8}$$

and hence

$$-\frac{u_{ji}}{l_{ji}} = \boldsymbol{E} \cdot \frac{\boldsymbol{l}_{ji}}{|\boldsymbol{l}_{ji}|} = E_{ij}, \tag{2.9}$$

with $\boldsymbol{l}_{ji} := \boldsymbol{x}_j - \boldsymbol{x}_i$ and $l_{ij} := |\boldsymbol{l}_{ji}|$. Substituting this into Eq. (2.7) results in the final form of the BM discretization for Poisson's equation:

$$F_i^u := -\sum_{j\neq i} \frac{A_{ij}}{l_{ij}} u_{ij} - V_i(p_i - n_i + N_i) = 0, \tag{2.10}$$

where we have used $l_{ij} = l_{ji}$ and $u_{ij} = -u_{ji}$. Consequently, in the oxide, the discretization of Laplace's equation (1.48) reads

$$\sum_{j\neq i}\left(-\frac{\varepsilon_{ox}}{\varepsilon_{semi}} \frac{A_{ij}}{l_{ij}} u_{ij}\right) = 0. \tag{2.11}$$

At interfaces, the appropriate equation, Eq. (2.10) or Eq. (2.11), must be chosen separately for the semiconductor and the insulator part of the box.

Continuity equations

For the continuity equations (1.42,1.43), Eq. (2.6) translates into

$$\begin{aligned} -\sum_{j\neq i} A_{ij}J_{ij}^n + V_i(R_i + \dot{n}_i) &= 0, &(2.12)\\ \sum_{j\neq i} A_{ij}J_{ij}^p + V_i(R_i + \dot{p}_i) &= 0. &(2.13) \end{aligned}$$

To determine J^n_{ij}, the component of the electron current density along an edge, we use the 1d solution first derived by Scharfetter and Gummel [64]: We consider Eq. (1.44) on the edge $\overline{ij}$:

$$J^n_{ij} = \mu^n_{ij}(nE_{ij} - \frac{dn}{dl}). \tag{2.14}$$

Under the assumption that J^n_{ij}, μ^n_{ij}, and E_{ij} are constant on that edge, we can integrate the ordinary differential equation (ODE) (2.14) and obtain the solution

$$J^n_{ij} = \frac{\mu^n_{ij}}{l_{ij}}[n_j B(u_{ji}) - n_i B(u_{ij})], \tag{2.15}$$

with the *Bernoulli function*

$$B(x) = \frac{x}{e^x - 1}. \tag{2.16}$$

The hole continuity equation (1.43) can be treated in an analogous fashion, yielding a 1d hole current density of

$$J^p_{ij} = \frac{\mu^p_{ij}}{l_{ij}}[p_j B(u_{ij}) - p_i B(u_{ji})], \tag{2.17}$$

Substituting this into Eqs. (2.12,2.13) results in the discretized continuity equations

$$F^n_i := -\sum_{j \neq i} \frac{A_{ij}}{l_{ij}} \mu^n_{ij}[n_j B(u_{ji}) - n_i B(u_{ij})] + V_i(R_i + \dot{n}_i) = 0, \tag{2.18}$$

$$F^p_i := -\sum_{j \neq i} \frac{A_{ij}}{l_{ij}} \mu^p_{ij}[p_j B(u_{ij}) - p_i B(u_{ji})] + V_i(R_i + \dot{p}_i) = 0. \tag{2.19}$$

2.1.3 Limitations of the box method

In Section 2.1.1 we postulated that the boxes delimited by the mid-perpendiculars of the edges form a partition of the simulation domain. This imposes a serious restriction on the grid, the well-known *obtuse angle problem* of the BM (see e.g. Pinto [57]). In 2d the restriction is that the sum of opposite angles of adjacent triangles must not exceed $\pi/2$, and a similar characterization of a *well shaped* grid exists in 3d [21].

While in principle the BM allows the use of grids that can model general geometries, and allows good adjustment of the point density, it is quite difficult to construct irregular grids that are well shaped. Sophisticated grid generation algorithms are required for truly 3d grids (see Conti et al. [23]).

2.1.4 Other Spatial Discretization Methods

Finite differences

The simplest (and probably most straight-forward) method for solving a PDE is by *finite differences* (FD, for a detailed presentation see Smith [68]). This method is based on replacing differential operators by difference operators. For example the 2d Laplace equation

$$\frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} = 0 \tag{2.20}$$

is, at point $(x_i, y_j) = (ih, jh)$ of a uniform grid, replaced by the difference equation

$$\frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2} + \frac{f(y_{i-1}) - 2f(y_i) + f(y_{i+1})}{h^2} = 0. \tag{2.21}$$

If the grid is non-uniform, but still regular, a similar but significantly messier expression holds.

For the continuity equations this simple scheme is not useful, because the exponential variation of the carrier densities is poorly fitted by the linear approximation underlying FD. Higher order difference methods are possible but of not much help in this case. The 1d Scharfetter-Gummel solution of the current equations must therefore be used along the edges as in the case of the BM (cf. Section 2.1.2). On a rectangular mesh the BM is actually equivalent to FD, so that the former can be considered a generalization of the latter.

Standard FD requires a regular (though not necessarily uniform) grid consisting of points $(x_i, y_j, \ldots)$, $i = 1, \ldots, N_x$, $j = 1, \ldots, N_y, \ldots$ Since such d dimensional grids are the tensor product of d one-dimensional grids, they are often called *tensor product grids*.

The regularity of tensor product grids is reflected in a regularity of the structure of the sparse linear systems emerging from the discretization|they exhibit a simple band structure (d bands at either side of the main diagonal). This allows for the use of very simple sparse data structures and algorithms which can be implemented very efficiently with little effort, making FD methods very appealing from the implementation point of view.

The drawback of FD is the poor control one has over the point density of the grid|in order to have a sufficiently high point density in the physically active device regions one gets many more points than are actually needed in other regions, an effect that is drastically worse in 3d than in 2d. Since the number of grid points determine both, memory and CPU time requirements of a simulation, this seriously limits the utility of FD. Furthermore it is difficult to accurately model non-rectangular device features with purely rectangular grids.

One possibility to reduce the number of grid points is to allow terminating grid lines. The resulting variant of FD is often called *finite boxes*, see Franz et al. [33] for details. Terminating lines, however, immediately destroy the regularity of the structure of the resulting systems of linear equations, thus giving away the main advantage of the FD method. Moreover, as Pinto [57] has shown, severe limitations are posed on the aspect ratios of the boxes containing a termination node, seriously restricting the flexibility of varying the point density. It is therefore questionable whether finite boxes have any real advantage over the BM, and the method does not seem to be in widespread use.

Finite element methods

The *finite element method* (FEM), originally introduced for the numerical solution of problems in structural mechanics, has established itself in the last two or three decades as one of the most popular methods for solving PDEs.

The basic idea behind FEM is to replace a PDE by an equivalent *variational problem*. The domain Ω is partitioned into elements Ω_i, and a solution of the variational problem is then sought by solving it approximately within each element (see Strang and Fix [70] for details).

The advantage of the FEM is that no hard restrictions, comparable to the angle conditions of the BM, are imposed on the grid. (There do exist \soft" angle conditions in that the solution error increases with a decrease of the smallest angle [70], but this is far less restrictive than the angle condition in the BM.)

The disadvantage is that the interpolation functions used within the elements, which are usually linear or of low polynomial order, are unsuitable for the exponentially varying densities in the continuity equations. Various attempts to use exponential interpolation have apparently not been successful [65, 57]. Other approaches, like hybrid methods or upwinding schemes [13], have failed to provide solutions general enough to allow simulating devices under arbitrary operating conditions. They all suffer from truncation problems when potential differences across elements exceed a few U_T. This forces extremely high point densities when simulating reversely biased p-n-junctions, e.g. in MOSFETs.

Because of these problems, FEM based methods are not common in the field of device simulation. We are not aware of any general-purpose device simulator using FEM.

2.2 Time Discretization

The spatially discretized semiconductor equations (2.10, 2.11, 2.18, 2.19) can be written as

$$\boldsymbol{F}(\boldsymbol{z}(t)) = \dot{\boldsymbol{q}}(\boldsymbol{z}(t)) + \boldsymbol{f}(\boldsymbol{z}(t)) = 0, \tag{2.22}$$

where $\boldsymbol{f} = (\boldsymbol{f}^\nu), \nu = u, n, p$ stands for the terms arising from the spatial discretization of the stationary device equations ($\dot{n} = \dot{p} = 0$) and $\boldsymbol{F} = (\boldsymbol{F}^\nu)$ for the full (transient) equations,

$$\boldsymbol{z}(t) := \begin{pmatrix} \boldsymbol{u}(t) \\ \boldsymbol{v}(t) \\ \boldsymbol{w}(t) \end{pmatrix} \tag{2.23}$$

is the transient solution, with $\boldsymbol{u} = (u_i)$, $\boldsymbol{v} = (v_i)$, and $\boldsymbol{w} = (w_i)$, and

$$\boldsymbol{q}(t) := \begin{pmatrix} 0 \\ (V_i n_i) \\ (V_i p_i) \end{pmatrix}. \tag{2.24}$$

Various methods are known for integrating equations like (2.22), for example the *Euler* or *backward Euler methods* or the *trapezoidal rule* (TR) [35].

The problem is that Eq (2.22) is extremely stiff, i.e. the time constants vary over several orders of magnitude. For the usual one-step methods, which are typically used in conjunction with a (repeated) Richardson extrapolation, this results in an unacceptably small time step.

Another major concern is the stability of the algorithm. An often used criterion is *A stability* [35]: a one-step method

$$y_{n+1} = A(h\lambda) y_n, \tag{2.25}$$

where (hopefully) $y_n \approx y(nh)$, is one stable if, applied to the test problem

$$\frac{dy}{dt} = \lambda y \tag{2.26}$$

with $\mathrm{Re}\lambda < 0$, it satisfies the condition

$$|A(h\lambda)| < 1. \tag{2.27}$$

The second order TR is the A-stable multistep method with the smallest local truncation error [26]. However, A-stability is not sufficient for very stiff problems since it does not prevent oscillations in the computed solution unless the time step becomes very small. We therefore require the quadrature method to be *L stable* [47], where a method is said to be L stable if it is A stable and

$$|A(h\lambda)| \to 0 \text{ as } |h\lambda| \to 0. \tag{2.28}$$

This is the case for the method proposed by Bank et al. [7]: They use a time step composed of a TR step of length γh_n followed by a *second order backward differential formula* (BDF2) step of length $(1-\gamma)h_n$ to go from time t to $t_{n+1} := t_n + h_n$. For the TR step one has to solve

$$\boldsymbol{F}_{n+\gamma} := \boldsymbol{f}_{n+\gamma} + \boldsymbol{f}_n + \frac{2}{\gamma h_n}(\boldsymbol{q}_{n+\gamma} - \boldsymbol{q}_n) = 0 \tag{2.29}$$

and for the BDF2 step

$$\boldsymbol{F}_{n+1} := \boldsymbol{f}_{n+1} + \frac{2-\gamma}{(1-\gamma)h_n}\boldsymbol{q}_{n+1} - \frac{1}{\gamma(1-\gamma)h_n}\boldsymbol{q}_{n+\gamma} + \frac{1-\gamma}{\gamma h_n}\boldsymbol{q}_n = 0. \tag{2.30}$$

Here we have written $\boldsymbol{q}_n$ for $\boldsymbol{q}(\boldsymbol{z}(t_n))$ etc. It turns out that the optimal value of $\gamma = 2 - \sqrt{2}$ minimizes the *local truncation error* (LTE) of the composite scheme. The advantage of this method is that the composite scheme is second order, yet a one-step algorithm that does not need several previous time steps for (re)starting.

Bank et al. also propose a scheme for controlling the size of the time step based on an estimate of the LTE defined as

$$\boldsymbol{\tau} = 2Ch_n \left[\frac{\boldsymbol{f}_n}{\gamma} - \frac{\boldsymbol{f}_{n+\gamma}}{\gamma(1-\gamma)} + \frac{\boldsymbol{f}_{n+1}}{(1-\gamma)} \right], \tag{2.31}$$

where

$$C = \frac{-3\gamma^2 + 4\gamma - 2}{12(2-\gamma)}. \tag{2.32}$$

From the previous step size, h_n, a candidate step size, $\tilde{h}$, is determined as

$$\tilde{h} = h_n r^{-1/3}, \tag{2.33}$$

where

$$r^2 = \frac{1}{N} \sum_i \left(\frac{\tau_i}{e_i} \right)^2, \tag{2.34}$$

and

$$e_i = \epsilon_R |q_{n+1,i}| + \epsilon_A, \tag{2.35}$$

with the absolute and relative error parameters ϵ_R and ϵ_A. If $r \leq 5$ the time step is accepted and the scheme continues with the next step $h_{n+1} := \tilde{h}$, otherwise the step is rejected and repeated with $h_n := 0.9\tilde{h}$. If the nonlinear solver does not converge (within a given number of iterations) the time step is also rejected and repeated with $h_n := h_n/2$.

2.3 Non-linear Equation Solution

2.3.1 Damped Newton Iteration

The discretized equations are nonlinear and are linearized for numerical solution. The usual linearization procedure is the (quadratically convergent) *Newton method*. Given a nonlinear system of equations

$$\boldsymbol{F}(\boldsymbol{z}) = 0, \tag{2.36}$$

the Newton procedure iteratively computes a new solution

$$z_i^{k+1} := z_i^k + \delta z_i^k \tag{2.37}$$

from an old one $\boldsymbol{z}^k$, where the update $\boldsymbol{\delta z}^k$ is obtained as the solution of the linear system

$$\sum_j \frac{\partial F_i(\boldsymbol{z}^k)}{\partial z_j^k} \delta z_j^k = -F_i(\boldsymbol{z}^k). \tag{2.38}$$

This basic Newton procedure suffers from a phenomenon called *overshoot*: the update $\boldsymbol{\delta z}$ frequently overestimates (often by many orders of magnitude) the difference to the solution of (2.36). If such an excessive update is applied, the resulting intermediate solution may lie outside of the convergence region of the Newton procedure, or numerical problems (like exponent overflow) may prevent convergence.

To control this overshoot, *damping* is introduced: Eq. (2.37) is replaced by

$$z_i^{k+1} := z_i^k + s^k \delta z_i^k, \tag{2.39}$$

where a damping factor s^k, $0 < s^k \leq 1$ is introduced. The question remains how to determine that damping factor. Bank and Rose [8] showed that, under certain conditions, global and quadratic convergence is achieved if s^k satisfies the *sufficient decrease condition*

$$1 - \frac{\|\boldsymbol{F}^{k+1}\|}{\|\boldsymbol{F}^k\|} \geq \epsilon s^k, \tag{2.40}$$

where $\epsilon > 0$ is some fixed, small value, usually taken to be the machine epsilon. Note that this algorithm will still converge if some reasonable approximation is used instead of the exact Jacobian $\partial F_i(\boldsymbol{z})/\partial z_j$.

To determine a damping factor satisfying Eq. (2.40) without a large number of evaluations of $\boldsymbol{F}^{k+1}$, Coughran et al. [25] propose the following scheme: An initial damping factor s^{k+1} for a new step is determined from the last successful one as

$$s^{k+1} := \frac{s^k}{s^k + 0.2(1 - s^k)\|\boldsymbol{F}^{k+1}\|/\|\boldsymbol{F}^k\|}. \tag{2.41}$$

If this step does not satisfy (2.40), the following values are tried in turn:

$$s^{k+1} := s^k \left(\frac{\epsilon \|\boldsymbol{z}^k\|}{\boldsymbol{\delta z}} \right)^{j^2/l^2}, j = 1, \ldots, l. \tag{2.42}$$

2.3.2 Coupled and Decoupled Solution

For the stationary case ($\dot{n} = \dot{p} = 0$), the discretized equations (2.10, 2.18, 2.19) can be summarized as

$$F_i^u(\boldsymbol{u}, \boldsymbol{n}, \boldsymbol{p}) = 0, \tag{2.43}$$

$$F_i^n(\boldsymbol{u}, \boldsymbol{n}, \boldsymbol{p}) = 0, \tag{2.44}$$

$$F_i^p(\boldsymbol{u}, \boldsymbol{n}, \boldsymbol{p}) = 0. \tag{2.45}$$

These are three times N_v equations in $3N_v$ unknowns, where N_v is the number of grid points. One possibility to solve the equations is by applying the above Newton procedure to the whole $3N$-dimensional system. This is called the *coupled solution* or *full Newton* approach.

Alternatively one can first solve (2.43) for $\boldsymbol{u}$, use the new $\boldsymbol{u}$ and the original $\boldsymbol{n}$ and $\boldsymbol{p}$ to solve (2.44) for $\boldsymbol{n}$, and use the new values of $\boldsymbol{u}$ and $\boldsymbol{n}$ together with the original $\boldsymbol{p}$ to solve (2.45) for $\boldsymbol{p}$. This must then be iterated until a self-consistent solution is achieved, effectively performing a nonlinear block Gauss-Seidel iteration. The method is usually called *decoupled solution* or *Gummel* or *plugin iteration.*

The advantage of the coupled scheme is that the coupling between the PDEs is fully taken into account and convergence is generally much faster than with the Gummel method. On the other hand, when the coupling is weak (low injection case), the Gummel method may actually converge just as quickly as the full Newton scheme. In that case it is certainly preferable to use the former, since the latter requires far more memory due to the fact that the linear system to be solved have three times the number of unknowns.

Experience shows that the full Newton method only converges if started from a reasonably good initial solution. For a truly general purpose device simulator, a good initial guess is not possible without significant effort (comparable to the total solution effort, cf. Section 4.5). Hence it must be possible to start the simulation from a poor initial guess. This is possible with the Gummel iteration, which converges for a very wide range of starting values. The Gummel method is therefore indispensable for a general purpose device simulator.

On the other hand, the decoupled scheme does converge very slowly (or not at all) if the PDEs are strongly coupled (high injection case). Here one is forced to use the full Newton iteration. The same holds true for transient simulations. Hence, both methods must be implemented in the device simulator.

2.3.3 Choice of Variables

The choice of the variables strongly influences the nonlinear convergence. So far we have expressed most equations in terms of the variables (u, n, p). Alternatives are to use quasi-Fermi levels in place of the densities, (u, v, w), or the *Slotboom variables* $(u, \nu := \exp(-v), \omega := \exp(w))$.

At a first glance, the variable set (u, n, p) seems attractive for the decoupled method, since the equations (2.10, 2.18, 2.19) are linear (ignoring the dependence of the mobilities and recombination rates on the variables). However, it turns out that the Gummel iteration does in most practical cases not converge in these variables [57]. Using the set (u, v, w) for Poisson's equation results in a stable Gummel iteration. Note that for the decoupled method there is no need to use the same set of variables for the different equations, it is therefore possible to use the carrier densities for the continuity equations and thus keep these linear.

In the coupled case, the equations become nonlinear, even when expressed in densities, due to the Bernoulli functions in Eqs. (2.44, 2.45). When using quasi-Fermi levels we have in addition the exponential dependencies on the variables in the density terms of all three equations. It is therefore to be expected that the variable set (u, n, p) is preferable in the coupled case, which is exactly what Pinto [57] finds.

There is a problem, however, in the scaling of the variables. While u typically varies over one or two orders of magnitude, the carrier densities vary over ten to twenty orders of magnitude. This causes severe problems when linear systems are solved by iterative methods (see next section). A linear solver will in general not be able to resolve the small variations in the potential when solving for densities at the same time. This essentially renders the concentration variables useless when performing a full Newton iteration while using iterative linear solvers. One might hope that some smart scaling of the equation could help, but currently no such scaling is known. The quasi-Fermi levels, on the other hand, are scaled comparably to the potential and are therefore appropriate for the full Newton scheme.

The Slotboom variables have the advantage that the continuity equations become self-adjoined and symmetric positive definite, a property the other sets of variables do not have. However, their scaling is even worse than that of the densities, so that they are of no help in the coupled case.[1]

To summarize this discussion, we found that the variable set (u, n, p) works well for solving the continuity equations in decoupled mode, while for Poisson's equation and in the coupled case the variables of choice are (u, v, w).

2.4 Sparse Linear Systems

Owing to the fact that the box discretization produces coupling between different grid points only if the points are edge neighbours, the linear system of equations, e.g. (2.38), are very sparse. We found that with the irregular grids we are using, there are in average only about eight non-zeros in each row of the coefficient matrix. In order to keep time and memory requirements of the linear solves within reasonable limits, it is mandatory to employ algorithms and data structures that make use of the sparsity, so-called *sparse linear solvers*. Since the computer time required for a simulation is usually dominated by the time needed for linear solves, it is mandatory to use the fastest methods available.

The linear systems can be solved by *sparse direct methods* (i.e. variants of *Gaussian elimination*) [29, 28, 9, 4, 1] or by *sparse iterative methods*, usually generalization of the basic *conjugate gradient method* (CG) [44].

Direct methods have traditionally been used in device simulation, and enjoy continued popularity in 2d [57]. The major reason is that they reliably produce a solution, while most iterative methods cannot handle the ill-conditioned matrices arising in device simulation. However, due the huge grid sizes typical for 3d device simulations, problems with memory size made the use of iterative methods a necessity.

The memory requirements of an iterative method are fixed, known in advance, and fairly low. Only a few vectors of length N are required as working space (typically between three and twelve, depending on the

---

[1] Pinto reports typical condition numbers of 10^{15} to 10^{16} for densities, up to 10^{20} for Slotboom variables, and as low as 10^{3} for quasi-Fermi levels. This is a clear indication that only the latter choice is useful when applying iterative solvers.

method), plus, in the case of no-fill incomplete factorization preconditioning, one matrix with the same sparsity pattern as the original system matrix. For direct methods, the memory requirements depend highly on the structure of the matrix and particularly on the ordering of the rows in the matrix. Although there are heuristics to reduce the *fill*,[2] like the minimum degree algorithm or bandwidth reduction techniques like the reverse Cuthill-McKee scheme [58], the storage requirements for direct solvers on general sparse matrices are unpredictable and grow superlinearly with the problem size. The difference in the storage requirements of direct and iterative solvers is depicted in Fig. 2.2, based on experimental data.

