1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

**A Two-Dimensional Steady-State Finite Difference Simulator For
Semiconductor Magnetic Field Sensors**

by Brian D. Gieschen, B. Eng

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfilment of
the requirements for the degree

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Faculty of Engineering
Carleton University
April 1995

© copyright
1995, Brian D. Gieschen

**National Library
of Canada**

**Acquisitions and
Bibliographic Services Branch**

**395 Wellington Street
Ottawa, Ontario
K1A 0N4**

**Bibliothèque nationale
du Canada**

**Direction des acquisitions et
des services bibliographiques**

**395, rue Wellington
Ottawa (Ontario)
K1A 0N4**

Your file Votre référence

Our file Notre référence

**THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.**

**L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.**

**THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.**

**L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.**

ISBN 0-612-03021-0

Canada

Name

Brian D. Gioschen

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Electronics & Electrical

SUBJECT TERM

0547

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0507
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0272
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Science 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce-Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0578
Gerontology 0578
History 0578
General

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0322
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and 0617
Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family 0630
Studies 0629
Industrial and Labor 0630
Relations 0700
Public and Social Welfare 0344
Social Structure and 0709
Development 0999
Theory and Methods 0453
Transportation 0453
Urban and Regional Planning 0453
Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and 0476
Nutrition 0359
Animal Pathology 0478
Food Science and 0479
Technology 0480
Forestry and Wildlife 0817
Plant Culture 0777
Plant Pathology 0746
Plant Physiology 0306
Range Management 0287
Weed Technology 0308
Biology 0309
General 0379
Anatomy 0329
Botanics 0353
Botany 0369
Cell 0793
Ecology 0410
Entomology 0307
Genetics 0317
Limnology 0416
Microbiology 0433
Molecular 0821
Neuroscience 0778
Oceanography 0472
Physiology 0786
Radiation 0760
Veterinary Science 0760
Zoology 0425
Biophysics 0996
General 0425
Medical

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polymology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0549
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and 0381
Therapy 0571
Ophthalmology 0419
Pathology 0572
Pharmacology 0382
Pharmacy 0574
Physical Therapy 0575
Public Health 0574
Radiology 0575
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0495
Pharmaceutical 0754
Physical 0405
Polymer 0605
Radiation 0986
Mathematics 0606
Physics 0608
General 0748
Acoustics 0607
Astronomy and 0798
Astrophysics 0759
Atmospheric Science 0610
Atomic 0752
Electronics and Electricity 0756
Elementary Particles and 0611
High Energy 0346
Fluid and Plasma 0984
Molecular 0621
Nuclear 0622
Optics 0623
Radiation 0624
Solid State 0625
Statistics 0626
Applied Sciences 0346
Applied Mechanics 0984
Computer Science

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0545
Heat and Thermodynamics 0546
Hydraulic 0547
Industrial 0794
Marine 0548
Materials Science 0743
Mechanical 0551
Metallurgy 0552
Mining 0549
Nuclear 0765
Packaging 0554
Petroleum 0790
Sanitary and Municipal 0428
System Science 0796
Geotechnology 0795
Operations Research 0994
Plastics Technology 0994
Textile Technology

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis:

"A Two-Dimensional Steady-State Finite Difference Simulator For Semiconductor Magnetic Field Sensors"

Submitted by Brian D. Gieschen in partial fulfilment of the requirements for the degree of Master of Engineering

Prof. N.G. Tarr
Thesis Supervisor

Prof. T. Smy
Thesis Supervisor

for

Prof. J.S. Wight
Chairman,
Department of Electronics

Carleton University
April 1995

Abstract

This thesis details the development of a simulator for semiconductor magnetic field sensors starting from the governing differential equations. In the past magnetic field sensors have been fabricated without the aid of extensive simulation. The lack of availability of suitable simulation tools has contributed to this. This simulator differs from most recently developed simulators for semiconductor magnetic field sensors by using the finite-difference method rather than the finite element method. The relative simplicity and ease of modification by others makes this simulator accessible to a wide range of users. The process of building this simulator from deriving the governing equations to creating difference expressions to implementing simulation algorithms to generating initial values and boundary conditions is explored here. The result is a working simulator capable of being used to simulate a range of semiconductor magnetic field sensors. This simulator is tested by application to a Lateral Magneto Transistor (LMT) and a Suppressed Sidewall Injection Magneto Transistor (SSIMT). This is one of the first finite-difference simulations of a Suppressed Sidewall Injection Magneto Transistor.

Table of Contents

List of Tables	iv
List of Figures	v
List of Symbols	vii
1 Introduction	1
1.1 Overview of Semiconductor Magnetic Field Sensors	1
1.2 The Need for a Simulator	5
1.3 Outline	6
2 Background Analysis	7
2.1 Purpose	7
2.2 Simulation Methods	7
2.3 The Basic Equations	9
2.4 Modifying the Transport Equation to include the Lorentz Force	10
2.5 The Finite-Difference Method	13
2.6 Overall Matrix Solution and Successive Line Over-Relaxation (SLOR)	16
2.7 Gridding	17
3 The Simulator	19
3.1 The Scharfetter-Gummel form of the Transport Equations	19
3.2 Difference Equations	25
3.3 Singularities in the Difference Equations	28
3.4 Fixed Boundary Conditions	28
3.5 Floating Boundary Conditions	29
3.6 Initial Values	30
3.7 The Simulator Algorithm	32
3.8 Testing	35
4 Method and Results	37
4.1 The Device Structures	37
4.2 Interpreting Simulation Results	39
4.3 Constructing the Simulation	39
4.4 Simulations	46
5 Conclusions	52
5.1 Finite-Difference Simulator	52
5.2 Future Work	52

6	References	54
7	Appendix A - Medici PN-Junction Results	57
8	Appendix B - Finite-Difference Simulator PN-Junction Results	69
9	Appendix C - Finite-Difference Simulator LMT and SSIMT Results	73
10	Appendix D - Simulator Code	89
	10.1 Overview of Simulator Files	89
	10.2 Type.h	89
	10.3 Vector.h	90
	10.4 Matrix.h	90
	10.5 Gauss.h	92
	10.6 Node.h	92
	10.7 Region.h	96
	10.8 Thesis1.h	97
	10.9 Thesisrc.h	98
	10.10Thesis1.cpp	98
	10.11Region.cpp	105
	10.12Node.cpp	118
	10.13Node1.cpp	142
	10.14Fileio.cpp	144
	10.15Thesis1.rc	147
	10.16Thesis1.def	147

List of Tables

Table 4.1	Changes in Collector Current at 1 Gauss Magnetic field	50
-----------	--	----

List of Figures

Figure 1.1	Examples of Magneto Transistors.	2
Figure 1.2	Example of a Suppressed Sidewall Injection Magneto Transistor	4
Figure 2.1	Lorentz Effects on Carriers	10
Figure 2.2	Gridding Terminology	18
Figure 4.1	LMT structure simulated.	37
Figure 4.2	The SSIMT structure simulated	38
Figure 4.3	Orientation and bounds for graphs	40
Figure 4.4	LMT Vemitter=0 volt, Vbase=1.2 volt, Vcollector=-0.4 volt, B=0 Gauss	47
Figure 4.5	LMT Vemitter=0 volt, Vbase=1.2 volt Vcollector=-0.4 volt, B=1 Gauss	48
Figure 7.1	MEDICI - Zero Bias PN-Junction (Potential)	57
Figure 7.2	MEDICI - Zero Bias PN-Junction (Electron Concentration)	58
Figure 7.3	MEDICI - Zero Bias PN-Junction (Hole Concentration)	59
Figure 7.4	MEDICI - 0.6 volt Forward Bias PN-Junction (Potential)	60
Figure 7.5	MEDICI - 0.6 volt Forward Bias PN-Junction (Electron Concentration)	61
Figure 7.6	MEDICI - 0.6 volt Forward Bias PN-Junction (Hole Concentration)	62
Figure 7.7	MEDICI - 0.8 volt Forward Bias PN-Junction (Potential)	63
Figure 7.8	MEDICI - 0.8 volt Forward Bias PN-Junction (Electron Concentration)	64
Figure 7.9	MEDICI - 0.8 volt Forward Bias PN-Junction (Hole Concentration)	65
Figure 7.10	MEDICI - 1.0 volt Forward Bias PN-Junction (Potential)	66
Figure 7.11	MEDICI - 1.0 volt Forward Bias PN-Junction (Electron Concentration)	67
Figure 7.12	MEDICI - 1.0 volt Forward Bias PN-Junction (Hole Concentration)	68
Figure 8.1	PN-Junction - Zero Bias	69
Figure 8.2	PN-Junction - 0.6 volt forward bias	70
Figure 8.3	PN-Junction - 0.8 volt forward bias	71

		vi
Figure 8.4	PN-Junction - 1.0 volt forward bias	72
Figure 9.1	LMT - Zero Bias, Zero Magnetic Field	73
Figure 9.2	LMT - Vbase=1.2 volt, B=0 Gauss	74
Figure 9.3	LMT- Vbase=1.2 volt, Vcollector=-0.25 volt, B=0 Gauss	75
Figure 9.4	LMT- Vbase=1.2 volt, Vcollector=-0.25 volt, B=1 Gauss	76
Figure 9.5	LMT - Vbase=1.2 volt, Vcollector=-0.4 volt, B=0 Gauss	77
Figure 9.6	LMT - Vbase=1.2 volt, Vcollector=-0.4 volt, B=1 Gauss	78
Figure 9.7	SSIMT Zero bias, Zero Magnetic Field	79
Figure 9.8	SSIMT - Vbase=1.2 volt, B=0 Gauss	80
Figure 9.9	SSIMT - Vbase=1.2 volt, Vcollector=-0.4 volt, B=0 Gauss	81
Figure 9.10	SSIMT - Vbase=1.2 volt, Vcollector=-0.4 volt, B=1 Gauss	82
Figure 9.11	SSIMT - Vbase=1.2 volt, Vcollector=-0.8 volt, B=0 Gauss	83
Figure 9.12	SSIMT - Vbase=1.2 volt, Vcollector=-0.8, B=1 Gauss	84
Figure 9.13	SSIMT - Vbase=1.2 volt, Vcollector=-0.8 volt, Vbias stripe=0.2 volt, B=0 Gauss ...	85
Figure 9.14	SSIMT - Vbase=1.2 volt, Vcollector=-0.8 volt, Vbias stripe=0.2 volt, B=1 Gauss ...	86
Figure 9.15	SSIMT - Vbase=1.2 volt, Vcollector=-0.8 volt, Vbias stripe=0.3 volt, B=0 Gauss ...	87
Figure 9.16	SSIMT - Vbase=1.2 volt, Vcollector=-0.8 volt, Vbias stripe=0.3 volt, B=1 Gauss ...	88

List of Symbols

B	Magnetic induction T
D_n	Electron diffusion coefficient m^2/s
D_p	Hole diffusion coefficient m^2/s
ϵ	Permittivity (of silicon) 1.064×10^{-12} F/cm
E	Electric field V/m
G	Carrier generation rate $m^{-3} \cdot s^{-1}$
h	Distance between two simulation nodes m
$\vec{H}$	Magnetic field. A/m
J_n	Electron current density A/m^2
J_p	Hole current density A/m^2
K	Boltzmann's constant 1.38066×10^{-23} J/K
N_a	Acceptor doping concentration m^{-3}
N_d	Donor doping concentration m^{-3}
n	Electron concentration m^{-3}
n_0	Initial electron concentration m^{-3}
p	Hole concentration m^{-3}
p_0	Initial hole concentration m^{-3}
q	Elementary electronic charge 1.60218×10^{-19} C

R_n	Hall coefficient for electrons	$m^{-3}C^{-1}$
R_p	Hall coefficient for holes	$m^{-3}C^{-1}$
τ_p	Hole lifetime	s
τ_n	Electron lifetime	s
T	Temperature	300. K
U	Carrier recombination rate	$m^{-3}s^{-1}$
v	Carrier drift velocity	m/s
V	Applied bias	V
μ_0	Magnetic permeability of free space	A/m-T
μ_r	Relative magnetic permeability	-
μ_n	Electron drift mobility	$m^2/V\cdot s$
μ_n^h	Electron hall mobility	$m^2/V\cdot s$
μ_p	Hole drift mobility	$m^2/V\cdot s$
μ_p^h	Hole hall mobility	$m^2/V\cdot s$
Γ	Doping	m^{-3}
σ_n	Electron conductance	$(\Omega\cdot m)^{-1}$
σ_p	Hole conductance	$(\Omega\cdot m)^{-1}$
ϕ_n, ϕ_p	Quasi-Fermi potentials	V
ψ	Potential	V

θ	Short for $\frac{q}{RT}$ 38.6815. V^{-1}
Y	Column vector of primary variables p, n, ψ
δY	Column vector of increments of primary variables
C, N, S, E, W	Matrices relating initial values of Y to δY
F	Constant column vector relating initial values of Y to δY
M, N	X and Y axis indices to an array

1 Introduction

1.1 Overview of Semiconductor Magnetic Field Sensors

The first semiconductor device designed to sense magnetic fields was the Hall Plate. This device is easily fabricated in the most basic semiconductor fabrication processes. Other semiconductor devices have been designed in the search for more sensitive magnetic sensors. A class of these devices which has sparked interest in current literature and research is called magneto transistors [1,15,16,17,18,21,24,25,26]. These devices are specially designed bipolar transistors, optimized to sense magnetic fields. There are a variety of magneto transistor designs which differ in physical principle used to sense magnetic field as well as direction of magnetic field detected.

A magneto transistor is simply a bipolar transistor designed to be sensitive to magnetic fields. There are two categories based on the direction of current flow in the transistor: Vertical and Lateral. There are also subcategories based on the number of collectors in the device. Figure 1.1 . shows several typical magneto transistors. The lateral magneto transistor (LMT) with one collector is thought to function exclusively through carrier deflection. In terms of electrical operation, the device can be broken into two separate transistors; a vertical NPN transistor formed from emitter, base and substrate and a lateral transistor formed from emitter, base and collector. In the absence of a magnetic field, electrons injected by the emitter are swept towards the base and substrate contacts. A portion of these electrons fall into the collector potential well. Although a collector current in an NPN transistor is normal, this zero magnetic field collector current is not desirable because it limits the sensitivity of the device. Optimally, no electrons swept from emitter

Lateral Magneto Transistor with a single collector (side view)

Lateral Magneto Transistor with two collectors (top view)

Vertical Magneto Transistor with two collectors (side view)

Figure 1.1 Examples of Magneto Transistors

to base contact should fall into the collector unless pushed by the magnetic field. In continuous sensing operation, the substrate current is also a major limitation. A second base contact can also be placed to the left of the emitter to help sweep electrons laterally. In the presence of a magnetic field, electrons are deflected either toward or away from the collector depending on the orientation of the magnetic field, altering the collector current. There has been considerable debate regarding the physical principle governing the operation of the lateral magneto transistor with two collectors [21,22,23,25]. Both emitter mod-

ulation and carrier deflection play a role to some degree in this device, although carrier deflection is thought to be more dominant. The emitter modulation theory [21,22,23,25] describes a lateral magneto transistor structure as being similar to a Hall plate. A Hall voltage, due to magnetic field, is induced normal to the current flow which travels between emitter and collector. This Hall voltage causes the emitter to be biased asymmetrically causing a higher level of injection on the side which is more forward biased. The current flowing from emitter to base is deflected (or modulated) to one side increasing the current in one collector while simultaneously decreasing the current in the other collector.

The vertical magneto transistor (VMT) with two collectors operates similarly to the lateral magneto transistor with two collectors. The electrons injected across the forward biased emitter/base junction are collected at the base/collector junction. The electrons flow away from the surface towards the two collectors. Ideally, each collector collects half the current. When a magnetic field is applied a portion of the current collected by one collector is deflected to be collected by the other collector. The difference between the two collector currents is directly related to the magnitude of the magnetic field.

The suppressed sidewall injection magneto transistor (SSIMT) is an improvement on the lateral magneto transistor [1,17]. An SSIMT is shown in Figure 1.2 . The bias stripes reduce current which travels from the side of the emitter to the collector. This current is insensitive to magnetic field and is undesirable. The bias stripes thus effectively increase the sensitivity to small magnetic fields. The bias stripe can also be used to drive the injected carriers from the emitter deeper towards the substrate. Applying a bias to the

Figure 1.2 Example of a Suppressed Sidewall Injection Magneto Transistor

stripe will drive carriers further from the surface. When properly biased the collector current will be very small in the absence of a magnetic field. In the presence of a magnetic field, current which is passing by the collector will be deflected into the collector. This optimizes the device's sensitivity to magnetic fields and allows the device to be calibrated, reducing the effects of semiconductor fabrication process variations.

Possible additional improvements have also been discussed in current literature. One possible improvement is the elimination of substrate leakage through the substrate transistor in the LMT and SSIMT. In a typical MOS process, devices such as the SSIMT are fabricated in a P type well on an N type substrate. The N type emitter, P type well, N type substrate combination forms an NPN bipolar transistor referred to as a substrate transistor. Reducing the current due to this transistor could make the device more energy efficient for continuous sensing applications. This could be accomplished by implementing a reverse biased junction between device well and substrate, for example, depositing a P type epitaxial layer on top of a P+ substrate. In addition, control over the biasing of this junction would also aid in focusing the current flowing parallel to the surface of the device. By biasing the Bias Stripe and Device Well/Substrate junctions the current may be

made to flow through a small aperture as it flows toward the base. Since the current would flow in an area with a smaller width, deflection due to magnetic field would affect a larger proportion of the current. This could improve the sensitivity of the collector current to magnetic field.

1.2 The Need for a Simulator

Semiconductor magnetic field sensors are often complicated devices involving up to 10 contacts [1] to control their behavior. The complicated geometries and electrical activity in a sensor make it difficult to predict behavior based on intuition or analytical methods. Different processes involved in the production of a sensor can drastically change the required biases and signals which the sensor produces [2,3,19,20]. This thesis will simulate both a lateral magneto transistor and a suppressed sidewall injection magneto transistor with an emphasis on useful design insight.

Numerical simulation has seen at least limited use in the design of semiconductor magnetic field sensors for some time. The simulators described to date in literature have generally made use of finite-element techniques to solve the differential equations governing carrier motion [26,27,28], although the finite-difference method has also been used successfully [29]. Unfortunately, few if any of these simulators are publicly available, and most have been written for specific computing platforms, often making use of specialized library routines, which has generally blocked their portability. Perhaps for this reason, simulators are still not widely used, and many sensors are still designed using crude analytic approximations.

This thesis is concerned with making a semiconductor magnetic field simulator accessible. The approach taken is to produce a straight forward simulator which designers can modify themselves in order to tailor the simulation for a specific device. The code is accessible, written for a PC/AT clone running Microsoft Windows. The code is portable, written in C++ and encapsulates the user interface and file I/O in a small number of routines. The code uses dynamic data structures, accommodating simulations which are limited only by memory. This simulator is also concerned with simplicity, while providing useful insight into a semiconductor magnetic field sensor.

1.3 Outline

This thesis is separated into four main chapters: Background Analysis, The Simulator, Results and Conclusions. The background analysis chapter describes the theory underlying the simulator. The simulator chapter describes the derivations involved with and implementation of the simulator. The results chapter presents a sample use of the simulator, presenting a simulation of an LMT and SSIMT. The conclusions chapter discusses the achievements of this simulator as well as future improvements.

2 Background Analysis

2.1 Purpose

This chapter provides important background information for use in constructing the simulator. It provides an overview of methods used and discusses some of the choices made. The derivation integrating the Lorentz force into the transport equations is presented. An overview of the numerical methods and gridding algorithms used in this simulator is included at the end of this chapter.

2.2 Simulation Methods

Much of the simulation on complicated devices is now performed with finite-element simulators [3,4,10,11,12,26,27,28]. This is a relatively new type of simulation which has gained popularity in the last fifteen years for semiconductor devices. One of the main advantages of the finite-element method is the ability to approximate problems using a triangular grid allowing more realistic representation of complex geometries. The finite-element method has been used more extensively in mechanical engineering design. The theory involved with the finite-element method is more complex than finite-difference theory [4].

Stability and convergence are some of the most important aspects of a simulator [5]. In finite-element methods the underlying equations are converted to integral forms, as opposed to the difference forms used by finite-difference. The integral forms can provide better stability. Convergence generally improves with the use of higher order approxima-

tions in both the integral and difference forms. In the traditional finite-difference theory, the grid is usually rectangular to facilitate the use of Successive Line Over-Relaxation [6.7]. In the finite-element theory, the grid is rectangular and/or triangular. The finite-difference theory can be adapted to triangular gridding but the traditional theory (which was developed originally for hand calculation) does not attempt this. Triangular gridding for complex, real, devices is preferable from a simulation perspective, but implementing the gridding can be more complex than actually implementing an alternative simulation method (such as finite-element).

Triangular gridding is clearly the superior choice for reducing the size of a simulation when applied to an exact representation of a device which does not have rectangular boundaries or rectangular junctions. Triangular gridding would require significant modification to both the application of finite difference theory to semiconductor device simulation and the approach taken to implement Successive Line Over-Relaxation. Rectangular gridding facilitates the use of Successive Line Over-Relaxation and can still accommodate non-uniform gridding. Rectangular gridding practically also means the simulator may not be able to exactly describe the real geometry of the device being simulated.

This simulator will allow several gridded regions to be butted together to make a larger, more complicated device. This overcomes some of the limitations of rectangular gridding. The matrix equations will also be solved using a graph approach rather than sparse matrix techniques. This facilitates multi-region gridding.

2.3 The Basic Equations

The equations used to describe the motion of electrons and holes in semiconductor magnetic field sensors are the Boltzman transport equations. Poisson's equation and the Lorentz equation. Shockley-Read-Hall recombination.[8,9] is also included. For completeness all the equations which are relevant are listed below:

Boltzman transport equation for electrons:

$$J_n = qD_n \nabla n - q\mu_n n \nabla \psi \quad (2.1)$$

Boltzman transport equation for holes:

$$J_p = -qD_p \nabla p - q\mu_p p \nabla \psi \quad (2.2)$$

Continuity equation for electrons:

$$\frac{\partial n}{\partial t} = \frac{1}{q} \text{div} J_n + G - U \quad (2.3)$$

Continuity equation for holes:

$$\frac{\partial p}{\partial t} = -\frac{1}{q} \text{div} J_p + G - U \quad (2.4)$$

Poisson's equation;

$$\text{div} \nabla \psi = -\frac{q}{\epsilon} (N_d - N_a + p - n) \quad (2.5)$$

Lorentz equation;

$$\vec{F} = q\vec{v} \times \vec{B} \quad \vec{B} = \mu_r \mu_0 \vec{H} \quad (2.6)$$

Shockley-Read-Hall Recombination:

$$U_{SRH} = \frac{pn - n_i^2}{\tau_p(n + n_0) + \tau_n(p + p_0)} \quad (2.7)$$

2.4 Modifying the Transport Equation to include the Lorentz Force

The Lorentz force must be included in the transport equations to include the effect of magnetic fields on device operation, an example of such can be found in Andor et al. [29]. This thesis takes an approach similar to that of Andor et al.

The Lorentz force deflects charge carriers travelling perpendicular to the magnetic field. In the case of a Hall plate this causes an excess of carriers at one side of the current channel and a deficiency at the other [1]. This non-uniform distribution of carriers along the cross-section of the channel creates an electric field acting in the opposite direction to the Lorentz force see Figure 2.1.

Figure 2.1 Lorentz Effects on Carriers

First, an expression for the current density component arising from the interaction

of carrier transport and magnetic field is needed. This is accomplished by substituting the electric field generated by the Lorentz force into Ohm's Law.

Relating the electric field to the Lorentz force ($\vec{F}_L = e\vec{E}_H$). Substituting ($-e = q$) into the Lorentz equation to account for the negative charge and ($\vec{j}_n = ne\vec{v}_{Dn}$) relating electron velocity to electron current density yields

$$\vec{E}_{H,n} = -R_n \vec{j}_n \times \vec{B} \quad (2.8)$$

where $R_n = \frac{-1}{ne}$, similarly,

$$\vec{E}_{H,p} = -R_p \vec{j}_p \times \vec{B} \quad (2.9)$$

where $R_p = \frac{1}{pe}$, and using Ohm's law

$$\vec{j} = \sigma \vec{E} \quad (2.10)$$

these can be rewritten as

$$\vec{j}_{H,n} = -\sigma_n R_n \vec{j}_n \times \vec{B} \quad \vec{j}_{H,p} = -\sigma_p R_p \vec{j}_p \times \vec{B} \quad (2.11)$$

The electron current density $\vec{j}_n(\vec{B})$ in the presence of a magnetic field is the summation of both the electron current density in the absence of a magnetic field $\vec{j}_n(0)$ and the current resulting from the Lorentz force $\vec{j}_{H,n}(\vec{B})$.

$$\vec{j}_{H,n}(\vec{B}) = -\sigma_n R_n \vec{j}_n(\vec{B}) \times \vec{B} \quad \vec{j}_{H,p}(\vec{B}) = -\sigma_p R_p \vec{j}_p(\vec{B}) \times \vec{B} \quad (2.12)$$

This interaction is recursively dependent on the electron current density due to magnetic field $\vec{j}_n(\vec{B})$. The Boltzman transport equation now becomes

$$\vec{j}_n(\vec{B}) = \vec{j}_n(0) - \mu_n^* \vec{j}_n(\vec{B}) \times \vec{B} \quad \vec{j}_p(\vec{B}) = \vec{j}_p(0) - \mu_p^* \vec{j}_p(\vec{B}) \times \vec{B} \quad (2.13)$$

where $\mu_n^* = \sigma_n R_n$ and $\mu_p^* = \sigma_p R_p$.

This expression for a Lorentz-Boltzman transport equation is difficult to manipulate because of the cross product on the right hand side (RHS). With a fair amount of manipulation this equation can be reduced to a more easily tractable form.

Bringing all $\hat{J}_n(\vec{B})$ terms to the RHS and expanding into matrix notation

$$\begin{bmatrix} J_{nx}(\vec{B}) \\ J_{ny}(\vec{B}) \\ J_{nz}(\vec{B}) \end{bmatrix} \begin{bmatrix} 1 & \mu_n^* B_z & -\mu_n^* B_y \\ -\mu_n^* B_z & 1 & \mu_n^* B_x \\ \mu_n^* B_y & -\mu_n^* B_x & 1 \end{bmatrix} = \begin{bmatrix} J_{nx}(0) \\ J_{ny}(0) \\ J_{nz}(0) \end{bmatrix} \quad (2.14)$$

Multiplying both sides by the inverse of the matrix yields

$$\begin{bmatrix} J_{nx}(\vec{B}) \\ J_{ny}(\vec{B}) \\ J_{nz}(\vec{B}) \end{bmatrix} = \begin{bmatrix} J_{nx}(0) \\ J_{ny}(0) \\ J_{nz}(0) \end{bmatrix} \frac{\begin{bmatrix} 1 + (\mu_n^* B_x)^2 & \mu_n^* B_z + \mu_n^{*2} B_x B_y & -\mu_n^* B_y + \mu_n^{*2} B_x B_z \\ -\mu_n^* B_z + \mu_n^{*2} B_x B_y & 1 + (\mu_n^* B_y)^2 & \mu_n^* B_x + \mu_n^{*2} B_y B_z \\ \mu_n^* B_y + \mu_n^{*2} B_x B_z & -\mu_n^* B_x + \mu_n^{*2} B_y B_z & 1 + (\mu_n^* B_z)^2 \end{bmatrix}}{1 + (\mu_n^* |\vec{B}|)^2} \quad (2.15)$$

The matrix on the RHS can be separated into three sub-matrices to facilitate the compactation of the equation into vector notation

$$\hat{J}_n(\vec{B}) = \frac{\hat{J}_n(0) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \mu_n^* \hat{J}_n(0) \begin{bmatrix} 0 & B_z & -B_y \\ -B_z & 0 & B_x \\ B_y & -B_x & 0 \end{bmatrix} + \mu_n^{*2} \hat{J}_n(0) \begin{bmatrix} B_x^2 & B_x B_y & B_x B_z \\ B_y B_x & B_y^2 & B_y B_z \\ B_z B_x & B_z B_y & B_z^2 \end{bmatrix}}{1 + (\mu_n^* |\vec{B}|)^2} \quad (2.16)$$

This equation can now be re-written in vector notation

$$\hat{J}_n(\vec{B}) = \frac{\hat{J}_n(0) + \mu_n^* (\hat{J}_n(0) \times \vec{B}) + \mu_n^{*2} (\vec{B} \cdot \hat{J}_n(0)) \vec{B}}{1 + (\mu_n^* |\vec{B}|)^2} \quad (2.17)$$

similarly

$$\vec{j}_p(\vec{B}) = \frac{\vec{j}_p(0) + \mu_n^{-1}(\vec{j}_p(0) \times \vec{B}) + \mu_p^{-1}(\vec{B} \cdot \vec{j}_p(0))\vec{B}}{1 + (\mu_p^{-1}|\vec{B}|)^2} \quad (2.18)$$

Note that the last term on the RHS could also be written $\mu_n^{-1}\vec{j}_n(0)(\vec{B}\vec{B}^T)$.

2.5 The Finite-Difference Method

With all numerical techniques, the finite-difference method approximates the continuous analytical solution with a set of discrete relations. The continuous analytic functions which solve the governing partial differential equations are represented by their values at a discrete set of points. Derived equations relate the values of the primary variables at adjacent points.

In this description of the finite-difference method, the location in a device where there is a set of primary variables describing one of the localized relations, is termed a node. The network of nodes related to each other through sets of equations, is termed a grid. In the two-dimensional case, this network of nodes resides on a two-dimensional plane. In the typical application of finite-difference method [4,29], which this simulator uses, nodes may be related to other nodes which are adjacent directly along the x-axis or y-axis only. No diagonal relations are permitted. This limits the grid to being rectangular although it can be non-uniform.

The method of constructing the equations relating the primary variables located at

each of the nodes is as follows:

1. Discretization of the governing equations;
2. Linearization of the discretized equations if they are non-linear (using the Newton-Raphson scheme);
3. Solution of the linear discretized equations.

The governing equations are the electron and hole Lorentz-Boltzman transport equations and Poisson's equation. Discretization involves substituting an approximation for the differential components of the equation. A first order approximation is common and also simple to implement,

$$\left. \frac{\partial \psi}{\partial x} \right|_{x=x(M, N)} \equiv \frac{\psi(M+1, N) - \psi(M, N)}{x(M+1, N) - x(M, N)} \quad (2.19)$$

Higher order approximations are not used in this simulator. They can speed convergence but introduce a large amount of complexity to a simulator. Higher order approximation methods require the values of variables in a radius equal to their order from the central node at which the differential is being determined. In the first order approximation, values from an adjacent node are required. In higher order approximations, values from nodes two, three, four etc. nodes (or grid spacings in the case of a rectangular grid) away from the central node may be required. This slows computation considerably.

Once the governing equations have been discretized, they must be linearized. Typically, the Newton-Raphson method is applied. The exact value of a variable is assumed to be an initial value plus an increment, calculated using the Newton-Raphson method. The sum of the initial value and the increment are used as an initial value for the calcula-

tion of another increment. This continues until the computed increment is within a specified error.

The linearized discretized governing equations for each node can be written as,

$$CY(M, N) + NY(M, N + 1) + SY(M, N - 1) + EY(M - 1, N) + WY(M + 1, N) = F \quad (2.20)$$

Where $Y(M, N)$ is a column vector of primary variables (in this case hole density, electron density and potential) located at the node M, N . F is a vector which is constant for each Newton iteration. C, N, S, E, W are matrixes whose values relate the primary variables at other nodes to this node. This equation is a localized relation relating the value of variables located at one node and the values of variables located at other adjacent nodes. A system of these equations, one for each node in the grid, comprises an overall linear matrix equation, which when combined with boundary constraints, forms a numerical representation of the problem.

The solution of the overall matrix equation,

$$AX = B \quad (2.21)$$

represents a one iteration improvement on the initial value. X is a vector where each element is a Y (from 2.20) pertaining to a node. B is a vector where each element is an F (from 2.20) pertaining to a node. A is a matrix, each row whose components are (C, N, S, E, W) pertaining to a node. Each row in the overall matrix equation represents equation (2.20) for a particular node. Equation 2.20 represents a system of equations, so equation 2.21 can be considered a system of systems of equations. Alternatively, equation 2.20 could be expanded into equation 2.21 multiplying the dimensions of the vectors and matrixes in equation 2.21 by a factor equal to the dimension of the vectors and matrixes in

equation 2.20.

The overall matrix equation cannot practically be solved using a direct method such as Gaussian-Elimination because the computational requirements are prohibitive on an IBM PC compatible computer [4,6,7]. Instead, the overall matrix equation is solved using matrix iterative methods. One of the techniques which has been used successfully in semiconductor simulators is Successive Line Over-Relaxation (SLOR) [6,7]. This technique is employed in this simulator because of its simplicity.

2.6 Overall Matrix Solution and Successive Line Over-Relaxation (SLOR)

SLOR takes an initial guess at the solution of an arbitrary linear matrix equation and refines it toward the exact solution [6,7]. Given a matrix equation,

$$AX = B \quad A = (a_{ij}), B = (b_i), X = (x_i) \quad (i, j) = 1, 2, \dots, N \quad (2.22)$$

The following equations show the next iteration in the progression to a solution.

$$\bar{x}_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^N a_{ij} x_j^k \right) \quad (2.23)$$

$$x_i^{k+1} = x_i^k + \omega (\bar{x}_i^{k+1} - x_i^k) \quad (2.24)$$

where, the superscript k indicates the iteration number and the coefficient ω is called the over-relaxation parameter. An iterative method related to SLOR is the Gauss-Seidel method. The only difference between SLOR and Gauss-Seidel methods is the over-relaxation parameter. In the Gauss-Seidel method the over-relaxation parameter is 1. In the SLOR method the over-relaxation parameter is greater than 1. The Gauss-Seidel method is more stable. The SLOR method takes the change between the iterations and multiplies

it by a number greater than 1 in the hope that this will hasten the convergence to a solution. This can also hinder convergence by changing the initial value too much which requires the next iteration to compensate rather than refining the initial value. More sophisticated approaches do not use a constant over-relaxation parameter but change the parameter depending on circumstance. These approaches can reduce the computation required for a solution, but also significantly increase the complexity of the simulator.

In this simulator, the relaxation factor can be set arbitrarily allowing both SLOR and Gauss-Seidel methods to be used.

2.7 Gridding

For this simulator a rectangular grid was chosen [4,29]. The labeling terminology is as follows: The node (M,N) denotes the current node on which an equation is based, $M+1$ is the next node in the x direction and $N+1$ is the next node in the y direction.

Coordinates increase from the bottom left. The x coordinates are horizontal and the y coordinates are vertical. In the description of the simulator code there is liberal use of relational positions like north, south, east or west. North refers to the node $(M,N+1)$. South refers to the node $(M,N-1)$. East refers to the node $(M+1,N)$. West refers to the node $(M-1,N)$.

Some of the equations, such as for current density, refer to a quantity which is accurate at a point mid-way between the actual nodes. These points are denoted by a prime on the node coordinate. The node (M',N) refers to a quantity which is accurate at a point mid-way between (M,N) and $(M+1,N)$. Figure 2.2 is a diagram depicting most of

Figure 2.2 Gridding Terminology

the gridding terminology.

3 The Simulator

3.1 The Scharfetter-Gummel form of the Transport Equations

It is very important to achieve a Scharfetter-Gummel form for the transport equation in order to minimize instability in the simulators matrix solver [4,13]. The Scharfetter-Gummel form of the transport equation makes the overall matrix more diagonally dominant than just using the regular transport equation. This diagonal dominance speeds the convergence of the simulator and reduces the numerical instability introduced through the use of floating point numbers.

Non-singularity for a matrix equation $AX = B$ is usually guaranteed by the condition,

$$|a_{ii}| > \sum_{k=1, L}^{k \neq i} |a_{ik}| \quad 1 \leq i \leq L \quad (3.1)$$

which describes diagonal dominance. Using the one-dimensional transport equation for holes.

$$J_p = -qD_p \frac{\partial p}{\partial x} - q\mu_p p \frac{\partial \psi}{\partial x} \quad (3.2)$$

Linearizing the transport equation,

$$J_p(M') = J_p^0(M') + \frac{\partial}{\partial p(M)} J_p^0(M') \delta p(M) + \frac{\partial}{\partial p(M+1)} J_p^0(M') \delta p(M+1) + \frac{\partial}{\partial \psi(M)} J_p^0(M') \delta \psi(M) + \frac{\partial}{\partial \psi(M+1)} J_p^0(M') \delta \psi(M+1) \quad (3.3)$$

Transforming the linearized transport equation to difference form.

$$J_p(M) = -\frac{q\mu_p(M')}{\theta h(M')} (p(M+1) - p(M)) - \frac{q\mu_p(M')}{h(M')} (\psi(M+1) - \psi(M)) \frac{p(M) + p(M+1)}{2} \quad (3.4)$$

Substituting into the continuity equation.

$$\begin{aligned} & -\frac{\mu_p(M'-1)}{2h(M'-1)} \left(\frac{2}{\theta} + \psi(M-1) - \psi(M) \right) p(M-1) \\ & + \left[\frac{\mu_p(M'-1)}{2h(M'-1)} \left(\frac{2}{\theta} - \psi(M-1) + \psi(M) \right) + \frac{\mu_p(M')}{2h(M')} \left(\frac{2}{\theta} + \psi(M) - \psi(M+1) \right) \right] p(M) \\ & + \frac{\mu_p(M')}{2h(M')} \left(-\frac{2}{\theta} + \psi(M) - \psi(M+1) \right) p(M+1) = h(M) (G(M) - U(M)) \end{aligned} \quad (3.5)$$

Assuming the potential variation is linear at the three points: $M-1$, M , and $M+1$ and that μ_p and h are constant.

$$\begin{aligned} \psi(M-1) - \psi(M) &= \psi(M) - \psi(M+1) = V_D \\ \frac{\mu_p}{2h} \left(\left(\frac{2}{\theta} + V_D \right) p(M-1) + \frac{4}{\theta} p(M) + \left(V_D - \frac{2}{\theta} \right) p(M+1) \right) &= h (G(M) - U(M)) \end{aligned} \quad (3.6)$$

The diagonal dominant condition is satisfied provided $|V_D| \ll \frac{2}{\theta}$ and the generation-recombination terms are not of great enough magnitude to deform the matrix coefficients on the left side. If this is not true then diagonal dominance is lost. The only way to satisfy the condition in this case is to reduce the spacing h so that V_D is small enough. By using an integral form of the current equations as per Scharfetter and Gummel this problem can be removed.

For the purposes of this simulator, the x and y components of the current density are of interest. The z component of the magnetic field is the major factor affecting these current density components. Using only these components of current density and magnetic

field the Lorentz-Boltzman transport equation arrived at in the previous section can be written

$$\vec{j}_n(\vec{B}) = \frac{\vec{j}_n(0) + \mu_n^* (\vec{j}_n(0) \times \vec{B})}{1 + (\mu_n^* |\vec{B}|)^2} \quad \vec{j}_p(\vec{B}) = \frac{\vec{j}_p(0) + \mu_p^* (\vec{j}_p(0) \times \vec{B})}{1 + (\mu_p^* |\vec{B}|)^2} \quad (3.7)$$

Writing this equation as two, one for each current density component

$$j_{nx}(B_z) = \frac{j_{nx}(0) + \mu_n^* j_{ny}(0) B_z}{1 + (\mu_n^* B_z)^2} \quad j_{px}(B_z) = \frac{j_{px}(0) + \mu_p^* j_{py}(0) B_z}{1 + (\mu_p^* B_z)^2} \quad (3.8)$$

$$j_{ny}(B_z) = \frac{j_{ny}(0) - \mu_n^* j_{nx}(0) B_z}{1 + (\mu_n^* B_z)^2} \quad j_{py}(B_z) = \frac{j_{py}(0) - \mu_p^* j_{px}(0) B_z}{1 + (\mu_p^* B_z)^2} \quad (3.9)$$

The following Scharfetter-Gummel scheme is derived in a similar fashion to Andor et al. [29]. Working with the electron current density expression for the x direction only, and substituting drift and diffusion component and rearranging

$$j_{nx}(B_z) = \frac{qD_n}{A} \left(\frac{dn}{dx} + \mu_n^* B_z \frac{dn}{dy} \right) + \frac{q\mu_n n}{A} (E_x + \mu_n^* B_z E_y) \quad (3.10)$$

where $A = 1 + (\mu_n^* B_z)^2$. Solving for n

$$n = \frac{j_{nx}(B_z) A}{q\mu_n (E_x + \mu_n^* B_z E_y)} - \frac{kT \left(\frac{dn}{dx} + \mu_n^* B_z \frac{dn}{dy} \right)}{q (E_x + \mu_n^* B_z E_y)} \quad (3.11)$$

rearranging and adding integrals

$$\int_{n(M)}^{n(M+1)} \frac{dn}{j_{nx}(B_z) A} = - \int_{x(M), y(N)}^{x(M+1), y(N+1)} \theta(E_x + \mu_n^* B_z E_y) \left(\frac{dx dy}{dy + \mu_n^* B_z dx} \right) \quad (3.12)$$

The terms M and N refer to nodes in the simulator's gridding scheme.

At this point, a departure is taken from Andor et al.. Andor et al. apply Gauss' theorem to the continuity equation obtaining a general result useful with non-rectangular gridding. The following is a direct integral method utilizing the $\mu_n * B_z \ll 1$ assumption which is valid for small magnetic fields (typically 1 Gauss or smaller) [1].

Integrating the LHS

$$\int_{n(M)}^{n(M+1)} \frac{dn}{n - \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n * B_z E_y)}} = \ln \left| \frac{n(M+1) - \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n * B_z E_y)}}{n(M) - \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n * B_z E_y)}} \right| \quad (3.13)$$

integrating the RHS

$$- \int_{x(M), y(N)}^{x(M+1), y(N+1)} \theta(E_x + \mu_n * B_z E_y) \left(\frac{dx dy}{dy + \mu_n * B_z dx} \right) = -\theta(E_x + \mu_n * B_z E_y) \int_{x(M), y(N)}^{x(M+1), y(N+1)} \frac{dx dy}{dy + \mu_n * B_z dx} \quad (3.14)$$

if $B_z = 0$ or $\mu_n * B_z \ll 1$:

$$\int_{x(M), y(N)}^{x(M+1), y(N+1)} \frac{dx dy}{dy + \mu_n * B_z dx} \equiv \int_{x(M)}^{x(M+1)} dx \quad (3.15)$$

If $\mu_n * B_z \gg 1$ then

$$\int_{x(M), y(N)}^{x(M+1), y(N+1)} \frac{dx dy}{dy + \mu_n * B_z dx} \equiv \int_{y(N)}^{y(N+1)} \frac{dy}{\mu_n * B_z} \quad (3.16)$$

An approximation to this integral which includes these two cases is:

$$-\theta(E_x + \mu_n * B_z E_y) \int_{x(M), y(N)}^{x(M+1), y(N+1)} \frac{dx dy}{dy + \mu_n * B_z dx} \equiv -\theta(E_x + \mu_n * B_z E_y) \left(\frac{h_x h_y}{h_y + \mu_n * B_z h_x} \right) \quad (3.17)$$

Where $h_x = h_x(M') = x(M+1) - x(M)$ and $h_y = h_y(N) = \frac{1}{2} (h_y(N'+1) + h_y(N'))$, $h_y(N') = y(N+1) - y(N)$.

Using this approximation yields

$$\ln \left| \frac{n(M+1) - \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n^*B_zE_y)}}{n(M) - \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n^*B_zE_y)}} \right| = -\theta(E_x + \mu_n^*B_zE_y) \left(\frac{h_x h_y}{h_y + \mu_n^*B_z h_x} \right) \quad (3.18)$$

Re-arranging this equation

$$n(M+1) = n(M)e^{-\theta(E_x + \mu_n^*B_zE_y) \frac{h_x h_y}{h_y + \mu_n^*B_z h_x}} + \frac{J_{nx}(B_z)A}{q\mu_n(E_x + \mu_n^*B_zE_y)} \left(1 - e^{-\theta(E_x + \mu_n^*B_zE_y) \frac{h_x h_y}{h_y + \mu_n^*B_z h_x}} \right) \quad (3.19)$$

This is now the Scharfetter-Gummel form of the Lorentz-Boltzman transport equation.

Re-writing this equation in difference form

$$J_{nx}(M', N) = \frac{q}{H_{nx}(M', N) (1 + (\mu_n^*B_z)^2)} (\lambda_{nx1}(M', N)n(M, N) + \lambda_{nx2}(M', N)n(M+1, N)) \quad (3.20)$$

similarly

$$J_{ny}(M, N') = \frac{q}{H_{ny}(M, N') (1 + (\mu_n^*B_z)^2)} (\lambda_{ny1}(M, N')n(M, N) + \lambda_{ny2}(M, N')n(M, N+1)) \quad (3.21)$$

$$J_{px}(M', N) = \frac{q}{H_{px}(M', N) (1 + (\mu_p^*B_z)^2)} (\lambda_{px1}(M', N)p(M, N) + \lambda_{px2}(M', N)p(M+1, N)) \quad (3.22)$$

$$J_{py}(M, N') = \frac{q}{H_{py}(M, N') (1 + (\mu_p^*B_z)^2)} (\lambda_{py1}(M, N')p(M, N) + \lambda_{py2}(M, N')p(M, N+1)) \quad (3.23)$$

where

$$\lambda_{nx1}(M', N) = \frac{\mu_n(M')}{\theta(M')} \frac{\beta_{nx}(M', N)}{(1 - e^{-\beta_{nx}(M', N)})} \quad \lambda_{ny1}(M, N') = \frac{\mu_n(N')}{\theta(N')} \frac{\beta_{ny}(M, N')}{(1 - e^{-\beta_{ny}(M, N')})} \quad (3.24)$$

$$\lambda_{nx2}(M', N) = \frac{\mu_n(M')}{\theta(M')} \frac{\beta_{nx}(M', N)}{(1 - e^{-\beta_{nx}(M', N)})} \quad \lambda_{ny2}(M, N') = \frac{\mu_n(N')}{\theta(N')} \frac{\beta_{ny}(M, N')}{(1 - e^{-\beta_{ny}(M, N')})} \quad (3.25)$$

$$\lambda_{px1}(M', N) = \frac{\mu_p(M')}{\theta(M')} \frac{\beta_{px}(M', N)}{(1 - e^{-\beta_{px}(M', N)})} \quad \lambda_{py1}(M, N') = \frac{\mu_p(N')}{\theta(N')} \frac{\beta_{py}(M, N')}{(1 - e^{-\beta_{py}(M, N')})} \quad (3.26)$$

$$\lambda_{px2}(M', N) = \frac{\mu_p(M')}{\theta(M')} \frac{\beta_{px}(M', N)}{(1 - e^{-\beta_{px}(M', N)})} \quad \lambda_{py2}(M, N') = \frac{\mu_p(N')}{\theta(N')} \frac{\beta_{py}(M, N')}{(1 - e^{-\beta_{py}(M, N')})} \quad (3.27)$$

$$\begin{aligned} \beta_{nx}(M', N) &= \theta(E_x(M', N) + \mu_n * B_z E_y(M', N)) H_{nx}(M', N) \\ \beta_{ny}(M, N') &= \theta(E_y(M, N') - \mu_n * B_z E_x(M, N')) H_{ny}(M, N') \end{aligned} \quad (3.28)$$

$$\begin{aligned} \beta_{px}(M', N) &= \theta(E_x(M', N) + \mu_p * B_z E_y(M', N)) H_{px}(M', N) \\ \beta_{py}(M, N') &= \theta(E_y(M, N') - \mu_p * B_z E_x(M, N')) H_{py}(M, N') \end{aligned} \quad (3.29)$$

$$H_{nx}(M', N) = \frac{h_x(M') h_y(N)}{h_y(N) + \mu_n * B_z h_x(M')} \quad H_{ny}(M, N') = \frac{h_y(N') h_x(M)}{h_x(M) + \mu_n * B_z h_y(N')} \quad (3.30)$$

$$H_{px}(M', N) = \frac{h_x(M') h_y(N)}{h_y(N) + \mu_p * B_z h_x(M')} \quad H_{py}(M, N') = \frac{h_y(N') h_x(M)}{h_x(M) + \mu_p * B_z h_y(N')} \quad (3.31)$$

$$\Xi_x(M', N) = \frac{-(\psi(M+1, N) - \psi(M, N))}{h_x(M', N)} \quad E_y(M, N') = \frac{-(\psi(M, N+1) - \psi(M, N))}{h_y(M, N')} \quad (3.32)$$

$$\Xi_x(M, N') = \frac{-(\psi(M+1, N') - \psi(M-1, N'))}{h_x(M', N) + h_x(M'-1, N)} \quad E_y(M', N) = \frac{-(\psi(M', N+1) - \psi(M', N-1))}{h_y(M, N') + h_y(M, N'-1)} \quad (3.33)$$

$$\psi(M', N+1) = \frac{\psi(M, N+1)(x(M', N) - x(M, N+1)) + \psi(M+1, N+1)(x(M+1, N+1) - x(M', N))}{h_x(M', N+1)} \quad (3.34)$$

$$\psi(M', N-1) = \frac{\psi(M, N-1)(x(M', N) - x(M, N-1)) + \psi(M+1, N-1)(x(M+1, N-1) - x(M', N))}{h_x(M', N-1)} \quad (3.35)$$

$$\psi(M+1, N') = \frac{\psi(M+1, N)(y(M, N') - y(M+1, N)) + \psi(M+1, N+1)(y(M+1, N+1) - y(M, N'))}{h_y(M+1, N')} \quad (3.36)$$

$$\psi(M-1, N') = \frac{\psi(M-1, N)(y(M, N') - y(M-1, N)) + \psi(M-1, N+1)(y(M-1, N+1) - y(M, N'))}{h_y(M-1, N')} \quad (3.37)$$

3.2 Difference Equations

The magnetic effects are included in this simulator solely through the transport equations. Without substituting the transport equations into the continuity equations, there is no difference compared to other non-magnetic semiconductor device simulators [4]. Simulation of magnetic field semiconductor devices is primarily concerned with steady state results. This simulator concentrates on steady-state solutions so the time derivatives are zero. To include transient response solution into this simulator, the time derivatives in the continuity equations must simply be included and solved in a way similar to Kurata's [4].

The difference forms of the continuity equations and Poisson's equation are:

$$\frac{1}{q} \frac{J_{px}(M', N) - J_{px}(M' - 1, N)}{h'_x(M, N)} + \frac{1}{q} \frac{J_{py}(M, N') - J_{py}(M, N' - 1)}{h'_y(M, N)} = -U(M, N) \quad (3.38)$$

$$\frac{1}{q} \frac{J_{nx}(M', N) - J_{nx}(M' - 1, N)}{h'_x(M, N)} + \frac{1}{q} \frac{J_{ny}(M, N') - J_{ny}(M, N' - 1)}{h'_y(M, N)} = U(M, N) \quad (3.39)$$

$$\begin{aligned} & \frac{1}{h'_x(M, N)} \left(\frac{\psi(M+1, N) - \psi(M, N)}{h_x(M', N)} - \frac{\psi(M, N) - \psi(M-1, N)}{h_x(M' - 1, N)} \right) \\ & + \frac{1}{h'_y(M, N)} \left(\frac{\psi(M, N+1) - \psi(M, N)}{h_y(M, N')} - \frac{\psi(M, N) - \psi(M, N-1)}{h_y(M, N' - 1)} \right) = -\frac{q}{\epsilon} (\Gamma(M, N) + p(M, N) - n(M, N)) \end{aligned} \quad (3.40)$$

In the non-magnetic equations, J_x relates nodes (M, N) and $(M+1, N)$ and J_y relates (M, N) and $(M, N+1)$. This means that the continuity equations relate (M, N) , $(M-1, N)$, $(M+1, N)$, $(M, N-1)$ and $(M, N+1)$ nodes. This information is very important for developing the Newton approximations for Newton iteration. It also affects the algorithm for solving

the overall matrix equation, because it expands the band of non-zero entries in the matrix and complicates the solution.