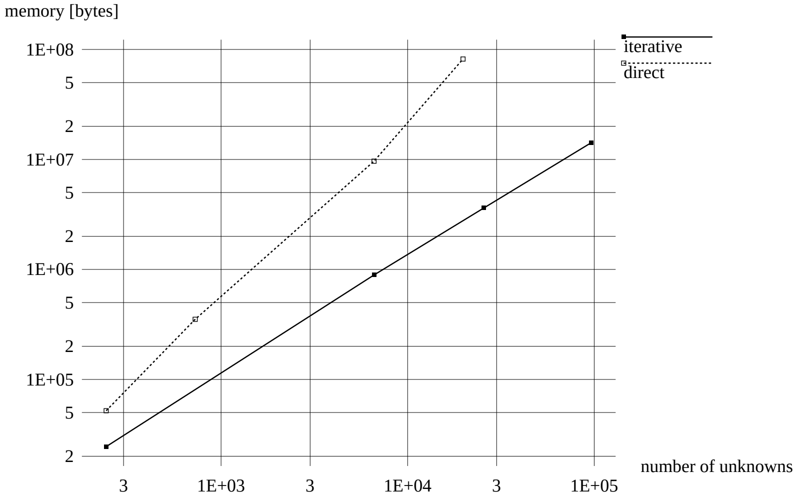

Figure 2.2: *Memory requirements of direct and iterative solvers as a function of problem size*

If we extrapolate the curves to, say, 300 000 unknowns (corresponding to a coupled solve with a 100 000 point grid) we expect memory requirements in the 10 to 100 Gbyte range, which is more than even the biggest machines can offer today. It is obvious that direct methods can no longer be used once the grid sizes exceed some ten or twenty thousand points.

Time considerations also favor iterative methods for large problems. The

---

[2] Non-zero entries in the obtained factor matrix at positions where the original matrix was zero are called *fill*.

time to solve a general linear system by a direct method is as unpredictable as its storage requirements (with an upper limit of $n^3/3$). For a given matrix structure and row ordering, however, this time is fixed, it does not depend on the actual numerical values in the matrix. Conversely, an iterative method requires a fixed amount of work *per iteration*, and the number of iterations required to achieve a certain precision depends strongly on the numerical values of the coefficients. As a result, direct methods are usually faster on small problems. For large problems the iterative methods tend to be faster due to the fact that for a given problem the required number of iterations depends only weakly on the problem size N_v.[3]

The use of iterative schemes has only recently become a topic for device simulation [60, 74]) and the performance of these methods has often been disappointing. However, in the last few years significant progress has been made and currently the CG variants *BiCG* [32], *CGS* [69] and especially *CGSTAB* [79], all in combination with *ILU preconditioning* [52, 53], seem to be most promising. For a detailed comparison of iterative methods in device simulation see Heiser et al. [43].

---

[3] The meaning of \small" and \large" here depends on the machine used for the calculation.

3

Semiconductor Device Modelling in Three Dimensions

In this chapter we discuss semiconductor device simulation from the viewpoint of 3d modelling. The first section reviews the most important 3d simulation projects published so far. The next section illustrates the problems that are particular to device simulation in 3d. The last section of the chapter outlines the approach we have taken with our simulator **Second**.

3.1 Previous Work

The oldest published accounts of 3d device simulation seems to be on the FIELDAY program developed by Buturla et al. [14, 63] at IBM, and the work done by Yoshii et al. [84, 46] at NTT.

FIELDAY is a 1, 2 and 3-dimensional FEM code. For 3d simulations a grid consisting of triangular prisms is used. This grid is obtained as a tensor product of a 2d triangular mesh and a 1d grid. The approach chosen allows good modelling of device features, including non-rectangular boundaries, in two dimensions, while in the third dimension the grid is regular (and possesses translational symmetry).

FIELDAY already allowed the steady-state or transient solution of the semiconductor equations, using either a full Newton scheme or a Gummel iteration. To save computing time it allowed suppressing one or both continuity equations in cases where carrier flow is unipolar or negligible. The program's applicability was mainly limited by the fact that direct methods were used for the solution of linear systems, apparently due to the poor reliability of the iterative methods available at that time, particularly in the case of irregular FEM grids. This, together with memory sizes available ten years only allowed simulation of grids containing no more than a few thousand points.

Conversely, the NTT effort used a FD method (with regular grids). Only steady state solutions using the Gummel iteration were possible. An analytical (linear) approximation of the variation of the electrostatic potential was used within the oxide for MOSFET simulations. Linear systems were solved by relaxation methods, which allowed grid sizes of up to 20 000 nodes.

The TOPMOST MOSFET simulator by Dang et al. from Toshiba [27, 66] was the first to use preconditioned CG to solve the linear equations. This purportedly required them to use Slotboom variables for the continuity equations, with all the adverse effects these variables have on the condition of the linear systems (cf. Section 2.3.3). The authors also report using the BM, however their grids are purely tensor product type, and the BM is only used in order to treat some non-rectangular boundaries. An interesting feature of TOPMOST is that it also incorporates a 3d process simulator [55].

Toyabe et al. [51, 74] from Hitachi, with their program CADDETH, were the first to report the use of CG-based methods for solving non-symmetric linear systems, namely BiCG and CR [62]. Usage of tensor product grids enabled them to highly vectorize their code. Their simulator can model avalanche breakdown of MOSFETs and has been extensively used in the investigation of α-particle induced soft errors [72].

Notable recent work includes the SMART program by Odanaka et al. [54] from Matsushita, which also combines 3d process and device simulation, and which can simulate GaAs-MESFETs [76]. The well-known MOSFET simulator MINIMOS by Selberherr et al. from the Technical University of Vienna, which includes energy balance [38], has been extended to 3d [73] and recently also to GaAs-MESFETs [49] and to non-rectangular Si-SiO$_2$ interfaces [75]. SMART and MINIMOS both use tensor product grids.

The SIERRA program by Chern et al. [17] from Texas Instruments is a 3d extension of the well established Pisces-II simulator [56]. It uses the BM with prismatic elements for stationary and transient simulations as well as small signal analysis. The geometry specification is extracted from layout and process descriptions. This is probably the most versatile 3d device simulator published to date. However, the usage of prismatic grids still significantly restricts the generality of devices that can be modelled. The HFIELDS-3D simulator by Baccarani et al. [19] from the University of Bologna also uses prismatic grids.

If we compare the recent publications with the oldest ones, we can see that recent progress has chiefly been made in two areas: improved numerical methods, in particular improved iterative solvers, have made 3d simulation more practical. Improved physical models have made them more realistic (energy balance) or applicable to a wider range of problems (GaAs).

With respect to geometrical generality the improvements have been rather modest: while some progress has been made by allowing some limited form of non-rectangular geometries, all projects use grids that are essential tensor products of one- or two-dimensional meshes and are therefore not well suited to model truly 3d geometry and device features. The result is a grid that is much bigger (in terms of the number of grid points) than what is really needed and wanted, resulting in excessive memory consumption of the simulator. Furthermore the grids are still essentially 1+1+1 dimensional (rectangular grids) or 1+2 dimensional (prismatic grids), implying limited capability to model general device geometries.

3.2 What Makes 3D Harder Than 2D?

In this section we will examine some of the main difficulties that are inherent in 3d device simulation.

3.2.1 Computational Complexity

As can be seen from Figure 2.2, the memory requirements of an iterative linear solver grow approximately linearly with the problem size, i.e. the number of unknowns. The same holds true for the device simulator in general, so that one can say that the required memory size is proportional to the grid size. Similarly, the time *per iteration* of the linear solver is proportional to the number of equations. Since the condition of the linear systems tends to deteriorate with increasing number of unknowns, the time required for the simulation grows in general superlinearly with the grid size. Note that this is only a rough \back of the envelope" calculation, since the condition of the linear systems will also depend on the grid geometry, sometimes a simulation may actually be faster on a bigger grid than on a smaller one. This, however, is exceptional. The general tendency of a slightly superlinear dependence of simulation time on grid size is certainly correct.

The transition from 2d to 3d is obviously connected with a huge increase in grid sizes. Typical 2d simulations with irregular grids use several hundred up to a few thousand points, and for simulations with regular grids a few thousand points are certainly a necessity for non-trivial device geometries. If we assume symmetric treatment of all space dimensions, the transition from 2d to 3d will increase the grid size by a power of $3/2$, that is from 1 000 to 30 000 or from 4 000 to 80 000. Such grid sizes are sufficient to fill the memories of the largest supercomputers available today, and typical supercomputer run times for a grid with close to 100 000 points are of the order of hours for stationary and days for transient simulations, which is at the edge or beyond of what can be considered practical or \tractable". For smaller machines, like mainframes or mini-supercomputers, the maximum size of \tractable" problems is maybe four times smaller than for supercomputers.

3.2.2 Numerical Aspects

We have already pointed out in Section 2.4 that direct linear solvers cannot be used for realistic 3d simulations since their time and space complexity is too high. This poses new problems. The linear systems arising from the discretization of the semiconductor equations are notoriously ill-conditioned (cf. Section 2.3.3), and the condition tends to deteriorate with increasing grid size. The use of preconditioned solver algorithms is therefore mandatory.

Because of the long run times typically associated with 3d device simulations, it is imperative to make optimal use of the hardware, in particular the parallel processing capabilities of vector or multiprocessor computers. However, as Heiser et al. [43] have shown, the reordering of unknowns required to achieve this goal can be counterproductive|condition deteriorates further and the number of iterations required for convergence is increased, sometimes convergence is even fully destroyed.

Hence, to perform realistic 3d simulations we need very stable iterative solvers that can handle ill-conditioned systems, and \good" grids, that prevent the condition number from becoming too big.

3.2.3 Geometry Definition

There is a qualitative difference in the difficulty of specifying the geometry of a 2d or a 3d object. While a 2d geometry can basically be defined by a simple drawing (e.g. using a mouse and a graphic display) this is not the case in 3d. Sophisticated geometric modelling tools are required, and even with a good solid modeller construction of a 3d device geometry is much more difficult and time consuming than it is in 2d.

Another problem is caused by the need to use process simulator output or measured doping data. 3D process simulators are not yet widely available, even 2d simulation is not yet generally done in process modelling. The problem is worse with experimental data|it is difficult (and inaccurate) to measure doping profiles in 1d and measured 2d profiles are an exception. Hence for the purpose of device simulation, the 3d impurity information must be constructed out of 1d or 2d process simulation or measurement data.

3.2.4 Grid Generation

Since 3d grids are generally much bigger than 2d grids, and since 3d simulation is at the limit of today's computers, every attempt must be made to keep the number of grid points as small as possible. This is only possible if irregular grids are used, otherwise many grid points are wasted in device regions where a low point density suffices.

The generation of grids adapted to needs in three dimensions is by itself a difficult problem. While in 2d it is possible (in principle) to place points manually, this is not possible in 3d. The grid generation process must be fully automated. The angle conditions imposed by the BM add enormously to that difficulty, since it is a hopeless task to \regularize" a grid that does not fulfill these conditions. The grid generation process must take the angle conditions into account right from the beginning [21]. Furthermore the elements of the grids must have bounded aspect ratios to avoid unnecessarily poor condition numbers in the linear systems resulting from the discretization.

3.2.5 Visualization of Results

Visualization of 3d simulation results is both important and difficult. While it may be possible to examine 1d results in the form of tables or simple curves, this becomes already impractical in 2d. In 3d the sheer amount of data necessitates some condensed graphical representation.

On the other hand, there is again a qualitative difference between the visualization of 2d and 3d results. A scalar function on a 2d domain produces a surface embedded in a 3d space, which is relatively easy to visualize since the world we experience is three dimensional. In the same way a function on a 3d domain would require a four dimensional representation, which is beyond the imagination of most humans. New approaches must therefore been taken for the visualization of 3d results.

3.3 Our Approach

In the preceding sections we attempted to give the reader an impression of the difficulties involved in constructing a truly general 3d device simulation system, much more than a single person can handle within a reasonable amount of time. It is therefore necessary to break the whole problem into several parts. However, the full extend of the problem must be kept in mind when going about to solve the partial problems.

We will from now on focus our attention back on the \simulator proper", i.e. the program that, when supplied with a suitable description of the device to be simulated, including the grid, will solve the semiconductor equations and produce results in a form which allows their visualization using the appropriate tools. We will take another look at the complete simulation system in Section 5.3.

The reader surely couldn’t help noticing our view that currently only the BM allows modelling general device geometries while being applicable to arbitrary device operating conditions|we therefore adopted the BM for our simulator system. In order to have sufficient flexibility we allow grids to be composed of four element types (*shapes"*): *tetrahedra*, *quadrilateral pyramids* (with a parallelogram base), *triangular prisms*, and *parallelepipeds* (sheared cuboids).

With these elements every plane faced device geometry (or internal interface) can be modelled. Furthermore, they allow to interpolate between (quasi-regular) grid regions of various coarseness, thus permitting the point density of the grid to vary in all directions. While this would be possible by using tetrahedra alone (all the other elements can be divided into at most five tetrahedra), by using cuboids we can significantly reduce the number of elements and edges in the grid.

The admissible element shapes are defined such that the type and three edge vectors are sufficient to define each element [41]. This permits the use of efficient data structures for describing the grid within the simulator.

4

Methods

In this chapter we describe some of the methods used in the actual implementation of the concepts presented in the previous chapters.

4.1 Assembly

At the very heart of the simulation lies the problem of solving the linear equations (2.38) arising from the Newton procedure. Before such a system can actually be solved it must first be assembled, i.e. the coefficient matrix (LHS) and the right hand sides (RHS) must be computed.

4.1.1 Poisson's Equation

From the discretized Poisson's and Laplace's equations (2.10, 2.11) we obtain the LHS as

$$-\frac{\partial F_i^u}{\partial u_k} = \varepsilon \frac{A_{ik}}{l_{ik}} + \delta_{ik} \sum_{j \neq i} \varepsilon \frac{A_{ij}}{l_{ij}} - \delta_{ik} V_i (n_i + p_i). \qquad (4.1)$$

Here, $\varepsilon = 1$ within silicon, while within the oxide, the charge terms $\delta_{ik} V_i (n_i + p_i)$ are to be omitted, and $\varepsilon = \varepsilon_{ox} / \varepsilon_{semi}$. Obviously the first two terms in (4.1) only depend on the grid, not the current values of the unknowns. This part, which represents the discretized Laplace operator, can therefore be computed once for the whole simulation.

When setting up this matrix, which we denote (a_{ij}), one has to make sure that the proper value of ε is used for each edge. If the box section A_{ij} does not fully belong to a single material (e.g. if the edge is on a material interface) the box parts from the different materials must be multiplied with the proper value of ε and added together.

Once the matrix (a_{ij}) is set up, the LHS for the Poisson equation can be assembled by simply copying (a_{ij}) and adding the charge terms to the diagonal. The RHS then becomes

$$F_i^u = \sum_j a_{ij} u_k + (n_i - p_i) V_i - N_i V_i. \tag{4.2}$$

The last term on the right is again independent of the solution and can be precomputed. Since doping values are not needed afterwards, there is no storage penalty for this preprocessing. The box volumes V_i can also be precomputed as

$$V_i = \sum_{j \neq i} A_{ij} l_{ij} / 6, \tag{4.3}$$

so that the assembly of the RHS of the Poisson equation reduces to a sparse matrix-times-vector operation plus a few very simple vector operations.

4.1.2 Continuity Equations

The BM discretization of the continuity equations (2.18, 2.19) is expressed completely in u, n, and p. This is an appropriate form for the RHS if we use this set of variables (cf. Section 2.3.3). The LHS then take the form

$$-\frac{\partial F_i^n}{\partial n_k} = \frac{A'_{ik}}{l_{ik}} \mu_{ik}^n B(u_{ki}) - \delta_{ik} \left[\sum_{j \neq i} \frac{A'_{ij}}{l_{ij}} \mu_{ij}^n B(u_{ij}) + V'_i \frac{\partial R_i}{\partial n_i} \right], \tag{4.4}$$

$$-\frac{\partial F_i^p}{\partial p_k} = \frac{A'_{ik}}{l_{ik}} \mu_{ik}^p B(u_{ik}) - \delta_{ik} \left[\sum_{j \neq i} \frac{A'_{ij}}{l_{ij}} \mu_{ij}^p B(u_{ji}) + V'_i \frac{\partial R_i}{\partial p_i} \right], \tag{4.5}$$

where

$$V_i' = \sum_{j \neq i} A_{ij}' l_{ij}/6, \tag{4.6}$$

and A_{ij}' is the part of the box section that lies within the semiconducting material; A_{ij}' is zero for edges that lie completely within insulating material.

If we use the variables u, v, and w, we prefer a form of the discrete equations that is expressed in these variables. Substituting v and w for n and p in Eq. (2.18, 2.19) and making use of the properties of the Bernoulli functions, we obtain the alternative forms

$$\begin{aligned}
F_i^v &= -\sum_{j \neq i} \frac{A_{ij}'}{l_{ij}} \mu_{ij}^n n_i B(u_{ij})(e^{v_{ij}} - 1) + V_i' R_i, && (4.7)\\
F_i^w &= -\sum_{j \neq i} \frac{A_{ij}'}{l_{ij}} \mu_{ij}^p p_i B(u_{ji})(e^{w_{ji}} - 1) + V_i' R_i, && (4.8)\\
-\frac{\partial F_i^v}{\partial v_k} &= -\frac{A_{ik}'}{l_{ik}} \mu_{ik}^n n_i B(u_{ik}) e^{v_{ik}} \\
&\quad + \delta_{ik} \left[\sum_{j \neq i} \frac{A_{ij}'}{l_{ij}} \mu_{ij}^n n_i B(u_{ij}) + V_i' n_i \frac{\partial R_i}{\partial n_i} \right], && (4.9)\\
-\frac{\partial F_i^w}{\partial w_k} &= +\frac{A_{ik}'}{l_{ik}} \mu_{ik}^p p_i B(u_{ki}) e^{w_{ki}} \\
&\quad + \delta_{ik} \left[\sum_{j \neq i} \frac{A_{ij}'}{l_{ij}} \mu_{ij}^n p_i B(u_{ji}) + V_i' p_i \frac{\partial R_i}{\partial p_i} \right], && (4.10)
\end{aligned}$$

with $v_{ij} := v_i - v_j$ and $w_{ij} := w_i - w_j$. We can remove most direct references to the densities in these equations by scaling them with the appropriate densities. That way we obtain for the linear systems

$$\begin{aligned}
(RHS)_i^v &= +\frac{1}{n_i} F_i^v, && (4.11)\\
(RHS)_i^w &= +\frac{1}{p_i} F_i^w, && (4.12)\\
(LHS)_{ik}^v &= -\frac{1}{n_i} \frac{\partial F_i^v}{\partial v_k}, && (4.13)\\
(LHS)_{ik}^w &= -\frac{1}{p_i} \frac{\partial F_i^w}{\partial w_k}. && (4.14)
\end{aligned}$$

Unlike Poisson's equation, the LHS of the continuity equations depend on the solution, not just the grid. However, the matrix elements contain the invariant factors A'_{ik}/l_{ik}. The matrix (A'_{ik}/l_{ik}), which is different from the matrix (a_{ij}) that is used for Poisson's equation, can again be pre-computed.

The Scharfetter-Gummel discretization of the continuity equations assumes the mobilities to be constant along an edge. Because we can compute the mobilities at the grid points, we approximate the mobility along an edge as the average of the nodal values at the edge's end points:

$$\mu_{ij} = \frac{1}{2}(\mu_i + \mu_j). \tag{4.15}$$

The Bernoulli functions and the terms $e^x - 1$ must be evaluated very accurately, otherwise the currents will not be conserved. Truncation of significant digits make the computation of $\exp(x) - 1$ inaccurate for $|x| \ll 1$. We therefore use a polynomial expansion [39] of $B(x)$ and $e^x - 1$ for small values of x.

4.2 Treatment of Boundary Conditions

4.2.1 Dirichlet Boundaries

At Dirichlet boundary points (called *Dirichlet points* from now on) the solution is known from the beginning. The treatment of such grid points within the PDE solver thus consists of two parts:

1. setting up the initial solution so that it satisfies the Dirichlet boundary condition, and
2. ensuring that the values of the unknowns at the Dirichlet points do not change.

The first point is straightforward and needs no further explanation. The second implies that in the Newton update step

$$\boldsymbol{z}^{k+1} := \boldsymbol{z}^k + s^k \boldsymbol{\delta z}^k \tag{4.16}$$

all the components $i, i \in \Gamma_0$ of $\boldsymbol{\delta z}$ that correspond to Dirichlet points are zero. Since $\boldsymbol{\delta z}$ is the solution of a linear system

$$\sum_j A_{ij} \delta z_j = b_i, \tag{4.17}$$

this means that we already know part of the solution to Eq. (4.17). Let us assume that the equations are numbered such that Dirichlet points have the highest numbers, i.e.

$$\forall i \in \Gamma_0 : \forall j \notin \Gamma_0 : i > j. \tag{4.18}$$

We can now drop all coefficients of (A_{ij}) which are multiplied with the Dirichlet components of the solution, since their removal will not change the solution of the system. These are the components (A_{ij}) with $j \geq N_D$, if N_D is the Dirichlet point with the smallest number. In the schematic representation of Figure 4.1 this includes all the coefficients in block II of the matrix.

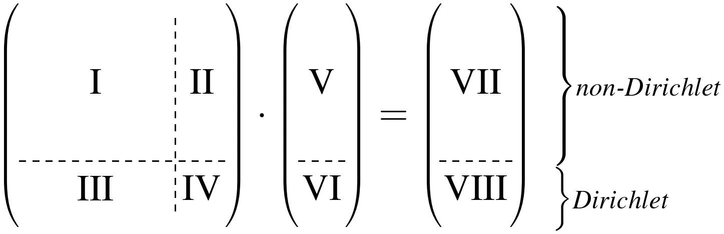

Figure 4.1: *Schematic view of Dirichlet and non-Dirichlet regions in the linear system of equations*

The dropping of these coefficients in fact completely decouples the equations for the non-Dirichlet points from the equations for the Dirichlet points. Hence it is no longer necessary to solve the equations for the Dirichlet points at all, rather than solving the $N \times N$ system (4.17) we can solve the reduced $N_D \times N_D$ system which is obtained from (4.17) by dropping the last $(N - N_D)$ equations and the last $(N - N_D)$ columns of the coefficient matrix. Alternatively we can zero all the coefficients in regions II and III of Figure 4.1, make region IV a unit matrix, and set the right hand sides in region VIII to zero.

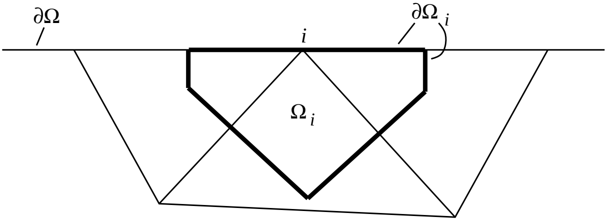

Figure 4.2: *Box of a boundary point*

4.2.2 Neumann Boundaries

Neumann boundary conditions are even easier to treat|they basically look after themselves. If we focus on Eq. (2.4), the integral $\int_{\partial\Omega_i} \boldsymbol{F} \cdot d\boldsymbol{n}$ gives a zero contribution on $\partial\Omega_i \cap \Gamma_h$, the part of the box boundary that coincides with the Neumann boundary; due to the fact that $\boldsymbol{F} \cdot d\boldsymbol{n} = 0$ by definition. In the discretized equation (2.6) this is ensured automatically, since no edges correspond to the box boundary section $\partial\Omega_i \cap \Gamma_h$, and therefore no term in (2.6) corresponds to that part of the boundary (see also Fig. 4.2). In other words, Neumann vertices can be treated exactly like internal vertices.

4.2.3 Internal Boundaries

The situation is basically the same for internal boundaries between different materials (so-called *internal interfaces*). The only kind of internal interfaces we have to deal with are the boundaries between semiconducting material and insulator. The interface condition for the continuity equations is that no current can flow across the interface, which is a Neumann boundary condition for the continuity equations (cf. Section 1.2.2).

For Poisson's equation, the interface condition in the absence of surface charges is the continuity of the electric displacement $\boldsymbol{D} = \varepsilon\boldsymbol{E}$. This is again satisfied automatically due to the fact that the charge density, which determines the electric displacement, is continuous. Hence internal boundaries require no special treatment.

4.3 Terminal Currents

As has been shown by Bürgler [13], the current $I_i^{n,p}$ flowing out of the i-th contact due to electron or hole conduction can be obtained by summing the right hand sides obtained when assembling the continuity equations:

$$I_i^{n,p} = - \sum_{\omega_j \subset R_i} F_j^{n,p}. \tag{4.19}$$

Here R_i is a region (part) of the device that contains contact i but no other contact. Note that here we need to use the RHS as assembled independent of the Dirichlet boundary conditions. In other words, if we treat Dirichlet boundary conditions as suggested in Section 4.2.1, we must do the computation of the terminal currents according to Eq. (4.19) *before* setting the RHS to zero for Dirichlet points.

According to Bürgler the total current (including the displacement current in transient simulations) can be obtained by solving

$$\boldsymbol{\nabla} \cdot \boldsymbol{J}^n + \boldsymbol{\nabla} \cdot \boldsymbol{J}^p - \boldsymbol{\nabla} \cdot \varepsilon \boldsymbol{\nabla} \dot{u} = 0. \tag{4.20}$$

The linear system resulting from the discretization of this equation has the RHS

$$F_i^t := f_i^n + f_i^p + \sum_j a_{ij} \dot{u}_k, \tag{4.21}$$

while the LHS is, except for the sign, equal to the discretized Laplacian (cf. Section 4.1.1).

After solving Eq. (4.20) we can obtain the total current through contact i as

$$I_i^t = \sum_{\omega_j \subset R_i} F_j^t \tag{4.22}$$

just as in the case of the conduction currents. The displacement current is then the difference of the total and conduction currents:

$$I_i^d = I_i^t - (I_i^n + I_i^p). \tag{4.23}$$

4.4 Electric Field

The physical models employed in device simulation often contain references to the electric field or the electron and hole current densities (cf. Section 1.4). We must therefore find a way to compute these entities *at the grid points*.

As shown in Section (2.1.2), under the assumption of a constant electric field, the projection $E_{ij} = \boldsymbol{E} \cdot \boldsymbol{l}_{ij}/|\boldsymbol{l}_{ij}|$ of $\boldsymbol{E}$ onto the edge $\overline{ij}$ is known to be $-u_{ji}/l_{ji}$. The problem now is to reconstruct the vector $\boldsymbol{E}$ from its projections.

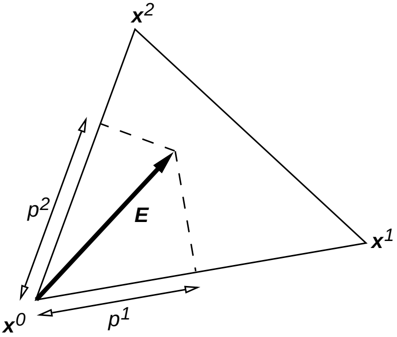

Figure 4.3: *Electric field vector within an element*

Consider the situation in Figure 4.3: In d space dimensions ($d = 2$ in the Figure) we have a corner of an element where d edges meet. The projections, p_i, of the vector $\boldsymbol{E}$ onto the edges are

$$p_i = \boldsymbol{E} \cdot \frac{\boldsymbol{l}^{i0}}{|\boldsymbol{l}_{i0}|} = \sum_{j=1}^{d} E_j \frac{l_{i,j}}{|\boldsymbol{l}_{i0}|}, \tag{4.24}$$

where $l_{i,j}$ is the j-th coordinate of $\boldsymbol{l}_{i0}$.

Since the vectors $\boldsymbol{l}_{i0}$ are assumed to be linearly independent, the matrix $(l_{i,j}/|\boldsymbol{l}_{i0}|)$ is regular and has an inverse $(a_{ij}) := (l_{i,j}/|\boldsymbol{l}_{i0}|)^{(-1)}$ with the property

$$\sum_{i=1}^{d} a_{ki} \frac{l_{i,j}}{|\boldsymbol{l}_{i0}|} = \delta_{kj}. \tag{4.25}$$

Multiplying (4.24) with this inverse from the left we obtain

$$\sum_{i=1}^{d} a_{ki} p_i = \sum_{j=1}^{d} \sum_{i=1}^{d} a_{ki} \frac{l_{i,j}}{|\boldsymbol{l}_{i0}|} E_j = \sum_{j=1}^{d} \delta_{kj} E_j = E_k. \tag{4.26}$$

This means that we can compute the electric field vector as

$$E_k = -\sum_{i=1}^{d} a_{ki} \frac{u_i - u_0}{|\boldsymbol{l}_{i0}|}. \tag{4.27}$$

If the electric field is not constant, (4.27) holds only approximately, and the value obtained for E will depend on the d-tuple of edges used; none of these values will, in general, be equal to the \true" value of E. However, if the grid sufficiently resolves the physics of the device, the values obtained should be a reasonable approximation.

The question remains how to obtain a value for E at a grid point. The obvious approach is to compute in each element, ω, that is incident in the vertex, $\boldsymbol{x}$, an approximation, $\boldsymbol{E}^\omega$, to the electric field vector according to (4.27). We can then get an approximation to the field as a weighted average of the values obtained for the individual elements:

$$\boldsymbol{E}(\boldsymbol{x}) = \frac{1}{S} \sum_{\omega \ni \boldsymbol{x}} s^\omega \boldsymbol{E}^\omega, \tag{4.28}$$

where

$$S := \sum_{\omega \ni \boldsymbol{x}} s^\omega. \tag{4.29}$$

Note that in general $\boldsymbol{E}^\omega$ depends on the corner of ω at which it is evaluated. Within each element ω, the value of the corner corresponding to vertex $\boldsymbol{x}$ must be taken.

As weight factors s^ω we choose the angle spanned by ω. In 3d this means computing the solid angle spanned by three vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$. According to [12] this angle is equal to

$$s^\omega = 4\,\mathrm{arctan}\left(\tan\frac{\alpha+\beta+\gamma}{4}\right.$$
$$\left.\times\tan\frac{-\alpha+\beta+\gamma}{4}\tan\frac{\alpha-\beta+\gamma}{4}\tan\frac{\alpha+\beta-\gamma}{4}\right)^{\frac{1}{2}}, \qquad (4.30)$$

where α, β and γ are the (2d) angles enclosed between each pair of edges. This choice of s^ω yields

$$S = 4\pi. \qquad (4.31)$$

An additional complication exists if there are 3d elements which have vertices where more than three edges meet | as is the case at the tip of our quadrilateral pyramids. We can handle this case by (temporarily) treating such a pyramid as two tetrahedra. A more symmetric treatment is to compute a value of $\boldsymbol{E}^\omega$ for each of the four triples of edges, and extending the sum in (4.28) over all four subelements. In this case we must divide the weight s^ω by two, since each part of the element is used twice.

We can summarize the electric field computation as

$$E_i(\boldsymbol{x}) = -\frac{1}{S}\sum_{\omega\ni\boldsymbol{x}} s^\omega \sum_{j=1}^{d} \frac{a_{ij}}{|\boldsymbol{l}^{\omega j}|}[u(\boldsymbol{x}_j^\omega) - u(\boldsymbol{x})]. \qquad (4.32)$$

In 3d this is rather expensive to compute, due to the many inversions of 3 × 3 matrices required. It is therefore advantageous to accumulate all the factors $(s/S)(a/|\boldsymbol{l}|)$ in a matrix b_{ijk}. The storage requirement for this matrix is d times the number of edges. Once the matrix is set up, the electric field in *all* vertices can be computed as a simple linear transformation of the potential differences along the edges:

$$E_i(\boldsymbol{x}_j) = \sum_{k\neq i} b_{ijk} u_{jk}. \qquad (4.33)$$

The current density vectors at the grid points can be computed in the same fashion: The Scharfetter-Gummel solutions (2.15,2.17) are in fact the projections of the current densities onto the grid edges. We can therefore compute the current densities at the grid points by means of the same transformation (4.33). However, since there exist no current densities within the insulator, an additional weight factor must be used for points at the interface.

4.5 Initial Solution

If we want to simulate a device we need some initial solution to start the iteration. Physical reasoning and experience can often be used to determine an approximation to the final solution. Such an approximation can in principle be used as an initial guess in a device simulator. While this may be feasible in the case of a special purpose simulator, building such \intelligence" into a truly general program is a task whose complexity is at least comparable to the construction of the remainder of the simulator. It must also be remembered that determination of the initial guess should be cheap (in terms of computing time) compared to the solution of the full problem. We are therefore forced to use a rather coarse initial guess that is easy to compute.

A frequently used method to start up a simulation is to solve first for thermal equilibrium, i.e. no applied bias, and then step the terminal voltages up to the required bias conditions. Pinto [57], however, reports that the \local quasi-Fermi" guess leads to a faster convergence. Hence the majority quasi-Fermi potential in each device region is set equal to the bias applied to that region, while the minority quasi-Fermi potential is set such that minimum minority carrier densities result. This is achieved by setting the electron quasi-Fermi potential in p-regions to the maximum applied voltage V_{max}, while in n-regions the hole quasi-Fermi potential is set to the minimum voltage V_{min}. The electrostatic potential is set such that the majority carrier density equals the local impurity concentration, which means that the potential is set to the applied voltage plus the built-in voltage.

This initial guess, while significantly better than the thermal equilibrium solution, is not sufficient for good convergence under all conditions. If the applied bias is too high, the nonlinear iteration may not converge from this starting solution, or may converge very slowly. In such a case it is necessary to first solve for some reduced bias value and step from there up to the required bias.

Often a simulation is not started from scratch, but from the results of an earlier simulation. Since the previous simulation will in general have been performed with different bias conditions, we need to adjust the solution before using it as an initial guess for the new simulation run. For this we use a method similar to the initial guess: If a contact voltage is changed by a certain amount, the electrostatic potential and the majority quasi-Fermi potential is changed by the same amount at all points in the device region belonging to that contact. Minority quasi-Fermi potentials are left unmodified if this does not increase the minority carrier density, otherwise they are changed by the same amount as the electrostatic and the majority quasi-Fermi potential (thus keeping the minority carrier densities constant).

Note that this way of adjusting an earlier result to obtain an initial guess works well if applied voltages are increased in the new simulation (so the old bias values represent something like an intermediate working point). It will perform worse if the applied voltage range is actually reduced in the new simulation, and may fail miserably if going from forward to reverse bias or vice versa.

4.6 Stopping Criteria

4.6.1 Non-linear Iterations

An important aspect of any iterative algorithm is the decision of when the iteration is considered converged and can therefore be stopped. There are principally two kinds of stopping criteria: *relative* and *absolute*.

For the Newton iteration (cf. Section 2.3.1) a relative criterion would be

$$\frac{\|\boldsymbol{\delta z}^k\|}{\|\boldsymbol{z}^k\|} \leq \epsilon_r, \tag{4.34}$$

while an absolute criterion might read

$$\|\boldsymbol{\delta z}^k\| \leq \epsilon_a, \tag{4.35}$$

where ϵ_r, ϵ_a are the relative and absolute error limits resectively.

Obviously, an absolute error criterion does not make much sense for the concentration variables, which vary over ten to twenty orders of magnitude, and even if only majority carriers are considered, the range is still at least six orders of magnitude. In regions where the (majority) carrier densities are relatively small, the (absolute) fluctuations of the densities near convergence are so small that an absolute convergence criterion sensitive to these changes would, in regions where the densities are large, translate into a relative error of, say, 10^{-10} or smaller. It may not even be possible to obtain such a high accuracy.

An absolute error criterion can still be applied if the error is monitored in terms of the quasi-Fermi levels. This is easily possible, even if the iteration is actually performed in terms of the carrier densities. For quasi-Fermi levels, as for the electrostatic potential, a (uniform) absolute error criterion makes sense, and reasonable values are in the range $10^{-3}U_T \cdots 10^{-2}U_T$.

We generally use a combination of both kinds of criteria: an iteration is considered converged if either the relative or the absolute criterion is met. For the relative criterion, usual values are $10^{-6} \cdots 10^{-4}$. We use the l_2 (*Euclid*) norm for relative, and the l_∞ (*maximum*) norm for absolute error criteria.

Pinto [57] recommends an absolute stopping criterion of $10^{-5}U_T$, reasoning that, due to the quadratic convergence of the Newton method, this does not really cost much computing time. For VLSI applications, where typical voltages are of the order of $5\,\mathrm{V} \approx 200U_T$, this corresponds to a relative tolerance of 5×10^{-8}. The linear solver error (see next section) should, of course, not be greater than the nonlinear tolerance, otherwise the latter does not make sense. That implies that linear systems must be solved to at least the same tolerance of 5×10^{-8}. This is no problem when using direct solvers, an iterative solver, however, may take very many iterations or not converge at all when trying to solve a system so accurately. For that reason we have to use a less strict nonlinear stopping criterion.

4.6.2 Linear Iterations

When using iterative linear solvers, the question arises how accurately the linear systems are to be solved. Clearly, an insufficiently converged linear solve may prevent convergence of the nonlinear solve. We generally require a relative error for the linear solve that is by a factor $0.1 \cdots 0.5$ smaller than the nonlinear tolerance expected.

At the beginning of a Newton iteration, when the variables are still far away from the solution values, it seems a waste of effort to solve the linear systems too accurately. Indeed, as Bank and Rose [8] have shown, the quadratic convergence of the Newton iteration is preserved, if, in the k-th Newton step, the linear solver error is less than

$$\alpha_k := \alpha_0 \frac{\|\boldsymbol{F}^{k-1}\|}{\|\boldsymbol{F}^0\|}, \tag{4.36}$$

for some $\alpha_0 \in (0, 1)$. We have found when that using $\alpha_0 = 0.5$ for Poisson's equation, the Newton iteration usually converged in the same number of iterations as if all linear solves were performed with high accuracy, and up to 50% of computing time was saved. For coupled solves we found the value $\alpha_0 = 0.1$ to be safer.

In order to avoid requiring unreasonable linear solver tolerances, a lower limit for the tolerance of the linear solver is taken, if the tolerance as required by the above formula becomes too small. This minimum tolerance is user settable, its default value is one tenth of the nonlinear tolerance.

In transient simulations the Newton iteration often converges in one or two iterations. In such a case the tolerance determined by Eq. (4.36) is actually too big and may lead to unnecessarily large errors, resulting in an increase of Newton iterations. It is then advisable to turn off the automatic adjustment of linear solver tolerances by setting α_0 to a very small value. Since the linear solver tolerance is limited to a minimum, setting $\alpha_0 = 0$ will do.

4.6.3 Transient Simulations

In transient simulations, the time step is controlled by the error parameters ϵ_R and ϵ_A in Eq. (2.35). We usually set the former to ten times the nonlinear tolerance, while the latter is set to

$$\epsilon_A = \frac{\epsilon_R^2 \|\boldsymbol{V}\|_2}{\sqrt{2N_v}}, \tag{4.37}$$

where $\boldsymbol{V} = (V_i)$ is the vector of box volumes. These are, admittedly, purely heuristic criteria, but they seem to work.

5

Implementation

This chapter describes the implementation of **Second** in some detail.

5.1 Software Engineering Aspects

5.1.1 Hardware and Software Environment

The design and implementation of **Second** was strongly influenced by certain constraints imposed by hard- and software.

The code was developed at the Integrated Systems Laboratory at ETH Zürich, where a variety of hardware exists ranging from Sun workstations to Alliant (and later also Convex) mini-supercomputers, plus in the later phase access to ETH's Cray X-MP and Cray-2 supercomputers. These machines differ widely in architecture, performance (and price), but all run the same kind of operating system (all Berkeley UNIX or UNIX System V). Hence the *development* environment was characterized by very diverse hardware but relatively homogeneous software.

On the other hand the code was supposed to run on other machines available to industrial partners, like a Siemens/Fujitsu VP-200 supercomputer running MSP, an operating system largely compatible to IBM's MVS. Hence the *application* environment was characterized by diverse hardware *and* software.

The traditional implementation language in the area of scientific computing is FORTRAN. Because of its many shortcomings, which will be explained in detail later on, we were seriously investigating the possibility of using a different implementation language.

There were only two other candidates: C and Pascal. Both were available on most UNIX systems, both were just becoming available on Cray supercomputers and both we considered much better languages than FORTRAN. We did expect the C or Pascal compilers to generate less efficient code that the FORTRAN compilers, but felt that this could be handled by writing the (usually quite simple) inner loops in FORTRAN.

The real problem were the industrial partners. On the VP-200 system there was no C compiler. There was a vectorizing Pascal compiler announced for the first half or 1988, and, in fact, Siemens coding regulations [78] called for all new software to be implemented in Pascal.

Unfortunately, this Pascal compiler never materialized, and we were finally left with the choice between continuing without the support from Siemens or biting the bullet and writing in FORTRAN. The next section attempts to outline the implications of that decision.

5.1.2 Drawbacks of FORTRAN

The FORTRAN language was developed in the mid ’50s by Backus et al. [6]. The language has evolved since, but even the most recent standard [2] describes a rather archaic language that has roughly the power of Algol-60 [5] while completely lacking the latter’s elegance.

The three most frequently voiced reasons why people continue to use FORTRAN are

- the huge world-wide investment in FORTRAN code,
- the efficiency of the produced machine code, and
- the portability of FORTRAN programs.

The first of these points is obviously of little consequence for new software. The second one is indeed true and is due to the fact that FORTRAN is the most frequently used programming language for problems where this kind of efficiency is important. Consequently, the manufacturers invest most into optimizing FORTRAN compilers. This, of course, leads to a vicious circle|people use FORTRAN because it's efficient, and it's efficient, because people use it a lot. However, at least in an environment like UNIX, where inter-language calls pose no unsurmountable problems, this argument is not really a decisive one, one can code most of a system in another language and fall back on FORTRAN for the few really critical algorithms.

The last point, portability, works really against FORTRAN, if we look closer. There are various reasons for this.

Supersets

One is the proliferation of *supersets* that is typical for FORTRAN. Because FORTRAN lacks so much of the power of modern computer languages, most manufacturers implemented supersets of the standard language. These supersets, of course, differ between compilers from different manufacturers, and more often than not even between different compilers from the same manufacturer. Some of these non-standard features have actually developed into *\de facto*-standards" which many (if not most) users of the language actually consider part of standard FORTRAN. The bad surprise often comes much later when a relatively mature code is ported to a machine whose compiler only supports the standard language, or an incompatible superset.

Because these *de facto*-standards are so deeply entrenched into the FORTRAN community, writing portable FORTRAN code is quite difficult. Most of the programs one sees use non-standard features, most language manuals do not clearly differentiate between standard features and extensions, most compilers do not consistently point out non-standard usage, and most people who teach FORTRAN to their students do not know the difference either. The only help comes from [2], which, of course, is quite hard to read and one must almost know it by heart in order to find all the portability catches.

Control- and data structures

The reason for this notorious supersetting in FORTRAN implementations originates from the fact that FORTRAN is such an old-fashioned language lacking so many of the features that are natural for users of other computer languages. One example is the lack of support for everything that is considered \good" or \modern" programming style. FORTRAN is very poor in control structures, making it almost impossible to adopt a \structured" programming approach. The only control structures available are block `IF`s, a loop construct with an iteration count that is established before execution of the loop begins, and unstructured `GOTO`s. The often needed `WHILE` loop construct is missing completely and can only be emulated with `GOTO` statements.

Much more serious than the lack of control structures is the lack of data structures. The only data types available are simple types (numeric and `LOGICAL`) and arrays of simple types. Pointers and structured types are missing, not to mention any support for *abstract data types*. This means that when programming in FORTRAN one has to forget all the advances in computer languages of the last thirty years, and map all data structures onto primitive objects, a task that is nowadays considered the compiler's job.

While such a \manual compilation" of data structures is, of course, always possible, it defeats the purpose of using sophisticated data structures in the first place. All advantages with respect to readability and maintainability of the code is lost. The time required to write the code is increased, and modifications in the existing code are much harder to do. But worst of all, coding is significantly more error-prone, and the bugs are much harder to find.

Additional problems exist for writing large programs. Since the \software crisis" has been perceived in 1969 [15], modularization is considered one of the most potent weapons to counter the crisis. This, however, is one more place where the FORTRAN language is no help at all. The only modules known in FORTRAN are subprograms, which cannot be nested. Data sharing between program units is possible only via procedure parameters and COMMON blocks.

Parameters are only of limited use, since the lack of structured types and pointers would require huge, unwieldy parameter lists. COMMON blocks, on the other hand, make data completely global, accessible by any program unit. There is no way to ensure that certain data can easily be accessed by a group of related subprograms but remain hidden from others. This means in particular that there is no support for data abstraction and data hiding.

COMMON blocks are furthermore potentially dangerous, since the programmer is largely responsible for their layout. This means that the declaration of a COMMON block must be identically repeated in each program unit that is to access some data from the COMMON block. The only reasonable means to ensure this identical definition is to do it once in a file and include that file in all program units accessing the COMMON block. The problem is that standard FORTRAN does not provide an include statement. Of course, every implementation we are aware of provides some form of an include statement, but since this is not part of the standard, the syntax (if not semantics) of include statements differ between implementations.

Numeric precision

Standard FORTRAN supports two floating point numeric types, `REAL` and `DOUBLE PRECISION`. This reflects the fact that most computers have a word length (nowadays 32 bits, in the old days often 36 bits) that is insufficient for many numerical problems. Semiconductor device simulation is such a problem, where ill-conditioned systems have to be solved and floating point numbers with a mantissa of at least 40 bits are required. Hence, on most computers a device simulator requires the `DOUBLE PRECISION` type.

There exist, however, machines with long words, most notably Cray supercomputers with a word length of 64 bits (48 bit mantissa). On this machines the `REAL` type is obviously sufficient. Furthermore, since `DOUBLE PRECISION` operations are implemented in software on Cray machines, their usage is prohibitively expensive. Hence the program must use `REAL` variables on Crays and `DOUBLE PRECISION` variables on most other computers. Obviously, a fully portable program is not possible in FORTRAN.

Most compilers support some kind of switch that instructs the compiler to automatically treat every `REAL` declaration as `DOUBLE PRECISION`. This still does not solve the problem, since many subprogram libraries (e.g. [48]) have entry points for both `REAL` and `DOUBLE PRECISION` parameters, and the entry point name is used to differentiate between the precisions. This means that, besides variable declarations, the names used in subroutine calls have to be changed too.