In the magnetic current density equations more nodes are related. The x component of the current density now has to take into account the y component of the current density because of the inclusion of the Lorentz equation cross product. This makes the Newton approximations larger. The magnetic current density equations relate additional nodes for the potential only. For carrier concentration J_x and J_y relate the same nodes as non-magnetic J_x and J_y . For potential, J_x relates nodes (M,N) , $(M+1,N)$, $(M,N-1)$, $(M+1,N-1)$, $(M,N+1)$ and $(M+1,N+1)$ and J_y relates (M,N) , $(M,N+1)$, $(M-1,N)$, $(M-1,N+1)$, $(M+1,N)$ and $(M+1,N+1)$. The continuity equations relate the same nodes for carrier concentration, but for potential they relate $(M-1,N-1)$, $(M,N-1)$, $(M+1,N-1)$, $(M-1,N)$, (M,N) , $(M+1,N)$, $(M-1,N+1)$, $(M,N+1)$, $(M+1,N+1)$. This is much larger set of relations and the additional complexity impacts the algorithms and equations for Newton approximation, singularities, handling boundary values, and the matrix solver. These issues will be addressed later.

The Newton approximations for current density follow:

$$\begin{aligned} p(M, N) &= p^o(M, N) + \delta p(M, N), \\ n(M, N) &= n^o(M, N) + \delta n(M, N), \\ \psi(M, N) &= \psi^o(M, N) + \delta \psi(M, N) \end{aligned} \tag{3.41}$$

where p^o, n^o, ψ^o are trial values from a previous iteration and $\delta p, \delta n, \delta \psi$ are the incremental improvements on the trial values.

$$\begin{aligned}
J_{px}(M', N) = & J_{px}^0(M', N) + \frac{\partial}{\partial p(M, N)} J_{px}^0(M', N) \delta p(M, N) + \frac{\partial}{\partial p(M+1, N)} J_{px}^0(M', N) \delta p(M+1, N) + \\
& \frac{\partial}{\partial \psi(M, N-1)} J_{px}^0(M', N) \delta \psi(M, N-1) + \frac{\partial}{\partial \psi(M+1, N-1)} J_{px}^0(M', N) \delta \psi(M+1, N-1) + \\
& \frac{\partial}{\partial \psi(M, N)} J_{px}^0(M', N) \delta \psi(M, N) + \frac{\partial}{\partial \psi(M+1, N)} J_{px}^0(M', N) \delta \psi(M+1, N) + \\
& \frac{\partial}{\partial \psi(M, N+1)} J_{px}^0(M', N) \delta \psi(M, N+1) + \frac{\partial}{\partial \psi(M+1, N+1)} J_{px}^0(M', N) \delta \psi(M+1, N+1)
\end{aligned} \quad (3.42)$$

$$\begin{aligned}
J_{py}(M, N') = & J_{py}^0(M, N') + \frac{\partial}{\partial p(M, N)} J_{py}^0(M, N') \delta p(M, N) + \frac{\partial}{\partial p(M, N+1)} J_{py}^0(M, N') \delta p(M, N+1) + \\
& \frac{\partial}{\partial \psi(M-1, N)} J_{py}^0(M, N') \delta \psi(M-1, N) + \frac{\partial}{\partial \psi(M-1, N+1)} J_{py}^0(M, N') \delta \psi(M-1, N+1) + \\
& \frac{\partial}{\partial \psi(M, N)} J_{py}^0(M, N') \delta \psi(M, N) + \frac{\partial}{\partial \psi(M, N+1)} J_{py}^0(M, N') \delta \psi(M, N+1) + \\
& \frac{\partial}{\partial \psi(M+1, N)} J_{py}^0(M, N') \delta \psi(M+1, N) + \frac{\partial}{\partial \psi(M+1, N+1)} J_{py}^0(M, N') \delta \psi(M+1, N+1)
\end{aligned} \quad (3.43)$$

$$\begin{aligned}
J_{nx}(M', N) = & J_{nx}^0(M', N) + \frac{\partial}{\partial n(M, N)} J_{nx}^0(M', N) \delta n(M, N) + \frac{\partial}{\partial n(M+1, N)} J_{nx}^0(M', N) \delta n(M+1, N) + \\
& \frac{\partial}{\partial \psi(M, N-1)} J_{nx}^0(M', N) \delta \psi(M, N-1) + \frac{\partial}{\partial \psi(M+1, N-1)} J_{nx}^0(M', N) \delta \psi(M+1, N-1) + \\
& \frac{\partial}{\partial \psi(M, N)} J_{nx}^0(M', N) \delta \psi(M, N) + \frac{\partial}{\partial \psi(M+1, N)} J_{nx}^0(M', N) \delta \psi(M+1, N) + \\
& \frac{\partial}{\partial \psi(M, N+1)} J_{nx}^0(M', N) \delta \psi(M, N+1) + \frac{\partial}{\partial \psi(M+1, N+1)} J_{nx}^0(M', N) \delta \psi(M+1, N+1)
\end{aligned} \quad (3.44)$$

$$\begin{aligned}
J_{ny}(M, N') = & J_{ny}^0(M, N') + \frac{\partial}{\partial n(M, N)} J_{ny}^0(M, N') \delta n(M, N) + \frac{\partial}{\partial n(M, N+1)} J_{ny}^0(M, N') \delta n(M, N+1) + \\
& \frac{\partial}{\partial \psi(M-1, N)} J_{ny}^0(M, N') \delta \psi(M-1, N) + \frac{\partial}{\partial \psi(M-1, N+1)} J_{ny}^0(M, N') \delta \psi(M-1, N+1) + \\
& \frac{\partial}{\partial \psi(M, N)} J_{ny}^0(M, N') \delta \psi(M, N) + \frac{\partial}{\partial \psi(M, N+1)} J_{ny}^0(M, N') \delta \psi(M, N+1) + \\
& \frac{\partial}{\partial \psi(M+1, N)} J_{ny}^0(M, N') \delta \psi(M+1, N) + \frac{\partial}{\partial \psi(M+1, N+1)} J_{ny}^0(M, N') \delta \psi(M+1, N+1)
\end{aligned} \quad (3.45)$$

Recombination is also non-linear, so it also needs an approximation similar to those above.

$$U(M, N) = U^0(M, N) + \frac{\partial}{\partial p(M, N)} U^0(M, N) \delta p(M, N) + \frac{\partial}{\partial n(M, N)} U^0(M, N) \delta n(M, N) \quad (3.46)$$

3.3 Singularities in the Difference Equations

The continuity equations have no singularities themselves unless the grid points

are infinitely close together. Singularities arise from the transport equations. Both magnetic and non-magnetic transport equations have terms like $\frac{\theta(\psi(M, N) - \psi(M+1, N))}{1 - e^{\theta(\psi(M, N) - \psi(M+1, N))}}$ in the denominator. The problem with these terms occurs when potential at two adjacent nodes is the same. Typically, this condition occurs quite frequently. The limit of this expression as the potentials of the two nodes approach each other is -1. If this portion of the equation was implemented without modification, the simulator would at some point cause a non-recoverable floating point error involving division by 0, directly attributable to the case where two adjacent nodes had the same potential.

The simulator must detect this situation and return the limit. This situation is compounded by the fact that in the magnetic simulator there are eight adjacent nodes and the limit must be returned for only those nodes which are at the same potential. This situation complicates the simulator, but this modification can not be avoided.

3.4 Fixed Boundary Conditions

On metal contacts it can be assumed that the carrier concentrations do not change from thermal-equilibrium [4]. Space charge is assumed not to exist,

$$\begin{aligned} pn &= n_i^2, \\ \Gamma + p - n &= 0 \end{aligned} \tag{3.47}$$

For integrating bias applied to metal contacts into the simulator, quasi-Fermi potentials ϕ_n, ϕ_p are useful. At metal contacts the quasi-Fermi potentials reflect applied

biasing conditions. By definition,

$$\psi = \phi_p - \frac{1}{q} \ln \frac{p}{n_i} \quad (3.48)$$

$$\psi = \phi_n + \frac{1}{q} \ln \frac{n}{n_i} \quad (3.49)$$

If one of the contacts is used as a reference, the applied bias on other contacts with respect to the reference is equal to either ϕ_p , ϕ_n depending on whether the doping at the contact is P-type or N-type respectively.

3.5 Floating Boundary Conditions

For surfaces which have no metal contact it can be assumed there is no current component normal to the surface [4]. When this is substituted into the current equations,

$$p_1 e^{\theta(\psi_1 - \psi_2)} - p_2 = 0 \quad (3.50)$$

$$n_1 e^{-\theta(\psi_1 - \psi_2)} - n_2 = 0 \quad (3.51)$$

Where the subscript 1 refers to a node on the surface and the subscript 2 refers to a node adjacent to and displaced one node in a direction normal to the surface from node 1. In order to define the three fundamental variables, an additional relation is required. It is also assumed that the change in space-charge between the two nodes is zero [4].

$$\Gamma_1 + p_1 - n_1 = \Gamma_2 + p_2 - n_2 \quad (3.52)$$

In some situations, such as some surfaces well away from junction depletion regions, it is reasonable to assume zero space-charge at the surface rather than zero change

in space-charge. If the surface node is located in a junction, then a zero space-charge assumption would not be appropriate. A node located in a junction depletion region must have space-charge.

3.6 Initial Values

For zero-bias conditions, the thermal-equilibrium equations under space charge neutrality conditions provide a suitable starting point. For applied bias conditions the voltage ramping technique must often be used. Voltage ramping consists of solving the zero-bias case then solving incrementally, increasing biasing conditions. This is performed because typically the jump from zero-bias to the desired bias would cause the numerical method to diverge. This is a limitation of numerical methods [5]. When ramping the voltage of a contact towards a desired bias, the carrier concentrations do not need to be modified, only the potential needs to be modified. On each contact, the applied bias with respect to a common reference point is simply added to the potential on that contact as per the quasi-Fermi potentials discussed in the Fixed Boundary Conditions section (5.4).

The potentials on each of the nodes between the contacts must also be changed by varying degrees to avoid a severe voltage discontinuity between contact and non-contact nodes. Some simple algorithms are shown below. One which works well for two and even three contacts is,

$$\delta\psi_n = \delta\psi_1 \frac{|\psi_n - \psi_2|}{|\psi_1 - \psi_2|} \quad (3.53)$$

The subscript n denotes the arbitrary node in the device to which a change in bias is being calculated. The subscript 1 refers to the contact to which a change in bias has occurred. The subscript 2 refers to another contact which no change in bias has occurred, and which represents a maximum difference in bias with respect to contact 1. This equation works best if all the potentials in the device are between the potentials of contact 1 and contact 2. If the device potentials are not then this algorithm breaks down quickly.

A more sophisticated algorithm for modifying potential in a four contact device is,

$$\psi_n = \delta\psi_1 \frac{|\psi_n + \psi_2| + |\psi_n + \psi_3| + |\psi_n + \psi_4|}{|\psi_n + \psi_1| + |\psi_n + \psi_2| + |\psi_n + \psi_3| + |\psi_n + \psi_4|} \frac{\sum_{i=1}^4 d_i}{\frac{1}{d_1} + \frac{1}{d_2} + \frac{1}{d_3} + \frac{1}{d_4}} \quad (3.54)$$

The subscript n denotes the node on which the change in bias is being calculated. The numbered subscripts denote the contacts, contact 1 is the contact on which the bias has changed. The variable d represents the distance from the node to the respective contact.

One problem with changing the bias of a node by simply looking at the potentials occurs when bias is first applied after the zero-bias solution has been calculated. If there are four contacts, as above, and two have the same doping (for example 10^{20} cm^{-3}) then the potential of these two contacts will be the same at zero-bias. If the voltage of one of the two contacts was ramped using the algorithm above, ignoring the distance portion on the right hand side of the equation, the nodes adjacent to the contact which has no applied bias would be at the same bias as those of the nodes adjacent to the contact which has an

applied bias. This would not be correct. To combat this tendency, the algorithm incorporates a smoothing by distance. The closer a node is to a contact, the more it is affected by the bias on that contact and the less it is affected by the bias on other contacts.

3.7 The Simulator Algorithm

Since this simulator was written in C++ the components are object-oriented. There are four main objects: node, region, gaussian_doping, and main. These objects can be thought of as C-type structures where all the functions which operate on the structure's data are also part of the structure. The node object encompasses information such as the carrier concentrations, potential, matrix equations and position coordinates. The gaussian_doping object holds information related to doping profile. The region object holds gridding information such as a matrix of nodes and a gaussian_doping object. The main object contains overall device information such as regions, biases and magnetic field.

The functions for this simulator are distributed amongst the objects according to the information they work on. An example of this distributing of functions is the calc_newton_matrices function. Both the region and node objects contain information needed. The region::calc_newton_matrices function goes through the matrix of nodes and instructs the nodes to calculate their individual Newton matrices by calling node::calc_newton_matrices. These Newton matrices are elements of the overall matrix equation described in the finite-difference method section (4.3). The region contains information to do with overall gridding, while the node contains information to do with the

actual difference equations. This separation of information greatly eases the construction of the simulator and reduces programming errors and debugging.

There are two functions controlling the overall simulation. The first function (the initialization function) is: `main::CMInit`. This function constructs a set of regions and the associated grid, attaches contacts and sets the bias and magnetic field to zero. The second function (the startup function) is: `main::CMStart`. This function ramps biases, controls the calculation of Newton matrices, controls the solution of Newton matrices and saves results in files.

The `main::CMInit` function is the constructor. For each of the regions of the simulation it creates a doping function. The doping function is tailorable from junction depth to width and height of the region, to maximum and minimum doping. The `main::CMInit` function then creates an initial specification of the grid through lists of x and y coordinates to assure a maximum distance between nodes. It then passes this information to the region creation algorithm. This algorithm constructs a grid, initializes the nodes using the doping profile, adds nodes to maintain a maximum potential difference between nodes, and adds nodes to have grids between adjoining regions line up. The `main::CMInit` function then initializes the regions by linking up the regions and their adjoining nodes. Finally, it tags each contacted region to have calculations treated differently at the contact (fixed boundary values).

Gridding is performed in several steps by the region and is coordinated by `main::CMInit` function. The first step creates lists of x and y coordinates to define an initial starting point in the gridding. The algorithm then goes through these lists comparing

differences in potential at the locations defined by the coordinates using the defined doping profile. If a potential difference is greater than the allowed maximum (a constant defined in the simulator code), the algorithm adds coordinates to reduce the potential difference. Regions which are adjacent to each other must have their grids line up node for node. This is performed after potential differences are dealt with. These steps are performed on all regions before continuing.

Once the coordinate lists are refined for all regions, the nodes are created. For each set of coordinates a node is initialized and linked with adjacent nodes. Nodes which are adjacent to other regions are linked across regions to facilitate SLOR line calculations.

The `main::CMStart` function iterates through a set of bias points: ramping biases between each point, going through a set of Newton iterations for each bias point, and going through a set of SLOR iterations for each Newton iteration, finally saving the values of primary variables at each node in a file. The voltage ramping is accomplished through the use of the `region::ramp_voltage` function, which goes through all the non-contacted nodes and applies the potential ramping algorithm. The region on which the contacted bias is being changed also calls `region::ramp_contact_voltage`. This function encapsulates code which would make the `region::ramp_voltage` function more difficult to read. This function simply changes the potential on the contacted nodes. After the bias ramping is complete the Newton iteration begins. For each region, `main::CMStart` calls `region::calc_newton_matrices`, which goes through all the interior nodes and applies difference equations, fixed and floating boundary equations to create linear matrix-vector equations at each node. For each Newton iteration, `main::CMStart` calls `region::calc_slor`.

The lines along which the semi-Gaussian-elimination occurs require only the `calc_slor` function to be called for the substrate region. Once the SLOR calculation is complete for the Newton iteration, another starts until all the Newton iterations are complete. Once all the Newton iterations are complete, `main::CMStart` saves the results in a file, and starts the bias ramping again.

3.8 Testing

Much of this simulator is based on Kurata [4]. In developing this simulator, both the one-dimensional and two-dimensional simulators described in [4] were reproduced. These simulators provided a basis for comparison with the derived equations in this simulator. The results of PN-junction simulations from this simulator were compared with the results from the Kurata reproduction simulators. The shapes of the hole and electron concentration curves and the potential curve were compared.

All the components in this simulator underwent extensive black and white box testing. A one-dimensional PN-junction was compared to an analytic depletion approximation calculation. A one-dimensional PN-junction was calculated in both x and y directions to assure symmetry in x and y direction solutions.

Finally, a PN-junction is calculated and compared with a MEDICI simulation in this section. The results of the MEDICI simulations are shown in Appendix A. The results from this simulator for comparison are shown in Appendix B.

There are few specific results published in recent literature to provide an extensive comparison of the magnetic simulation capability of this simulator. There are, however,

general results published which provide a comparison of trends in potential and currents resulting from an applied magnetic field. Some of these results will be compared with the LMT and SSIMT simulations in the results chapter.

4 Method and Results

4.1 The Device Structures

Two devices have been simulated. The first is a Lateral Magneto Transistor (LMT) [14,15,16]. Figure 4.1 describes the geometry of the LMT simulated. The second

Figure 4.1 LMT structure simulated

is a Suppressed Sidewall Injection Magneto Transistor (SSIMT) [1,17,18]. Figure 4.2 describes the geometry of the SSIMT simulated.

The LMT dopings are as follows: Emitter = $10^{20}/\text{cm}^3$, Collector = $10^{20}/\text{cm}^3$, Base = $-10^{20}/\text{cm}^3$, Substrate = $10^{17}/\text{cm}^3$. The junctions in the emitter, collector and base are at a depth of $1.5 \mu\text{m}$. The number of nodes used in this LMT simulation is 5900.

The SSIMT dopings are as follows: Emitter = $10^{20}/\text{cm}^3$, Collector = $10^{20}/\text{cm}^3$, Base = $-10^{20}/\text{cm}^3$, Substrate = $10^{17}/\text{cm}^3$, Bias Stripe = $-10^{20}/\text{cm}^3$. The junctions in the

Figure 4.2 The SSIMT structure simulated

emitter, collector, base and bias stripe are at a depth of $1.5 \mu\text{m}$. The number of nodes used in this SSIMT simulation is 7200.

There are some important differences between this simulator's representation and fully accurate representations of these devices. There are no spherical junction corners and there is no substrate junction leakage current.

The lack of spherical junction corners is due to the inability of rectangular gridding in this simulator to accommodate it. The number of grid points for a simple circular corner is well beyond the storage capacity of a 32MByte machine. Even if more memory were available, the simulation time would be prohibitive. This simulator uses one-dimensional doping profiles in each region. Sophisticated device structures are obtained in this simulator by butting regions together. Although the resulting simulation does not perfectly represent the physical device, the operation of a device is reasonably simulated.

Contacts and isolation on a device are accomplished by using fixed and floating boundary conditions on regions.

The lack of substrate current leakage is due to the floating boundary conditions. This can be easily circumvented by increasing the depth of the substrate and adding a contact at the bottom of the substrate. The reason this was not done was purposeful. The effects of a retrograde well are reasonably approximated in this simulation. A retrograde well prevents the current from travelling too far from the surface. This has been identified in current literature as a possible improvement on these devices which can enhance sensitivity.

4.2 Interpreting Simulation Results

Since the doping and geometries are the same for both the LMT and SSIMT (excepting the bias stripe) results of both can be compared. The graphs of simulation results are in the orientation shown in Figure 4.3. The simulation results show potential, hole and electron concentrations.

4.3 Constructing the Simulation

The simulation is constructed and executed in two functions. These functions are `main::CMInit` and `main::CMStart` in the "Thesis.cpp" file. The coding required for a simulation is kept minimal. This section will cover the coding for the SSIMT device. The coding for the LMT device is similar.

The `main::CMInit` function contains the coding for creating a description of the

Figure 4.3 Orientation and bounds for graphs

device to be simulated. Doping and region objects are created and associated together. Region objects are stitched together to create more sophisticated devices. Contacts are added.

Each of the regions have a specific doping profile with an associated C++ object. The gaussian doping object is already present in this simulator. This doping object is used by the region which is created. The code to create a region and associated Gaussian doping object follows.

```
d=new gaussian_doping ( north, 1e20, 1e17, 1.5e-4, 0.0, 3e-4, 2e-4, 4e-4);
```

```

for (i=0; i<xcontact; i++) x[i]=(2e-4*(scalar)i/ ((scalar)xcontact-1.0));

for (i=0; i<ycontact; i++) y[i]= 3e-4 + (4e-4*(scalar)i/((scalar)ycontact-1.0));

re=new region(xcontact, ycontact, x, y, d, north, east, 0, NULL);

```

The gaussian_doping parameters required for creation denote:

- 1) The direction of gaussian profile (north, south, east or west),
- 2) The surface doping (/cm³),
- 3) The background doping (/cm³),
- 4) The depth of the junction (cm),
- 5) The x offset of this region with respect to the origin,
- 6) The y offset of this region with respect to the origin,
- 7) The width of this region in the x direction,
- 8) The width of this region in the y direction.

The gaussian_doping created through this process is itself a parameter in the creation of the region object. The region parameters required for creation denote:

- 1) & 2) The number of elements in arrays of x and y coordinates denoting an initial grid for refining.
- 3) & 4) the x and y arrays of coordinates denoting the initial grid for refining.
- 5) The doping object representing the doping in this region.
- 6) The direction (north, south, east or west) in which the SLOR lines progress (e.g. "north" means an east-west line is calculated then another east-west line is calculated

directly north of the previous).

7) The direction which the SLOR line is calculated (e.g.. if parameter 6 was set to "north" and this parameter was set to "east" east-west line would be calculated from west-most node in an eastern direction.

8) The number of adjacent regions to this region. Only one of two adjacent regions sets this to a non-zero value.

9) An array of pointers to adjacent regions.

Once each of the regions is created, they must be initialized and connected. The code to initialize and connect a region follows.

```
re->init(0,NULL);
```

The parameters in this function should be exactly the same as the last two parameters in the call to create the region object. Connection is accomplished by going through the nodes defining the refined grids for each of the adjacent regions and connecting them. If there is no adjacent node one is created and the grid of the respective region is changed to accommodate this new node.

Once each of the regions is initialized and connected, sides of regions are tagged as contacts. An example of code to accomplish this follows.

```
re->contact(north);
```

This function tags the side of the region defined in the parameter as contacted.

This is used during the simulation to determine boundary conditions.

The main::CMStart function performs the voltage ramping and simulation at each bias point. It also saves and retrieves results to facilitate large multi-step simulations. The general algorithm follows.

```

for (each bias point) {
    optionally retrieve previously saved results:
    ramp voltage on any combination of contacts:
    ramp magnetic field:
    for (each newton iteration) {
        calculate matrices for newton iteration;
        for (each SLOR iteration) {
            calculate SLOR solution for overall matrix equation;
        }
    }
}

```

To retrieve previously saved results:

```

f.open("file.sim");
f>>>(*this);
Bz=bz;

```

```
f.close();
```

The file is opened using the C++ stream classes. The ">>" operator has been overridden in this simulator to automatically read the file and create all the data structures. A global variable controls the value of the magnetic field, Bz. In order to save this value it is written as `main::bz`, and this must be written back to Bz in order to bring the simulator back to the state previous to saving. Lastly, the file is closed.

To ramp voltage:

```
re->ramp_voltage(0.1 , ri, north, re, north, rb, north, rc, north);
ri->ramp_voltage(0.1, ri, north, re, north, rb, north, rc, north);
rc->ramp_voltage(0.1, ri, north, re, north, rb, north, rc, north);
rb->ramp_voltage(0.1, ri, north, re, north, rb, north, rc, north);
rs->ramp_voltage(0.1, ri, north, re, north, rb, north, rc, north);
ri->ramp_contact_voltage(0.1,north);
vi+=0.1;
```

The `ramp_voltage` function ramps the voltage in the body of a region to maintain a continuous voltage function. The parameters are:

- 1) The increment (or decrement) in bias on the contact being ramped.
- 2) The region on which the contact being ramped is tagged.
- 3) The side of the region on which the contact is tagged.

4) & 6) & 8) are other contacted regions (whose bias is not being ramped).

5) & 7) & 9) are the sides of the other contacted regions which are tagged.

The `ramp_contact_voltage` function is applied only to the region on which the contact bias is being ramped. The parameters are:

- 1) The increment (or decrement) in the bias.
- 2) The side of the region which is tagged as contacted.

Lastly, a variable to keep track of the applied bias is also incremented. Note also the region objects denoted by "re", "ri", "rc", "rb". "rs" are the emitter, bias stripe, collector, base and substrate regions respectively.

To calculate newton matrices for a region:

```
re->calc_newton_matrices(0);
```

Each region must calculate its own Newton matrixes. The parameter is always 0.

To calculate SLOR results:

```
rs->calc_slor();
```

The SLOR calculation ignores region boundaries necessarily. The simulator code included with this thesis has created each of the regions with SLOR progression from south to north and line progression from west to east. The substrate extends along the complete length (x direction) of the device and the emitter, bias stripe, collector and base are butted to the north of the substrate. It is only necessary to call `calc_slor` for the sub-

strate. The SLOR progression will progress from the bottom of the substrate to the top then it will progress into each of the emitter, bias stripe, collector and base regions ignoring region boundaries.

All of the complicated inner workings of the simulator are hidden from the user. The user must only define the device to be simulated and the biasing conditions required for a set of simulations. The interface is both powerful and simple.

4.4 Simulations

When a magnetic field is applied to a device, the equi-potential lines in the device rotate [1]. This is the basic principle involved in the operation of a Hall plate. More sophisticated devices also display this rotation. Figure 4.4 shows an LMT with no magnetic field. Figure 4.5 shows the same LMT structure with the same bias in a 1 Gauss magnetic field. When the magnetic field is applied, the potential on the right side of the emitter, collector, and base drop, as well as the potential at the bottom of the substrate. The electron concentration graph is denoted by an "N", the hole concentration graph is denoted by a "P" and the potential graph is denoted by "V".