Besides the problem of the control of the numeric precision, there exists the problem of determining the numeric precision that can be achieved. FORTRAN does not provide any standard means to allow a program to determine the value of the machine epsilon, which is of utmost importance for many numerical algorithms. Since different computers use different representations of floating point numbers, the machine epsilon can differ by more than a factor of 500 between machines, even if the same number of bits are used to store a number.[1]

Access to environment

For any large program it is highly desirable, if not mandatory, to have some access to the computing environment. The most commonly used function of this kind is the processor time consumed by the program. This is important information for tuning the code as well as for comparing computers. Furthermore it is desirable to have the program print a time stamp on its output, and maybe even identify the computer system on which the program was run.

Information on processor time consumption can be important for an entirely different reason. A 3d device simulator will typically run for several hours even on a supercomputer. This implies that interactive usage is often not possible, the program must be run through some kind of batch system, which usually means that the amount of processor time that the program may use is limited, and execution is aborted when the limit is exhausted. A program abort due to exceeded time limits means that some hours of precious supercomputer time may be wasted. Naturally, this must be avoided|the program must terminate in an orderly fashion before it is aborted by the operating system. To this end one may impose, via input data, some limit on the number of iterations the program may perform in its outermost loop. However, such an approach is not always practical since it is not always clear *a priori* what a reasonable limit would be. A better solution is to have the program monitor its time consumption and compare with the allotted limit. The program must then realize when it is about to exceed that limit and terminate in time. This, of course, requires that the program be able to determine the time limit.

For really long computations that must be broken not only in two or three, but maybe in five or ten parts, it is preferable to automate the process of

---

[1] Example: ε for Cray in single precision is 7.1×10^{-15} while on a VAX in `D_FLOATING` (double precision) format it is 1.4×10^{-17}, both using 64 bits!

starting the next job after the successor has terminated. For that reason one wishes to be able to determine, at the command language level, whether the program terminated successfully, ran out of time, or encountered an error. Hence the program should somehow signal its success to the command level. On most operating systems this is done by some kind of exit status that can be set by the program and tested at the command level. The mechanism for setting is, of course, system dependent.

Furthermore, in order to make usage of the program as convenient as possible, one would like to employ some machine specific means of passing information, like input file names, to the program. Most systems have some notion of \command line parameters" that specify file names or options.

None of the above features are a strict prerequisite for making a program perform its task on a computer system. However, they are highly desired to make the program truly useful for a wide range of applications and environments. *None of them are available in standard FORTRAN* and in order to implement them one has to refer to non-portable features.

Memory management

One of the most serious shortcomings of the FORTRAN language is the lack of memory management. All storage assignment happens at compile, link or load time; once a program has been loaded into a computer's main memory, all variables have a fixed address. This makes the compiler's and linker's job easy|on the programmer's and user's back. The implication is that all arrays must be dimensioned with the maximum size they may ever assume. Changing this maximum requires recompilation of at least one program unit. This is, of course, extremely impractical. For once it is not always possible to know in advance how arrays are to be dimensioned as a function of the size of the input data. For example, it is in general not possible to predict how much working storage is required to factorize a sparse input matrix with a given rank and fill. Thus one is faced with the choice of either being conservative, and waste lots of computer memory in most cases, or risk that the program aborts in the middle of the calculation due to insufficient memory.

Another problem is that, since the program size is fixed independently of the input data, even computations with small data sets require the full amount of memory set aside for the biggest jobs. This is clearly unacceptable for an environment where the user pays for memory occupancy. Compiling different versions of the program for various data sizes is a makeshift measure, not a solution of the problem.

Naming restrictions

Variable, COMMON block and program unit names are limited to six characters (letters or digits) in FORTRAN. This is one of the main reasons for the poor readability of FORTRAN code. While six characters may often suffice for local variables, for global entities like subprograms this is clearly inappropriate. Any attempt to use meaningful names for writing \self documenting" code is doomed from the beginning.

Most FORTRAN compilers available today allow names up to at least 31 characters in length. However, since there are still many compilers that follow the six character rule, we cannot rely on the availability of long names.

5.1.3 Further Complications

The preceding section made it (hopefully) obvious that plain standard FORTRAN code would not suffice for the implementation of a big program. On the other hand, portability of the code was essential:

During the development phase it was frequently necessary to move the code to another machine, e.g. for running larger test cases on a faster computer. This, together with the diversity of the target platforms, put high demands on the portability of the code. The frequent re-installations require an automated process for installation and compilation|manually editing the ported code to produce a runnable version on the target system is out of the question. For the required ease of installation we needed a *fully portable source code*. With \fully portable" we mean that, after some initial installation procedure, any later version of the code can be copied to the target system, and can there be compiled and run without any further changes.

5.1.4 Preprocessing

In the preceding two sections we have seen that we had to reconcile two conflicting objectives, the requirement for full portability on one side, and the need for system dependent code on the other. The only reasonable solution to that dilemma, short of dropping FORTRAN, is to use a preprocessor.

The choice we had was whether to use the UNIX tools `ratfor` or `m4`, the C language preprocessor, `cpp`, or to write a new preprocessor. While the latter choice offers the greatest flexibility, it is also the most expensive version and would make sense only if the other possibilities proved to be impractical.

The `ratfor` program was originally written to enhance FORTRAN-66. It does not offer significant improvement over standard FORTRAN and was therefore no help for our problem.

Between the remaining two alternatives we decided in favour of `cpp`, since this program is virtually guaranteed to be available on any UNIX system, and because it is already used by all C programs and it would be helpful when interfacing FORTRAN with C.

Usage of the (UNIXish) C preprocessor does not restrict the simulator to UNIX machines. The preprocessing does not need to be done on the target system, any UNIX workstation will do, and the preprocessor output can then be transferred to the target machine.

By using a preprocessor we were able to find reasonable solutions for most of the problems mentioned above. Some of them are still quite clumsy, and the implementation of all these measures cost a significant amount of time and effort|all for things that are really the compiler's job. C, while not the author's favourite language, provides all the features we consider essential for a project as ours, at no extra cost.

Considering all the effort we had to invest to cope with FORTRAN's shortcomings and pitfalls (not all of which we have mentioned), even the efficiency argument that is regularly used by FORTRAN advocates becomes dubious. In a time where computer power becomes cheaper and cheaper at an astonishing rate, and where programmer time becomes more and more expensive, it is more than questionable whether there is any *overall* efficiency to be gained from using FORTRAN. We certainly feel that we would have had a working program at least a year earlier if we had used a suitable programming language.

5.2 Description of the Implementation

5.2.1 Modules and Files

As mentioned earlier, our development environment featured a variety of different machines all running UNIX operating systems. Consequently we employed the usual UNIX conventions for program sources. In particular we use the file name extension \\`.F`" for FORTRAN source files requiring preprocessing, \\`.f`" for the preprocessed sources, i.e. the \plain" FORTRAN sources, and \\`.h`" for \header" files. The header files are included into the sources by means of `cpp #include` statements. They mainly contain preprocessor directives, code that has to be inserted at several places (like COMMON block declarations) and comments.

From now on we will use the term \module" to designate a set of data structures (in FORTRAN: COMMON blocks) and operations (SUBROUTINEs or FUNCTIONs). In our case, a module typically consists of three files: a source file *modulename*`.F` that contains the procedures, a header file *modulename*`.h` that contains definitions of macros and data to support access to the module by client routines, and possibly another header file *modulename*`_int.h` that contains definitions for internal use by the module.

The convention is that the *modulename*`.h` header file is all a client of the module needs. In particular this file contains comments describing the meaning of the exported data structures and the calling sequences for the exported procedures.

The internal header file should only be used (included) by procedures belonging to the module. Its main purposes is to define data that are shared between different parts of the module.

For illustration, Program 5.1 shows the header file of a sample module `sumint`. Program 5.2 shows the internal header file and Program 5.3 shows the source file. Finally, Program 5.4 shows a client module. The example demonstrates how the data structures can be accessed in the source files, after including the appropriate header files.

The appropriate header files must be included by every procedure that is to access global data. This implies that in general a header file is included several times by the same source file. Since header files also contain macro definitions

```
/* sumint, a package for summing integers.

Data structures:
   the_sum: sum of all numbers processed so far

Entry points:
   add_this (I)
   INTEGER I
adds I

   FUNCTION the_average ()
returns the average value

*/
#ifdef INCLUDE_BODY
      COMMON /sumint_common/ SUM
      INTEGER the_sum
      SAVE /sumint_common/
# ifdef DEFINE_FUNCTIONS
      REAL the_average
# endif /* DEFINE_FUNCTIONS */
#endif /* INCLUDE_BODY */
```

Program 5.1: `sumint.h`

that must not be executed more than once, a mechanism is needed to ensure that certain parts of the header file are seen only once by the preprocessor, while others are seen several times.

To avoid problems with multiple inclusions and to reduce order-dependence of the `#includes` as much as possible, we use the following convention: *FORTRAN code (COMMON definitions) is only included if the macro* `INCLUDE_BODY` *is defined.* This leads to the usage as demonstrated in Programs 5.3 and 5.4: Header files are included for the first time in the header of the source file, i.e. before any FORTRAN code, then the macro `INCLUDE_BODY` is defined. All further `#include` statements are in the declaration parts of the individual procedures.

A special case is the return type of FUNCTIONs. If a module exports a FUNCTION, its return type should also be declared in the header file. This can cause problems with BLOCK DATA subprograms, since some compilers do

```
#include "sumint.h"
#ifdef INCLUDE_BODY
      COMMON /sumint_private_common/ NRINTS
      INTEGER NRINTS
      SAVE /sumint_private_common/
#endif /* INCLUDE_BODY */
```

Program 5.2: `sumint_int.h`

not allow the declaration of non-COMMON variables in this kind of program units. For that reason, FUNCTION return types will only be declared if the macro `DEFINE_FUNCTIONS` is defined.

5.2.2 Macros for Portability and C/C++ Interface

All our macros are based on a set of general-purpose macros defined in the header file `CF_macros.h`. This header file contains only macro definitions, no data declarations. It is not part of a module, i.o.w. there does not exist a corresponding source file. Because it defines many macros that are used in other header files, `CF_macros.h` should always be included first. There is no point of including it more than once since it does not declare any data.

`CF_macros.h` is the main vehicle for the solution of the portability problems mentioned in Section 5.1.2. Furthermore it supports the interface between FORTRAN and C routines. To distinguish FORTRAN from C code, the macro `_FORTRAN_` must be defined. For this reason FORTRAN modules include the header file `F_macros.h`, which in turn defines `_FORTRAN_` and includes `CF_macros.h`.

With all these macros, care has been taken to assure that the string into which the macro expands is at most as long as the macro's name. This is to avoid bad surprises with the FORTRAN 72 column limit: a macro that expands into a string longer than its name could cause some of the generated FORTRAN code to extend beyond column 72 of the preprocessed source file, even when the original source did not. This could result in obscure compiler messages, or even in wrong code.

```
#include "sumint_int.h"
#define INCLUDE_BODY
      BLOCK DATA sumint_bd
#include "sumint_int.h"
       DATA SUM, NRINTS /2*0/
       END

#define DEFINE_FUNCTIONS

      SUBROUTINE add_this (I)
      INTEGER I
#include "sumint_int.h"
      NRINTS = NRINTS + 1
      SUM    = SUM + I
      END

      FUNCTION the_average ()
#include "sumint_int.h"
      the_average = sum / REAL (NRINTS)
      END
```

Program 5.3: `sumint.F`

Numeric precision

As a remedy for the problem of controlling the numeric precision (see Section 5.1.2) `CF_macros.h` provides a macro `normal_precision` (`NORMAL_PRECISION` for C programs). This should be used as a type name when declaring floating point variables. It will expand into `REAL` on Cray computers and into `DOUBLE PRECISION` on 32-bit machines.

The macros `single_precision` and `double_precision` are available for code that needs to use the maximum or minimum precision available. They should be used instead of the (normally equivalent) `REAL` and `DOUBLE PRECISION` since the macros allow consistent use of non-standard types on machine that provide more than two floating point types.

Other macros serve to allow programs to access the most important machine characteristics. These are summarized in Table 5.1. The macros `..._normal_precision` correspond to the `normal_precision` type.

```
#include "sumint.h"
#define INCLUDE_BODY
#define DEFINE_FUNCTIONS
      PROGRAM client
#include "sumint.h"
      INTEGER I
      REAL AV

      DO 100 I = 1, 10
         CALL add_this (I)
 100     CONTINUE
      PRINT *, 'sum = ', SUM
      PRINT *, 'average = ', the_average ()
      END
```

Program 5.4: *Client module*

There exist a few more macros for supporting the use of the `normal_precision` type: `_ints_per_real_` is the size of a `normal_precision` datum in units of `INTEGER` words. The macros `_0_`, `_1_`, `_2_` and `_05_` expand into proper `normal_precision` literals representing the constants 0.0, 1.0, 2.0 and 0.5 respectively.

Long names

To improve readability of the program code it is desirable to make use of long names (longer than the usually allowed six characters). Since not all compilers support these, there must be a means for mapping long names onto standard conforming ones. However, we do not wish to do this mapping unconditionally, since this implies that even on systems where long names are legal, we would have to cope with the unreadable six character names whenever we have to look at the preprocessed sources, e.g. when using a source level debugger.

Therefore `CF_macros.h` defines the macro `LONG_NAMES` for compilers that allow long names for variables, and `LONG_EXTERNALS` on those systems that also allow them for subprogram and COMMON block names. Header files use these to conditionally map long names onto short ones. Using these we get the improved version of the `sumint` header file (Program 5.5).

FORTRAN name	C name	typical value
`max_integer`	`Max_INTEGER`	2 147 483 647
`min_integer`	`Min_INTEGER`	$-2\,147\,483\,647$
`max_single_precision`	`Max_SINGLE_PRECISION`	$3.4 \times 10^{+38}$
`min_single_precision`	`Min_SINGLE_PRECISION`	$-3.4 \times 10^{+38}$
`least_single_precision`	`Least_SINGLE_PRECISION`	1.2×10^{-38}
`eps_single_precision`	`Eps_SINGLE_PRECISION`	1.2×10^{-7}
`max_double_precision`	`Max_DOUBLE_PRECISION`	$1.8 \times 10^{+308}$
`min_double_precision`	`Min_DOUBLE_PRECISION`	$-1.8 \times 10^{+308}$
`least_double_precision`	`Least_DOUBLE_PRECISION`	2.2×10^{-308}
`eps_double_precision`	`Eps_DOUBLE_PRECISION`	2.2×10^{-16}
`max_normal_precision`	`Max_NORMAL_PRECISION`	$1.8 \times 10^{+308}$
`min_normal_precision`	`Min_NORMAL_PRECISION`	$-1.8 \times 10^{+308}$
`least_normal_precision`	`Least_NORMAL_PRECISION`	2.2×10^{-308}
`eps_normal_precision`	`Eps_NORMAL_PRECISION`	2.2×10^{-16}

Table 5.1: *Macros defining machine characteristics for use by FORTRAN and C programs*

Further portability support

The macros `std_in`, `std_out` and `std_err` expand into the FORTRAN logical unit numbers for standard input, standard output and standard error output respectively. Note that the first two of these are equivalent to the implied units designated by the use of an asterisk in an I/O statement. The macros are mainly needed when a unit number must be passed as a parameter. Most non-UNIX systems do not have a notion of a \standard error output" file, on such systems `std_err` will be the same as `std_out`.

`implicit_none` can be used to disable the dangerous implicit typing in FORTRAN. For compilers that support this FORTRAN-8X [3] extension, the macro expands into `IMPLICIT NONE`, otherwise into `IMPLICIT CHARACTER*7 (A-Z)`.

In cases where non-portable constructs are necessary, a machine dependent macro can be used to hide such a construct from other compilers. For example, on the Alliant (and only there) the macro `_alliant_` is defined. This allows for the safe use of nonportable code by protecting it with

```
#ifndef LONG_EXTERNALS
# define add_this SIADD
# define the_average SIAVER
# define sumint_common SICOMN
#endif /* LONG_EXTERNALS */

#ifndef LONG_NAMES
# define the_sum SUM
#endif /* LONG_NAMES */

/* sumint, a package for summing integers.
   ...   */

#ifdef INCLUDE_BODY
      COMMON /sumint_common/ the_sum
      INTEGER the_sum
      SAVE /sumint_common/
# ifdef DEFINE_FUNCTIONS
      REAL the_average
# endif /* DEFINE_FUNCTIONS */
#endif /* INCLUDE_BODY */
```

Program 5.5: *An improved version of* `sumint.h`

`#ifdef _alliant_.`

For testing purposes, non-standard features can be disabled by defining `STANDARD_ONLY` prior to including `CF_macros.h`. This will in particular prevent the definition of `LONG_EXTERNALS` and `LONG_NAMES`.

Optimization directives

In order to fully exploit the power of vector computers or multiprocessors, it is often necessary to use compiler directives, telling the compiler that it is safe to optimize a certain loop. These directives vary from compiler to compiler. `CF_macros.h` provides a portable means for the insertion of these directives. It works by including certain files: If a loop can be safely optimized by the compiler, the statement

```
#include O_nodep.h
```

should be used immediately before the loop. Similarly, including `O_vector.h` instructs a vectorizing compiler to vectorize the loop, without exploiting any other forms of parallelism, `O_concur.h` does the same for concurrency, and `O_vect_conc.h` tries to force both, vectorization and concurrency (provided the machine is a vector multiprocessor). `O_noopt.h` can be used to prevent optimization (useful e.g. in the case of nested loops to prevent the compiler from optimizing the \wrong" loop).

C/C++ interface

To solve the problems mentioned in Section 5.1.2, it was necessary to call procedures written in C. Furthermore, other parts of the 3d simulation package that are written in C++ should be able to call the same libraries as **Second**. Therefore it was necessary to provide a portable means to interface FORTRAN to C and C++. This is done by providing the macros `fortran_name` and `fortran_common_block` for declaring FORTRAN entities in C or C++. Their usage is demonstrated in Program 5.6 for a library routine that can be called from either FORTRAN or C.

The macros `fortran_name` and `fortran_common` have two parameters, the all-lower-case and the all-upper-case versions of the name used in the FORTRAN program. This is necessary since on some systems the FORTRAN compiler exports procedure names up-cased, on others down-cased. The same holds for COMMON blocks.

Naturally there is no guarantee that the chosen scheme for the C/FORTRAN interface will work on all computers, not even on all UNIX systems. There are too many possible variations in the way things might be done. The currently implemented scheme works at least on all the UNIX systems we know.

We would like to note here that the present implementation of **Second** does not depend on C code, it is possible to install a pure FORTRAN version|with significantly reduced comfort.

5.2.3 Libraries and Tools

This Section presents the library functions used by **Second**. They fall into three categories: general-purpose FORTRAN utilities (`f_util`), device

```
/* Simple plot module */
#ifndef _FORTRAN_
# ifdef LONG_EXTERNALS
#  define Cur_pos fortran_common(cur_pos,CUR_POS)
#  define Move_to fortran_name(move_to,MOVE_TO)
# else /* ! LONG_EXTERNALS */
#  define Cur_pos fortran_common(drcups,DRCUPS)
#  define Move_to fortran_name(drmove,DRMOVE)
# endif /* ! LONG_EXTERNALS */
fortran_common_block
   struct {INTEGER x, y}
   common_declaration(Cur_pos);
Move_to (INTEGER *x, *y);
#else /* _FORTRAN_ */
# ifndef LONG_EXTERNALS
#  define Cur_pos DRCUPS
#  define Move_to DRMOVE
# endif /* ! LONG_EXTERNALS */
# ifdef INCLUDE_BODY
      COMMON /current_pos/ X, Y
      INTEGER X, Y
      SAVE /current_pos/
# endif /* INCLUDE_BODY */
#endif /* _FORTRAN_ */
```

Program 5.6: *Header file for library routine callable by both, FORTRAN and C*

simulation specific utilities (`sim_util`) and linear algebra kernels and sparse linear solver packages.

General-Purpose FORTRAN Utilities

The subprogram library `f_util` contains procedures that are of general use for FORTRAN programs, as well as routines that support a C/FORTRAN interface on a file level. These library routines are discussed in this section.

Heap To solve the dynamic memory problem (Section 5.1.2) we implemented a heap module. There are basically two ways to implement a heap: Purely in FORTRAN, using a big array as a heap, or outside FORTRAN, e.g. by calling the C function `malloc`. The latter method is, of course, not portable to systems that do not have C, while the former has the disadvantage that the heap is not really dynamic and the total size of a program is still independent of the data.

We therefore decided to implement both methods. The FORTRAN module `heap` allocates storage from a static array. If no more space is available, a C function is called (module `heap_dyn`), which in turn calls `malloc`. On systems where `malloc` is available, the static heap array is made small so that the C routines are used, otherwise the static array is dimensioned big enough and the C interface is disabled.

```
#ifdef ...
#define real_heap single_heap
#else
#define real_heap double_heap
#endif
      SUBROUTINE allocate_real (SIZE, INDEX, ERROR)
         INTEGER SIZE, INDEX, ERROR
      SUBROUTINE deallocate_real (INDEX, ERROR)
         INTEGER INDEX, ERROR
      SUBROUTINE resize_real (NEWSZE, INDEX, ERROR)
         INTEGER NEWSZE, INDEX, ERROR
      SUBROUTINE print_heap_statistics ()
      SUBROUTINE print_heap ()
      SUBROUTINE set_heap_debug (DBGLEV, UNIT)
         INTEGER DEGBLV, UNIT
C
      INTEGER HEAPSZ, DHEAPS
      PARAMETER (HEAPSZ = s_heap_size)
      PARAMETER (DHEAPS = (HEAPSZ+1)/2)
      COMMON /heap_common/ double_heap
      double_precision double_heap (0:DHEAPS-1)
      single_precision single_heap (0:HEAPSZ-1)
      EQUIVALENCE (single_heap,double_heap)
```

Program 5.7: *Simplified heap interface*

Program 5.8 shows a simplified interface to the `heap` module (only showing routines for allocating `normal_precision` data). The allocation function `allocate_real` returns, if successful, an index into the heap array (`real_heap` for type `normal_precision`) which points to the first word of the allocated segment. With `deallocate_real` the allocated storage can be returned, with `resize_real` the size of an allocated segment can be increased or decreased.

Corresponding calls exist for the dynamic allocation of `INTEGER`, `LOGICAL`, `single_precision` or `double_precision` storage.

If the dynamic heap option is used, the allocated storage segment is in general not part of the heap array, rather the address returned by `malloc` is converted into an offset from the address of the array. This only works as long as the system does not perform array bounds checking at run time. Since such checks are normally very expensive, they are not done by most FORTRAN systems. There are exceptions, however, like on the Burroughs B-6700 series where these checks are automatically done by hardware. On such a machine our dynamic memory management would not work, but on \normal" systems there should be no problems.

The `heap` module also provides some check and debug functions. The procedure `print_heap_statistics` prints statistics on the amount of heap storage used and the amount of memory fragmentation. The usage and fragmentation figures only include what was allocated by the `heap` module, if any routines call `malloc` directly this storage will not be counted as \used" (and increase the \fragmentation" figure). As a side-effect `print_heap_statistics` performs a consistency check of the heap segment lists.

The procedure `print_heap` prints the current layout of the heap, i.e. the list of free and used segments. This can be useful for debugging a program that overwrites storage. The procedure `set_heap_debug` turns debugging on or off. A value greater than zero for `DBGLEV` turns on debugging output, a zero or negative value turns debugging off. Debug output is written to the logical unit designated by `UNIT`.

Program 5.8 shows the typical use of heap memory. Macros are used to access dynamically allocated memory using the normal array notation. That way the code looks exactly as if the arrays were \real" ones.

Note that a relatively long name is used for the array macro (`global_...`),

```
#define n_mat 3*n_vec
#define global_vec(i)real_heap(x_vec+i)
#define global_mat(i,j)real_heap(x_mat+3*i+j)
         ...
      INTEGER n_vec, x_vec, x_mat, I, J
         ...
      CALL allocate_real (n_vec, x_vec, ERR1)
      CALL allocate_real (n_mat, x_mat, ERR2)
         ...
      DO 100 I = 0, n_vec-1
         READ (*,*) global_vec(I),
     $             (global_mat(I,J), J=0,2)
 100     CONTINUE
         ...
      CALL deallocate_real (x_vec, ERR1)
      CALL deallocate_real (x_mat, ERR2)
```

Program 5.8: *Typical usage of heap memory.*

to prevent the expanded code from being longer than the original macro call. This is easy to do for one dimensional arrays. For more dimensions the names need to be quite long and become unhandy. We therefore stick with the above used names which are safe in the case of one dimensional arrays and must otherwise be used with care.

Simbad The **Simbad** binary I/O interface facilitates data transfer between C/C++ and FORTRAN programs on the file level. It provides for hardware independent binary files and thus allows moving data in a compressed form between dissimilar computers. **Simbad** is discussed in detail in [40]. The module `simbad.h` implements the **Simbad** specifications.

Smaller utility modules

f_strings The module `f_strings` supplies a few often needed string processing functions. These include a function returning the actual length of a string, and procedures to remove extra spaces in strings, to append strings, and to up- or down-case strings. Finally there is a procedure for extracting the value of a keyword field set up by RCS.

tiny The module `tiny` exports a set of small utilities that help to solve the problems pointed out in Section 5.1.2. Most of them are actually implemented in C, on non-UNIX systems these may return null values of a kind that allow the calling program to continue in a reasonable fashion.

Procedures exported by `tiny` allow programs to inquire command line parameters, the total amount of processor time consumed or remaining, or the current date and time. Other entry points flush output units or remove files. Functions for determining whether or not standard input/output is from/to an interactive terminal are provided, as is a function returning the identification number of the current process. Finally there is a procedure to terminate the program with an indication on the success of the execution (*exit status*), and one to force a controlled program crash (producing information that may be helpful for debugging).

clock The module `clock` provides a convenient interface for timing sections of a program. It exports a type `time_type` and the procedures `init_clock`, `start_clock` and `stop_clock`. To use the clock, a variable of type `time_type` must be declared, which is initialized by passing it to `init_clock`. After that, the clock can be started, stopped, re-started etc. by calling the procedures `start_clock` and `stop_clock`. The latter will return the time since the last time the clock was started, the accumulated time during which the clock was running, and the number of intervals for which it was running.

By defining several `time_type` variables a program can use several different stopwatches to time various parts of code.

arsinh The *arsinh* function is frequently needed in device simulations, but is not part of the set of standard FORTRAN intrinsic functions. Hence the module `arsinh.h` provides a (not particularly accurate) implementation of *arsinh*.

Kernels and linear solvers

blas `blas.h` is a header file providing a generic interface to the *Basic Linear Algebra Subprograms* (BLAS) [48]. These are a set of elementary vector × vector operations. On most vector computers highly optimized implementations of the BLAS are available in some system library. For those systems only the header file `blas.h` is needed to compile the simulator. For systems where no BLAS library is available, ready-to-compile sources are supplied (source subdirectory `blas`).

BLAS routines come in four categories: `REAL`, `DOUBLE PRECISION`, `COMPLEX` and, where available, `DOUBLE COMPLEX`. No complex data types are used throughout the simulator, hence the latter two categories are of no interest to us. For the other ones we require a interface that is consistent with our type `normal_precision`. This is provided by generic name macros like `_axpy`, which expands into `SAXPY` or `DAXPY`, depending on whether `normal_precision` is the same as `REAL` or `DOUBLE PRECISION`. Defining these macros is the main purpose of the header file `blas.h`.

math_aux The module `math_aux` provides simple vector operations that are similar to, but not part of, the BLAS. These include vector assignment operations ($x_i := a$) and ternary vector operations like $z_i := z_i + x_i y_i$. These can be efficiently optimized on vector computers. However, the main reason for their existence is the wish to improve readability of the code by using subroutine calls for such basic operations rather than cluttering the code with many loops.

solvers Finally there are subroutines for the solution of sparse linear systems of equations. These are packages of their own [9, 59] and are not documented here.

Simulation-related utilities

General definitions Certain conventions are required for the efficient internal or external storage of simulation grids. These conventions are defined in **Datex** [41]. For convenient access by programs the conventions are specified in the form of macros in `general.h`.

The macros defined in `general.h` include constants identifying element and face types (*shape codes*), vertex location types and material types. Additional constants like `Max_elt_corners` and `Max_face_corners` are typically used for defining arrays within client modules as well as the `general` module.

Data structures exported by `general` are arrays specifying properties of elements and element faces, such as the numbers as well as the start and end points of edges.

The only procedure exported by the module is an initialization routine that must be called before any of the arrays are being used. In addition, the C/C++ interface defines types that are useful for processing grid data.

data_codes The module `data_codes` contains macros specifying the type codes for various data that may be output from a simulation. This allows using symbolic names rather than the constants defined in [41].

Furthermore the module exports a procedure `get_data_label`, which, given a data type code, returns the factor used to scale data of this kind when externally stored in **Datex** files. The procedure also returns strings for representing the name of the datum in human readable form. This is useful for labelling graphs or tables that are output of various tools.

5.2.4 Program Structure

Figure 5.1 gives a rough representation of the overall structure of **Second**. The meaning of the various symbols is as follows:

A box, like **module:** `procedure`, symbolizes one or more procedures. The bold, colon terminated string denotes the module name while the other string(s) give the procedure name(s). We will frequently make use of the notation *mod*:*proc* to designate that the procedure *proc* is exported from module *mod*.

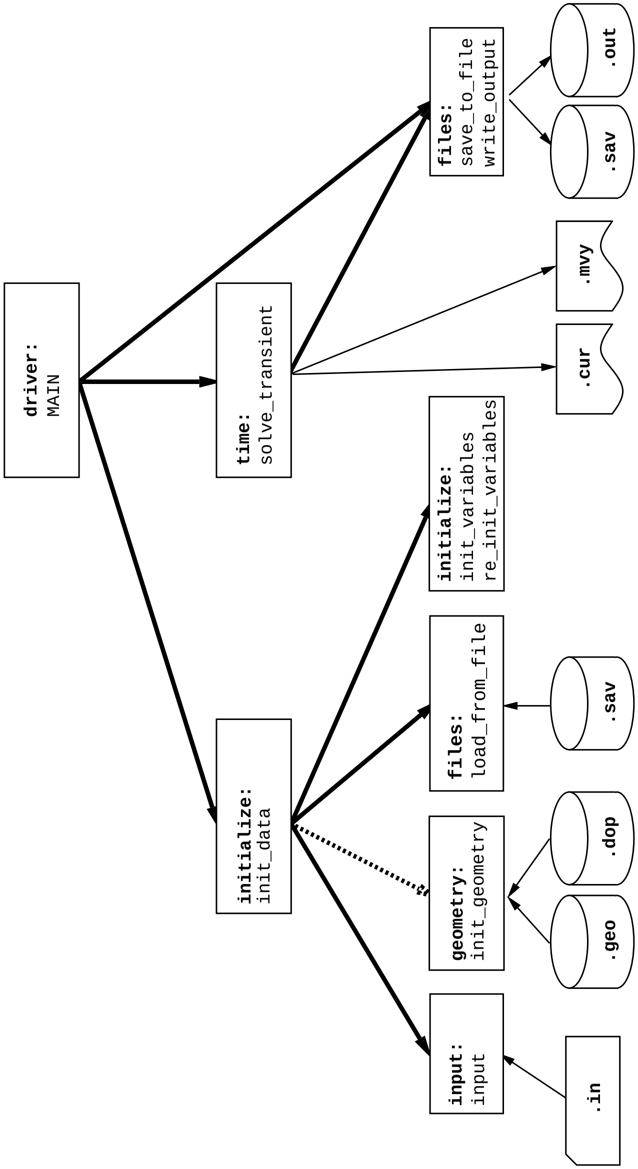

Figure 5.1: *The rough structure of* **Second**

Bold solid arrows symbolize procedure calls (pointing from the caller to the callee), while dotted bold arrows designate indirect calls (i.e. calls through intermediate procedures). Thin solid lines denote data flow, particularly from or to I/O units. The symbols ▭, 🗎, and ⛁ designate such I/O units: input decks, output lists and mass storage units respectively. The symbols are labelled with a string which gives the conventional file name extension for that kind of file. These extensions are used to distinguish within a directory the various files that are used in a particular simulation run.

We use the \input deck" symbol for human readable input files that may come from a small file or directly from the terminal, or may be produced by another program. Similarly, the \output lists" stand for human readable output files. The \mass storage" symbol denotes files that are generally large and not human readable (i.e. stored in binary), such files are used to pass large amounts of data between different programs or between different runs of the same program. Note that some of the \listing files" are also read by other programs, usually for some kind of post-processing.

The input deck that is read by the input processor contains directives to the simulator. Its contents determine which simulation is to be performed, specifying the grid to use, the bias conditions and the physical models to apply etc. It also specifies what kind of output (e.g. for plotting the results) is to be produced. The contents of the input deck, called *parameter input*, is described in the User Manual [42].

The *grid* file (extension `.geo`) and the *doping* file (`.dop`) contain the physical description of the device to be simulated, as well as a simulation grid. These two files are produced by the grid generator Ω. The *result* files (`.out`) are used to plot the results with the graphic tool Picasso [82]. More details on the interaction between **Second** and the various pre- and post-processing tools are given in Section 5.3.

Save files (`.sav`) are used to save the results of one simulation so that a future simulation can continue from the point where an earlier one finished. The *current* file (`.cur`) records, in transient or quasi-stationary simulations, the values of the terminal currents after every time step.

Several *result* files may be written in the case of transient or quasi-stationary simulations, provided the *parameter* file says so. In that case a *movie* file (`.mvy`) records the names of the intermediate result files, together with the simulated time to which they belong.

The procedure `time:solve_transient` is the root of the \real" PDE solver. Its structure is shown in Figure 5.2. Here the dashed boxes refer to several different modules and the thin lines denote important data flow.

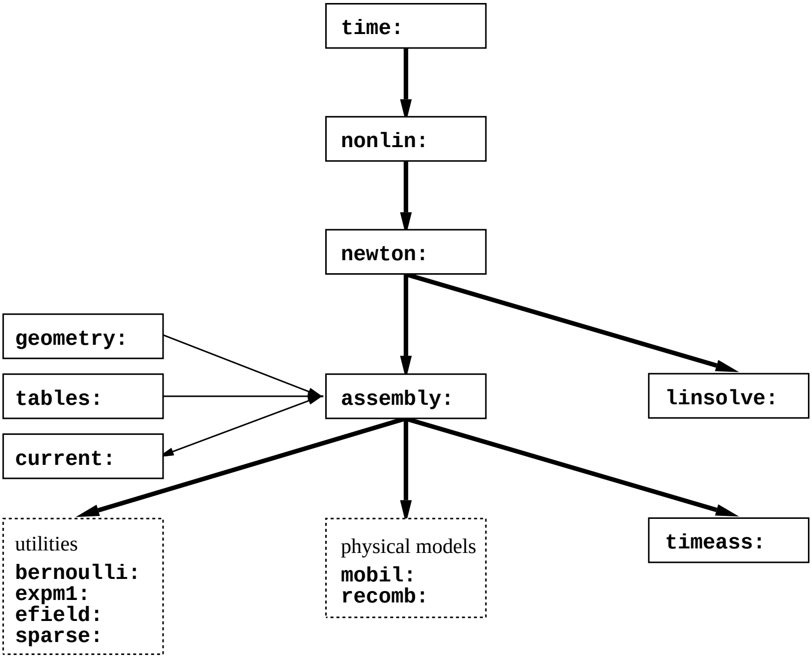

Figure 5.2: *Structure and main data flow of the PDE solver*

The utility modules `bernoulli` and `expm1` are fast and accurate implementations of the Bernoulli and $e^x - 1$ functions, and `sparse` only contains the procedure `_saxp` which multiplies a sparse matrix with a vector. `efield` contains procedures for the accurate computation of the electric field and current density vectors, as explained in Section 4.4.

The module `mobil` computes the carrier mobilities according to one of several available models, while `recomb` computes carrier recombination and generation rates. The other modules will be discussed in the following sections.

5.2.5 Data Structures

The modules `geometry` and `tables` are the main containers of global data for the simulator. Further data of general interest are exported by the modules `par_files`, `par_math`, and `par_physics`. The modules `assembly`, `current`, `efield`, `mobil`, and `recomb` contain data that are of relevance only for a few modules.

Geometry As the name implies, the `geometry` module contains data describing the geometry of the simulated object. Almost every module needs to access some of these data.

Data exported by `geometry` fall in three categories: simple variables, small, fixed size arrays, and large, dynamic arrays. Variables of the first category contain counts like the number of vertices in the grid or the number of contacts in the device. The second kind of variables are quite similar: arrays containing, for example, the number of elements of each possible shape. The third kind contains the actual geometry data, like the coordinates or doping values of the vertices, or the shape codes of the elements.

These arrays are initialized from the *grid* file. *Grid* file information is also used to initialize other data, particularly the arrays exported by `tables`. After this initialization most of the dynamic arrays in `geometry` are no longer required and are hence deallocated.

Reordering of elements and vertices In order to simplify some of the algorithms (particularly in the assembly of the linear equations) and to improve vectorization of several loops, the vertices and elements are reordered while reading the geometry file. With a few (particularly documented) exceptions, all data structures and algorithms of the simulator assume this internal order. The only places where the original order is used are the input and output routines, and some initialization procedures.

The vertices are internally ordered by material: first come all the vertices belonging to a semiconductor material (including those at the interfaces), then all the vertices in the insulator (if any), and finally all the Dirichlet (contact) vertices irrespective of the material. The entries in the array `dom_points` contain the starting point numbers of each of these *domains*. Figure 5.3 illustrates the internal order.

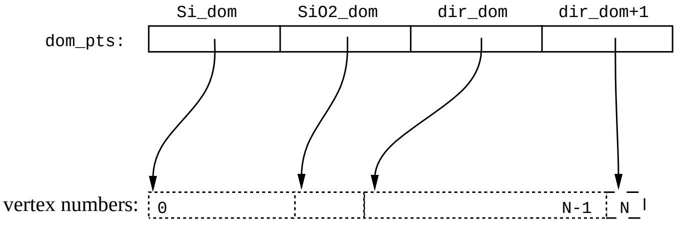

Figure 5.3: *Internal order of vertices*

Similarly the elements are ordered by shape, first all tetrahedra, followed by all pyramids, followed by all prisms, followed by all cuboids. Within each shape the elements are ordered by material. The array `shape_elts` contains pointers to the beginning of each part.

The permutation indices for vertices and elements are contained in the arrays `global_pt_permut` and `global_elt_permut` respectively. These arrays actually contain both, the permutation from external to internal order and the inverse. The element permutation is actually not needed by the program, it is only stored for debugging purposes. At the end of the initialization phase the element permutation array is deallocated. The vertex permutations are needed for outputting results in external order and are therefore kept until the end.

Tables The module `tables` contains the bulk of the global numerical data, most of which falls in two categories: vectors and sparse matrices. Vector data, like the box volumes for each vertex, are straightforward and need not be discussed in detail.

Sparse matrix representation Sparse matrices must be stored in a form that suppresses zero entries, otherwise the memory requirements as well as the time needed for processing the matrices would be unreasonably large. There is, however, no established standard for the representation of sparse matrices.

Since they originate from the box discretization of a 3d mesh, each nonzero off-diagonal entry in our sparse matrices corresponds to an edge in the mesh. Therefore the sparsity structure of our matrices is equivalent to a list of edges. If we have a data structure for edges we can interpret this as a sparse matrix data format.

We choose the following conventions for edges: Each edge is identified by a starting and an ending vertex. For edges between non-Dirichlet vertices, we take as the starting vertex the one with the smaller (internal) number, for edges where at least one of the vertices is a Dirichlet point, the vertex with the larger number is taken as the starting vertex. (We will see the advantages of this convention shortly.) Edges are sorted by ascending starting vertex number, with ascending ending vertex number as the minor sort key. This uniquely determines the order of the edges.

We now define the array `global_edg_index` to contain, at position `i`, the number of the first edge whose starting index is `i`. Two further arrays `global_edg_pt` and `global_edg_oth_pt` contain each edge's starting index (`i`) and ending index (`j`) respectively. Figure 5.4 shows how the off-diagonal elements of a sparse symmetric matrix are stored: the coefficient a_{ij}, which corresponds to the edge from vertex `i` to vertex `j`, is stored in an array (here `global_edg_fact`) at the same position as the indices `i` and `j` in their respective arrays.

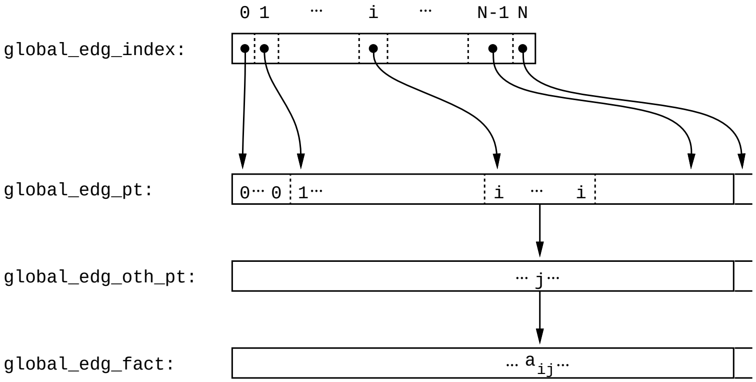

Figure 5.4: *Schematic representation of the sparse data structure*

Note that the array `global_edg_pt` is redundant. However, its availability often allows processing of an entire sparse matrix in a single loop rather than two nested loops (with an additional indirection). The one-loop variant, when usable, leads to code that can be efficiently vectorized. Program 5.9 gives examples of processing matrices with our data structure.

```
--  compute $\mathrm{du}_{ij} = u_i - u_j$
e_0:=global_edg_index[0]
e_1:=global_edg_index[N]
for  e:=e_0  to  e_1-1
  i:=global_edg_pt[e]
  j:=global_edg_oth_pt[e]
  du[e]:=u[i] - u[j]
```

```
--  compute $b_i = \sum_j a_{ij} x_j$
off:=global_edg_index[N] -
     global_edg_index[0]
for  i:=0  to  N-1
  e_0:=global_edg_index[i]
  e_1:=global_edg_index[i+1]
  for  e:=e_0  to e_1-1
    j   :=global_edg_oth_pt[e]
    b[i]:=b[i] + x[j]*a[e]
    b[j]:=b[j] + x[i]*a[e+off]
```

Program 5.9: *Examples of sparse matrix usage: single loop version (left) and nested loop version (right)*

We observe that by virtue of our ordering of the edges, the previously discussed separation of non-Dirichlet and Dirichlet vertices translates into a corresponding separation of edges between non-Dirichlet points and edges that belong to at least one Dirichlet point. This allows for easy and efficient treatment of the boundary conditions (cf. Section 4.2.1). In fact, the order insulator-semiconductor-Dirichlet points would be even more advantageous, since it would automatically separate insulator edges from semiconductor edges and thus save additional `IF` statements in the assembly of the continuity equations. However, this would require a linear solver that allows vertex numbers to start at an arbitrary value, which is not supported by the linear solvers we have at our disposal.

What has been said so far only explains how the off-diagonal coefficients of a symmetric matrix, or the strict upper triangle of a non-symmetric matrix, are stored. The diagonal, which is simply a vector of length N, is either stored separately, or immediately preceding the off diagonals (indices $-N \cdots -1$). For non-symmetric matrices we can either use a two-dimensional array, or simply store the coefficients of the lower triangle after the upper triangle (with a constant offset between the coefficients a_{ij} and a_{ji}). This usage is already demonstrated in Program 5.9. Most internally used sparse matrices are symmetric, moreover most of them have zero diagonals.

Our sparse matrix data format is quite similar to the BLSMP data structure [9], which is a variant of the YSMP format [30]. The main difference between the latter two is that BLSMP makes use of the structural symmetry of the matrices and avoids storing redundant information when dealing with symmetric matrices. Our format differs from BLSMP in the use of zero-relative indices and by avoiding the mixing of pointers and indices in the same array.

Other arrays The coefficient matrix of the linear systems, `assembly:global_lhs`, which is non-symmetric and has a non-zero diagonal, is kept in the BLSMP format dictated by the linear solver. Its sparsity structure (BLSMP's notorious \`JA`" array) is contained in `global_index_list`. The array `global_edg_jac_index` serves to translate between the two data structures: the array contains the BLSMP indices of our edges. The array `global_dphi_dphi` contains the discretized Laplace operator. This one is also kept in BLSMP format, because that way it only needs to be copied when assembling the coefficient matrix.

There are several temporary arrays that contain data in \external" vertex order: `global_raw_e_ndx`, `global_raw_e_oth` etc. These are used to hold the box information supplied in the *grid* file, until enough of the simulator data structures are set up to store the box sections at their final place. This is done at the end of the initialization phase, the temporary arrays are afterwards deallocated.

The arrays `global_vect_trafo` and `global_vect_Si_wgt` are used for the computation of electric fields and current densities along the lines laid out in Section 4.4.

Other modules The module `assembly` exports the arrays to hold the coefficient matrices (LHS) and the residual vectors (RHS) of the sparse linear systems to be solved. The LHS arrays are kept in BLSMP format (see above).

The module `current` exports the array containing the current density vectors for electrons and holes, while `efield` exports the electric field as well as the gradients of the electron and hole quasi-Fermi potentials. Mobilities are exported by `mobil` while recombinations, effective intrinsic densities and the bandgap narrowing values are exported by `recomb`. These modules also export procedures, some of which will be discussed in Section 5.2.6.