Improvement to magnetic sensitivity by adding a biasing stripe to an LMT, as shown in Figure 4.2 has been demonstrated in current literature. This improvement essentially defines the SSIMT [1]. An early version of this improvement to the LMT was suggested by Popovic et al. [18]. Adding a second base contact to the left of the emitter would provide an accelerating field between the two base contacts. This improves the ratio of collector current to substrate current, sweeping injected electrons towards the col-

Figure 4.4 LMT $V_{emitter}=0$ volt, $V_{base}=1.2$ volt, $V_{collector}=-0.4$ volt, $B=0$ Gauss

lector. The biasing stripe of the SSIMT provides much more than this. The biasing stripes suppress carrier injection from the emitter laterally towards the collector, and, in conjunction with the base contact, create an accelerating field toward the collector[1].

This discussion will concentrate mainly on the accelerating field improvement and constraining carriers in the base close to the surface of the device. The construction of the regions in this simulation assure the suppression of carrier injection from the emitter laterally towards the collector in both the LMT and SSIMT devices simulated, similar to making each contact a mesa structure in a physical device. The structure also eliminates any

Figure 4.5 LMT $V_{emitter}=0$ volt, $V_{base}=1.2$ volt $V_{collector}=-0.4$ volt, $B=1$ Gauss

current leakage into the substrate and constrains lateral current flow relatively close to the surface compared to recently fabricated devices [1,15,18], simulating the effects of a retrograde well. The biasing stripe in the SSIMT provides an accelerating field and a potential hill forcing carriers deeper toward the substrate.

The figure of merit for magneto transistors most often cited in recent literature is relative sensitivity [1,15,18]. This figure of merit will be used with the results of this simulation to aid comparison. The most common units associated with this figure of merit are % collector current change per Tesla.

$$S_r = \frac{\nabla I_{c1}}{I_c B} \times 100 \quad (4.4.1)$$

Recent literature has reported relative sensitivities as high as 3000 %/Tesla for an SSIMT [15]. These devices do not contain retrograde wells and absolute current containment. The simulations presented here indicate substantial improvement in sensitivity when retrograde well structures are implemented. A retrograde well constrains lateral current flow much closer to the collector so that a magnetic field will deflect more of the current flow into the collector. The LMT simulation for $V_{\text{collector}} = -0.4$ volt and $V_{\text{base}} = 1.2$ volt results in 19.6% change in collector current for a 1 Gauss magnetic field. The SSIMT simulation for $V_{\text{collector}} = -0.4$ volt and $V_{\text{base}} = 1.2$ volt results in 191% change in collector current for a 1 Gauss magnetic field. These are 1.96×10^5 %/T and 1.91×10^6 %/T relative sensitivities.

This simulation can be considered ideal and relative sensitivities for physical devices will be lower. Constraining lateral current flow near the surface of the device can be accomplished by depositing a thin lightly doped P type layer on a highly doped P+ substrate creating a reverse biased junction between the epitaxial layer and the substrate. Such a junction will still have current leakage and during the deposition the highly doped substrate may diffuse into the epitaxial layer. This diffusion will make the junction less abrupt and reduce its effectiveness as a barrier to current. Other physical and process dependent variables will also limit performance.

The simulation results show the SSIMT relative sensitivity is largest when the electron concentration difference in the base between emitter and collector is largest. This

differential allows minority carriers to travel at a higher velocity making them more sensitive to magnetic field and deflecting them through larger distances. Table 4.1 details these results.

Table 4.1 Changes in Collector Current at 1 Gauss Magnetic field

Device	V bias stripe	V collector	V base	change in I_c
LMT	N/A	-0.4	1.2	19.6%
LMT	N/A	-0.35	1.2	13.4%
LMT	N/A	-0.3	1.2	-2.5%
LMT	N/A	-0.25	1.2	-25.9%
SSIMT	0	-0.4	1.2	191.0%
SSIMT	0	-0.5	1.2	78.2%
SSIMT	0	-0.6	1.2	26.2%
SSIMT	0	-0.7	1.2	-3.2%
SSIMT	0	-0.8	1.2	-20.6%
SSIMT	0.2	-0.8	1.2	8.4%
SSIMT	0.3	-0.8	1.2	14%

Recently, Riccobene et al. [30] simulated an LMT structure with a buried layer in an attempt to see if the substrate current could be reduced or eliminated. Two buried layers were tried. The PNP LMT was created in an N-well on a P-substrate. The buried N+ layer was centered on the N-well / P-substrate boundary under the emitter. Their calculations displayed that the substrate current simply flowed around the buried layer to the substrate contact. Their calculations also displayed that the high conductivity of the buried layer prevents the formation of a Hall field, further reducing the sensitivity of the LMT.

These findings further support the construction of an LMT or SSIMT in a P-epi layer grown on a P+ substrate of which these simulations are an approximation.

Appendix C contains graphs of selected results for both the LMT and SSIMT simulations.

5 Conclusions

5.1 Finite-Difference Simulator

This simulator has achieved accessibility and portability through its design in standard C++ language for a common computing platform. The steps involved in the design have been illustrated in detail. The improvement of multi-region simulation has enabled this simulator to tackle aggressive simulation problems previously not attempted with other finite-difference semiconductor magnetic field simulators. The results of application to a Lateral Magnetotransistor (LMT) and Suppressed Sidewall Injection Magnetotransistor (SSIMT) have demonstrated the use of this simulator in simulating some effects of the biasing stripe improvement present in the SSIMT which have been presented in recent literature on the subject [1,14,15,16,17,18].

This thesis has presented evidence that a relatively simple and inexpensive simulator can perform some important functions which have typically been limited to complex, expensive simulators. Discussion has been presented on choices made during the construction of this simulator. These choices may aid any person contemplating modifying and using this simulator for other aggressive simulation projects.

5.2 Future Work

Future work can be split into three categories: device description improvements, simulation control improvements and simulator improvements.

Device description improvements would involve the integration of a graphical

device description editor which would serve the same purpose as coding the `main::CMInit` and `main::CMStart` functions for controlling simulation. This graphical device description would also benefit from being able to read in device layouts from other popular layout software. These improvements would aid accessibility to the simulator through improved ease of use.

Simulation control improvements would involve menu-based simulation control, precision based rather than iteration based simulation endpoint detection, and batch simulation capabilities. Menu-based simulation control would improve ease of use. Precision based endpoint detection would aid the quality of the simulation results by assuring the simulation is in a pre-specified precision range. Currently, if the results are not in a desired precision range the bias point must be re-simulated with more iterations. Batch simulation capabilities would greatly aid the simulation of large devices or large numbers of bias points.

Simulator improvements would involve triangular gridding and higher order approximation capabilities. Triangular gridding would make this simulator very precise. The finite difference method would have to be modified significantly to accommodate this. The SLOR matrix solver would have to allow line calculation based on a weighted cost routing method or other routing method similar to an autorouter. This improvement could have beneficial effects on simulator performance. Higher order approximation would aid convergence speed and quality, but would involve either multiple coded routines to calculate equations depending on the order of the calculation required, or a dynamic routine with symbolic manipulation capabilities.

6 References

- [1] T. Smy, The SSIMT: A CMOS Based Magnetic Sensor, PhD. Thesis, University of Alberta, Canada, 1990.
- [2] R. Pallas-Areny and J. G. Webster, Sensors and Signal Conditioning, New York, New York, W.J. Wiley, 1991.
- [3] Sensor Technology and Devices, Boston, Massachusetts, Artech House, 1994.
- [4] M. Kurata, Numerical Analysis for Semiconductor Devices, Lexington, Massachusetts, D.C. Heath and Company, 1982.
- [5] J. Vlach and K. Singhal, Computer Methods for Circuit Analysis and Design, New York, New York, Van Nostrand Reinhold, 1983.
- [6] A. Jennings and J.J. McKeown, Matrix Computation, 2nd Edition, New York, New York, W.J. Wiley, 1992.
- [7] R.S. Varga, Matrix Iterative Analysis, Englewood Cliffs, NJ, Prentice-Hall, 1962.
- [8] S.M. Sze, Physics of Semiconductor Devices, 2nd Edition, New York, New York, W.J. Wiley, 1981.
- [9] J. Allison, Electronic Engineering Semiconductors and Devices, London, McGraw Hill, 1990.
- [10] A. Nathan, Carrier Transport in Magnetotransistors, PhD. Thesis, University of Alberta, Canada, 1988.
- [11] A. Nathan, A. M.J. Huizer and H. P. Baltes, "Two-Dimensional Numerical Modeling of Magnetic-Field Sensors in CMOS Technology", *IEEE Transactions on Electron Devices*, Vol. ED-32, No. 7, pp 1212-1219, July 1985.
- [12] L. Andor, H. P. Bales, A. Nathan and H. G. Schmidt-Weinmar, "Numerical Modeling of Magnetic-Field-Sensitive Semiconductor Devices", *IEEE Transactions on Electron Devices*, Vol. ED-32, No.7, pp. 1224-1230, July 1985.
- [13] D.L. Scharfetter and H.K. Gummel, "Large-Signal Analysis of a Silicon Read Diode Oscillator", *IEEE Transactions on Electron Devices*, Vol. ED-16, No.1, pp 64-77, January 1969.

- [14] S. Chou, "An Investigation of Lateral Transistors - D.C. Characteristics", *Solid-State Electronics*, Pergamon Press, Vol. 14, pp. 811-826, 1971.
- [15] L. Ristic, T. Smy and H. P. Baltes, "A Lateral Magnetotransistor Structure with a Linear Response to the Magnetic Field", *IEEE Transactions on Electron Devices*, Vol. 36, No. 6, pp 1076-1086, June 1989.
- [16] R.S. Popovic and H.P. Baltes, "An Investigation of the Sensitivity of Lateral Magnetotransistors", *IEEE Electron Device Letters*, Vol. EDL-4, No. 3, pp 51-53, March 1983.
- [17] L.J. Ristic, H.P. Baltes, T. Smy and I. Filanovsky, "Suppressed Sidewall Injection Magnetotransistor with Focused Emitter Injection and Carrier Double Deflection", *IEEE Electron Device Letters*, Vol EDL-8, No. 9, pp 395-397, September 1987.
- [18] R. S. Popovic and R. Widmer, "Magnetotransistor in CMOS Technology", *IEEE Transactions on Electron Devices*, Vol ED-33, No. 9, pp 1334-1341, September 1986.
- [19] W. G. Wolber and K. D. Wise, "Sensor Development in the Microcomputer Age", *IEEE Transactions on Electron Devices*, Vol. ED-26, No.12, pp 1864-1874, December 1979.
- [20] S. Kordic, "Integrated Silicon Magnetic-Field Sensors", *Sensors and Actuators*, 10, pp 347-378, 1986.
- [21] A.W Vinal and N.A. Masnari, "Magnetic Transistor Behaviour Explained by Modulation of Emitter Injection, Not Carrier Deflection", *IEEE Electron Device Letters*, vol. EDL-3, pp. 203-205, 1982.
- [22] V. Zieren, S.Kordic, and S. Middelhoek, "Comment on 'Magnetic Transistor Behaviour Explained by Modulation of Emitter Injection, Not Carrier Deflection'", *IEEE Electron Device Letters*, vol. EDL-3, pp. 394-395, 1982.
- [23] A.W. Vinal and N.A. Masnari, "Response to 'Comment on 'Magnetic Transistor Behaviour Explained by Modulation of Emitter Injection, Not Carrier Deflection'", *IEEE Electron Device Letters*, vol EDL-3, pp. 396-397, 1982.
- [24] I.M. Mitnikova, T.V. Perysiyanov, B.I. Relakova and G. Shtyubner, "Investigation of the Characteristics of Silicon Lateral Magnetotransistors with Two Measuring Collectors", *Sov. Phys. Semicond.*, vol. 12, pp. 26-28, 1978.

- [25] A.W. Vinal and N.A. Masnari, "Operating Principles of Bipolar Transistor Magnetic Sensors", IEEE Transactions on Electron Devices, Vol. ED-31, No. 10, pp. 1486-1494, 1984.
- [26] A. Nathan, Carrier Transport in Magnetotransistors, Ph.D. Thesis, University of Alberta, Canada, 1988.
- [27] A.M.J. Huizer and H.P. Baltes, "Numerical Modeling of Vertical Hall-Effect Devices", IEEE Electron Device Letters, Vol. EDL-5, No. 11, pp. 482-484, 1984.
- [28] A. Nathan, A.M.J. Huizer and H.P. Baltes, "Two-Dimensional Numerical Modeling of Magnetic Field Sensors in CMOS Technology", IEEE Transactions on Electron Devices, Vol. ED-32, No. 7, pp. 1212-1219, 1985.
- [29] L. Andor, H.P. Baltes, A. Nathan and H.G. Schmidt-Weinmar, "Numerical Modeling of Magnetic-Field-Sensitive Semiconductor Devices", IEEE Transactions on Electron Devices, Vol. ED-32, No. 7, pp. 1224-1230, 1985.
- [30] C. Riccobene, G. Wachutka, J. Burgler, H. Baltes, "Operating Principle of Dual Collector Magnetotransistors Studied by Two-Dimensional Simulation", IEEE Transactions on Electron Devices, Vol. ED-41, No. 1, pp. 32-43, 1994.

7 Appendix A - Medici PN-Junction Results

The device simulated is $3\text{ }\mu\text{m}$ by $3\text{ }\mu\text{m}$ length and depth with a P-type Gaussian doping of $10^{19}/\text{cm}^3$ with a junction at $1.5\text{ }\mu\text{m}$ depth on an N-type uniform doping of $10^{15}/\text{cm}^3$.

Figure 7.1 MEDICI - Zero Bias PN-Junction (Potential)

Figure 7.2 MEDICI - Zero Bias PN-Junction (Electron Concentration)

Figure 7.3 MEDICI - Zero Bias PN-Junction (Hole Concentration)

Figure 7.4 MEDICI - 0.6 volt Forward Bias PN-Junction (Potential)

Figure 7.5 MEDICI - 0.6 volt Forward Bias PN-Junction (Electron Concentration)

Figure 7.6 MEDICI - 0.6 volt Forward Bias PN-Junction (Hole Concentration)

Figure 7.7 MEDICI - 0.8 volt Forward Bias PN-Junction (Potential)

Figure 7.8 MEDICI - 0.8 volt Forward Bias PN-Junction (Electron Concentration)

Figure 7.9 MEDICI - 0.8 volt Forward Bias PN-Junction (Hole Concentration)

Figure 7.10 MEDICI - 1.0 volt Forward Bias PN-Junction (Potential)

Figure 7.11 MEDICI - 1.0 volt Forward Bias PN-Junction (Electron Concentration)

Figure 7.12 MEDICI - 1.0 volt Forward Bias PN-Junction (Hole Concentration)

8 Appendix B - Finite-Difference Simulator PN-Junction Results

The device simulated is $3\text{ }\mu\text{m}$ by $3\text{ }\mu\text{m}$ length and depth with a P-type Gaussian doping of $10^{19}/\text{cm}^3$ with a junction at $1.5\text{ }\mu\text{m}$ depth on an N-type uniform doping of $10^{15}/\text{cm}^3$. On the graphs: Electron Concentration is denoted by an "N"; Hole concentration is denoted by an "P"; Potential is denoted by a "V".

Figure 8.1 PN-Junction - Zero Bias

Figure 8.2 PN-Junction - 0.6 volt forward bias

Figure 8.3 PN-Junction - 0.8 volt forward bias

Figure 8.4 PN-Junction - 1.0 volt forward bias

9 Appendix C - Finite-Difference Simulator LMT and SSIMT Results

73

On the graphs: Electron Concentration is denoted by an "N"; Hole concentration is denoted by an "P"; Potential is denoted by a "V".

Figure 9.1 LMT - Zero Bias, Zero Magnetic Field

Figure 9.2 LMT - $V_{\text{base}} = 1.2$ volt, $B = 0$ Gauss

Figure 9.3 LMT- $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.25$ volt, $B=0$ Gauss

Figure 10.3 LMT- $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.25$ volt, $B=0$ Gauss

Figure 10.4 LMT- $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.25$ volt, $B=1$ Gauss

Figure 10.5 LMT - $V_{base}=1.2$ volt, $V_{collector}=-0.4$ volt, $B=0$ Gauss

Figure 10.6 LMT - $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.4$ volt, $B=1$ Gauss

Figure 10.7 SSIMT Zero bias, Zero Magnetic Field

Figure 10.8 SSIMT - $V_{\text{base}} = 1.2$ volt, $B = 0$ Gauss

Figure 10.9 SSIMT - $V_{base}=1.2$ volt, $V_{collector}=-0.4$ volt, $B=0$ Gauss

Figure 10.10 SSIMT - $V_{base} = 1.2$ volt, $V_{collector} = -0.4$ volt, $B = 1$ Gauss

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Figure 10.11 SSMT - $V_{base}=1.2$ volt, $V_{collector}=-0.8$ volt, $B=0$ Gauss

Figure 10.12 SSIMT - $V_{base}=1.2$ volt, $V_{collector}=-0.8$, $B=1$ Gauss

Figure 10.13 SSIMT - $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.8$ volt, $V_{\text{bias stripe}}=0.2$ volt, $B=0$ Gauss

Figure 10.14 SSIMT - $V_{base}=1.2$ volt, $V_{collector}=-0.8$ volt, $V_{bias\ stripe}=0.2$ volt, $B=1$ Gauss

Figure 10.15 SSIMT - $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.8$ volt, $V_{\text{bias stripe}}=0.3$ volt, $B=0$ Gauss

Figure 10.16 SSIMT - $V_{\text{base}}=1.2$ volt, $V_{\text{collector}}=-0.8$ volt, $V_{\text{bias stripe}}=0.3$ volt, $B=1$ Gauss

10 Appendix D - Simulator Code

10.1 Overview of Simulator Files

Type.h: constants and variable types used globally in the simulator.

Vector.h: storage and operators for a 3D vector quantity.

Matrix.h: storage and operators for a 3D matrix quantity.

Gauss.h: object describing doping for a region.

Node.h: object describing a simulation node.

Region.h: object describing a rectangular simulation region.

Thesis1.h: contains the main object describing the structure of the device and current state of the simulation.

Thesisrc.h: contains constants which relate the resource file (Thesis1.rc) with the main simulation object so that the main functions can be reached through menu items in the application's window.

Thesis1.cpp: contains all the functions related to the main object.

Region.cpp: contains all the functions related to the region object.

Node.cpp, Node1.cpp: contains all the functions related to the node object.

Fileio.cpp: contains operators for reading and writing to file.

Thesis1.rc: the windows resource file describing the menus and dialog boxes used as the user interface to this simulator.

Thesis1.def: a windows definition file which specifies how this application is treated by windows, how much memory to be used by the stack, how modules are loaded etc.

10.2 Type.h

```

    copyright 1995, Brian Greenham
#include TYPE_INCLUDE1
#define TYPE_INCLUDE1
typedef double scalar

#define north
#define south 1
#define east 2
#define west 3

#define ni 1.48E17          intrinsic carrier density of Si
#define q 1.6 22E-19        charge on an electron
#define V 1.3406E-23        silicon band constant (eV)
#define T 300               temperature of operation (K)
#define es 1.64E-12          permittivity of silicon (F/cm)
#define theta 34.4415        q / k T

    // mobility parameters
#define Nrefp 5.3E16          cm-3
#define Alpha 0.76           cm-2 V-1/2sec-1
#define Umaxp 495.0          cm-2 V-1/2sec-1
#define Uminp 47.7           cm-2 V-1/2sec-1
#define Ecp 1.95E4           volt/cm
#define Betap 1.0
#define Nrefn 9.5E16         cm-3
#define Alphan 0.72
#define Umaxn 1330.0         cm-2 V-1/2sec-1
#define Uminn 65.0          cm-2 V-1/2sec-1
#define Ecn 9E3             volt/cm
#define Betan 2.0

```

```
// impact ionization carrier generation parameters
#define apo 2.25E7 // (/cm)
#define Epo 3.26E6 // (volt/cm)
#define ano 3.8E6
#define Eno 1.75E6

// recombination parameters
#define tp 2.5E-3
#define tn 2.5E-3
#define raug 0.0
*/
#endif
```

10.3 Vector.h

```
// Copyright 1995, Brian Gieschen
#include <iostream.h>
#include "type.h"
#ifdef VECTOR_INCLUDED
#define VECTOR_INCLUDED
// this class includes all the mathematical operators for 3D vector operations
class vector;
extern ostream& operator >>(ostream&,vector&);
extern istream& operator >>(istream&,vector&);
class vector {
    friend ostream& operator >>(ostream&,vector&);
    friend istream& operator >>(istream&,vector&);
protected:
    scalar v[3];
public:
    vector() {
        v[0]=v[1]=v[2]=0.0;}
    vector(scalar a) {
        v[0]=v[1]=v[2]=a;}
    scalar& operator [](unsigned int i) {
        return v[i];} // error handle i>2
    vector& operator =(vector& a) {
        v[0]=a[0];v[1]=a[1];v[2]=a[2];return *this;}
    vector& operator =(scalar a) {
        v[0]=a;v[1]=a;v[2]=a;return *this;}
    vector operator *(scalar a) {
        vector b;b[0]=v[0]*a;b[1]=v[1]*a;b[2]=v[2]*a;return b;}
    vector operator /(scalar a) {
        vector b;b[0]=v[0]/a;b[1]=v[1]/a;b[2]=v[2]/a;return b;}
    vector operator +(vector& a) {
        vector b;b[0]=v[0]+a[0];b[1]=v[1]+a[1];b[2]=v[2]+a[2];return b;}
    vector operator -(vector& a) {
        vector b;b[0]=v[0]-a[0];b[1]=v[1]-a[1];b[2]=v[2]-a[2];return b;}
    vector& operator +=(vector& a) {
        v[0]=v[0]+a[0];v[1]=v[1]+a[1];v[2]=v[2]+a[2];return *this;}
    vector& operator +=(scalar a) {
        v[0]=a;v[1]=a;v[2]=a;return *this;}
    vector& operator /=(scalar a) {
        v[0]=v[0]/a;v[1]=v[1]/a;v[2]=v[2]/a;return *this;}
/*
    vector& abs() {
        sabs(v[0]);sabs(v[1]);sabs(v[2]);return *this;}
    int operator ==(scalar a) {
        return ((v[0]==a)|| (v[1]==a)|| (v[2]==a));}
    int operator ==(vector& a) {
        return ((v[0]==a[0])&&(v[1]==a[1])&&(v[2]==a[2]));}
    int operator >(vector& a) {
        return (sabs(v[2])>sabs(a[2]));}
    int operator <(vector& a) {
        return (sabs(v[2])<sabs(a[2]));}
*/
};
#endif
```

10.4 Matrix.h

```
// Copyright 1995, Brian Gieschen
#include <iostream.h>
#include "type.h"
#include "vector.h"
```

```

#ifdef MATRIX_INCLUDED
#define MATRIX_INCLUDED
// this class contains all the operators to perform all the 3D
// matrix operations necessary it performs inversion and calculates
// the determinant and performs matrix multiplication
// this allows the program to treat matrix equations much
// the way it would treat scalar equations.
// the equation X=A*B+C would be typed the same way for
// matrix or scalar. This clarifies the code immensely.
class matrix;
extern ostream& operator >>(ostream&,matrix&);
extern istream& operator <<(istream&,matrix&);
class matrix {
    friend ostream& operator >>(ostream&,matrix&);
    friend istream& operator <<(istream&,matrix&);
protected:
    vector m[3];
public:
    matrix() {}
    vector& operator [] (unsigned int i) {
        return m[i]; // error handle i>2
    }
    matrix& operator =(matrix &a) {
        m[0][0]=a[0][0];m[0][1]=a[0][1];m[0][2]=a[0][2];
        m[1][0]=a[1][0];m[1][1]=a[1][1];m[1][2]=a[1][2];
        m[2][0]=a[2][0];m[2][1]=a[2][1];m[2][2]=a[2][2];
        return *this;
    }
    matrix& operator =(scalar a) {
        m[0]=a;m[1]=a;m[2]=a;return *this;
    }
    matrix operator *(matrix &a) {
        matrix b;
        b[0][0]=m[0][0]*a[0][0]+m[0][1]*a[1][0]+m[0][2]*a[2][0];
        b[0][1]=m[0][0]*a[0][1]+m[0][1]*a[1][1]+m[0][2]*a[2][1];
        b[0][2]=m[0][0]*a[0][2]+m[0][1]*a[1][2]+m[0][2]*a[2][2];
        b[1][0]=m[1][0]*a[0][0]+m[1][1]*a[1][0]+m[1][2]*a[2][0];
        b[1][1]=m[1][0]*a[0][1]+m[1][1]*a[1][1]+m[1][2]*a[2][1];
        b[1][2]=m[1][0]*a[0][2]+m[1][1]*a[1][2]+m[1][2]*a[2][2];
        b[2][0]=m[2][0]*a[0][0]+m[2][1]*a[1][0]+m[2][2]*a[2][0];
        b[2][1]=m[2][0]*a[0][1]+m[2][1]*a[1][1]+m[2][2]*a[2][1];
        b[2][2]=m[2][0]*a[0][2]+m[2][1]*a[1][2]+m[2][2]*a[2][2];
        return b;
    }
    vector operator *(vector &a) {
        vector b;
        b[0]=m[0][0]*a[0]+m[0][1]*a[1]+m[0][2]*a[2];
        b[1]=m[1][0]*a[0]+m[1][1]*a[1]+m[1][2]*a[2];
        b[2]=m[2][0]*a[0]+m[2][1]*a[1]+m[2][2]*a[2];
        return b;
    }
    matrix operator *(scalar a) {
        matrix b;
        b[0][0]=m[0][0]*a;b[0][1]=m[0][1]*a;b[0][2]=m[0][2]*a;
        b[1][0]=m[1][0]*a;b[1][1]=m[1][1]*a;b[1][2]=m[1][2]*a;
        b[2][0]=m[2][0]*a;b[2][1]=m[2][1]*a;b[2][2]=m[2][2]*a;
        return b;
    }
    matrix operator +(matrix &a) {
        matrix b;
        b[0][0]=m[0][0]+a[0][0];b[0][1]=m[0][1]+a[0][1];b[0][2]=m[0][2]+a[0][2];
        b[1][0]=m[1][0]+a[1][0];b[1][1]=m[1][1]+a[1][1];b[1][2]=m[1][2]+a[1][2];
        b[2][0]=m[2][0]+a[2][0];b[2][1]=m[2][1]+a[2][1];b[2][2]=m[2][2]+a[2][2];
        return b;
    }
    matrix operator -(matrix &a) {
        matrix b;
        b[0][0]=m[0][0]-a[0][0];b[0][1]=m[0][1]-a[0][1];b[0][2]=m[0][2]-a[0][2];
        b[1][0]=m[1][0]-a[1][0];b[1][1]=m[1][1]-a[1][1];b[1][2]=m[1][2]-a[1][2];
        b[2][0]=m[2][0]-a[2][0];b[2][1]=m[2][1]-a[2][1];b[2][2]=m[2][2]-a[2][2];
        return b;
    }
    matrix& operator +=(matrix &a) {
        m[0][0]=m[0][0]+a[0][0];m[0][1]=m[0][1]+a[0][1];m[0][2]=m[0][2]+a[0][2];
        m[1][0]=m[1][0]+a[1][0];m[1][1]=m[1][1]+a[1][1];m[1][2]=m[1][2]+a[1][2];
        m[2][0]=m[2][0]+a[2][0];m[2][1]=m[2][1]+a[2][1];m[2][2]=m[2][2]+a[2][2];
        return *this;
    }
    matrix& operator -=(matrix &a) {
        m[0][0]=m[0][0]-a[0][0];m[0][1]=m[0][1]-a[0][1];m[0][2]=m[0][2]-a[0][2];
        m[1][0]=m[1][0]-a[1][0];m[1][1]=m[1][1]-a[1][1];m[1][2]=m[1][2]-a[1][2];
        m[2][0]=m[2][0]-a[2][0];m[2][1]=m[2][1]-a[2][1];m[2][2]=m[2][2]-a[2][2];
        return *this;
    }
    scalar det() {