5.2.6 Algorithms

Time Integration

```
-- time:solve_transient
if not restarted from transient simulation
    nonlin:solve_nonlinear()
foreach time_interval do
    -- time:time_interval
    time := start_time(time_interval)
    while time<end_time(time_interval) do
        if exceeded resources or step size limit
            exit
        -- time:time_step
        for step := TR_step, BDF2_step
            set the contact voltages for time
            -- time:time_extrapolation
            extrapolate variables from previous to current time
            nonlin:solve_nonlinear()
        estimate the LTE
        if LTE<LTE_limit
            time := time + time_step
            time:write_currents()
        else
            reject time step
        determine new time_step
```

Program 5.10: *Schematic control flow for time integration*

Program 5.10 shows schematically the procedure for the time integration. The comments introduced by \--" indicate which procedure contains the particular section of code. The algorithm follows the method laid out in Section 2.2.

The main work to be performed for the time integration is the solution (space integration) of the semiconductor equations for each individual time point. This solution is not different from the stationary case, except that some terms are added to the arising linear systems of equations. This is done during the linear equation assembly.

Quasi-stationary simulations are basically performed as transient simulations without adding the transient contributions to the linear systems. An artificial time is used to control the speed with which terminal voltages are stepped up.

Nonlinear equation solution For the space integration the procedure `nonlin:solve_nonlinear` is called. This performs a Gummel iteration, calling `newton:newton` for each individual equation, or a coupled solve, calling `newton:newton` once for the full system. In stationary simulations Gummel iterations are always performed, usually followed by a coupled solve. Transient simulations only use coupled iterations, and quasi-stationary simulations can be performed either way.

In order to keep the control over convergence criteria all in one place, `newton:newton` uses the procedure `nonlin:comp_rel_err` for computing the relative error. That procedure is passed as a parameter to `newton:newton`. The procedure `nonlin:extract_old_values` (also passed as a parameter) serves to pass the original values of the variables from `newton` to `nonlin`, so that `comp_rel_err` can compute the relative change due to the last Newton iteration.

Program 5.11 shows the implementation of the damped Newton algorithm. The damping scheme has been discussed in Section 2.3.1, while the control of the linear solver has been described in Section 4.6.2.

A safeguard not shown in Program 5.11 is to impose an upper limit on the damping factor if some components of the solution of the linear system for potential variables (electrostatic or quasi-Fermi potentials) become too big. We limit the damping factor such that no potential component may be updated during a single Newton step by more than approximately one volt, thus avoiding overflow when computing the carrier densities from the updated potentials. This is a rather coarse method that is sufficient for simulations where applied voltages are in the one to ten volts range. For high voltage devices it is inappropriate.

Assembly The real work for the solution of the differential equations is done by the assembly procedures and by the sparse linear solver. The latter is a separate piece of software and is not discussed here in any detail. The assembly procedures `assemble_RHS` and `assemble_LHS` are exported by the module `assembly`.

```
assemble_RHS
rhsnrm  := ‖rhs‖
oldnrm  := rhsnrm
damp    := 1
for it := 1, max_it
   assemble_LHS
   solve_sparse_system (lhs, rhs, dx)
   dxnrm  := ‖dx‖
   damp   := damp / (damp + ((1-damp) * rhsnrm) /
                                (10 * oldnrm))
   oldnrm := rhsnrm
   oldamp := damp
   extract_old_values
   for j := 1 to j_max
      update_vars  -- $x := x + damp * dx$
      assemble_RHS
      rhsnrm := ‖rhs‖
      comp_rel_err
      if converged
          exit
      else if 1-rhsnrm/oldnrm > damp*delta
          exit
      else
          damp := oldamp*(delta/dxnrm)**((j/j_max)**2)
          reset   -- recover old x
   if converged
      exit
```

Program 5.11: *The damped Newton algorithm*

RHS assembly The assembly of the RHS proceeds in several steps. First the stationary RHS are assembled (without the recombination terms in the case of the continuity equations). Next the recombination terms are evaluated and added. This means that in the coupled case the recombination rates are computed only once for both, the electron and hole equations. The next step is to call `timeass:time_RHS` to add the transient contributions, if any. After that the terminal currents are extracted from the assembled RHS. Finally the Dirichlet boundary conditions are incorporated by zeroing the RHS components belonging to Dirichlet vertices (cf. Section 4.2.1).

As explained in Section 4.1.1, assembling the RHS for Poisson's equation

```
e_0 := global_edg_index[0]
e_1 := global_edg_index[N]
for e := e_0 to e_1-1
   i      := global_edg_pt[e]
   j      := global_edg_oth_pt[e]
   fact   := global_edg_fact[e]*(mobil[i]+mobil[j])/2
   tmp[e] := -fact * (dens[j]*B[0,e]-dens[i]*B[1,e])
for i := 0 to N-1
   e_0 := global_edg_index[i]
   e_1 := global_edg_index[i+1]
   for e := e_0 to e_1-1
      j      := global_edg_oth_pt[e]
      RHS[i] := RHS[i] + tmp[e]
      RHS[j] := RHS[j] - tmp[e]
```

Program 5.12: *Simplified RHS assembly for the electron continuity equation (using densities). The Bernoulli function values* `B` *are precomputed*

can be done by multiplying the discretized Laplacian with the solution vector, and adding a few vectors arising from the charge terms, hence the dominating operation is the *sparse matrix-times-vector product*. Program 5.12 shows that the situation is similar for RHS assembly for the continuity equations: First a sparse matrix is computed, then the RHS is obtained by summing the rows of that matrix. The computation of the matrix can be very efficiently vectorized, so this part is rather fast, even though many operations are involved. The second part, the summing of the rows, is just a simplified form of the matrix-times-vector product (the vector has all components equal to one).

The sparse matrix-times-vector type operations do not vectorize well. The inner loop runs over the non-zero entries of the upper triangular part of the matrix, which, owing to the extreme sparsity of our matrices,[2] makes a very short loop. Due to data dependencies, the loops cannot be exchanged either. Therefore this part of the algorithm executes essentially at scalar speed.

LHS Assembly The LHS assembly is quite similar. Looking at Eqs. (4.4, 4.5, 4.9, 4.10), we see that the LHS for the continuity equations has

---

[2] For the grids we are using, a vertex is in average incident in about seven edges, which means that the resulting sparse matrices have in average eight non-zeros per row (including the diagonal), so that a loop over rows in a strict upper triangular matrix has in average 3.5 iterations.

the form

$$(LHS)_{ik} = a_{ik} - \delta_{ik}\left(\sum_{j\neq i} a_{ij} + b_i\right), \qquad (5.1)$$

where a_{ik} is a matrix with a zero diagonal. Assembling the LHS therefore requires the computation of the matrix a_{ik} (which, again, can easily be vectorized), and adding two diagonal contributions, the first being the row sum of a_{ik}, while the second, arising from the recombination terms, is just a vector that does not cause problems. As in the RHS assembly, the summing of the rows of a_{ik} must be done essentially at scalar speed.

Clearinghouse The clearinghouse `clear` serves to avoid redundant evaluations of costly expressions. For example, when assembling the LHS, it is not necessary to recompute the mobilities if the RHS has just been assembled and the values of the unknowns did not change in the meantime.

If a quantity, such as the mobility, needs to be known, the procedure `must_compute` is called with one parameter specifying the quantity to be computed and the other the quantities on which the first quantity depends (i.e. the "*independent quantities*"). The procedure returns `false` if the dependent quantity is up-to-date, otherwise `true` is returned and the dependent quantity must be recomputed. Whenever one of the monitored quantities changes this must be reported to the clearinghouse by calling `notify_change`.

The clearinghouse is implemented by maintaining a "modification time" for each monitored variable. This is an artificial time value that starts with zero and is incremented on each call of `notify_change`. Consequently, `must_compute` only needs to check if any of the independent quantities are "younger" than the dependent one.

Quantities are identified by numbers that are all powers of two. They can therefore be treated as elements of a set. To specify the set of independent quantities, one only needs to add all their identifiers. This makes the clearinghouse very convenient to use. However, it must be kept in mind that FORTRAN does not really support sets—care must be taken, that no identifier is specified twice, since this would result in the wrong "set".

other modules Besides data structures (cf. Section 5.2.5, page 84), `geometry` also contains procedures, only one of them, `init_geometry`, is exported. These procedures do all the input processing of the *grid* and *doping* files, plus the setting up of many of the geometry data structures, including the temporary ones. A corresponding initialization procedure exists in `tables`, this does the remaining initializations of the global data structures, particularly those required for the equation assembly, plus the transformation matrix for the computation of electric fields (cf. Section 4.4). Other modules are initialized by `tables` by calling their respective initialization codes.

The module `current` exports three routines: an initialization routine called by `tables`, and the routines `comp_currents` and `get_cont_cur`. The former extracts the contact currents from the assembled RHS as explained in Section 4.3, and saves them in internal arrays. The latter procedure then returns the stored values.

All the physical models are implemented in `mobil` and `recomb`, the latter module also contains the evaluation of bandgap narrowing. The contributions to the Jacobian (LHS) due to derivatives of the recombination terms are also computed in `recomb`. LHS contributions due to the field dependence of the mobilities and the generation terms are currently ignored. The computation of the electric field, as well as the gradients of the quasi-Fermi levels, is done in `efield`, using the method presented in Section 4.4. Note that the electric field computation, like the assembly routines, contains a poorly vectorizing sparse matrix row sum.

Potential Future Improvements

We pointed out in Section 5.2.6 (page 91) that parts of the assembly procedure are sparse matrix-times-vector operations or row sums of sparse matrices and do not vectorize well. This has so far not been a serious problem, since CPU times of typical simulations are dominated by the linear solves (usually to 70-98 %). However, recent improvements in the solver algorithms have lead to cases where only about 60 % of the time was used for linear solves, assembly and electric field computations being responsible for most of the remainder.

The slow algorithms could be vectorized (and hence speeded up by factors of ten or more) if they could be processed in a different way (e.g. diagonal wise rather than row wise). This is normally not possible since data dependencies would then prevent vectorization completely. However, the rows can be reordered in such a way that data dependencies are avoided and inner loops with a large range result. This reordering is already done in the iterative solver, so using the solver's order could significantly speed up the assembly routines (probably by some factor of two to five).

So far this has not been worth the effort, since assembly times were usually only about 10 % of the total simulation time. However, if progress with the linear solver algorithms continues, it may be a worthwhile task to tackle. An improved linear solver interface is needed though, which would be incompatible with our direct solver (which has proved very important for debugging).

With a new solver interface we would also have the chance to get rid of the multiple sparse data structures that plague the present implementation.

5.2.7 Availability and Portability

Second has been implemented mainly on an Alliant FX-80 minisuper computer. During most of the development phase up-to-date versions were maintained on Cray X-MP and Cray-2 supercomputers, later also on a Convex C-220 minisuper. These were heavily used for testing the code and running examples.

The code was ported to various other UNIX machines, including NEC SX-3 supercomputers, Multiflow Trace and DECsystem mainframes and Sun workstations. An older (FORTRAN only) version ran on Fujitsu VP-200 and VP-2000 series supercomputers under both, the Super-UX (UNIX System V) and MSP (compatible to IBM's MVS) operating systems.

The good portability of the code is underscored by the fact that installation to a UNIX system on which **Second** has never been running before typically takes some three to five hours, including all system dependent parts. Most of this time is typically used up waiting for compilations to finish. The actual task of configuring **Second** for a new system typically takes less than one hour.

5.3 Integration Into a Simulation System

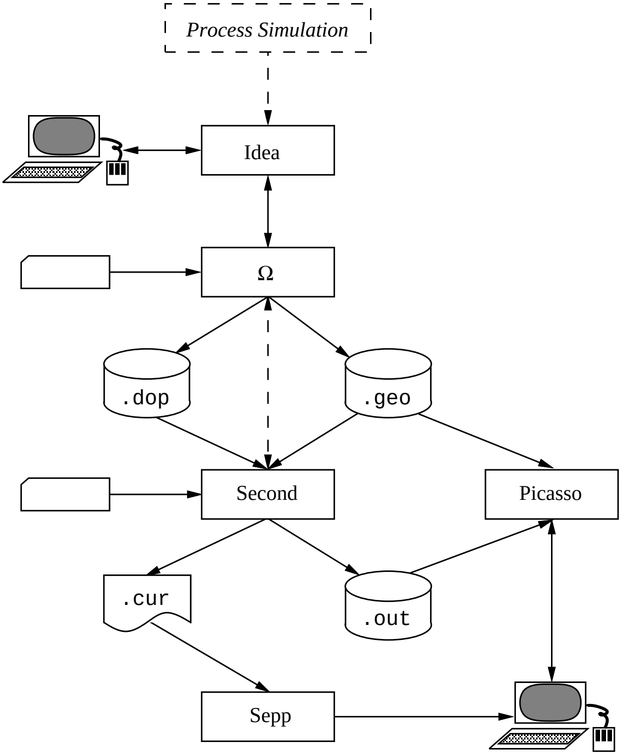

Figure 5.5: *Embedding* **Second** *into a simulation environment*

Figure 5.5 shows how **Second** fits into the 3d device simulation environment of the Integrated Systems Lab at ETH Zürich. **Idea** [80] is an interactive tool that allows the user to construct a device out of simple building blocks. *Process simulation* output can be used to define the doping profiles within the device under construction. (This feature is currently only rudimentary, the corresponding part in Figure 5.5 is therefore dashed.)

The user also supplies to **Idea** information on how the initial simulation grid is to be refined. This grid information, together with the constructed geometry, is used by **Idea** to build an input file for the grid generator Ω [24].

Ω can then generate the grid, which is deposited in the *grid* file (`.geo`), while the corresponding doping information is written to the *doping* file (`.dop`).

The grid and doping information is input to **Second**, together with some *parameter* input specifying e.g. terminal voltages. Simulation results, i.e. the values of physical quantities at the grid points, are written to *result* files (`.out`), while contact currents are written to the *current* file (`.cur`). The dashed line between Ω and **Second** indicates adaptive grid refinement which is not yet implemented.

Two tools exist for the visualization of the simulation results. **Sepp** is a small utility based on `xgraph` that reads the *current* file and plots $I(V)$ curves and similar graphs. **Picasso** [82] is a sophisticated and versatile 2d/3d graphic tool. It uses the *grid* and *result* files to render various representations of simulation data. In particular it allows the user to select an arbitrary view of the simulated object and to view scalar or vector data on the surface of the object. Colours are used to represent magnitudes. To examine data in the device interiour, an arbitrary plane can be used to cut away parts of the object, so that data become visible on the cut face. **Picasso** has proved to be an indispensable tool for both, the interpretation of simulation results and for debugging **Second**. All the plots in Chapter 6 were produced with **Sepp** and **Picasso**.

It is evident from Figure 5.5 that information is transferred between programs mostly through files. This may not always be the best method, however in our case there is not much of an alternative. **Picasso** will typically run on a workstation, while **Second** requires a supercomputer for large simulations, and often needs to be run in the background (e.g. over night). Files are currently the only reasonable way to hand the simulation results from **Second** to **Picasso**.

6

Results

In this chapter we present some typical results of 3d device simulations. These results are meant to show the wide range of possible applications and the flexibility of **Second**.

The first example in Section 6.1 shows a study of parasitic MOSFETs and demonstrates how design rules can be drawn up based on the simulation results. In the next example (Section 6.2) we investigate different CMOS designs with respect to their susceptibility to latchup. Section 6.3 finally presents the examination of the switching behaviour of a bipolar transistor.

6.1 Parasitic MOSFETs

In this section we want to use an example of MOSFET degradation by parasitic devices to demonstrate the necessity and possibilities of 3d simulation.

Figure 6.1 shows an idealized view of a sub-micron n-type MOSFET isolated by an oxide trench.[1] The channel between the source and drain n^+ regions is controlled by a gate which must be imagined to sit on top of the device, between source and drain. Due to the device geometry there exist two parasitic n-MOS devices, both gated through the trench oxide: a lateral

---

[1] This example has been proposed by Marius Orlowski from Motorola Inc., Austin.

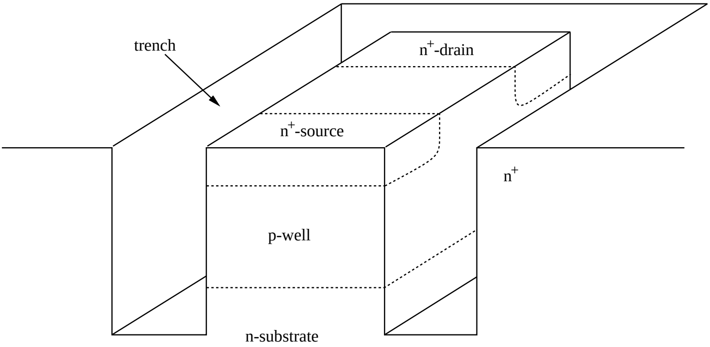

Figure 6.1: *Schematic view of a trench-isolated MOSFET*

parasitic device with the same source and drain as the \proper" device, and a vertical one whose \drain" is the substrate. If the n$^+$ region outside the trench is positively biased with respect to the p-well, a channel can be created at the surface of the trench oxide. This parasitic channel can carry a leakage current which may interfere with the normal transistor operation.

The \channel" of the lateral parasitic MOSFET will be a very thin layer along the (vertical) gate oxide interface, while the \true" transistor channel is, of course, a very shallow layer along the (horizontal) gate oxide interface. Since the two channels are in orthogonal planes it is impossible to simulate their interaction two dimensionally.

For this matter-of-principle investigation we make the simulation problem more manageable with the help of a few simplifications. First we restrict ourselves to examining the effect of the lateral parasitic MOSFET. In addition we note that the device as sketched in Figure 6.1 is symmetrical|hence we ignore the left half. Furthermore we notice that for the operation of the lateral device the front and back parts of the trench do not play any significant role|we ignore everything in front of the source and behind the drain. Since we are only interested in the steady-state, we do not need to simulate the substrate and the n$^+$ region outside the trench. We therefore replace the n$^+$ region by a contact at the outside of the trench oxide, and the substrate by a contact at the bottom of the p-well. To make for a good contact and to suppress the parasitic vertical MOSFET action, we introduce a thin p$^+$ layer at the bottom of the p-well.

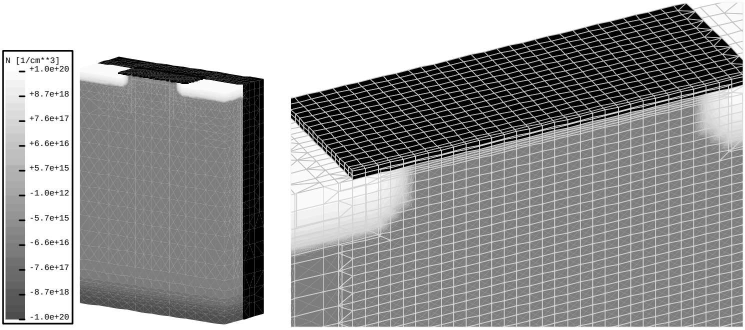

Figure 6.2: *Geometry and doping distribution for the lateral parasitic MOSFET simulation. The plot on the right shows the channel region (including the grid) viewed from the trench after removal of the trench oxide*

The resulting device is shown in Figure 6.2. We used a channel length $L = 0.5\,\mu m$ and a channel width $W = 0.5\,\mu m$ (meaning $0.25\,\mu m$ for our half device). The gate is 150 Å thick and the trench is $T = 0.3\,\mu m$ wide. The p-well doping is $10^{17} cm^{-3}$.

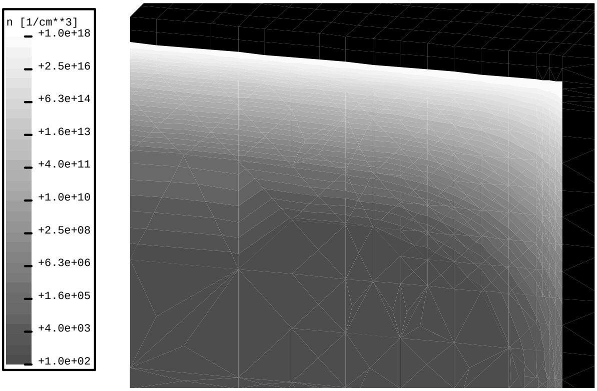

Figure 6.3: *Electron density plot showing the parasitic channel*

Figure 6.3 shows the electron density in the device with a gate voltage of 1 V and 5 V applied to the parasitic gate. The device is cut in the middle of the channel, the cut plane is orthogonal to the direction of the current flow. The parasitic channel can be clearly seen at the trench oxide interface.

In order to study the interaction of the parasitic device with the proper MOSFET we examine the effect of the parasitic gate bias on the *threshold voltage*. The latter we define as the gate bias for which the drain voltage is $10^{-7}\frac{W}{L}$ A. Figure 6.4 shows the result of the simulation: applying a bias of 5 V to the parasitic gate shifts the transistors threshold voltage by approximately 200 mV.

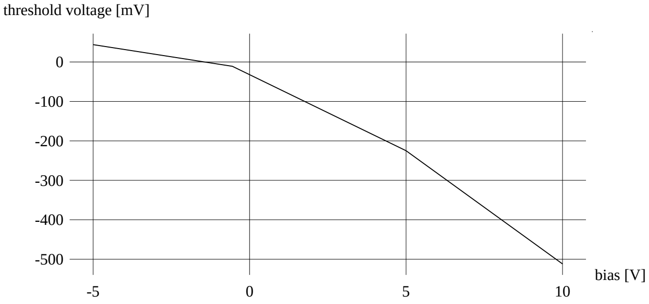

Figure 6.4: *MOSFET threshold voltage as a function of parasitic gate bias*

The device engineer is interested in design rules that ensure save device operation. In our example the question of interest to the device engineer might be: Given a certain value of an acceptable threshold shift, what are the critical values of the device geometry?

To answer this question we varied the trench thickness T from 0.3 to 0.5 μm. For each geometry the shift of the threshold voltage (for increasing the parasitic gate bias from 0 to 5 V) was determined. The result, which is plotted in Figure 6.5, shows that the threshold shift decreases with increasing trench thickness. This is, of course, expected, since the electric field created by the parasitic gate decreases with increasing T.

These simulations were performed with grids consisting of between 15 000 and 24 000 vertices. CPU times on a 6 processor Alliant FX-80 mini-supercomputer were on the order of 20 minutes per bias point, when

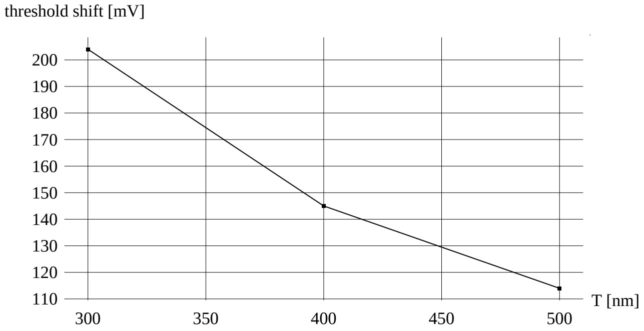

Figure 6.5: *Threshold voltage shift as a function of trench width*

the bias was stepped up smoothly. Approximately five to ten bias steps were normally used to determine the threshold voltage for a given voltage applied to the parasitic gate. The working point shown in Figure 6.3 was run from scratch (without stepping up voltages) in just over an hour. On a Cray-2 supercomputer the execution times are typically faster by a factor of five.

6.2 CMOS Latchup

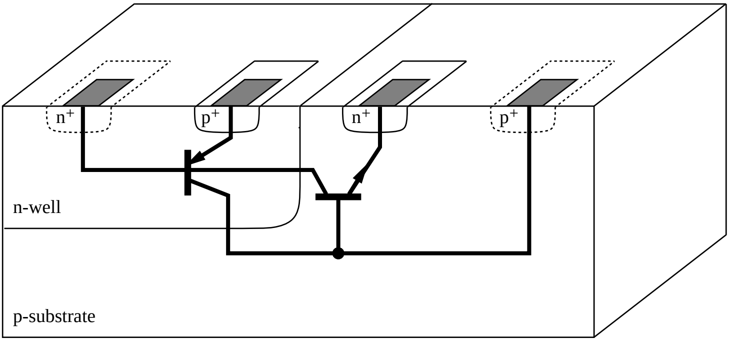

Figure 6.6: *Simplified CMOS latchup structure*

Latchup is an effect in CMOS devices where different parasitic bipolar transistors lock each other in a high current on-state. Once latchup occurs the device will be destroyed within microseconds due to the excessive heat development. Shrinking device dimensions ease device cross-talk and thus make latchup more likely. The avoidance of latchup is a major design goal when trying to produce even smaller devices. Simulation can be a extremely helpful in setting up latchup-proof design rules.

There are various possibilities for latchup to occur. We concentrate on a part of a CMOS inverter, which is the simplest configuration susceptible to latchup. Figure 6.6 shows an idealized sketch of the devices. The p^+ diffusion, the n-well, and the p-substrate form a vertical pnp transistor, while the n^+ diffusion, the substrate and the n-well form a lateral npn transistor. Both are coupled in a thyristor-like fashion.

Normally, both bipolar transistors are turned off since under normal operating conditions the substrate is biased at 0 V by the n^+-plug and the p-well is similarly held at V_{SS}. The contacts at the n^+- and p^+-diffusions are always between 0 V and V_{SS}, so that the base-emitter diode is never forward biased. However, a voltage glitch at one of the two emitters can turn on the corresponding transistor. If the transistor is conducting enough current for a sufficient amount of time, its collector current may cause a voltage drop along the other transistor's base-emitter diode high enough to turn on that transistor as well. If the two transistors have sufficient gain, they will lock each other in high injection mode even as the glitch that triggered the process is over, and the devices are latched.

In our experiments we always tried to induce latchup by applying a negative voltage pulse of 0.85 V to the p^+-diffusion, thus turning on the lateral npn transistor. The length of the pulse varied, while the steepness of its flanks was kept constant (1 ns rising time). Figure 6.7 shows the impurity concentrations for the basic configuration, which is in 1 μm technology, featuring a very shallow (1.35 μm) n-well. Minimum distances between active regions occur across tub edges.

The first investigation (see also [43]) was a comparison between 2d and 3d simulations. The former were also performed with **Second**, but a \quasi-2D" grid was obtained by replicating the front layer of the original 3d grid once in the third dimension. The 3d grid consisted of 56 562 vertices while the resulting \2D" grid had 2 × 2 247 points.

The result of the simulation with the triggering voltage pulse held for

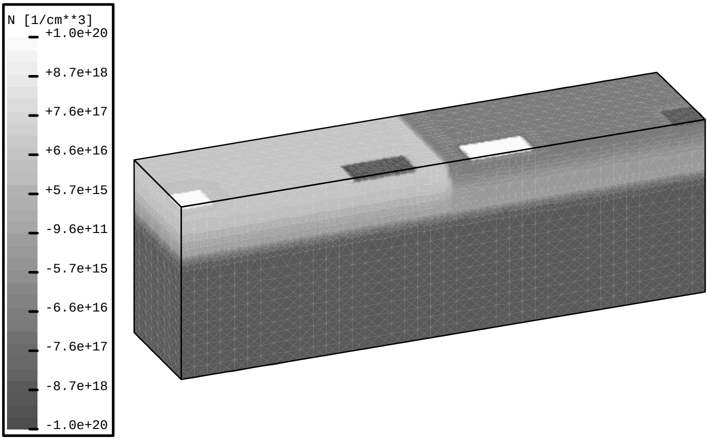

Figure 6.7: *Impurity distribution for basic CMOS latchup example*

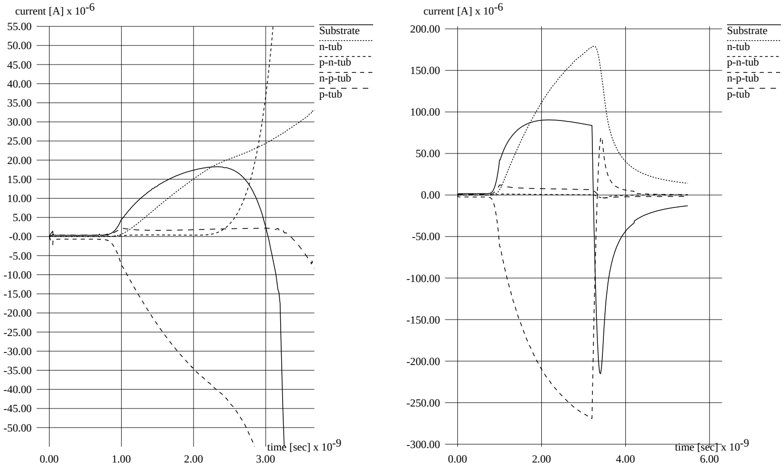

Figure 6.8: *Contact currents for 2d (left) and 3d (right) simulations*

2.2 ns is shown in Figure 6.8. The curve labels \p-n-tub" and \n-p-tub" refer to the p^+ and n^+ diffusions respectively. It can be clearly seen that according to the 2d simulation latchup occurs just as the voltage pulse is falling off after 3.2 ns, while according to the 3d simulation, the npn transistor turns off very quickly once the applied voltage goes towards zero. The voltage pulse needed to be held for more than 5 ns for the 3d simulation to indicate latchup. This is an example of the significant differences that can occur between two and three dimensional simulations.

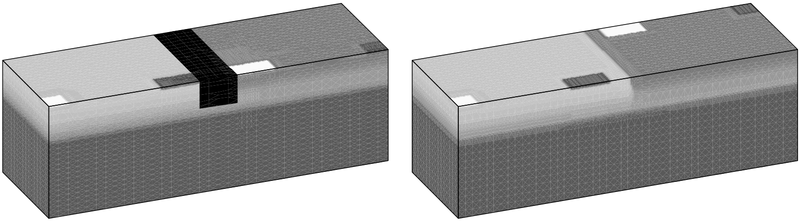

Figure 6.9: *Doping for the trench isolated (left) and shifted (right) CMOS structures. The black structure is the oxide*

Next we examined the effect of two design measures for inhibiting latchup [22]. One was to introduce a small oxide trench to isolate the active regions of the p-MOS and the n-MOS devices while the other was to shift one transistor in the third dimension (Figure 6.9). Figure 6.10 shows the currents for all three cases when the pulse was held for 10 ns. The curves clearly show that both measures effectively prevented latchup in this case.

These simulations were performed partially on a Cray-2 supercomputer and partially on a Convex C-220 minisuper. Typical run times were a few hours on the Cray-2 or days on the Convex for grid sizes ranging from 18 500 to 24 000 points. Memory usage was of the order of 240 Mbytes.

6.3 Transistor Switching

Bipolar transistors can be made to switch much faster than CMOS devices, at the expense of a higher power consumption (and hence heat dissipation). For that reason they are used in cases where speed matters more than price, e.g. for supercomputers.

Here we examine the switching behaviour of a high-speed ECL transistor.

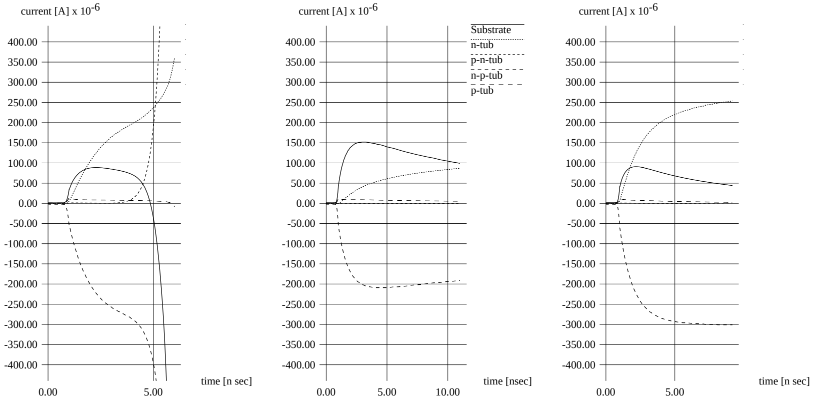

Figure 6.10: *Contact currents for three CMOS structures*

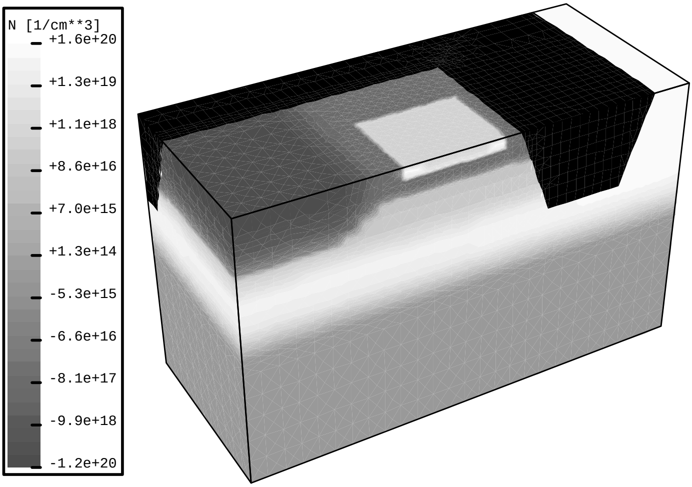

Figure 6.11: *Doping distribution for an ECL transistor*

Figure 6.11 shows the doping distribution within the device. Also visible is an insulating oxide trench. The emitter is the bright region in the middle top portion of the device, the base is contacted at the left and the collector contact is at the right. The collector current must therefore flow around the trench. Note the slanted walls of that trench, which **Second** can handle without problems. We know of no other program that can simulate such a device geometry.

We simulated the current waveform of the transistor in a common emitter configuration. The collector-emitter voltage, V_{CE}, was kept constant at 5 V and the base-emitter bias, V_{BE}, was initially at 0.8 V, so that the transistor was fully turned on. At $t = 0$ the device was being turned off by ramping V_{BE} to 0 V within 200 ps. At $t = 300$ ps the base voltage was again turned on, reaching its old value of 0.8 V at $t = 500$ ps. The simulation was then carried on for another 500 ps.

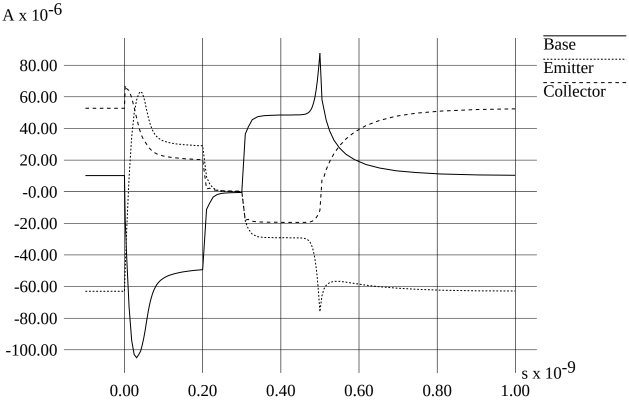

Figure 6.12: *ECL transistor switching waveform*

Figure 6.12 shows the computed contact currents. The discontinuities at 0, 200, 300, and 500 ps are due to the displacement current, which is discontinuous because of the discontinuous derivative of the applied voltage. The terminal currents are dominated by the displacement current after approx. 70 ps, meaning that the transistor is basically turned off at that time.

The picture is somewhat different when the device is switched on: The terminal current are completely dominated by the displacement current until the base voltage has reached its final value ($V_{BE} = 0.8$ V). After that, approx. 120 ps are needed until the collector current comes close to reaching the steady state value.

Such a simulation may be used to optimize the switching time of the transistor. To this end it is useful to observe the dynamic operation of the device by monitoring physical entities in the device interiour. Figure 6.13 shows a plot of the magnitude of the electron current density in the transistor after 100 ps, that is halfway through the switch-off phase. The plot shows that the active region of the transistor is already basically free of current, while carriers are still travelling through the highly doped collector and emitter regions. This clean-out time is obviously responsible for the latency of the switch. Pictures like this one can be very valuable for optimizing dynamic device characteristics.

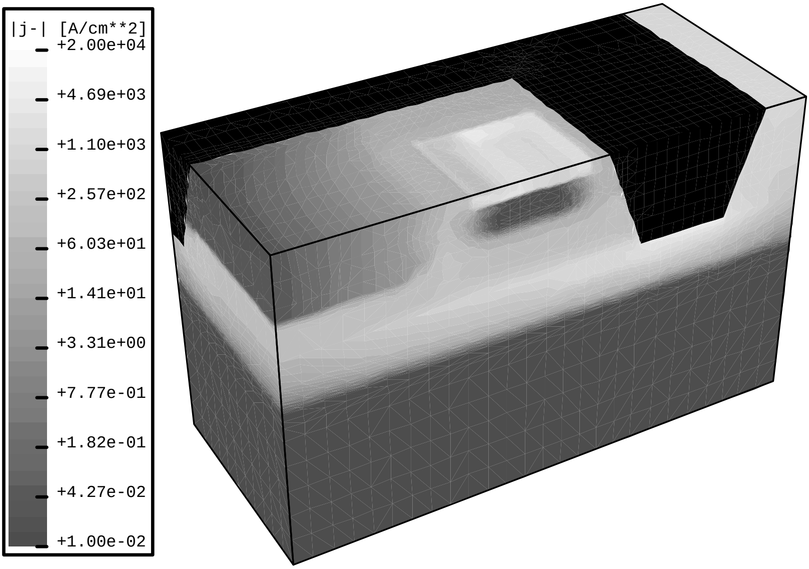

Figure 6.13: *Magnitude of the electron current density in the transistor during switch-off*

These computations were performed using a grid with 17 770 vertices. The simulations were run on a six processor Alliant FX-80 in about one day. Memory usage was 100 Mbytes. Similar runs took about five hours on a Cray-2.

7

Conclusions and Future Work

In this thesis we discussed the difficulties associated with the three dimensional simulation of general semiconductor devices. We presented the design and implementation of **Second**, a program that can be used to perform such simulations. We demonstrated the programs usefulness by applying it to a variety of different problems involving significantly differing geometries and operating conditions.

The results presented in Chapter 6 are meant to give an indication of the problems that can be approached with **Second**. However, they are only a small selection of a wide range of possible applications.

Our examples prove that 3d simulations, even transient, are possible with rather general device geometries. The examples also show, however, that these simulations are quite expensive in terms of CPU time and memory requirements. Typical simulations run for hours on modern supercomputers or days on minisupers. Optimizing a device generally requires many single simulations, increasing computer time requirements by another order of magnitude. Memory requirements are in the hundreds of megabytes range, which calls for big machines. It therefore appears safe to say that 3d simulation, while being a necessity for many problems, is not yet in a position to completely displace 2d simulation.

On the other hand, even running a supercomputer for days on a single problem is cheap compared to the cost of going through one more iteration of chip manufacturing, which can take months and cost millions. One has to keep that in mind when looking at the costs of 3d simulation. We would also like to add that the high costs are *not* a result of our general approach, the price we pay for generality is the size and complexity of the code, not the amount of computer time. Indeed, our general 3d grids often allow us to work with fewer mesh points than would be required by other programs, and hence save computer time.

Second meets its design goals of being a \general purpose device simulator" in as far as it permits the simulation of general plane faced geometries|the generality of the possible device geometries is only limited by what the grid generator can supply. No other 3d device simulator published so far supports such geometrical generality.

In other respects true generality is not yet achieved. Further improvements are mainly possible in two ways: adaptive grid refinement can improve speed and reliability of the simulation, and improved physical models can increase the domain of problems that can be tackled with **Second**.

Adaptive grid refinement offers the possibility to simulate with grids that are better adjusted to the problem than grids that are generated based on the doping information (and possibly additional hints by the user). Implementation of this feature requires two things: an improved interface between **Second** and the grid generator must enable the former to instruct the latter where the point density is to be increased or reduced, also some means must be available to interpolate data from the original to the revised grid. On the other hand, criteria must be found so the simulator can determine where it needs more grid points and where it needs less. As Bürgler [13] has shown, this problem is not easy to solve. The solution Bürgler has given is based on his new discretization scheme and is not applicable to the BM. More theoretical work must be done on error estimates before a good implementation of adaptive grid refinement is possible.

Various improvements are possible in the way physical devices are modelled. Generalized boundary conditions should include external resistors, capacitors and inductances, as well as current controlled (in addition to voltage controlled) contacts. The treatment of external magnetic fields must be possible for the simulation of magnetic field sensors. Special devices types, like thyristors, may require more sophisticated mobility and recombination models. Particularly power devices require the solution of extra equations for carrier and lattice temperature [81]. Finally, the recent interest in heterostructures creates a demand for their simulation in 3d (cf. [49]).

Most of these extensions follow work that has been done in 2d, since the physical models do not depend on the dimension. However, since better models generally imply an increase of the number of calculations to be done, computer speed poses a limit on what can be done|particularly for 3d simulations which are anyway at the edge of what is feasible with present-day computers.

It should finally be pointed out that there are a number of possible improvements in parts of the simulation system that are not discussed in this thesis. Improved grid generators can lower simulation costs by reducing the number of grid points or by allocating them in a way that results in better conditioned linear systems. In addition we can expect significant reductions in simulation costs from improved linear solver algorithms. During the development of **Second** we already experienced a dramatic improvement in the available iterative linear equation solvers, and there is good reason to expect further progress. Finally, the dramatic increase in computer speed and decrease in computer prices will certainly help to transform 3d simulation into a standard engineering tool.

List of Figures

0.1 Spreading of current at device edges causes 3d effects. 2

1.1 Simple MOSFET structure showing silicon, oxide, and contacts. 9

2.1 2D example of a box 21

2.2 Memory requirements of direct and iterative solvers as a function of problem size 33

4.1 Schematic view of Dirichlet and non-Dirichlet regions in the linear system of equations 47

4.2 Box of a boundary point 48

4.3 Electric field vector within an element 50

5.1 The rough structure of **Second** 81

5.2 Structure and main data flow of the PDE solver 83

5.3 Internal order of vertices 85

5.4 Schematic representation of the sparse data structure . 86

5.5 Embedding **Second** into a simulation environment . . . 96

6.1 Schematic view of a trench-isolated MOSFET 100

6.2 Geometry and doping distribution for the lateral parasitic MOSFET simulation 101
6.3 Electron density plot showing the parasitic channel . . . 101
6.4 MOSFET threshold voltage as a function of parasitic gate bias 102
6.5 Threshold voltage shift as a function of trench width . . 103
6.6 Simplified CMOS latchup structure 103
6.7 Impurity distribution for basic CMOS latchup example . 105
6.8 Contact currents for 2d and 3d simulations 105
6.9 Doping for the trench isolated and shifted CMOS structures 106
6.10 Contact currents for three CMOS structures 107
6.11 Doping distribution for an ECL transistor 107
6.12 ECL transistor switching waveform 108
6.13 Magnitude of the electron current density in the transistor during switch-off 109

List of Programs

5.1 `sumint.h` 67

5.2 `sumint_int.h` 68

5.3 `sumint.F` 69

5.4 Client module 70

5.5 An improved version of `sumint.h` 72

5.6 Header file for library routine callable by both, FORTRAN and C 74

5.7 Simplified heap interface 75

5.8 Typical usage of heap memory. 77

5.9 Examples of sparse matrix usage: single loop version and nested loop version 87

5.10 Schematic control flow for time integration 89

5.11 The damped Newton algorithm 91

5.12 Simplified RHS assembly for the electron continuity equation 92

List of Tables

1.1 De Mari scaling factors for T = 300 K. 12

1.2 Fundamental and material constants 13

1.3 Mobility parameters for electrons and holes 16

1.4 Parameters for Shockley-Read-Hall recombination model 17

1.5 Grant's coefficients for the impact ionization model after Chynoweth 18

5.1 Macros defining machine characteristics for use by FORTRAN and C programs 71

Bibliography

[1] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. Supercomp. Appl.*, 3(3):41{59, 1989.

[2] ANSI X3.9-1978. *American National Standard Programming Language FORTRAN*.

[3] ANSI X3J3. Fortran 8X draft. *ACM SIGPLAN Fortran Forum*, 8(4), 1989.

[4] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Inter. J. Supercomp. App.*, 1(4):10{30, 1987.

[5] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Numer. Math.*, 4:420{453, 1963.

[6] J. W. Backus et al. The FORTRAN automatic coding system. In *Proc. West Joint Comp. Conf.*, 1957.

[7] R. E. Bank, W. M. Coughran, Jr., W. Fichtner, E. H. Grosse, D. J. Rose, and R. K. Smith. Transient simulation of silicon devices and circuits. *IEEE Trans.*, CAD-4:436{451, 1985.

[8] R. E. Bank and D. J. Rose. Global approximate Newton methods. *Numer. Math.*, 37:279{295, 1981.

[9] R. E. Bank and R. K. Smith. General sparse elimination requires no permanent integer storage. *SIAM J. Sci. Stat. Comput.*, 8:574{584, 1987.

[10] W. E. Beadle, J. C. C. Tsai, and R. D. Plummer, editors. *Quick Refernce Manual for Silicon Integrated Circuit Technology*. Solid State Physics. John Wiley & Sons, New York, 1985.

[11] W. Bergner and R. Kircher. SITAR|an efficient 3 d-simulator for optimization of non-planar trench structures (DRAMs). In G. Baccarani and M. Rudan, editors, *SISDEP 3*, pages 165{74. Alma Mater Studiorum Saecularia Nona, Italy, 1988.

[12] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Harri Deutsch, 20th edition, 1981.

[13] J. F. Bürgler. *Discretization and Grid Adaptation in Semiconductor Device Modeling*. PhD thesis, ETH-Zürich, 1990. Publ. by Hartung-Gorre Verlag, Konstanz, Germany.