```

```

    scalar a,b,c;
    a=m[0][2]*((m[1][0]*m[2][1])-(m[2][0]*m[1][1]));
    b=m[1][2]*((m[0][0]*m[2][1])-(m[2][0]*m[0][1]));
    c=m[2][2]*((m[0][0]*m[1][1])-(m[1][0]*m[0][1]));
    return a-b+c;}
matrix inv() {
    matrix b;
    scalar a=det();
    b[0][0]=((m[1][1]*m[2][2])-(m[2][1]*m[1][2]))/a;
    b[1][0]=(-(m[1][0]*m[2][2])-(m[2][0]*m[1][2]))/a;
    b[2][0]=((m[1][0]*m[2][1])-(m[2][0]*m[1][1]))/a;
    b[0][1]=(-(m[0][1]*m[2][2])-(m[2][1]*m[0][2]))/a;
    b[1][1]=((m[0][0]*m[2][2])-(m[2][0]*m[0][2]))/a;
    b[2][1]=(-(m[0][0]*m[2][1])-(m[2][0]*m[0][1]))/a;
    b[0][2]=((m[0][1]*m[1][2])-(m[1][1]*m[0][2]))/a;
    b[1][2]=(-(m[0][0]*m[1][2])-(m[1][0]*m[0][2]))/a;
    b[2][2]=((m[0][0]*m[1][1])-(m[1][0]*m[0][1]))/a;
    return b;}
};
#endif

```

10.5 Gauss.h

```

// Copyright 1995, Brian Gieschen
#include <iostream.h>
#include "type.h"
#include <math.h>
#ifdef GAUSS_INCLUDED
#define GAUSS_INCLUDED
// this object contains all the info required for calculating
// the doping in a gaussian profile anywhere in a region.
// it can also calculate the thermal equilibrium, hole and electron
// concentration and potential at any point in a region.
class gaussian_doping;
extern ostream& operator <<(ostream&,gaussian_doping&);
extern istream& operator >>(istream&,gaussian_doping&);
class gaussian_doping {
    friend ostream& operator <<(ostream&,gaussian_doping&);
    friend istream& operator >>(istream&,gaussian_doping&);
protected:
    scalar Ne,Nb,ew,xw,yw,xo,yo;
    unsigned int dir;
public:
    gaussian_doping() {dir=0;Ne=Nb=xo=yo=ew=xw=yw=0.0;}
    gaussian_doping(unsigned int _dir,scalar _Ne,scalar _Nb,scalar _ew,
    scalar _xo,scalar _yo,scalar _xw,scalar _yw) {
        dir=_dir;Ne=_Ne;Nb=_Nb;xo=_xo;yo=_yo;ew=_ew;xw=_xw;yw=_yw;}
    scalar N(scalar x,scalar y) {
        scalar j;
        if ((dir==east)|| (dir==west)) j=x-xo; else j=y-yo;
        if (dir==west) j=xw-j; else if (dir==north) j=yw-j;
        if (Ne==0.0) return -Nb;
        return -Nb*Ne*exp(-j*j/ew/ew*log(fabs(Ne/Nb)));}
    scalar voltage(scalar x,scalar y) {
        scalar D=N(x,y);
        if (D>0.0) return K*T/q*log((D/2.0+sqrt(D*D/4.0+ni*ni))/ni);
        else if (D<0.0) return -K*T/q*log((-D/2.0+sqrt(D*D/4.0+ni*ni))/ni);
        return 0.0;}
    scalar hole(scalar x, scalar y) {
        scalar D=N(x,y);
        if (D>0.0) return ni*ni/(D/2.0+sqrt(D*D/4.0+ni*ni));
        else if (D<0.0) return -D/2.0+sqrt(D*D/4.0+ni*ni);
        return ni;}
    scalar electron(scalar x, scalar y) {
        scalar D=N(x,y);
        if (D>0.0) return D/2.0+sqrt(D*D/4.0+ni*ni);
        else if (D<0.0) return ni*ni/(-D/2.0+sqrt(D*D/4.0+ni*ni));
        return ni;}
};
#endif

```

10.6 Node.h

```

// Copyright 1995, Brian Gieschen
#include <iostream.h>

```

```

#include "type.h"
#include "vector.h"
#include "matrix.h"
#include "gauss.h"
#ifndef NODE_INCLUDED
#define NODE_INCLUDED

// flag settings (these should be treated as mutually exclusive
#define inside 0x01
#define floating_boundary 0x02
#define contacted_boundary 0x04

// used in the slor algorithm
#define north_travel 0x01
#define south_travel 0x02
#define east_travel 0x04
#define west_travel 0x08
#define north_include 0x10
#define south_include 0x20
#define east_include 0x40
#define west_include 0x80
#define start_of_travel 0x100
#define end_of_travel 0x200

// used to access components of the solution
#define hole 0
#define elec 1
#define volt 2

// the node object stores the hole, electron concentration
// and potential in the vector Y.
// D contains the doping, x and y contain the coordinates
// flag contains the inside, floating or contacted boundary info
// F is the same F vector as that found using Kurata's terminology
// C refers to the matrix relating this nodes Y vector to itself.
// N refers to the matrix relating this nodes Y vector to the north adjacent nodes Y.
// S refers to the matrix relating this nodes Y vector to the south adjacent nodes Y.
// E refers to the matrix relating this nodes Y vector to the east adjacent nodes Y.
// W refers to the matrix relating this nodes Y vector to the west adjacent nodes Y.
// NE refers to the matrix relating this nodes Y vector to the northeast adjacent nodes Y.
// NW, SW, SE etc..
// Ft and Ct are temporary variables used in the SLOR calculations for storing
// intermediate results.
// n,s,e,w are pointers to their respective adjacent nodes.
// Ynp, and Yip are the previous newton and slor iteration values of Y
class region;
extern void dump(region*,char*);
class node;
extern ostream& operator <<(ostream&,node&);
extern istream& operator >>(istream&,node&);
class node {
    friend void dump(region*,char*);
    friend ostream& operator <<(ostream&,node&);
    friend istream& operator >>(istream&,node&);
protected:
    node *n,*s,*e,*w;
    unsigned int flag;
    scalar D,x,y;
    vector Y,F,Ft,Ynp,Yip;
    matrix N,S,E,W,NW,NE,SW,SE,C,Ct;
    void set_flag();
public:
    node() (n=s=e=w=NULL;flag=0;D=x=y=0.0;Y=F=Ft=Ynp=Yip=0.0;
        N=S=E=W=NW=NE=SW=SE=C=Ct=0.0;)
    node(scalar,scalar,scalar);
    ~node();
    node& operator =(node& a) {
        Y=a.Y;return *this;}
    void link(node *,node *,node *,node *);
    void ramp_voltage(scalar,scalar,scalar,scalar,scalar,
        scalar,scalar,scalar,scalar);
    scalar hx();
    scalar hy();
    scalar Hx(unsigned int,gaussian_doping*);
    scalar Hy(unsigned int,gaussian_doping*);

```

```

scalar hxp();
scalar hyp();
scalar upx(gaussian_doping*);
scalar upy(gaussian_doping*);
scalar unx(gaussian_doping*);
scalar uny(gaussian_doping*);
scalar U();
scalar dU_dpMN();
scalar dU_dnMN();
scalar Jpx(gaussian_doping*);
scalar Jpy(gaussian_doping*);
scalar Jnx(gaussian_doping*);
scalar Jny(gaussian_doping*);
scalar Jx(gaussian_doping *d) {return Jpx(d)+Jnx(d);}
scalar Jy(gaussian_doping *d) {return Jpy(d)+Jny(d);}
int JxOK() {return ((n)&&(s)&&(w)&&(n->w)&&(s->w));}
int JyOK() {return ((n)&&(e)&&(w)&&(n->e)&&(n->w));}
scalar qI(unsigned int, gaussian_doping*);
scalar Ex();
scalar Ey();
int ExOK() {return (w!=NULL);}
int EyOK() {return (n!=NULL);}
scalar Exi(gaussian_doping*, scalar*, scalar*, scalar*, scalar*);
scalar Eyi(gaussian_doping*, scalar*, scalar*, scalar*, scalar*);
scalar Bx(unsigned int, gaussian_doping*, scalar*, scalar*, scalar*, scalar*);
scalar By(unsigned int, gaussian_doping*, scalar*, scalar*, scalar*, scalar*);
scalar dJpx_dpMN(gaussian_doping*);
scalar dJpx_dpM1N(gaussian_doping*);
scalar dJpy_dpMN(gaussian_doping*);
scalar dJpy_dpM1N(gaussian_doping*);
scalar dJnx_dnMN(gaussian_doping*);
scalar dJnx_dnM1N(gaussian_doping*);
scalar dJny_dnMN(gaussian_doping*);
scalar dJny_dnM1N(gaussian_doping*);
scalar dJpx_dvMN(gaussian_doping*);
scalar dJpx_dvM1N(gaussian_doping*);
scalar dJpx_dvM0(gaussian_doping*);
scalar dJpx_dvM1N0(gaussian_doping*);
scalar dJpx_dvMN1(gaussian_doping*);
scalar dJpx_dvM1N1(gaussian_doping*);
scalar dJpy_dvMN(gaussian_doping*);
scalar dJpy_dvMN1(gaussian_doping*);
scalar dJpy_dvMON(gaussian_doping*);
scalar dJpy_dvMON1(gaussian_doping*);
scalar dJpy_dvM1N(gaussian_doping*);
scalar dJpy_dvM1N1(gaussian_doping*);
scalar dJnx_dvMN(gaussian_doping*);
scalar dJnx_dvM1N(gaussian_doping*);
scalar dJnx_dvM0(gaussian_doping*);
scalar dJnx_dvM1N0(gaussian_doping*);
scalar dJnx_dvMN1(gaussian_doping*);
scalar dJnx_dvM1N1(gaussian_doping*);
scalar dJny_dvMN(gaussian_doping*);
scalar dJny_dvMN1(gaussian_doping*);
scalar dJny_dvMON(gaussian_doping*);
scalar dJny_dvMON1(gaussian_doping*);
scalar dJny_dvM1N(gaussian_doping*);
scalar dJny_dvM1N1(gaussian_doping*);
scalar ldJpx_dpMN(gaussian_doping*);
scalar ldJpx_dpM1N(gaussian_doping*);
scalar ldJpy_dpMN(gaussian_doping*);
scalar ldJpy_dpM1N(gaussian_doping*);
scalar ldJnx_dnMN(gaussian_doping*);
scalar ldJnx_dnM1N(gaussian_doping*);
scalar ldJny_dnMN(gaussian_doping*);
scalar ldJny_dnM1N(gaussian_doping*);
scalar ldJpx_dvMN(gaussian_doping*);
scalar ldJpx_dvM1N(gaussian_doping*);
scalar ldJpx_dvM0(gaussian_doping*);
scalar ldJpx_dvM1N0(gaussian_doping*);
scalar ldJpx_dvMN1(gaussian_doping*);
scalar ldJpx_dvM1N1(gaussian_doping*);
scalar ldJpy_dvMN(gaussian_doping*);
scalar ldJpy_dvMN1(gaussian_doping*);
scalar ldJpy_dvMON(gaussian_doping*);

```

```

scalar ldJpy_dvMON1(gaussian_doping*);
scalar ldJpy_dvM1N(gaussian_doping*);
scalar ldJpy_dvM1N1(gaussian_doping*);
scalar ldJnx_dvMN(gaussian_doping*);
scalar ldJnx_dvM1N(gaussian_doping*);
scalar ldJnx_dvMN0(gaussian_doping*);
scalar ldJnx_dvM1N0(gaussian_doping*);
scalar ldJnx_dvMN1(gaussian_doping*);
scalar ldJnx_dvM1N1(gaussian_doping*);
scalar ldJny_dvMN(gaussian_doping*);
scalar ldJny_dvMN1(gaussian_doping*);
scalar ldJny_dvMON(gaussian_doping*);
scalar ldJny_dvMON1(gaussian_doping*);
scalar ldJny_dvM1N(gaussian_doping*);
scalar ldJny_dvM1N1(gaussian_doping*);
matrix nNt(gaussian_doping*);
matrix eNt(gaussian_doping*);
matrix wNt(gaussian_doping*);
matrix sSt(gaussian_doping*);
matrix eSt(gaussian_doping*);
matrix wSt(gaussian_doping*);
matrix eEt(gaussian_doping*);
matrix nEt(gaussian_doping*);
matrix sEt(gaussian_doping*);
matrix wEt(gaussian_doping*);
matrix nWt(gaussian_doping*);
matrix sWt(gaussian_doping*);
matrix wNwt(gaussian_doping*);
matrix nNwt(gaussian_doping*);
matrix eNEt(gaussian_doping*);
matrix nNEt(gaussian_doping*);
matrix wSWt(gaussian_doping*);
matrix sSWt(gaussian_doping*);
matrix eSEt(gaussian_doping*);
matrix sSEt(gaussian_doping*);
matrix lnNt(gaussian_doping*);
matrix leNt(gaussian_doping*);
matrix lwNt(gaussian_doping*);
matrix lsSt(gaussian_doping*);
matrix leSt(gaussian_doping*);
matrix lwSt(gaussian_doping*);
matrix leEt(gaussian_doping*);
matrix lnEt(gaussian_doping*);
matrix lsEt(gaussian_doping*);
matrix lwEt(gaussian_doping*);
matrix lnWt(gaussian_doping*);
matrix lsWt(gaussian_doping*);
matrix lwWt(gaussian_doping*);
matrix lnNwt(gaussian_doping*);
matrix leNEt(gaussian_doping*);
matrix lnNEt(gaussian_doping*);
matrix lwSWt(gaussian_doping*);
matrix lsSWt(gaussian_doping*);
matrix leSEt(gaussian_doping*);
matrix lsSEt(gaussian_doping*);
matrix SST(unsigned int);
vector Rt(unsigned int);
void calc_LD_newton_matrices(unsigned int, gaussian_doping*);
void calc_newton_matrices(gaussian_doping* int);
void calc_slor(unsigned int);
void back_sub(unsigned int);
void calc_boundary(unsigned int);
void calc_corner();
void update_newton();
void set_floating_boundary() {
    flag&=(-inside)&(~contacted_boundary);
    flag|=floating_boundary;
}
int is_floating_boundary() {
    if ((flag&floating_boundary)!=0)
return 1;
    return 0;
}
void set_contacted() {
    flag&=(-inside)&(~floating_boundary);
    flag|=contacted_boundary;
}
int is_contacted() {

```



```

        if ((flag&contacted_boundary)!=0)
return 1;
        return 0; }
    int is_inside() {
        if ((flag&inside)!=0)
return 1;
        return 0; }
    int inside_corner() {
        if (((flag&inside)&&(n)&&(s)&&(e)&&(w))
return 1;
        return 0; }
    scalar qx() {return x;}
    scalar qy() {return y;}
    scalar qv(unsigned int i) {return Y[i];}
    node* qn() {return n;}
    node* qs() {return s;}
    node* qe() {return e;}
    node* qw() {return w;}
    void inc_voltage(scalar inc) {Y[volt]+=inc;}
    void slor_test_setup(unsigned int);
};
#endif

```

10.7 Region.h

```

// Copyright 1995, Brian Gieschen
#ifndef WIN31
#define WIN31
#endif
#include <iostream.h>
#include "owl.h"
#include "type.h"
#include "node.h"
#ifndef REGION_INCLUDED
#define REGION_INCLUDED
#define hole_graph 0x01
#define elec_graph 0x02
#define volt_graph 0x04
#define curx_graph 0x08
#define cury_graph 0x10
#define ex_graph 0x20
#define ey_graph 0x40
typedef struct ntag { scalar v; ntag *nxt;} node_list;
class region;
extern void dump(region*,char*);
extern ostream& operator <<(ostream&,region&);
extern istream& operator >>(istream&,region&);
// the region object holds mainly gridding information
// to be used to coordinate calculations such as slor
// line calculations.
// node_list *xl,*yl,*xb,*yb: these lists are used to store grid coordinates
// until the grid is created in the init() function
// xl and yl point to the top of the list (low coord)
// xb and yb point to the bottom of the list
// (high coord). These allow the dynamic addition
// of grid points when processing potential constraints
// node **m: this is an array of node pointers. this is initialized in the init()
// function, not when the region is created. The formula for accessing
// a node in grid is (x_index + y_index * x_index_width).
// unsigned int M,N: these are the widths of the x and y indexes. M for the x
// indices and N for the y indices.
// int slor_horizontal, from_low_coord: these tell the region how to execute
// the slor calculations. slor_horizontal is true(1) if the
// slor line is horizontal, otherwise it is false(0).
// from_low_coord is true(1) if the slor lines are calculated
// starting at the low coordinate of the axis normal to the
// direction of the slor line calculation. An example may
// clarify this. In this coordinate system. Eastern nodes
// and Southern nodes are low coordinates and Northern and
// Western nodes are high coordinates. The x axis low is east
// and high is west. The y axis low is south and high is north
// If the slor_horizontal param is true(1), then the slor line
// will be calculated from east to west. If the from_low_coord
// is true(1), then the slor line calculations will progress
// from the south to the north.

```

```
// gaussian_doping *dop: a pointer to the gaussian doping object for this region
class region {
    friend void dump(region*,char*);
    friend ostream& operator <<(ostream&,region&);
    friend istream& operator >>(istream&,region&);
protected:
    node_list *xl,*yl,*xb,*yb;
    node **m;
    unsigned int M,N;
    int slor_horizontal,from_low_coord;
    gaussian_doping *dop;
public:
    region() {xl=yl=xb=yb=NULL;m=NULL;M=N=0;
        slor_horizontal=from_low_coord=0;dop=NULL;}
    region(unsigned int,unsigned int,
        scalar*,scalar*,gaussian_doping*,
        unsigned int,unsigned int,unsigned int,region**);
    ~region();
    void init(unsigned int,region**);
    void link(unsigned int,region**);
    void resize(unsigned int);
    void contact(unsigned int);
    scalar contact_qv(unsigned int);
    scalar contact_qI(unsigned int);
    scalar contact_dist(unsigned int,scalar,scalar);
    void ramp_voltage(scalar,region*,unsigned int,region*,unsigned int,
        region*,unsigned int,region*,unsigned int);
    void ramp_contact_voltage(scalar,unsigned int);
    void calc_ld_newton_matrices();
    void calc_newton_matrices(int);
    void calc_line_gauss();
    void calc_slor();
    void update_newton();
    void graph(HDC,HPEI*,unsigned int,unsigned int,unsigned int,
        unsigned int,scalar,scalar);
    void slor_test_setup();
};
#endif
```

10.8 Thesis1.h

```
// Copyright 1995, Brian Gieschen
#include <iostream.h>
#include "thesisrc.h"
#include "region.h"
#ifdef THESIS1_INCLUDED
#define THESIS1_INCLUDED
class main;
extern ostream& operator <<(ostream&,main&);
extern istream& operator >>(istream&,main&);
// the main class inherits it's window functions (methods) from
// a borland c++ TWindow class similar classes exist in other
// windowing systems and compilers.
// Simulation controlling information is stored in this class.
// Design structure, bias, magnetic field and current graph type
// are stored here.
// region *re,*ri,*rs,*rc,*rb: are the regions for this simulation
// double ve,vi,vc,vb,bz: are the contact biases and magnetic field
// at the current point in the simulation
// unsigned int gtype: bits are set in this variable to denote
// what types of surfaces to graph.
class main: public TWindow {
    friend ostream& operator <<(ostream&,main&);
    friend istream& operator >>(istream&,main&);
public:
    region *re,*ri,*rs,*rc,*rb;
    double ve,vi,vc,vb,bz;
    unsigned int gtype;
    main(PTWindowsObject p,LPSTR name):TWindow(p,name) {
        re=ri=rs=rc=rb=NULL;ve=vi=vc=vb=bz=0.0;}
    main(LPSTR name): TWindow(NULL,name) {
        AssignMenu("COMMAND");
        Attr.W=Attr.H=480;
        re=ri=rs=rc=rb=NULL;
        ve=vi=vc=vb=bz=0.0;}
};
```

```

    gtype=hole_graph|elec_graph|volt_graph; }
main operator =(main a) {
    if ((re)&&(re!=a.re)) delete re;
    if ((ri)&&(ri!=a.ri)) delete ri;
    if ((rs)&&(rs!=a.rs)) delete rs;
    if ((rc)&&(rc!=a.rc)) delete rc;
    if ((rb)&&(rb!=a.rb)) delete rb;
    re=a.re;ri=a.ri;rs=a.rs;rc=a.rc;rb=a.rb;
    ve=a.ve;vi=a.vi;vc=a.vc;vb=a.vb;
    return (*this);}
virtual void WMPaint(RTMessage) = [WM_FIRST + WM_PAINT];
virtual void WMClose(RTMessage) = [WM_FIRST + WM_CLOSE] {
    if (re) delete re;
    if (ri) delete ri;
    if (rs) delete rs;
    if (rc) delete rc;
    if (rb) delete rb;
    DestroyWindow(HWindow); }
virtual void CMInit(RTMessage) = [CM_FIRST + CM_INIT];
virtual void CMStart(RTMessage) = [CM_FIRST + CM_START];
virtual void CMOpen(RTMessage) = [CM_FIRST + CM_OPEN];
virtual void CMPrint(RTMessage) = [CM_FIRST + CM_PRINT];
void set_graph(unsigned int g,unsigned int d) {
    if (gtype&g) { gtype&=~g;
CheckMenuItem(GetSubMenu(GetMenu(HWindow),2),d,MF_UNCHECKED); }
    else { gtype|=g;
CheckMenuItem(GetSubMenu(GetMenu(HWindow),2),d,MF_CHECKED); }
}
virtual void CMElec(RTMessage) = [CM_FIRST + CM_ELEC] {set_graph(elec_graph,CM_ELEC);}
virtual void CMHole(RTMessage) = [CM_FIRST + CM_HOLE] {set_graph(hole_graph,CM_HOLE);}
virtual void CMVolt(RTMessage) = [CM_FIRST + CM_VOLT] {set_graph(volt_graph,CM_VOLT);}
virtual void CMCurx(RTMessage) = [CM_FIRST + CM_CURX] {set_graph(curx_graph,CM_CURX);}
virtual void CMCurY(RTMessage) = [CM_FIRST + CM_CURY] {set_graph(cury_graph,CM_CURY);}
virtual void CMEx(RTMessage) = [CM_FIRST + CM_EX] {set_graph(ex_graph,CM_EX);}
virtual void CMEy(RTMessage) = [CM_FIRST + CM_EY] {set_graph(ey_graph,CM_EY);}
};

class app: public TApplication {
public:
    app(LPSTR aname,HANDLE hinstance,HANDLE hprevinstance,
        LPSTR lpcmdline,int ncmdshow): TApplication(aname,
            hinstance,hprevinstance,lpcmdline,ncmdshow) {}
    virtual void InitMainWindow() {MainWindow=new main(Name);}
};
#endif

```

10.9 Thesisrc.h

```

// Copyright 1995, Brian Gieschen
#ifndef THESISRC_INCLUDED
#define THESISRC_INCLUDED
#define CM_START 1
#define CM_OPEN 2
#define CM_HOLE 3
#define CM_ELEC 4
#define CM_VOLT 5
#define CM_CURX 6
#define CM_CURY 7
#define CM_EX 8
#define CM_EY 9
#define CM_PRINT 10
#define CM_INIT 11
#endif

```

10.10 Thesis1.cpp

```

// Copyright 1995, Brian Gieschen
#define WIN32
#include <owl.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <new.h>
#include <filedialog.h>
#include <dir.h>