[14] E. M. Buturla, P. E. Cottrell, B. M. Grossman, K. A. Salsburg, M. B. Lawlor, and C. T. McMullen. Three-dimensional finite element simulation of semiconductor devices. In *ISSCC*, pages 76{77. IEEE, 1980.

[15] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques*, Bruxelles, 1970. NATO Science Commitee.

[16] D. M. Caughey and R. E. Thomas. Carrier mobilities in silicon empirically related to doping and field. *Proc. IEEE*, pages 2192{93, 1967.

[17] J.-H. Chern, J. T. Maeda, L. A. Arledge, and J. P. Yang. SIERRA: a 3-D device simulator for reliability modeling. *IEEE Trans.*, CAD-8:516{27, 1989.

[18] A. G. Chynoweth. Ionization rates for electrons and holes in silicon. *Phys. Rev*, 109:1537{1540, 1958.

[19] P. Ciampolini, A. Pierantoni, M. Melanotte, C. Cecchetti, C. Lombardi, and G. Baccarani. Realistic device simulation in three dimensions. In *IEDM*, pages 131{134, 1989.

[20] E. R. Cohen and B. N. Taylor. The fundamental physical constants. *Physics Today*, Aug 1989. After the 1986 CODATA report.

[21] P. Conti. *Grid Generation for Three Dimensional Device Simulation*. PhD thesis, ETH-Zürich, 1991. Publ. by Hartung Gorre Verlag, Konstanz, Germany.

[22] P. Conti, G. Heiser, and W. Fichtner. Three dimensional transient simulation of complex silicon devices. *Jap. J. Appl. Phys. Letters*, 29(12), 1990.

[23] P. Conti, N. Hitschfeld, and W. Fichtner. Ω { an octree-based mixed element grid allocator for adaptive 3D device simulation. IEEE Trans. CAD ICAS, to appear.

[24] P. Conti and H. Wachter. Ω *User Manual*. Integrated Systems Lab, ETH-Zürich, 1990.

[25] W. M. Coughran, Jr., E. Grosse, and D. J. Rose. CAzM: A circuit analyzer with macromodelling. *IEEE Trans.*, ED-30:1207{1213, 1983.

[26] G. Dahlquist. A special stability problem for linear multistep methods. *BIT*, 3:27{43, 1963.

[27] R. Dang, N. Shigyo, T. Wada, S. Onga, and M. Konaka. An analysis of MOSFETs' narrow-channel effects. In *ICCC 82*, pages 286{9. IEEE, 1982.

[28] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package II: The nonsymmetric codes. Research Rep. 114, Yale Univ. Comp. Sci. Dept., 1977.

[29] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package I: The symmetric codes. *Int. J. Numer. Methods in Eng.*, 18:1145{1151, 1982.

[30] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Considerations in the design of software for sparse gaussian elimination. In *Sparse Matrix Computations*. Academic Press, 1976.

[31] W. Fichtner, D. J. Rose, and R. E. Bank. Semiconductor device simulation. *IEEE Trans.*, ED-30:1018{30, 1983.

[32] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Proc. of the Dundee Biennal Conference on Numerical Analysis*, pages 73{89, New York, 1975. Springer-Verlag.

[33] A. F. Franz, G. A. Franz, S. Selberherr, C. Ringhofer, and P. Markowich. Finite boxes|a generalization of the finite-difference method suitable for semiconductor device simulation. *IEEE Trans.*, ED-30:1070{1083, 1983.

[34] S. P. Gaur, G. R. Srinivasan, and I. Antipov. Verification of heavy doping parameters in semiconductor device modeling. In *IEDM*, pages 276{79, 1980.

[35] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall, 1971.

[36] W. N. Grant. Electron and hole ionization rates in epitaxial silicon and high electric fields. *Solid State Elec.*, 16:1189{1203, 1973.

[37] H. K. Gummel. A self-consistent iterative scheme for one-dimensional steady state transistor calculations. *IEEE Trans.*, ED-27:1520{1532, 1964.

[38] W. Hänsch and S. Selberherr. MINIMOS 3: A MOSFET simulator that includes energy balance. *IEEE Trans.*, ED-34:1074{8, 1987.

[39] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, J. Heny G. Thacher, and C. Witzgall. *Computer Approximations*. Robert E. Krieger, Huntington, New York, 1978.

[40] G. Heiser. SIMBAD { a simple binary data format. Technical Report 88/24, Integrated Systems Lab, ETH-Zürich, 1988. Specification and user's guide.

[41] G. Heiser. Device simulation: Supplementary documents. Technical Report 90/10, Integrated Systems Lab, ETH-Zürich, July 1990. Contains *DATEX format specification* and *Conventions for grid descriptions*.

[42] G. Heiser and K. Kells. **Second** user manual. Technical Report 90/12, Integrated Systems Lab, ETH-Zürich, July 1990. Version 2.0.

[43] G. Heiser, C. Pommerell, J. Weis, and W. Fichtner. Large-scale device simulation: Algorithms, computer architectures, results. IEEE Trans. CAD ICAS, to appear.

[44] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409{436, 1952.

[45] J. D. Jackson. *Classical Electrodynamics*. John Wiley & Sons, 2^{d} edition, 1975.

[46] R. Kasai, K. Yokoyama, A. Yoshii, and T. Sudo. Threshold-voltage analysis of short- and narrow-channel MOSFETs by three-dimensional computer simulation. *IEEE Trans.*, ED-29:870{6, 1982.

[47] J. Lambert. *Computational Methods in Ordinary Differential Equations*. Wiley, 1973.

[48] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM TOMS*, 5:308{323, 1979.

[49] P. Lindorfer and S. Selberherr. GaAs MESFET simulation with MINIMOS. In *11th Annual GaAs IC Symposium*, pages 277{80. IEEE, 1989.

[50] A. de Mari. An accurate numerical steady-state one-dimensional solution of the p-n junction. *Solid-State Electron.*, 11:33{58, 1968.

[51] H. Masuda, T. Toyabe, T. Haguwara, and Y. Ushiro. High-speed three-dimensional device simulator on a super computer: CADDETH. In *International Symposium on Circuits and Systems*, pages 1163{6. IEEE, 1984.

[52] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. of Comp.*, 137:148{162, 1977.

[53] J. A. Meijerink and H. A. van der Vorst. Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems. *J. Comp. Phys.*, 44:134{155, 1981.

[54] S. Odanaka, M. Wakabayashi, H. Umimoto, A. Hiroki, K. Ohe, K. Moriyama, H. Iwasaki, and H. Esaki. Smart: three-dimensional process/device simulator integrated on a super-computer. In *Intern. Symp. Circ. Syst.*, pages 534{7. IEEE, 1987.

[55] S. Onga, N. Shigoyo, M. Yoshimi, and K. Taniguchi. Analysis of submicron MOS device characteristics using a composite full three-dimensional process/device simulation system. In *Symp. VLSI Techn.*, pages 15{16. Japan Soc. Appl. Phys., Tokyo, Japan, 1986.

[56] M. R. Pinto, C. S. Rafferty, and R. W. Dutton. *PISCES-II User's Manual*. Stanford University, 1984.

[57] M. R. Pinto. *Comprehensive Semiconductor Device Simulation for Silicon ULSI*. PhD thesis, Stanford, 1990.

[58] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press, 1984.

[59] C. Pommerell and W. Fichtner. PILS: An iterative linear solver package for ill-conditioned systems. Subm. Supercomputing '91 (ACM), March 1991.

[60] C. S. Rafferty, M. R. Pinto, and R. W. Dutton. Iterative methods in semiconductor device simulation. *IEEE Trans.*, ED-10:2018{2027, 1985.

[61] W. van Roosbroeck. Theory of flow of electrons and holes in germanium and other semiconductors. *Bell System Tech. J.*, 29:560{607, 1950.

[62] Y. Saad. The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.*, 19:484{506, 1982.

[63] K. A. Salsburg, P. E. Cottrell, and E. M. Buturla. FIELDAY|finite element device analysis. In P. Antognetti, D. A. Antoniadis, R. W. Dutton, and W. G. Oldham, editors, *Proc. Dev. Sim MOS-VLSI Circ.*, pages 582{619. Martinus Nijhoff, The Hague, Netherlands, 1983.

[64] D. Scharfetter and H. K. Gummel. Large-signal analysis of a silicon Read diode oscillator. *IEEE Trans.*, ED-16:64{77, 1969.

[65] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer, 1984.

[66] N. Shigyo and R. Dang. Analysis of an anomalous subthreshold current in a fully recessed oxide MOSFET using a three-dimensional device simulator. *IEEE Trans.*, ED-32:441{5, 1985.

[67] J. W. Slotboom. The pn-Product in Silicon. *Solid-State Electron.*, 20:279{83, 1977.

[68] G. D. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford, 1978.

[69] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36{52, 1989.

[70] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice Hall, 1973.

[71] S. M. Sze. *Physics of Semiconductor Devices*. John Wiley & Sons, 2nd edition, 1981.

[72] E. Takeda, K. Takeuchi, D. Hisamoto, T. Toyabe, K. Ohshima, and K. Itoh. A cross section of alpha -particle-induced soft-error phenomena in VLSIs. *IEEE Trans.*, ED-36:2567{75, 1989.

[73] M. Thurner and S. Selberherr. The extension of MINIMOS to a three dimensional simulation program. In *Proc. of NASECODE V Conf.*, pages 327{332, 1987.

[74] T. Toyabe, H. Masuda, Y. Aoki, H. Shukuri, and T. Hagiwara. Three-dimensional device simulator CADDETH with highly convergent matrix solution algorithms. *IEEE Trans.*, ED-32:2038{44, 1985.

[75] M. Turner, P. Lindorfer, and S. Selberherr. Numerical treatment of nonrectangular field-oxide for 3-d MOSFET simulation. *IEEE Trans.*, CAD-9:1189{1197, 1990.

[76] T. Uenoyama, S. Odanaka, and T. Onuma. Analysis of narrow channel effect in small-size GaAs MESFET. In W. T. Lindley, editor, *Intern. Symp. GaAs*, pages 447{52. IEEE Electrochem. Soc, 1986.

[77] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, 1962.

[78] VENUS Systemplanung. *VENUS 2 Projektdokumentation*. Siemens ZTI, München, Oct 1986.

[79] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of CG{S for the solution of nonsymmetric linear systems. Subm. SIAM J. Sci. Stat. Comput.

[80] H. Wachter. *Idea User Manual*. Integrated Systems Lab, ETH-Zürich, 1991.

[81] G. K. Wachutka. Rigorous thermodynamic treatment of heat generation and conduction in semiconductor device modeling. *IEEE Trans.*, CAD-9:1141{1149, 1990.

[82] M. Westermann. *Picasso Reference Manual Version 1.0*. Integrated Systems Lab, ETH-Zürich, February 1990.

[83] K. Yamaguchi. A mobility model for carriers in the MOS inversion layer. *IEEE Trans.*, ED-30:658{63, 1983.

[84] A. Yoshii, S. Horiguchi, and T. Sudo. A numerical analysis for very small semiconductor devices. In *Intern. Solid-State Circ. Conf.*, pages 80{1. IEEE, 1980.

Index

A stability 27
abstract data type 60
acceptor 5
Algol-60 58
algorithm 4, 25, 29, 32, 38, 84, 89, 92, 94f
  critical 59
  iterative 54
  minimum degree 33
  Newton 91
  numerical 62
  reverse Cuthill-McKee 33
Alliant 57, 102, 109
applied voltage 10, 53, 55, 89f, 97, 102, 104, 108
array 60, 63, 75f, 80, 84, 86ff, 94
assembly 44, 84, 87ff, 94
Auger process 16
avalanche generation 17

backward differential formula 27
backward Euler method 26
bandgap narrowing 8, 14, 88, 94
bandwidth reduction 33
Bernoulli function 23, 31, 45, 83, 92
BiCG 34
BLSMP 88
Bologna 37
Boltzmann constant 7
Boltzmann statistics 7
boundary 9, 20, 48
  internal 11
  plain faced 20
boundary condition 8, 87, 112
  Dirichlet 10, 13, 46, 91
  Neumann 9, 13, 48
box 20, 23, 48
box method 19, 21, 23, 25, 32, 36, 39, 44, 85, 112
built-in voltage 10, 53
Burroughs 76

C 58, 65, 68f, 71, 73ff, 77f, 80
C++ 68, 73, 77, 80
CADDETH 36
capacitor 112
carrier concentration 6, 15, 31, 54
carrier density 6, 31, 53
carrier temperature 113
CG 32
CGS 34
CGSTAB 34
charge density 6, 48
CMOS 103
code generation 58, 68
compiler 58, 60, 64, 72f
computer 1, 3, 11, 19, 32, 39, 59, 61ff, 69, 73, 113
  multiprocessor 38, 72
  vector 38, 72, 79
computer language 58, 60, 65

concentration
  carrier 6, 15, 31, 54
  impurity 6, 10, 39, 53, 104
  intrinsic 7, 12, 14, 88
    effective 8, 14
condition
  sufficient decrease 29
condition number 32, 36ff
conduction current 49
conjugate gradient method 32
contact 9, 112
continuity equation 6, 22, 24, 26, 31, 36, 44, 48f, 87, 91f
control structure 60
control volume method 19
Convex 57, 106
coupled solution method 30, 56, 90
Cray 57, 61, 69, 103, 106, 109
cuboids 40
current 7, 11, 15, 48f, 82, 91, 94, 97, 102, 106f
  conduction 49
  displacement 6, 49, 108
  total 6, 49
current conservation 46
current density 6, 9, 15, 23, 50, 52, 83, 88

damping 29
data structure 4, 25, 32, 41, 60, 66f, 80, 84, 86ff, 93, 95
  sparse 86
data type 60
decoupled solution method 30
degeneracy 8
density
  carrier 6, 31, 53
  charge 6, 48
  current 6, 9, 15, 23, 50, 52, 83, 88
  electron 101
  intrinsic 7, 12, 14, 88
    effective 8, 14
  point 25, 39, 41, 112
design rule 99, 102, 104
device cross-talk 104
dielectric constant 6
diffusion coefficient 7
direct linear solver 32, 38
Dirichlet boundary condition 10, 13, 46, 91
discretisation
  time 26
discretization 3, 19, 38, 44, 49, 112
  box 85
  Scharfetter-Gummel 23, 46, 52
  spatial 19, 32
displacement current 6, 49, 108
donor 5
doping 14, 39, 44, 82, 84, 94, 96, 101, 107, 112
DRAM 1
drift-diffusion 3, 7
dynamic memory 63, 75

edge neighbour 20, 32
effective intrinsic concentration 8, 14
effective potential 8
electric field 5, 50ff, 94
electron density 101
electronics
  solid state 1
electrostatic potential 6, 8f, 22, 26, 31, 36, 52, 55, 90
element shape 40

energy balance 36
environment 62, 66
equation
    continuity 6, 22, 24, 26, 31, 36, 44, 48f, 87, 91f
    Laplace’s 11, 43
    Poisson’s 6, 11, 21, 43f, 46, 48, 56
    semiconductor 2f, 5, 8, 19, 26, 30f, 36, 38, 40, 89
        scaled 13
equilibrium
    thermal 7f, 53
        local 10
Euler method 26

Fermi level 7
Fermi potential 8, 10
FIELDAY 35
fill 33
finite boxes 25
finite differences 24, 36
finite elements 25, 35
finite volumes 19
floating point representation 62
floating point type 61, 69
FORTRAN 58ff, 71, 73ff, 93, 95
full Newton method 30, 36

GaAs 36
gain 104
Gaussian elimination 32
generation 6, 15, 17, 83, 94
    code 58, 68
    grid 24, 39, 82, 96, 112
geometry 10, 37ff, 84, 94, 96, 99, 108
    plane faced 41, 112
glitch 104
grid 20, 23, 25, 35, 37, 39, 41, 46, 50f, 79f, 82, 84, 88, 92, 94, 96, 101, 104
    irregular 24, 32, 36, 38f
    quasi-regular 41
    regular 24, 35ff
    tensor product 24, 35ff
    well shaped 23
grid generation 24, 39, 82, 96, 112
grid refinement
    adaptive 112
Gummel iteration 30, 36

hardware 38, 57, 76
heap 75
heavy doping effects 14
heterostructure 113
HFIELDS-3d 37
Hitachi 36
hybrid method 26

IBM 35
Idea 96
ILU preconditioning 33f
impact ionization 17
impurity 5
impurity concentration 6, 10, 39, 53, 104
imref 7
inductance 112
insulator 11
integrated circuit 1
interface 11
inter-language call 59
internal boundary 11
interpolation 26, 112
intrinsic concentration 7, 12, 14
intrinsic density 88
intrinsic Debye length 12
inverter 104

ionization
    impact 17
irregular grid 24, 32, 36, 38f
iterative linear solver 32, 37

Jacobian 94

kernel 74

L stability 27
language
    computer 58, 60, 65
    inter- | call 59
Laplace's equation 11, 43
latchup 103
lattice temperature 113
library 73
lifetime 16
lifetime engineering 17
linear solver 31, 55f, 74, 79, 87, 90, 94
    direct 32, 38
    iterative 32, 37
    preconditioned 38
    sparse 32
local quasi-Fermi guess 53

machine epsilon 62
macro 66
magnetic field 112
maintainability 60
`malloc` 75
matrix 32, 43, 47, 50, 52, 93f
    sparse 44, 63, 83, 85ff, 92, 94
Matsushita 36
memory 30, 36, 63
    dynamic 63, 75
memory fragmentation 76
memory requirement 3, 25, 32f, 37, 85, 106, 109

MESFET 36
microelectronics 1
MINIMOS 36
minimum degree algorithm 33
mini-supercomputer 38, 57, 102, 106
mobility 7, 14, 31, 46, 83, 93f, 112
model
    physical 112
modularization 60
module 66, 80
MOSFET 9, 15, 26, 36, 99f, 103, 106
multiprocessor 38, 72

Neumann boundary condition 9, 13, 48
Newton algorithm 91
Newton method 28ff, 43, 46, 54ff, 90
    full 30, 36
nonlinear solver 28
norm 55
Debye length
    intrinsic 12
NTT 35
number
    condition 32
numeric type 61

obtuse angle problem 23
Ω 82, 96
operating system 57
overshoot 29
oxide 9, 11, 13, 22, 36, 43, 99, 106f

parallel processing 38
parallelepiped 40
partition 20, 23, 25
Pascal 58

- permittivity 6
- **Picasso** 97
- Pisces-II 36
- plane faced geometry 41, 112
- plugin iteration 30
- point density 25, 39, 41, 112
- pointer 60
- Poisson’s equation 6, 11, 21, 43f, 46, 48, 56
- portability 58, 64, 68
- potential
  - effective 8
  - electrostatic 6, 8f, 22, 26, 31, 36, 52, 55, 90
  - Fermi 8, 10
  - quasi-Fermi 7, 10, 15, 53, 88, 90
  - scaled 12
- preconditioned solver 38
- preconditioning 36
  - ILU 33f
- preprocessor 65ff
- prism
  - triangular 40
- process simulation 96
- pyramid
  - quadrilateral 40

- quadrilateral pyramid 40
- quasi-Fermi
  - local | guess 53
- quasi-Fermi level 9, 11, 31f, 55, 94
- quasi-Fermi potential 7, 10, 15, 53, 88, 90

- readability 60
- recombination 6, 15, 31, 83, 88, 91, 93f, 112
  - Shockley-Read-Hall 16
- region 49, 53
  - active 25, 104, 106
  - convergence 29
- regular grid 24, 35f, 38
- representation
  - floating point 62
- resistor 112
- reverse Cuthill-McKee algorithm 33

- scaling 11, 31
- Scharfetter-Gummel discretization 23, 46, 52
- **Second** 3, 14, 35, 57, 73, 80, 82, 96, 99, 104, 108, 111
- semiconductor 5ff, 11, 13f, 22, 35, 84, 87, 111
- semiconductor equation 2f, 5, 8, 19, 26, 30f, 36, 38, 40, 89
  - scaled 13
- **Sepp** 97
- Shockley-Read-Hall recombination 16
- SIERRA 36
- silicon 11, 13f, 16, 43
- **Simbad** 77
- Slotboom variables 31, 36
- SMART 36
- software 57ff, 61, 90
- software crisis 60
- solid state electronics 1
- solution method
  - coupled 30, 56, 90
  - decoupled 30
- solver
  - linear 31, 55, 74, 79, 87, 90, 94
    - iterative 37
    - sparse 32

- nonlinear 28
sparse linear solver 32
sparse matrix 44, 63, 83, 85ff, 92, 94
standard 58f, 61ff, 69f, 72, 78, 85
  - *de facto* 59
stiff equation 27
stopping criterion
  - absolute 54f
  - nonlinear 55
  - relative 54
structured type 60
sufficient decrease condition 29
Sun workstation 57
supercomputer 38, 57, 61f, 97, 103, 106
  - mini 38, 57, 102, 106
surface scattering 15

temperature 7
  - carrier 113
  - lattice 113
tensor product grid 24, 35ff
tetrahedron 40
Texas Instruments 36
thermal equilibrium 7f, 53
  - local 10
thermal voltage 12
threshold voltage 102
thyristor 104, 112
TOPMOST 36
Toshiba 36
total current 6, 49
tractable problem 38
transistor 1
  - bipolar 104, 106
  - ECL 106
  - lateral 104
  - MOS 9, 99, 103, 106
  - npn 104
  - pnp 104
  - vertical 104
trapezoidal rule 26
trench 106
triangular prism 40
type 60
  - abstract 60
  - floating point 61, 69
  - numeric 61
  - structured 60

UNIX 57ff, 65f, 71, 73, 78
upwinding scheme 26
utility 77

variables
  - choice of 31
  - Slotboom 31, 36
variational problem 25
vector computer 38, 72, 79
velocity saturation 15
Vienna 36
VLSI 1
voltage
  - applied 10, 53, 55, 89f, 97, 102, 104, 108
  - built-in 10, 53
  - thermal 12
voltage glitch 104

well shaped grid 23
word length 61

Curriculum Vitae

I was born in Müllheim, Germany, on July 7, 1957. At the same place I visited elementary school and high school (*gymnasium*), graduating with the *abitur* in 1976. After 15 months of compulsory military service I started to study Physics and Mathematics at the University of Freiburg, Germany, where I obtained the B.Sc. degree in Physics in 1982. From 1981 to 1984 I was studying Computer Science and Physics at Brock University, St. Catharines, Canada, graduating with the M.Sc. degree in Physics. Since 1985 I am working as a researcher at ETH Zürich, first in the Institut für Informatik, and since 1987 at the Integrated Systems Laboratory with Prof. Dr. Wolfgang Fichtner. My publications have appeared in *Physical Review*, *IEEE Software*, the *Japanese Journal of Applied Physics Letters* and *IEEE Transactions*, as well as in several conference proceedings.