```

```

#include <commdlg.h>
#include "type.h"
#include "gauss.h"
#include "region.h"
#include "thesis1.h"

typedef void (*fptr)(int);
main* mainwindow;

extern scalar Bz;

// displays an ERROR message box when called with a message described
// by the parameter s
void error(char* s)
{
    MessageBox(mainwindow->HWindow.s,"ERROR",MB_OK|MB_ICONEXCLAMATION);
}

// this callback is called automatically by the system in the event
// the system is out of memory
void memerr()
{
    error("Out of Memory");
}

// this callback is called automatically by the system in the event
// the system encounters a hardware floating point error
void floaterr(int sig)
{
    if (sig==SIGFPE) error("SIGFPE: hardware floating point error");
}

// this callback is called automatically by the system in the event
// the system encounters a segmentation violation
void segverr(int sig)
{
    if (sig==SIGSEGV) error("SIGSEGV: memory access out of bounds");
}

// this callback is called automatically by the system in the event
// the system encounters a illegal instruction error
void illerr(int sig)
{
    if (sig==SIGILL) error("SIGILL: attempted to execute illegal instruction");
}

// this is the main function which is first called when a user runs the
// simulator. it sets up the error callbacks, creates an instance of the
// application and runs it, then returns the applications status on exit.
int PASCAL WinMain(HANDLE hinstance,HANDLE hpreinstance,
    LPSTR lpcmdline,int ncmdshow)
{
    set_new_handler(memerr);
    signal(SIGFPE,(fptr)floaterr);
    signal(SIGSEGV,(fptr)segverr);
    signal(SIGILL,(fptr)illerr);
    app simapp("DC Steady-State 2D Lorentz",hinstance,
        hpreinstance,lpcmdline,ncmdshow);
    simapp.Run();
    return simapp.Status;
}

// this function is automatically called by windows to signal
// the window to redraw it's contents. In this case three pens
// are created (blue, red, black) to draw with. Each of the
// simulation regions are checked to see if they exist, if they
// do then they are told to repaint themselves. the gtype parameter
// tells them what types of graphs to display (hole concentration,
// electron concentration, potential, x or y current density,
// x or y electric field). Info Text lines are also drawn to show
// bias, magnetic field, current at each contact and potential at
// each contact. The pens are deleted after the drawing to prevent
// the tying up of windows resources.
void main::WMPaint(RTMessage)
{
    PAINTSTRUCT ps;
    HDC dc;
    HPEN drawpen[3];
    RECT rct;
    char str[100];

```

```

dc=BeginPaint(HWindow,&ps);
drawpen[0]=CreatePen(PS_SOLID,0,RGB(0,0,255));
drawpen[1]=CreatePen(PS_SOLID,0,RGB(255,0,0));
drawpen[2]=CreatePen(PS_SOLID,0,RGB(0,0,0));
if (re!=NULL) re->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25.0,15.0);
if (ri!=NULL) ri->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25.0,15.0);
if (rc!=NULL) rc->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25.0,15.0);
if (rb!=NULL) rb->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25.0,15.0);
if (rs!=NULL) rs->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25.0,15.0);
sprintf(str,"(ve=%5.3lf,vi=%5.3lf,vc=%5.3lf,vb=%5.3lf), Bz=%6.4lf",ve,vi,vc,vb,Bz);
rct.left=0;rct.right=Attr.W;rct.top=0;rct.bottom=20;
ExtTextOut(dc,1,1,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);
if ((re!=NULL)&&(rc!=NULL)&&(rb!=NULL)) {

sprintf(str,"(ie=%9.2le,ii=%9.2le,ic=%9.2le,ib=%9.2le),pot=(e=%5.3lf,i=%5.3lf,c=%5.3lf,b=%5.3lf)");
    re->contact_qi(north),ri->contact_qi(north),rc->contact_qi(north),rb->contact_qi(north),
    re->contact_qv(north),ri->contact_qv(north),rc->contact_qv(north),rb->contact_qv(north));
    rct.left=0;rct.right=Attr.W;rct.top=Attr.H-85;rct.bottom=Attr.H-65;
    ExtTextOut(dc,1,Attr.H-86,ETO_OPAQUE,&rct,str,strlen(str),NULL);
}
DeleteObject(drawpen[0]);
DeleteObject(drawpen[1]);
DeleteObject(drawpen[2]);
EndPaint(HWindow,&ps);
}

// this function is called by the Simulate/Init menu item.
// this function initializes a device structure complete
// with several rectangular regions. This is intended to be
// tailored for each device application. For each region
// a gaussian doping is created. This is intended to be used
// by the region to specify exact dopings for each simulation
// node. Two arrays are created specifying an initial set of
// x and y coordinates for creating nodes in the region. This
// is intended to be used to specify the maximum spacing between
// nodes. The region is then created. The region refines the grid
// which was given to it using potential constraints. Also at this
// point directions are given to the region to tell it how to do the
// line solution for SLOR. The last two parameters in the region
// creation function are for specifying adjacent regions so that
// this is taken into account at creation. Both adjacent regions
// don't have to be called this way, only one, the creation algorithm
// automatically updates the other.
// After all the regions are created they are then initialized.
// Adjacent regions must also be specified at this point because
// the initialization algorithm stitches together all adjacent nodes.
// As with the region creation, only one of the adjacent regions
// must be called this way, the algorithm automatically updates
// the other. Contacts are then flagged on the respective region sides.
// The bias and magnetic field are set to zero.
#define ycontact 51
#define xcontact 16
#define xsub 101
#define ysub 11
void main::CMInit(RTMessage)
{
    gaussian_doping *d;
    scalar x[xcontact],y[ycontact],xs[xsub],ys[ysub];
    unsigned int i,j;
    region *r[4];
    ofstream f;
    if (rs!=NULL) delete rs;
    if (ri!=NULL) delete ri;
    if (re!=NULL) delete re;
    if (rc!=NULL) delete rc;
    if (rb!=NULL) delete rb;
    // emitter
    d=new gaussian_doping(north,1e20,1e17,1.5e-4,0.0,3e-4,2e-4,4e-4);
    for (i=0;i<xcontact;i++) x[i]=(2e-4*(scalar)i/((scalar)xcontact-1.0));
    for (i=0;i<ycontact;i++) y[i]=3e-4+(4e-4*(scalar)i/((scalar)ycontact-1.0));
    re=new region(xcontact,ycontact,x,y,d,north,east,0,NULL);
    // bias stripe
    d=new gaussian_doping(north,-1e20,1e17,1.5e-4,2.5e-4,3e-4,1e-4,4e-4);
    for (i=0;i<xcontact;i++) x[i]=2.5e-4+(1e-4*(scalar)i/((scalar)xcontact-1.0));
    for (i=0;i<ycontact;i++) y[i]=3e-4+(4e-4*(scalar)i/((scalar)ycontact-1.0));
}

```

```

ri=new region(xcontact,ycontact,x,y,d,north,east,0,NULL);
// base
d=new gaussian_doping(north,-1e20,1e17,1.5e-4,1.8e-4,3e-4,2e-4,4e-4);
for (i=0;i<xcontact;i++) x[i]=1.8e-4*(2e-4*(scalar)i/(scalar)xcontact-1.0);
for (i=0;i<ycontact;i++) y[i]=3e-4*(4e-4*(scalar)i/(scalar)ycontact-1.0);
rb=new region(xcontact,ycontact,x,y,d,north,east,0,NULL);
// collector
d=new gaussian_doping(north,1e20,1e17,1.5e-4,9e-4,3e-4,2e-4,4e-4);
for (i=0;i<xcontact;i++) x[i]=9e-4*(2e-4*(scalar)i/(scalar)xcontact-1.0);
for (i=0;i<ycontact;i++) y[i]=3e-4*(4e-4*(scalar)i/(scalar)ycontact-1.0);
rc=new region(xcontact,ycontact,x,y,d,north,east,0,NULL);
// build array of contacting regions
r[0]=re;r[1]=ri;r[2]=rc;r[3]=rb;
// substrate
d=new gaussian_doping(south,0.0,1e17,1e-4,0.0,0.0,20e-4,3e-4);
for (i=0;i<xsub;i++) xs[i]=20e-4*(scalar)i/(scalar)xsub-1.0);
for (i=0;i<ysub;i++) ys[i]=3e-4*(scalar)i/(scalar)ysub-1.0);
rs=new region(xsub,ysub,xs,ys,d,north,east,4,r);
// init & connect regions
re->init(0,NULL);
ri->init(0,NULL);
rc->init(0,NULL);
rb->init(0,NULL);
rs->init(4,r);
// tag contacts
re->contact(north);
ri->contact(north);
rc->contact(north);
rb->contact(north);
ve=vi=vc=vb=0.0;
Bz=bz=0.0;
// redraw window
InvalidateRect (HWindow,NULL,TRUE);
}
// this function is called by the Simulate Start menu item.
// this function performs all the simulation. It is meant to
// be changed for different sets of bias and magnetic field.
// pens are created for graphing the results in the window.
// A loop is entered in which each iteration increments either
// a bias or magnetic field, then a set of newton iterations is
// entered, finally the result is saved in a file.
// Bias is ramped using the region's ramp_voltage function,
// the bias increment is the first parameter, the second and
// third parameters are the contact region, and side of contact
// region which is contacted on which the bias is being incremented
// the following parameters are the regions and contact sides
// of all the other contacted regions in the device. This function
// may need to have additional or less parameters depending on how
// many contacts there are on the device. The potential on the
// contact nodes themselves are then incremented using the region's
// ramp_contact_voltage function, and the respective bias is also
// incremented.
// For magnetic field ramping, simply increment the Bz and bz
// variables (Bz is global to this program).
// The initial values for the iteration are graphed then the
// newton iteration cycle is entered. The region's calc_newton_matrices
// function coordinates the calculation of matrices based on the
// newton approximation of the transport and poisson equations
// For each newton iteration a series of SLOR iterations is performed
// to solve the massive matrix equation. The region's calc_slor
// function also coordinates the solution of the matrix equation
// using the SLOR method. This function though, does not limit
// itself to just one region only. It will follow a SLOR line calculation
// across region boundaries. Thus, depending on the direction of the
// SLOR line calculation, not every region will call it's respective
// calc_slor function. The graphs are updated after each SLOR iteration
// to show the progress (the screen is not cleared until the complete
// set of SLOR iterations is completed, so that changes in the shape
// of the surfaces are readily apparent).
void main::CMStart(RTMessage)
{
    unsigned int i,j,k,newti[16],slori[16];
    char str[100];
    ofstream f;
    HDC dc;

```

```

HBRUSH hbrush;
HMENU drawpen[3];
RECT rct;
ifstream fin;
// setup for drawing
dc=GetDC(HWND);
hbrush=CreateSolidBrush(GetSysColor(COLOR_WINDOW));
drawpen[0]=CreatePen(PS_SOLID,0,RGB(0,0,255));
drawpen[1]=CreatePen(PS_SOLID,0,RGB(255,0,0));
drawpen[2]=CreatePen(PS_SOLID,0,RGB(0,0,0));
// set magnetic field to zero
Bz=0.0;
// setup for simulation
for(i=0;i<16;i++) {
    if (i==0) newti[i]=30;
    else newti[i]=15;
    slori[i]=30;
}
// simulation
for(k=1;k<5;k++) {
    if (k>0) {
        // perform selective file open / initialization
        if (k<5) {
            switch (k) {
                case 1: fin.open("i3z0.sim"); break;
                case 2: fin.open("i2z0.sim"); break;
                case 3: fin.open("i3z0.sim"); break;
                case 4: fin.open("i1.sim"); break;
            }
            //sprintf(str,"i%d.sim",k-3);
            //f.open(str);
            if (fin) {
                fin>>(*this);
                Bz=bz;
            }
            fin.close();
        }
        // perform voltage or magnetic field ramping
        if (k<2) {
            re->ramp_voltage(0.1,ri,north,re,north,rb,north,rc,north);
            ri->ramp_voltage(0.1,ri,north,re,north,rb,north,rc,north);
            rc->ramp_voltage(0.1,ri,north,re,north,rb,north,rc,north);
            rb->ramp_voltage(0.1,ri,north,re,north,rb,north,rc,north);
            rs->ramp_voltage(0.1,r,north,re,north,rb,north,rc,north);
            ri->ramp_contact_voltage(0.1,north);
            vi+=0.1;
            //
        }
        else if (k<14) {
            //re->ramp_voltage(-0.1,rc,north,re,north,ri,north,rb,north);
            //ri->ramp_voltage(-0.1,rc,north,re,north,ri,north,rb,north);
            //rc->ramp_voltage(-0.1,rc,north,re,north,ri,north,rb,north);
            //rb->ramp_voltage(-0.1,rc,north,re,north,ri,north,rb,north);
            //rs->ramp_voltage(-0.1,rc,north,re,north,ri,north,rb,north);
            //rc->ramp_contact_voltage(-0.1,north);
            //vc-=0.1;
        }
        else {bz+=0.0001;Bz=bz;}
    }
    // graph initial values
    rct.left=0;rct.right=Attr.W;rct.top=0;rct.bottom=Attr.H;
    FillRect(dc,&rct,hbrush);
    re->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
    ri->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
    rc->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
    rb->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
    rs->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
    sprintf(str,"(ve=%5.3lf,vi=%5.3lf,vc=%5.3lf,vb=%5.3lf), Bz=%6.4lf",ve,vi,vc,vb,Bz);
    rct.left=0;rct.right=Attr.W;rct.top=0;rct.bottom=20;
    ExtTextOut(dc,1,1,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);

    sprintf(str,"(ie=%9.2le,ii=%9.2le,ic=%9.2le,ib=%9.2le),pot(e=%5.3lf,i=%5.3lf,c=%5.3lf,b=%5.3lf)",
    re->contact_qi(north),ri->contact_qi(north),rc->contact_qi(north),rb->contact_qi(north),
    re->contact_qv(north),ri->contact_qv(north),rc->contact_qv(north),rb->contact_qv(north));
    rct.left=0;rct.right=Attr.W;rct.top=Attr.H-65;rct.bottom=Attr.H-45;
    ExtTextOut(dc,1,Attr.H-64,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);

```

```

// start newton iteration
for(j=0;j<newti[k];j++) {
    sprintf(str,"(ve=%5.3lf,vi=%5.3lf,vx=%5.3lf,vb=%5.3lf), Bz=%6.4lf,
newt=%d",ve,vi,vx,vb,Bz,j+1);
    rct.left=0;rct.right=Attr.W;rct.top=0;rct.bottom=20;
    ExtTextOut(dc,1,1,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);
    re->calc_newton_matrices(0);
    ri->calc_newton_matrices(0);
    rc->calc_newton_matrices(0);
    rb->calc_newton_matrices(0);
    rs->calc_newton_matrices(0);
    for(i=0;i<slori[k];i++) {
        sprintf(str,"(ve=%5.3lf,vi=%5.3lf,vx=%5.3lf,vb=%5.3lf), Bz=%6.4lf, newt=%d,
slor=%d",ve,vi,vx,vb,Bz,j+1,i+1);
        rct.left=0;rct.right=Attr.W;rct.top=0;rct.bottom=20;
        ExtTextOut(dc,1,1,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);
        rs->calc_slor();
        re->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
        ri->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
        rc->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
        rb->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);
        rs->graph(dc,drawpen,gtype,(Attr.W<Attr.H?Attr.W:Attr.H),0,0,25,0,15,0);

        sprintf(str,"(e=%9.2le,ii=%9.2le,ic=%9.2le,ib=%9.2le),pct(e=%5.3lf,ii=%5.3lf,ic=%5.3lf,ib=%5.3lf)");
        re->contact_qi(north);ri->contact_qi(north);rc->contact_qi(north);rb->
        >contact_qi(north);
        re->contact_qv(north);ri->contact_qv(north);rc->contact_qv(north);rb->
        >contact_qv(north);
        rct.left=0;rct.right=Attr.W;rct.top=Attr.H-65;rct.bottom=Attr.H-45;
        ExtTextOut(dc,1,Attr.H-64,ETO_CLIPPED|ETO_OPAQUE,&rct,str,strlen(str),NULL);
    }
    sprintf(str,"i%d.sim",k);
    f=fopen(str);
    f.setf(ios::scientific);
    f<<("this,");
    f.close();
}
DeleteObject(hbrush);
DeleteObject(drawpen[0]);
DeleteObject(drawpen[1]);
DeleteObject(drawpen[2]);
ReleaseDC(HWindow,dc);
InvalidateRect(HWindow,NULL,TRUE);
}

int wildcards(char* s)
{
    unsigned int i;
    int wildcardfound=0;
    for(i=0;s[i]!='\0';i++)
        if ((s[i]=='*')||(s[i]=='?'))
            wildcardfound=1;
    return wildcardfound;
}

// this function is called by the File Open menu item
// This function calls a special dialog box to get a file name from
// the user to open. It strips off the pathname and changes directory
// if needed. Then it opens the file and reads in (and replaces) all
// the simulation info including regions, nodes, doping, bias, magnetic field.
// The reading in part is done through a redirection operator ">>" which
// hides the complexity very nicely. The redirection operators are
// defined in the "fileio.cpp" source file.
void main::CWOpen(RTMessage)
{
    char str[MAXPATH],cdir[MAXPATH];
    int i,j;
    ifstream f;
    sprintf(str,"*.sim");
    if (GetApplication()->ExecDialog(new TFileDialog(this,SD_FILEOPEN,str))==IDOK) {
        if (wildcards(str)) {
            for(i=j=0;str[i]!='\0';i++) if (str[i]!='\\') j++;
            for(i=0;i<j;i++) cdir[i]=str[i];
            cdir[i]='\0';

```



```

        for(i=0,j++;str[j]!='\0';i++,j++) str[i]=str[j];
        str[i]='\0';
        chdir(cdir);
        f.open(str);
        if (f) {
f>>(*this);
        Bz=bz;
        }
        f.close();
        InvalidateRect(HWindow,NULL,TRUE);
    }
}

// this function is called by the File/Print menu item
// this function mucks about with the printer device to
// specify correctly the right fonts and to query the
// dpi of the printer for graphing the device surfaces.
// It then prints the page with the same function that
// this program uses to graph the device on the screen.
// It also prints Text information for bias, current,
// magnetic field and potential at the contacts.
void main::CMPrint(RTMessage)
{
    PRINTDLG pd;
    LOGFONT lf;
    HFONT hfnt,hfntold;
    DOCINFO di;
    double tmp,tmpx;
    char s[132];
    RECT rct;
    HPEN drawpen[3];
    drawpen[0]=CreatePen(PS_SOLID,0,RGB(0,0,255));
    drawpen[1]=CreatePen(PS_SOLID,0,RGB(255,0,0));
    drawpen[2]=CreatePen(PS_SOLID,0,RGB(0,0,0));
    memset(&pd,0,sizeof(PRINTDLG));
    memset(&lf,0,sizeof(LOGFONT));
    pd.lStructSize=sizeof(PRINTDLG);
    pd.hwndOwner=HWindow;
    pd.Flags=PD_RETURNDC;
    if (PrintDlg(&pd)!=0) {
        tmp=((double)GetDeviceCaps(pd.hDC,LOGPIXELSY));
        tmpx=((double)GetDeviceCaps(pd.hDC,LOGPIXELSX));
        lf.lfHeight=(int)(10.0*tmp/72.0);
        GetTextFace(pd.hDC,132,s);
        strcpy(lf.lfFaceName,s);
        hfnt=CreateFontIndirect(&lf);
        hfntold=SelectObject(pd.hDC,hfnt);
        di.cbSize=sizeof(DOCINFO);
        sprintf(s,"2D Steady-State Lorentz");
        di.lpszDocName=s;
        di.lpszOutput=NULL;
        StartDoc(pd.hDC,&di);
        StartPage(pd.hDC);
        sprintf(s,"(ve=%5.3lf,vi=%5.3lf,vc=%5.3lf,vb=%5.3lf), Bz=%6.4lf",ve,vi,vc,vb,Bz);
        rct.left=1.0*tmpx;rct.right=7.5*tmpx;rct.top=1.0*tmp;rct.bottom=1.25*tmp;
        SelectObject(pd.hDC,drawpen[2]);
        ExtTextOut(pd.hDC,1.0*tmpx,1.0*tmp,ETO_CLIPPED|ETO_OPAQUE,&rct,s,strlen(s),NULL);
        if ((re)&&(rc)&&(rb)) {
            sprintf(s,"(ie=%9.2le,ii=%9.2le,ic=%9.2le,ib=%9.2le).pot=(e=%5.3lf,i=%5.3lf,c=%5.3lf,b=%5.3lf)"
            ,
                re->contact_qi(north),ri->contact_qi(north),rc->contact_qi(north),rb-
            >contact_qi(north),
                re->contact_qv(north),ri->contact_qv(north),rc->contact_qv(north),rb-
            >contact_qv(north));
            rct.left=1.0*tmpx;rct.right=7.5*tmpx;rct.top=8.5*tmp;rct.bottom=8.25*tmp;
            ExtTextOut(pd.hDC,1.0*tmpx+1.0,8.5*tmp+1.0,ETO_OPAQUE,&rct,s,strlen(s),NULL);
        }
        if (re!=NULL) re->graph(pd.hDC,drawpen,gtype,6.5*tmp,1.0*tmpx,1.5*tmp,25.0,15.0);
        if (ri!=NULL) ri->graph(pd.hDC,drawpen,gtype,6.5*tmp,1.0*tmpx,1.5*tmp,25.0,15.0);
        if (rc!=NULL) rc->graph(pd.hDC,drawpen,gtype,6.5*tmp,1.0*tmpx,1.5*tmp,25.0,15.0);
        if (rb!=NULL) rb->graph(pd.hDC,drawpen,gtype,6.5*tmp,1.0*tmpx,1.5*tmp,25.0,15.0);
        if (rs!=NULL) rs->graph(pd.hDC,drawpen,gtype,6.5*tmp,1.0*tmpx,1.5*tmp,25.0,15.0);
        EndPage(pd.hDC);
        EndDoc(pd.hDC);
    }
}

```

```

    SelectObject(pd.hDC,hfntold);
    DeleteObject(hfnt);
}
DeleteObject(drawpen[0]);
DeleteObject(drawpen[1]);
DeleteObject(drawpen[2]);
}

```

10.11 Region.cpp

```

// Copyright 1995, Brian Gieschen
#include <alloc.h>
#include "gauss.h"
#include "region.h"

// this defines the maximum potential difference tolerated
// between each node at grid creation time.
#define vconstraint 0.05
// this is the region creation algorithm
// unsigned int _M : this is the number of x points passed to this function
// unsigned int _N : this is the number of y points passed to this function
// scalar *x : this is an array of x points to initially start with
// scalar *y : this is an array of y points to initially start with
// gaussian_doping *d: this is an object which specifies the doping in this region
// unsigned int slor_dir: this is the direction of the slor calculation
// unsigned int line_dir: this is the direction along the slor line calc
// unsigned int rnum: this is the number of adjacent regions
// unsigned int **r: this is an array of pointers to adjacent regions.
// Adjacent regions only have to be specified once this way
// i.e. Both regions do not have to show each other in their
// r parameters, only one has to show the other.
// First this function saves the slor direction info. Then it creates linked
// lists of the x and y coordinates to simplify adding more grid points
// due to voltage constraints. It then applies voltage constraints and adds
// grid points. Then it goes through the list of adjacent regions and creates
// additional grid points so that the grids of both regions line up.
region::region(unsigned int _M,unsigned int _N,scalar *x,scalar *y,
    gaussian_doping *d,unsigned int slor_dir,unsigned int line_dir,
    unsigned int rnum,region **r)
{
    node_list *xt,*yt,*t,*rxt,*ryt;
    unsigned int i,j,k;
    scalar l,mi,h,lv,mv,hv,target;
    xl=yl=NULL;
    M=_M;
    N=_N;
    dop=d;
    // remember direction along which slor is calculated
    if ((slor_dir==east)|| (slor_dir==west)) {
        if ((line_dir==north) from_low_coord=0;
        else from_low_coord=1;
        slor_horizontal=1;
    } else {
        if ((line_dir==west) from_low_coord=0;
        else from_low_coord=1;
        slor_horizontal=0;
    }
    // build lists of x and y coordinates
    xl=xb=NULL;
    for(i=0;i<M;i++) {
        if (xl==NULL) (xl=(node_list*)malloc(sizeof(node_list)));xt=xl;
        else (xt->nxt=(node_list*)malloc(sizeof(node_list)));xt=xt->nxt;
        xt->v=x[i];xt->nxt=NULL;
    }
    if (M>1) xb=xt; else xb=xl;
    yl=yb=NULL;
    for(j=0;j<N;j++) {
        if (yl==NULL) (yl=(node_list*)malloc(sizeof(node_list)));yt=yl;
        else (yt->nxt=(node_list*)malloc(sizeof(node_list)));yt=yt->nxt;
        yt->v=y[j];yt->nxt=NULL;
    }
    if (N>1) yb=yt; else yb=yl;
    // insert nodes by voltage constraint
    for(xt=xl;xt->nxt!=NULL;xt=xt->nxt)
        for(yt=yl;yt->nxt!=NULL;yt=yt->nxt) {

```

```

    while (fabs(d->voltage(xt->v, yt->v) - d->voltage(xt->nxt->v, yt->v)) > vconstraint) {
        t = xt->nxt;
        xt->nxt = (node_list*) malloc(sizeof(node_list));
        l = xt->v; h = t->v;
        lv = d->voltage(l, yt->v); hv = d->voltage(h, yt->v);
        if (lv > hv) { mi = l; l = h; h = mi; mv = lv; lv = hv; hv = mv; }
        target = (lv + hv) / 2.0;
        mi = (l + h) / 2.0; mv = d->voltage(mi, yt->v);
        for (; (fabs(mv - target) > (vconstraint / 100.0)); )
            if (mv < target) { l = mi; lv = mv; mi = (l + h) / 2.0; mv = d->voltage(mi, yt->v); }
            else { h = mi; hv = mv; mi = (l + h) / 2.0; mv = d->voltage(mi, yt->v); }
        xt->nxt->v = mi; xt->nxt->nxt = t;
        M++;
    }
    while (fabs(d->voltage(xt->v, yt->v) - d->voltage(xt->v, yt->nxt->v)) > vconstraint) {
        t = yt->nxt;
        yt->nxt = (node_list*) malloc(sizeof(node_list));
        l = yt->v; h = t->v;
        lv = d->voltage(xt->v, l); hv = d->voltage(xt->v, h);
        if (lv > hv) { mi = l; l = h; h = mi; mv = lv; lv = hv; hv = mv; }
        target = (lv + hv) / 2.0;
        mi = (l + h) / 2.0; mv = d->voltage(xt->v, mi);
        for (; (fabs(mv - target) > (vconstraint / 100.0)); )
            if (mv < target) { l = mi; lv = mv; mi = (l + h) / 2.0; mv = d->voltage(xt->v, mi); }
            else { h = mi; hv = mv; mi = (l + h) / 2.0; mv = d->voltage(xt->v, mi); }
        yt->nxt->v = mi; yt->nxt->nxt = t;
        N++;
    }
}
// insert nodes for multi-region line-up
for(k=0; k<rnum; k++) {
    // butt east or west sides
    if ((xl->v == r[k]->xb->v) || (xb->v == r[k]->xl->v)) {
        yt = yl; ryt = r[k]->yl;
        while ((yt != NULL) && (ryt != NULL)) {
            if ((yt->nxt != NULL) && (yt->v < ryt->v) && (ryt->v < yt->nxt->v)) {
                // add node on this region's side
                t = (node_list*) malloc(sizeof(node_list));
                t->v = ryt->v;
                t->nxt = yt->nxt;
                yt->nxt = t;
                N++;
            } else if ((ryt->nxt != NULL) && (ryt->v < yt->v) && (yt->v < ryt->nxt->v)) {
                // add node on the other region's side
                t = (node_list*) malloc(sizeof(node_list));
                t->v = yt->v;
                t->nxt = ryt->nxt;
                ryt->nxt = t;
                r[k]->N++;
            }
        }
        if (yt->v < ryt->v) yt = yt->nxt;
        else if (yt->v > ryt->v) ryt = ryt->nxt;
        else if ((yt->nxt != NULL) && (ryt->nxt != NULL)) {
            if (yt->nxt->v < ryt->nxt->v) yt = yt->nxt;
            else if (yt->nxt->v > ryt->nxt->v) ryt = ryt->nxt;
            else { yt = yt->nxt; ryt = ryt->nxt; }
        } else { yt = yt->nxt; ryt = ryt->nxt; }
    }
    // butt north or south sides
    else if ((yl->v == r[k]->yb->v) || (yb->v == r[k]->yl->v)) {
        xt = xl; rxt = r[k]->xl;
        while ((xt != NULL) && (rxt != NULL)) {
            if ((xt->nxt != NULL) && (xt->v < rxt->v) && (rxt->v < xt->nxt->v)) {
                // add node on this region's side
                t = (node_list*) malloc(sizeof(node_list));
                t->v = rxt->v;
                t->nxt = xt->nxt;
                xt->nxt = t;
                M++;
            } else if ((rxt->nxt != NULL) && (rxt->v < xt->v) && (xt->v < rxt->nxt->v)) {
                // add node on the other region's side
                t = (node_list*) malloc(sizeof(node_list));
                t->v = xt->v;
                t->nxt = rxt->nxt;
            }
        }
    }
}

```

```

    rxt->nxt=t;
    r[k]->M++;
}
if (xt->v<rxt->v) xt=xt->nxt;
else if (xt->v>rxt->v) rxt=rxt->nxt;
else if ((xt->nxt!=NULL)&&(rxt->nxt!=NULL)) {
    if (xt->nxt->v<rxt->nxt->v) xt=xt->nxt;
    else if (xt->nxt->v>rxt->nxt->v) rxt=rxt->nxt;
    else (xt=xt->nxt;rxt=rxt->nxt;)
} else (xt=xt->nxt;rxt=rxt->nxt;)
}
}
}
// this function takes the grid point lists and creates a matrix
// of nodes for simulation
// unsigned int rnum: this is the number of adjacent regions
// unsigned int **r: this is an array of pointers to adjacent regions
// Adjacent regions only have to be specified once this way
// i.e. Both regions do not have to show each other in their
// r parameters, only one has to show the other
// A matrix is first created for the nodes. Then the nodes are created to populate
// this matrix. The lists of grid points can now be deleted. The nodes dynamic
// pointers are then linked (adjacent nodes point to each other). Finally, nodes
// between adjoining regions are linked.
void region::init(unsigned int rnum, region **r)
{
    node_list *xt,*yt;
    unsigned int i,j;
    // setup matrix
    m=(node **)malloc(sizeof(node*)*M*N);
    // Create nodes to populate matrix
    for(xt=xl,i=0;xt!=NULL;i++,xt=xt->nxt)
        for(yt=yl,j=0;yt!=NULL;j++,yt=yt->nxt)
            m[i+j*M]=new node(xt->v,yt->v,dcp->N(xt->v,yt->v));
    for(xt=xl->nxt;xl!=NULL;xl=xt,xt!=NULL?xt->nxt:NULL) delete xl;
    for(yt=yl->nxt;yl!=NULL;yl=yt,yt!=NULL?yt->nxt:NULL) delete yl;
    xl=yl=xb=yb=NULL;
    // link nodes which populate matrix & set floating boundary flag
    for (i=0;i<M;i++) {
        for (j=0;j<N;j++) {
            m[i+j*M]->link(j==N-1?m[i+(j+1)*M]:NULL, // north
                j==0?m[i+(j-1)*M]:NULL, // south
                i==0?m[i-1+j*M]:NULL, // east
                i==M-1?m[i+1+j*M]:NULL); // west
            // this isn't really necessary node::link() does this
            if ((i==0)|| (j==0)|| (i==M-1)|| (j==N-1))
                m[i+j*M]->set_floating_boundary();
        }
    }
    link(rnum,r);
}
// this function links nodes between adjoining regions
// unsigned int rnum: this is the number of adjacent regions
// unsigned int **r: this is an array of pointers to adjacent regions
// Adjacent regions only have to be specified once this way
// i.e. Both regions do not have to show each other in their
// r parameters, only one has to show the other.
// this function goes along the interface between adjoining regions
// and links adjoining nodes.
void region::link(unsigned int rnum, region **r)
{
    unsigned int k,aM,aN;
    region *ar;
    node *n,*an;
    // link nodes between adjoining regions
    for(k=0;k<rnum;k++) {
        ar=r[k];aM=ar->M;aN=ar->N;
        // other region is butted against east side
        if (m[0]->qx()==ar->m[aM-1]->qx()) {
            for(n=m[0],an=ar->m[aM-1];
                (n!=NULL)&&(an!=NULL);
                (n->qy())<(an->qy())?(n=n->qn()):
                (n->qy())>(an->qy())?(an=an->qn()):
                (n=n->qn(),an=an->qn())) {

```

```

if (n->qy()==an->qy())
n->link(NULL,NULL,an->qe(),NULL);
    }
    ar->resize(west);
}
// other region is butted against west side
else if (m[M-1]->qx()<=ar->m[0]->qx()) {
    for(n=m[M-1],an=ar->m[0];
(n!=NULL)&&(an!=NULL);
(n->qy()<an->qy())?(n=n->qn());
(n->qy()>an->qy())?(an=an->qn());
(n=n->qn(),an=an->qn())) {
if (n->qy()==an->qy())
n->link(NULL,NULL,NULL,an->qw());
    }
    ar->resize(east);
}
// other region is butted against south side
else if (m[0]->qy()>=ar->m[(aN-1)*aM]->qy()) {
    for(n=m[0],an=ar->m[(aN-1)*aM];
(n!=NULL)&&(an!=NULL);
(n->qx()<an->qx())?(n=n->qw());
(n->qx()>an->qx())?(an=an->qw());
(n=n->qw(),an=an->qw())) {
if (n->qx()==an->qx())
n->link(NULL,an->qs(),NULL,NULL);
    }
    ar->resize(north);
}
// other region is butted against north side
else if (m[(N-1)*M]->qy()<=ar->m[0]->qy()) {
    for(n=m[(N-1)*M],an=ar->m[0];
(n!=NULL)&&(an!=NULL);
(n->qx()<an->qx())?(n=n->qw());
(n->qx()>an->qx())?(an=an->qw());
(n=n->qw(),an=an->qw())) {
if (n->qx()==an->qx())
n->link(an->qn(),NULL,NULL,NULL);
    }
    ar->resize(south);
}
}
}
// this function deletes the region. It deletes all of it's
// nodes and then deletes it's matrix. This is automatically
// called when the region is deleted.
void region::~region()
{
    unsigned int i,j;
    if (dop!=NULL) delete dop;
    if (m!=NULL) {
        for(i=0;i<M;i++) {
            for(j=0;j<N;j++) {
if (m[i+j*M]!=NULL) delete m[i+j*M];
            }
        }
        free(m);
    }
}
// when two adjoining regions are connected they will both have nodes
// which exist at the interface. Two nodes should not overlap in this
// simulator. Therefore the nodes of one of the regions will have to be
// removed along the interface. This function performs this task.
void region::resize(unsigned int side)
{
    node **tm;
    int i,j;
    if ((side==east)||(side==west))
        tm=(node**)malloc(sizeof(node)*(M-1)*N);
    else if ((side==north)||(side==south))
        tm=(node**)malloc(sizeof(node)*M*(N-1));
    for(i=0;i<M;i++) {
        for(j=0;j<N;j++) {
            switch(side) {
case north: if (j<N-1) tm[i+j*M]=m[i+j*M]; else free(m[i+j*M]);

```

```

        break;
    case south: if (j>0) tm[i+(j-1)*M]=m[i+j*M]; else free(m[i+j*M]);
        break;
    case east: if (i>0) tm[i-1+j*M]=m[i+j*M]; else free(m[i+j*M]);
        break;
    case west: if (i<M-1) tm[i+j*M]=m[i+j*M]; else free(m[i+j*M]);
        break;
    )
    )
    if ((side==east)||side==west) M--;
    else if ((side==north)||side==south) N--;
    free(m);
    m=tm;
}
// this function tags all the nodes along one side of the region as being
// contacted (metal contact).
void region::contact(unsigned int side)
{
    unsigned int i,j;
    switch (side) {
        case north:
            for (i=0;i<M;i++) m[i+(N-1)*M]->set_contacted();
            break;
        case south:
            for (i=0;i<M;i++) m[i]->set_contacted();
            break;
        case east:
            for (j=0;j<N;j++) m[j*M]->set_contacted();
            break;
        case west:
            for (j=0;j<N;j++) m[M-1+j*M]->set_contacted();
            break;
    }
}
// this function queries all the potentials along one of the sides of the region
// and averages them to return an average potential along the contact. This
// is perhaps being too cautious, all the potentials should be the same, but
// one can never be too careful.
scalar region::contact_qv(unsigned int side)
{
    unsigned int i,j;
    scalar v;
    v=0.0;
    switch (side) {
        case north: for(i=0;i<M;i++) v+=m[i+(N-1)*M]->qY.volt;
        v/=(scalar)M;
        break;
        case south: for(i=0;i<M;i++) v+=m[i]->qY.volt;
        v/=(scalar)M;
        break;
        case east: for(j=0;j<N;j++) v+=m[j*M]->qY.volt;
        v/=(scalar)N;
        break;
        case west: for(j=0;j<N;j++) v+=m[M-1+j*M]->qY.volt;
        v/=(scalar)N;
        break;
    }
    return v;
}
// This function adds up all the current components along one side of
// a region to come up with an overall current at a contact.
scalar region::contact_qI(unsigned int side)
{
    unsigned int i,j;
    scalar I;
    I=0.0;
    switch (side) {
        case north: for(i=0;i<M;i++) if (m[i+(N-1)*M]->qS()->JyOK()) I+=m[i+(N-1)*M]->qI(side,dop);
        break;
        case south: for(i=0;i<M;i++) if (m[i]->qS()->JyOK()) I+=m[i]->qI(side,dop); break;
        case east: for(j=0;j<N;j++) if (m[j*M]->qS()->JxOK()) I+=m[j*M]->qI(side,dop); break;
        case west: for(j=0;j<N;j++) if (m[M-1+j*M]->qS()->JxOK()) I+=m[M-1+j*M]->qI(side,dop);
        break;
    }
}

```

```

    return I;
}
// this function calculates the distance between a pair of coordinates and a contact
// or use when doing voltage ramping.
scalar region::contact_dist(unsigned int dir, scalar x, scalar y)
{
    if (dir==east) {
        if (y>m[(N-1)*M]->qy())
            return sqrt(pow(x-m[(N-1)*M]->qx(),2)+pow(y-m[(N-1)*M]->qy(),2));
        else if (y<m[0]->qy())
            return sqrt(pow(x-m[0]->qx(),2)+pow(y-m[0]->qy(),2));
        else return fabs(x-m[0]->qx());
    } else if (dir==west) {
        if (y>m[(N-1)*M]->qy())
            return sqrt(pow(x-m[M-1+(N-1)*M]->qx(),2)+pow(y-m[M-1+(N-1)*M]->qy(),2));
        else if (y<m[0]->qy())
            return sqrt(pow(x-m[M-1]->qx(),2)+pow(y-m[M-1]->qy(),2));
        else return fabs(x-m[M-1]->qx());
    } else if (dir==north) {
        if (x>m[M-1]->qx())
            return sqrt(pow(x-m[M-1+(N-1)*M]->qx(),2)+pow(y-m[M-1+(N-1)*M]->qy(),2));
        else if (x<m[0]->qx())
            return sqrt(pow(x-m[(N-1)*M]->qx(),2)+pow(y-m[(N-1)*M]->qy(),2));
        else return fabs(y-m[(N-1)*M]->qy());
    } else if (dir==south) {
        if (x>m[M-1]->qx())
            return sqrt(pow(x-m[M-1]->qx(),2)+pow(y-m[M-1]->qy(),2));
        else if (x<m[0]->qx())
            return sqrt(pow(x-m[0]->qx(),2)+pow(y-m[0]->qy(),2));
        else return fabs(y-m[0]->qy());
    }
    return 0.0;
}
// this function ramps the potential of each node in a region.
// scalar ramp: the additional bias which will be applied to the contact on the region
// specified in the second and third parameters.
// region *r: the region to which the additional bias will be applied
// unsigned int rd: the side of the region on which the contact is.
// region *r1,*r2,*r3: other regions which have contacts.
// unsigned int r1d,r2d,r3d: sides of other regions which are contacted.
// Basically, this goes through all the nodes in this region and calls the nodes.
// voltage ramping equation to change the nodes potential in anticipation of an other
// series of newton iterations.
void region::ramp_voltage(scalar ramp, region *r, unsigned int rd, region *r1, unsigned int r1d,
    region *r2, unsigned int r2d, region *r3, unsigned int r3d)
{
    unsigned int i,j;
    scalar v,v1,v2,v3,nx,ny;
    v=r->contact_qv(rd);
    v1=r1->contact_qv(r1d);
    v2=r2->contact_qv(r2d);
    v3=r3->contact_qv(r3d);
    for(i=0;i<M;i++)
        for(j=0;j<N;j++) {
            nx=m[i+j*M]->qx();
            ny=m[i+j*M]->qy();
            m[i+j*M]->ramp_voltage(ramp,v,v1,v2,v3,r->contact_dist(rd,nx,ny),r1->contact_dist(r1d,nx,ny),
                r2->contact_dist(r2d,nx,ny),r3->contact_dist(r3d,nx,ny));
        }
}
// this function simply goes through all the nodes on one side of the region
// and increments the potential.
void region::ramp_contact_voltage(scalar ramp, unsigned int side)
{
    unsigned int i,j;
    switch (side) {
        case north: for(i=0;i<M;i++) m[i+(N-1)*M]->inc_voltage(ramp); break;
        case south: for(i=0;i<M;i++) m[i]->inc_voltage(ramp); break;
        case east: for(j=0;j<N;j++) m[j*M]->inc_voltage(ramp); break;
        case west: for(j=0;j<N;j++) m[M-1+j*M]->inc_voltage(ramp); break;
    }
}
// this function simply goes through all the nodes in the region and calls
// the node's 1D newton matrix calculator (this function isn't actually

```

```

// used right now, it is intended to be used to calculate more accurate
// initial guesses to start the newton iteration). Use this function
// carefully, there are pitfalls.
void region::calc_1D_newton_matrices()
{
    unsigned int i,j;
    i=0;j=0;
    for(i=0;i<M;i++)
        for(j=0;j<N;j++)
            if (m[i+j*M]->is_inside()) {
                if (slor_horizontal)
                    m[i+j*M]->calc_1D_newton_matrices(west,dop);
                else
                    m[i+j*M]->calc_1D_newton_matrices(north,dop);
            }
}
// this function calcs the 2D newton matrices for each node
// in the region.
// initial value is always 0.
void region::calc_newton_matrices(int initial_value)
{
    unsigned int i,j;
    for (i=0;i<M;i++)
        for (j=0;j<N;j++)
            if (m[i+j*M]->is_inside())
                m[i+j*M]->calc_newton_matrices(dop,initial_value);
}
// this is the companion function to calc_1D_newton_matrices.
// it performs an optimized gaussian elimination to solve 1D
// lines in the device for use as initial values.
// This is not used right now. Be very careful with this one.
void region::calc_line_gauss()
{
    unsigned int dir,tmp;
    int i,j;
    if (slor_horizontal) {
        if (from_low_coord) j=0;
        else j=N-1;
        while (((from_low_coord)&&(j<N))) {
            ((!from_low_coord)&&(j>0)) {
                dir=west_travel;
                i=0;
                while (i<M) {
                    while ((i<M)&&(m[i+j*M]->is_inside())) {i++;}
                    while ((i<M)&&(m[i+j*M]->is_inside())) {
                        tmp=dir;
                        if ((i==0)||(!m[i-1+j*M]->is_inside())) tmp|=start_of_travel;
                        if ((i==M-1)||(!m[i+1+j*M]->is_inside())) tmp|=end_of_travel;
                        m[i+j*M]->calc_slor(tmp);
                        i++;
                    }
                }
                dir=east_travel;
                i=M-1;
                while (i>=0) {
                    while ((i>=0)&&(m[i+j*M]->is_inside())) {i--;}
                    while ((i>=0)&&(m[i+j*M]->is_inside())) {
                        tmp=dir;
                        if ((i==M-1)||(!m[i+1+j*M]->is_inside())) tmp|=start_of_travel;
                        m[i+j*M]->back_sub(tmp);
                        i--;
                    }
                }
                if (from_low_coord) j++;
                else j--;
            }
        }
    }
    else {
        if (from_low_coord) i=0;
        else i=M-1;
        while (((from_low_coord)&&(i<M))) {
            ((!from_low_coord)&&(i>0)) {
                dir=north_travel;
                j=0;
                while (j<N) {

```



```

while ((j<N)&&(!m[i+j*M]->is_inside())) {j++;}
while ((j<N)&&(!m[i+j*M]->is_inside())) {
    tmp=dir;
    if((j==0)||(!m[i+(j-1)*M]->is_inside())) tmp=start_of_travel;
    if((j==N-1)||(!m[i+(j+1)*M]->is_inside())) tmp=end_of_travel;
    m[i+j*M]->calc_slor(tmp);
    j++;
}
}
dir=south_travel;
j=N-1;
while (j>=0) {
while ((j>=0)&&(!m[i+j*M]->is_inside())) {j--;}
while ((j>=0)&&(!m[i+j*M]->is_inside())) {
    tmp=dir;
    if((j==N-1)||(!m[i+(j+1)*M]->is_inside())) tmp=start_of_travel;
    m[i+j*M]->back_sub(tmp);
    j--;
}
}
if (from_low_coord) i++;
else i--;
}
}
for(i=0;i<M;i++) {
    if (m[i]->is_floating_boundary())
        (*m[i])=(*m[i+M]);
    if (m[i+(N-1)*M]->is_floating_boundary())
        (*m[i+(N-1)*M])=(*m[i+(N-2)*M]);
}
for(j=0;j<N;j++) {
    if (m[j*M]->is_floating_boundary())
        (*m[j*M])=(*m[1+j*M]);
    if (m[M-1+j*M]->is_floating_boundary())
        (*m[M-1+j*M])=(*m[M-2+j*M]);
}
m[0]->calc_corner();
m[M-1]->calc_corner();
m[(N-1)*M]->calc_corner();
m[(M-1)+(N-1)*M]->calc_corner();
}
// this is the slor solver for the 2D newton matrices.
// it goes through and does forward elimination and backward substitution
// along a line. It calculates new values at the boundaries as well (taking
// into account boundary conditions). It uses a topological algorithm to
// follow the boundaries around the device. If the slor calculations for
// this device cannot be done by calling this function for only one region,
// this function will have to be modified to that the topological algorithm
// is called only once for each SLOR iteration.
void region::calc_slor()
{
    unsigned int dir,tmp;
    int i,j;
    node *cn,*cnnxt;
    if (slor_horizontal) {
        if (from_low_coord) cnnxt=m[0];
        else cnnxt=m[(N-1)*M];
        while (cnnxt!=NULL) {
            dir=west_travel|north_include|south_include;
            cn=cnnxt;
            while (cn!=NULL) {
while ((cn!=NULL)&&(!cn->is_inside())) {cn=cn->qw();}
while ((cn!=NULL)&&(!cn->is_inside())) {
    tmp=dir;
    if((cn->qe()==NULL)||(!cn->qe()->is_inside())) tmp=start_of_travel;
    if((cn->qw()==NULL)||(!cn->qw()->is_inside())) tmp=end_of_travel;
    cn->calc_slor(tmp);
    cnnxt=cn;
    cn=cn->qw();
}
}
            dir=east_travel|north_include|south_include;
            cn=cnnxt;
            while (cn!=NULL) {
while ((cn!=NULL)&&(!cn->is_inside())) {cn=cn->qe();}

```

```

while ((cn!=NULL)&&( cn->is_inside())) {
    tmp=dir;
    if ((cn->qw()==NULL) || (!cn->qw()->is_inside())) tmp=start_of_travel;
    cn->back_sub(tmp);
    cnnxt=cn;
    cn=cn->qe();
}
if (from_low_coord) cnnxt=cnnxt->qn();
else cnnxt=cnnxt->qs();
}
else {
    if (from_low_coord) cnnxt=m[0];
    else cnnxt=m[M-1];
    while (cnnxt!=NULL) {
        dir=north_travel|east_include|west_include;
        cn=cnnxt;
        while (cn!=NULL) {
            while ((cn!=NULL)&&(!cn->is_inside())) (cn=cn->qn());
            while ((cn!=NULL)&&( cn->is_inside())) {
                tmp=dir;
                if((cn->qs()==NULL) || (!cn->qs()->is_inside())) tmp=start_of_travel;
                if((cn->qn()==NULL) || (!cn->qn()->is_inside())) tmp=end_of_travel;
                cn->calc_slor(tmp);
                cnnxt=cn;
                cn=cn->qn();
            }
            dir=south_travel|east_include|west_include;
            cn=cnnxt;
            while (cn!=NULL) {
                while ((cn!=NULL)&&(!cn->is_inside())) (cn=cn->qs());
                while ((cn!=NULL)&&( cn->is_inside())) {
                    tmp=dir;
                    if((cn->qn()==NULL) || (!cn->qn()->is_inside())) tmp=start_of_travel;
                    cn->back_sub(tmp);
                    cnnxt=cn;
                    cn=cn->qs();
                }
                if (from_low_coord) cnnxt=cnnxt->qw();
                else cnnxt=cnnxt->qe();
            }
        }
        // have to do a topological algorithm rather than this straight
        // array type rectangle
        cn=cnnxt=m[0];
        cn->calc_boundary(start_of_travel);
        if ((cn->qn()) (cn=cn->qn();dir=north;))
        else if (cn->qe()) (cn=cn->qe();dir=east;))
        else if (cn->qs()) (cn=cn->qs();dir=south;))
        else if (cn->qw()) (cn=cn->qw();dir=west;))
        while (cn!=cnnxt) {
            cn->calc_boundary(0);
            switch(dir {
                case north:
                    if (cn->qe()) (cn=cn->qe();dir=east;))
                    else if (cn->qn()) (cn=cn->qn();dir=north;))
                    else if (cn->qw()) (cn=cn->qw();dir=west;))
                    else if (cn->qs()) (cn=cn->qs();dir=south;))
                    break;
                case south:
                    if (cn->qw()) (cn=cn->qw();dir=west;))
                    else if (cn->qs()) (cn=cn->qs();dir=south;))
                    else if (cn->qe()) (cn=cn->qe();dir=east;))
                    else if (cn->qn()) (cn=cn->qn();dir=north;))
                    break;
                case east:
                    if (cn->qs()) (cn=cn->qs();dir=south;))
                    else if (cn->qe()) (cn=cn->qe();dir=east;))
                    else if (cn->qn()) (cn=cn->qn();dir=north;))
                    else if (cn->qw()) (cn=cn->qw();dir=west;))
                    break;
                case west:

```

```

if (cn->qn()) {cn=cn->qn();dir=north;}
else if (cn->qw()) {cn=cn->qw();dir=west;}
else if (cn->qs()) {cn=cn->qs();dir=south;}
else if (cn->qe()) {cn=cn->qe();dir=east;}
break;
}
}
// This is a null function, it is meant to maintain values
// for previous iterations, so that the simulator could perform
// end point detection to end the simulation on it's own without
// having a specified number of iterations. This is a very
// useful thing, but the algorithm balloons very quickly when
// many criterion are used.
void region::update_newton()
{
    unsigned int i,j;
    for (i=0;i<M;i++)
        for (j=0;j<N;j++)
            m[i+j*M]->update_newton();
}

void setminmax(vector t,int &tlx,int &thx,int &tly,int &thy)
{
    if (((int)t[0])<tlx) tlx=t[0];
    if (((int)t[0])>thx) thx=t[0];
    if (((int)t[1])<tly) tly=t[1];
    if (((int)t[1])>thy) thy=t[1];
}
// this is the graphing function for both the screen and the printer.
// it graphs all parameters and car. rotate in 3 dimensions to aid
// visualization.
// HDC dc: this is the windows device context which is used for all windows
// gdi functions (used for drawing text and graphics on windows devices)
// HPEN *drawpen: this is an array of three pens used for drawing.
// unsigned int gtype: bits are set in this param to signify which types
// of graphs to display.
// unsigned int pix: this is the number of pixels width and height (square)
// in which the function has to display the graphs
// unsigned int xo: this is the x offset coordinate (in pixels)
// unsigned int yo: this is the y offset coordinate (in pixels)
// scalar rotation: this is the counter clockwise rotation in degrees
// scalar tilt: this is the forward tilt in degrees
// First, a transformation matrix and vector are created.
// Then, the extent of the graph is calculated and used for modifications
// to the transformation matrix and vector so that the graph stays where
// it is supposed to. Then for each node in the region, lines are drawn
// to the node north and west for each of the variables to be displayed.
void region::graph(HDC dc,HPEN *drawpen,unsigned int gtype,
    unsigned int pix,unsigned int xo,unsigned int yo,scalar rotation,scalar tilt)
{
    HPEN oldpen;
    scalar lx,hx,ly,hy,txo,tyo,tm,l[7],h[7];
    vector tv,tb,t;
    node *tmp;
    matrix ta;
    unsigned int i,j,k,tr[7];
    int tlx,thx,tly,thy;
    tr[0]=hole_graph;tr[1]=elec_graph;tr[2]=volt_graph;
    tr[3]=curx_graph;tr[4]=cury_graph;tr[5]=ex_graph;tr[6]=ey_graph;
    oldpen=SelectObject(dc,drawpen[0]);
    // mapping matrix and vector
    ta[0][0]=cos(3.14/180.0*rotation);
    ta[0][1]=-sin(3.14/180.0*rotation);
    ta[0][2]=0.0;
    ta[1][0]=-sin(3.14/180.0*tilt)*sin(3.14/180.0*rotation);
    ta[1][1]=-sin(3.14/180.0*tilt)*cos(3.14/180.0*rotation);
    ta[1][2]=-cos(3.14/180.0*tilt);
    ta[2][0]=ta[2][1]=ta[2][2]=0.0;
    tb[0]=0.0;
    tb[1]=pix;
    tb[2]=0.0;
    l[0]=0.0; l[1]=0.0; l[2]=-1.0; l[3]=-1e-2; l[4]=-1e-2; l[5]=-1e10; l[6]=-1e10;
    h[0]=20.0; h[1]=20.0; h[2]=1.0; h[3]=1e-2; h[4]=1e-2; h[5]=1e10; h[6]=1e10;
    lx=0.0; hx=20e-4;
}

```

```

ly=0.0; hy=10e-4;
// calculate offsets and multiplier for keeping graph within bounds
if (hx>hy) (txo=1.05; tyo=0.05*hy/hx;)
else (txo=0.05*hy/hx; tyo=1.05;)
tv[0]=-0.05*pix; tv[1]=-0.05*pix; tv[2]=0.0; t=ta*tv+tb; tlx=thx+t[0]; tly=thy+t[1];
tv[0]=-0.05*pix; tv[1]=-0.05*pix; tv[2]=pix; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=-0.05*pix; tv[1]=tyo*pix; tv[2]=0.0; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=-0.05*pix; tv[1]=tyo*pix; tv[2]=pix; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=txo*pix; tv[1]=-0.05*pix; tv[2]=0.0; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=txo*pix; tv[1]=-0.05*pix; tv[2]=pix; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=txo*pix; tv[1]=tyo*pix; tv[2]=0.0; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
tv[0]=txo*pix; tv[1]=tyo*pix; tv[2]=pix; t=ta*tv+tb; setminmax(t, tlx, thx, tly, thy);
txo=txo; tyo=tyo;
if (tlx!=0) txo=tlx;
if (tly!=0) tyo=tly;
tm=((scalar)pix)/1.1/(scalar)(thx-tlx);
if (tm>(((scalar)pix)/1.1/(scalar)(thy-tly)))
tm=(((scalar)pix)/1.1/(scalar)(thy-tly));
// draw graph
for (i=0; i<M; i++)
for (j=0; j<N; j++) {
// draw vertical
if (m[i+j*M]->qn()!=NULL) {
// hole, electron, voltage
for (k=0; k<3; k++) if (gtype&tr[k]) {
SelectObject(dc, drawpen[k<3?k:2]);
tmp=m[i+j*M];
tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
switch (k) {
case 0: tv[2]=(tmp->qY(hole)>0.0?log10(tmp->qY(hole)):0.0); break;
case 1: tv[2]=(tmp->qY(elec)>0.0?log10(tmp->qY(elec)):0.0); break;
case 2: tv[2]=tmp->qY(volt); break;
}
tv[2]=(tv[2]-1[k])/h[k]-1[k])*pix;
t=ta*tv+tb;
MoveTo(dc, txo+t[0]*tm, tyo+t[1]*tm);
tmp=tmp->qn();
tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
switch (k) {
case 0: tv[2]=(tmp->qY(hole)>0.0?log10(tmp->qY(hole)):0.0); break;
case 1: tv[2]=(tmp->qY(elec)>0.0?log10(tmp->qY(elec)):0.0); break;
case 2: tv[2]=tmp->qY(volt); break;
}
tv[2]=(tv[2]-1[k])/h[k]-1[k])*pix;
t=ta*tv+tb;
LineTo(dc, txo+t[0]*tm, tyo+t[1]*tm);
}
// current
for (k=3; k<5; k++) {
tmp=m[i+j*M];
if ((gtype&tr[k])&&(tmp->JxOK())&&(tmp->JyOK())&&
(tmp->qn()->JxOK())&&(tmp->qn()->JyOK())) {
SelectObject(dc, drawpen[2]);
tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
switch (k) {
case 3: tv[2]=tmp->Jx(dop); break;
case 4: tv[2]=tmp->Jy(dop); break;
}
tv[2]=(tv[2]-1[k])/h[k]-1[k])*pix;
t=ta*tv+tb;
MoveTo(dc, txo+t[0]*tm, tyo+t[1]*tm);
tmp=tmp->qn();
tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
switch (k) {
case 3: tv[2]=tmp->Jx(dop); break;
case 4: tv[2]=tmp->Jy(dop); break;
}
tv[2]=(tv[2]-1[k])/h[k]-1[k])*pix;
t=ta*tv+tb;
LineTo(dc, txo+t[0]*tm, tyo+t[1]*tm);
}
}

```

```

)
// electric field
for (k=5;k<7;k++) {
    tmp=m[i+j*M];
    if ((gtype&tr[k])&&(tmp->ExOK())&&(tmp->EyOK())&&
        (tmp->qn()->ExOK())&&(tmp->qn()->EyOK())) {
        SelectObject(dc,drawpen[2]);
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
        switch (k) {
            case 5: tv[2]=tmp->Ex(); break;
            case 6: tv[2]=tmp->Ey(); break;
        }
        tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
        t=ta*tv+tb;
        MoveTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
        tmp=tmp->qn();
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
        switch (k) {
            case 5: tv[2]=tmp->Ex(); break;
            case 6: tv[2]=tmp->Ey(); break;
        }
        tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
        t=ta*tv+tb;
        LineTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
    }
}
// draw horizontal
if (m[i+j*M]->qw()!=NULL) {
// hole, electron, voltage
for (k=0;k<3;k++) if (gtype&tr[k]) {
    SelectObject(dc,drawpen[k<3?k:2]);
    tmp=m[i+j*M];
    tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
    tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
    switch (k) {
        case 0: tv[2]=(tmp->qY(hole)>0.0?log10(tmp->qY(hole)):0.0); break;
        case 1: tv[2]=(tmp->qY(elec)>0.0?log10(tmp->qY(elec)):0.0); break;
        case 2: tv[2]=tmp->qY(volt); break;
    }
    tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
    t=ta*tv+tb;
    MoveTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
    tmp=tmp->qw();
    tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
    tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
    switch (k) {
        case 0: tv[2]=(tmp->qY(hole)>0.0?log10(tmp->qY(hole)):0.0); break;
        case 1: tv[2]=(tmp->qY(elec)>0.0?log10(tmp->qY(elec)):0.0); break;
        case 2: tv[2]=tmp->qY(volt); break;
    }
    tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
    t=ta*tv+tb;
    LineTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
}
// current
for (k=3;k<5;k++) {
    tmp=m[i+j*M];
    if ((gtype&tr[k])&&(tmp->JxOK())&&(tmp->JyOK())&&
        (tmp->qw()->JxOK())&&(tmp->qw()->JyOK())) {
        SelectObject(dc,drawpen[2]);
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
        switch (k) {
            case 3: tv[2]=tmp->Jx(dop); break;
            case 4: tv[2]=tmp->Jy(dop); break;
        }
        tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
        t=ta*tv+tb;
        MoveTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
        tmp=tmp->qw();
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
    }
}

```

```

switch (k) {
    case 3: tv[2]=tmp->Jx(dop); break;
    case 4: tv[2]=tmp->Jy(dop); break;
}
tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
t=ta*tv+tb;
LineTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
}

// electric field
for (k=5;k<7;k++) {
    tmp=m[i+j*M];
    if ((gtype&tr[k])&&(tmp->ExOK())&&(tmp->EyOK())&&
        (tmp->qw()->ExOK())&&(tmp->qw()->EyOK())) {
        SelectObject(dc,drawpen[2]);
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
        switch (k) {
            case 5: tv[2]=tmp->Ex(); break;
            case 6: tv[2]=tmp->Ey(); break;
        }
        tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
        t=ta*tv+tb;
        MoveTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
        tmp=tmp->qw();
        tv[0]=(tmp->qx()-lx)/(hx-lx)*pix;
        tv[1]=(tmp->qy()-ly)/(hy-ly)*pix;
        switch (k) {
            case 5: tv[2]=tmp->Ex(); break;
            case 6: tv[2]=tmp->Ey(); break;
        }
        tv[2]=(tv[2]-1[k])/(h[k]-1[k])*pix;
        t=ta*tv+tb;
        LineTo(dc,txo+t[0]*tm,tyo+t[1]*tm);
    }
}

if (oldpen!=NULL) SelectObject(dc,oldpen);
}

a simple test for SLOF
void region::slof_test_setup()
{
    unsigned int i,j;
    M=5;N=4;
    for(i=0;i<M;i++)
        for(j=0;j<N;j++)
            if ((i==0)||i==M-1)||j==0||j==N-1)
                f=0x00;
            else {
                f=0x3F;
                if (i==1) f=(f&~0x04)|0x40;
                else if (i==2) f=f|0x80;
                else if (i==3) f=(f&~0x08)|0x40;
                if (j==1) f=(f&~0x02);
                else if (j==2) f=(f&~0x01);
            }
    m[i+j*M]->slof_test_setup.f;
}

// this dumps the simulator contents in Maple readable form
#include <iostream.h>
void dump(region *reg,char *str)
{
    unsigned int i,j,r,c,t;
    ofstream f(str);
    for(t=0;t<9;t++) {
        switch (t) {
            case 0: f<<"Y:=array(1.."<<reg->M<<".1 "<<reg->N<<".["; break;
            case 1: f<<"F:=array(1.."<<reg->M<<".1.."<<reg->N<<".["; break;
            case 2: f<<"Ft:=array(1.."<<reg->M<<".1. "<<reg->N<<".["; break;
            case 3: f<<"N:=array(1.."<<reg->M<<".1.."<<reg->N<<".["; break;
            case 4: f<<"S:=array(1.."<<reg->M<<".1. "<<reg->N<<".["; break;
            case 5: f<<"Ea:=array(1.."<<reg->M<<".1.."<<reg->N<<".["; break;
            case 6: f<<"W:=array(1.."<<reg->M<<".1.."<<reg->N<<".["; break;
        }
    }
}

```

```

        case 7: f<<"C:=array(1.."<<reg->M<<"1.."<<reg->N<<"["; break;
        case 8: f<<"Ct:=array(1.."<<reg->M<<"1.."<<reg->N<<"["; break;
    }
    for(i=0;i<reg->M;i++) {
        if (i<reg->M-1) f<<"\n";
        if (i>0) f<<"["; else f<<"[";
        for(j=0;j<reg->N;j++) {
            if (j>0) f<<"[";
            if 't<3) f<<"array(1..3,1..1,[";
            else f<<"array(1..3,1..3,[";
        for(r=0;r<3;r++) {
            if (r>0) f<<"[";
            if (t<3) {
                switch(t) {
                    case 0: f<<"["<<reg->m[i+j*reg->M]->Y[r]<<"["; break;
                    case 1: f<<"["<<reg->m[i+j*reg->M]->F[r]<<"["; break;
                    case 2: f<<"["<<reg->m[i+j*reg->M]->Ft[r]<<"["; break;
                }
            }
            else {
                f<<"[";
                for(c=0;c<3;c++) {
                    if (c>0) f<<"[";
                    switch(t) {
                        case 3: f<<reg->m[i+j*reg->M]->N[r][c]; break;
                        case 4: f<<reg->m[i+j*reg->M]->S[r][c]; break;
                        case 5: f<<reg->m[i+j*reg->M]->E[r][c]; break;
                        case 6: f<<reg->m[i+j*reg->M]->W[r][c]; break;
                        case 7: f<<reg->m[i+j*reg->M]->C[r][c]; break;
                        case 8: f<<reg->m[i+j*reg->M]->Ct[r][c]; break;
                    }
                }
                f<<"[";
            }
        }
        f<<"[";
    }
    f<<"[";
    f<<"[";
    f<<"[";
    f<<"[";
    f<<"[";
    f.close();
}

```

10.12 Node.cpp

```

// Copyright 1995, Brian Gieschen
#include <math.h>
#include "node.h"
#ifdef NULL
#define NULL 0x0
#endif

#define theta 38.6815
#define es 1.064e-12
#define Nrefp 6.3e16
#define Nrefn 8.5e16
#define alphap 0.76
#define alphan 0.72
#define upmax 495
#define unmax 1330
#define upmin 47.7
#define unmin 65
#define Ecp 1.95e4
#define Ecn 8e3
#define betap 1
#define betan 2
#define tp 2.5e-3
#define tn 2.5e-3

scalar Bz=0.0;
// this is the node creator. _x and _y are the coordinates
// d is the doping. The electron and hole concentrations and potential
// are initialized at this point.
node::node(scalar _x,scalar _y,scalar d)

```

```

{
    x=_x;y=_y;D=d;
    n=s=e=w=NULL;
    flag=inside;
    if (d>0.0) { // n-type
        Y[elec]=D/4.0*sqrt(D*D/4.0+ni*ni);
        Y[hole]=ni*ni/Y[elec];
        Y[volt]=K*T/q*log(Y[elec]/ni);
    }
    else if (d<0.0) { // p-type
        Y[hole]=-D/2.0*sqrt(D*D/4.0+ni*ni);
        Y[elec]=ni*ni/Y[hole];
        Y[volt]=-K*T/q*log(Y[hole]/ni);
    }
    else { // intrinsic
        Y[hole]=ni;
        Y[elec]=ni;
        Y[volt]=0.0;
    }
}
// this deletes the node gracefully, repointing
// adjacent node's pointers so that they don't
// point to a non-existent node.
void node::~node()
{
    if (n!=NULL) n->s=NULL;
    if (s!=NULL) s->n=NULL;
    if (e!=NULL) e->>w=NULL;
    if (w!=NULL) w->>e=NULL;
    if (n!=NULL) {
        n->set_flag();
        if (n->e!=NULL) n->e->set_flag();
        if (n->w!=NULL) n->w->set_flag();
    }
    if (s!=NULL) {
        s->set_flag();
        if (s->e!=NULL) s->e->set_flag();
        if (s->w!=NULL) s->w->set_flag();
    }
    if (e!=NULL) e->set_flag();
    if (w!=NULL) w->set_flag();
    n=s=e=w=NULL;
}
// this sets the flag variable based on the nodes position in
// the device. It detects that a node is on the edge or on
// the inside of the device. This important to be able to
// properly deal with boundary conditions.
void node::set_flag()
{
    if ((n==NULL) || (s==NULL) || (e==NULL) || (w==NULL) ||
        (n->e==NULL) || (n->w==NULL) || (s->e==NULL) ||
        (s->w==NULL) || (e->n==NULL) || (e->s==NULL) ||
        (w->n==NULL) || (w->s==NULL))
        flag=floating_boundary;
    else
        flag=inside;
}
// this links adjacent nodes. It will not set a pointer
// to null, but it will change a pointer to another node.
// it also calls set_flag afterward to maintain the integrity
// of the grid.
// node *_n,*_s,*_e,*_w: pointers to adjacent nodes in the respective
// directions. If one or all are null nothing
// is done with this pointer.
void node::link(node *_n,node *_s,node *_e,node *_w)
{
    if (_n!=NULL) (n=_n;n->s=this;);
    if (_s!=NULL) (s=_s;s->n=this;);
    if (_e!=NULL) (e=_e;e->w=this;);
    if (_w!=NULL) (w=_w;w->e=this;);
    if (n!=NULL) {
        n->set_flag();
        if (n->e!=NULL) n->e->set_flag();
        if (n->w!=NULL) n->w->set_flag();
    }
}

```



```

if (s!=NULL) {
    s->set_flag();
    if (s->e!=NULL) s->e->set_flag();
    if (s->w!=NULL) s->w->set_flag();
}
if (e!=NULL) e->set_flag();
if (w!=NULL) w->set_flag();
set_flag();
}
// this is the potential ramping equation. It takes into account voltage and
// position to come up with a fairly good approximation to start the next
// newton iteration.
// scalar ramp: the bias increment which is applied to the potential in the next
// parameter.
// scalar v: the bias of the contact which the increment will be applied to
// scalar v1,v2,v3: the bias of the other contacts in the device.
// scalar td,td1,td2,td3: the distances between this node and the respective
// contacts.
void node::ramp_voltage(scalar ramp,scalar v,scalar v1,scalar v2, scalar v3,
    scalar td,scalar td1,scalar td2,scalar td3)
{
    scalar tmp,tv,tv1,tv2,tv3,tdtotal;
    if (!is_contacted()) {
        tdtotal = td+td1+td2+td3;
        td=tdtotal/td;
        td1=tdtotal/td1;
        td2=tdtotal/td2;
        td3=tdtotal/td3;
        tdtotal = td+td1+td2+td3;
        tv= fabs(Y[volt]-v);
        tv1=fabs(Y[volt]-v1);
        tv2=fabs(Y[volt]-v2);
        tv3=fabs(Y[volt]-v3);
        tmp=ramp*(tv1+tv2+tv3)/(tv+tv1+tv2+tv3)*td/tdtotal;
        Y[volt]+=tmp;
    }
}
// the following functions directly calculate equation parameters
// for the transport and poisson equations. They are not optimized
// for speed, rather for ease of debugging. If speed optimization
// were applied, the simulator would run easily an order of magnitude
// faster.
// functions like dU_dpMN coorespond to d U(M,N)
// -----
//                                     d p(M,N)
// functions like ldJpx_dvM1N0 coorespond to d [ limit Jpx(M',N) ]
// ----- [ v(M,N)->v(M+1,N) ]
//                                     d v(M+1,N-1) [
scalar node::hx()
{
    return w->x-x;
}

scalar node::hy()
{
    return n->y-y;
}

scalar node::Hx(unsigned int np,gaussian_doping *d)
{
    if (np==hole)
        return hx()*hy()/(hy()+upx(d)*Bz*hx());
    else if (np==elec)
        return hx()*hy()/(hy()+unx(d)*Bz*hx());
    return 0.0;
}

scalar node::Hy(unsigned int np,gaussian_doping *d)
{
    if (np==hole)
        return hy()*hx()/(hx()+upy(d)*Bz*hy());
    else if (np==elec)
        return hy()*hx()/(hx()+uny(d)*Bz*hy());
    return 0.0;
}

```

```

scalar node::hxp()
{
    return (hx()+e->hx())/2.0;
}

scalar node::hyp()
{
    return (hy()+s->hy())/2.0;
}

scalar node::upx(gaussian_doping *d)
{
    return (((upmax-upmin)/(1.0*pow(fabs(d->N(x/2.0+w->x/2.0,y))/Nrefp,alphap))))+upmin)
        /pow(1.0*pow((Y[volt]-w->Y[volt])/hx()/Ecp,betap),1.0/betap);
}

scalar node::upy(gaussian_doping *d)
{
    return (((upmax-upmin)/(1.0*pow(fabs(d->N(x,y/2.0+n->y/2.0))/Nrefp,alphap))))+upmin)
        /pow(1.0*pow((Y[volt]-n->Y[volt])/hy()/Ecp,betap),1.0/betap);
}

scalar node::unx(gaussian_doping *d)
{
    return (((unmax-unmin)/(1.0*pow(fabs(d->N(x/2.0+w->x/2.0,y))/Nrefn,alphan))))+unmin)
        /pow(1.0*pow((Y[volt]-w->Y[volt])/hx()/Ecn,betan),1.0/betan);
}

scalar node::uny(gaussian_doping *d)
{
    return (((unmax-unmin)/(1.0*pow(fabs(d->N(x,y/2.0+n->y/2.0))/Nrefn,alphan))))+unmin)
        /pow(1.0*pow((Y[volt]-n->Y[volt])/hy()/Ecn,betan),1.0/betan);
}

scalar node::U()
{
    return (Y[hole]*Y[elec]-ni*ni)/(tn*(Y[hole]+ni)+tp*(Y[elec]+ni));
}

scalar node::dU_dpMN()
{
    return (Y[elec]-tn*U)/(tn*(Y[hole]+ni)+tp*(Y[elec]+ni));
}

scalar node::dU_dnMN()
{
    return (Y[hole]-tp*U)/(tn*(Y[hole]+ni)+tp*(Y[elec]+ni));
}

scalar node::Ex()
{
    return (Y[volt]-w->Y[volt])/hx();
}

scalar node::Ey()
{
    return (Y[volt]-n->Y[volt])/hy();
}

scalar node::Exi(gaussian_doping *d,scalar &le,scalar &he,scalar &lw,scalar &hw)
{
    scalar me,mw;
    me=d->N(e->x,(y+n->y)/2.0);me=me>1.0?log(me):me<-1.0?-log(-me):0.0;
    mw=d->N(w->x,(y+n->y)/2.0);mw=mw>1.0?log(mw):mw<-1.0?-log(-mw):0.0;
    le=e->D; le=le>1.0?log(le):le<-1.0?-log(-le):0.0;le=fabs(me-le);
    lw=w->D; lw=lw>1.0?log(lw):lw<-1.0?-log(-lw):0.0;lw=fabs(mw-lw);
    he=e->n->D;he=he>1.0?log(he):he<-1.0?-log(-he):0.0;he=fabs(me-he);
    hw=w->n->D;hw=hw>1.0?log(hw):hw<-1.0?-log(-hw):0.0;hw=fabs(mw-hw);
    if (le==he) le=he=1.0;
    if (lw==hw) lw=hw=1.0;
    return ((e->Y[volt]*he+e->n->Y[volt]*le),(le+he)
        -(w->Y[volt]*hw+w->n->Y[volt]*lw),(lw+hw)),(hx()+e->hx());
}

```

```

scalar node::Ey(gaussian_doping *d, scalar &ls, scalar &hs, scalar &ln, scalar &hn)
{
    scalar ms, mn;
    ms=d->N((x+w->x)/2.0, s->y); ms=ms>1.0?log(ms):ms<-1.0?-log(-ms):0.0;
    mn=d->N((x+w->x)/2.0, n->y); mn=mn>1.0?log(mn):mn<-1.0?-log(-mn):0.0;
    ls=s->D; ls=ls>1.0?log(ls):ls<-1.0?-log(-ls):0.0; ls=fabs(ms-ls);
    ln=n->D; ln=ln>1.0?log(ln):ln<-1.0?-log(-ln):0.0; ln=fabs(mn-ln);
    hs=s->w->D; hs=hs>1.0?log(hs):hs<-1.0?-log(-hs):0.0; hs=fabs(ms-hs);
    hn=n->w->D; hn=hn>1.0?log(hn):hn<-1.0?-log(-hn):0.0; hn=fabs(mn-hn);
    if (ls==hs) ls=hs=1.0;
    if (ln==hn) ln=hn=1.0;
    return ((s->Y[volt]*hs+s->w->Y[volt]*ls)/(ls+hs)
            -(n->Y[volt]*hn+n->w->Y[volt]*ln)/(ln+hn))/(hy()+s->hy());
}

scalar node::Bx(unsigned int np, gaussian_doping *d,
    scalar &ls, scalar &hs, scalar &ln, scalar &hn)
{
    scalar ey;
    ey=Ey(d, ls, hs, ln, hn);
    if (np==hole)
        return theta*(Ex()+upy(d)*Bz*ey)*Hx(np, d);
    else if (np==elec)
        return theta*(Ex()+uny(d)*Bz*ey)*Hx(np, d);
    return 0.0;
}

scalar node::By(unsigned int np, gaussian_doping *d,
    scalar &le, scalar &he, scalar &lw, scalar &hw)
{
    scalar ex;
    ex=Exi(d, le, he, lw, hw);
    if (np==hole)
        return theta*(Ey()-upx(d)*Bz*ex)*Hy(np, d);
    else if (np==elec)
        return theta*(Ey()-unx(d)*Bz*ex)*Hy(np, d);
    return 0.0;
}

scalar node::Jpx(gaussian_doping *d)
{
    scalar ls, hs, ln, hn;
    scalar b=Bx(hole, d, ls, hs, ln, hn);
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    if (b==0.0) return K*T*upx(d)/Hx(hole, d)/m*(Y[hole]-w->Y[hole]);
    return K*T*upx(d)/Hx(hole, d)/m
        *(Y[hole]*(b/(1.0-exp(-b)))+w->Y[hole]*(b/(1.0-exp(b))));
}

scalar node::Jpy(gaussian_doping *d)
{
    scalar le, he, lw, hw;
    scalar b=By(hole, d, le, he, lw, hw);
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    if (b==0.0) return K*T*upy(d)/Hy(hole, d)/m*(Y[hole]-n->Y[hole]);
    return K*T*upy(d)/Hy(hole, d)/m
        *(Y[hole]*(b/(1.0-exp(-b)))+n->Y[hole]*(b/(1.0-exp(b))));
}

scalar node::Jnx(gaussian_doping *d)
{
    scalar ls, hs, ln, hn;
    scalar b=Bx(elec, d, ls, hs, ln, hn);
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    if (b==0.0) return K*T*unx(d)/Hx(elec, d)/m*(w->Y[elec]-Y[elec]);
    return K*T*unx(d)/Hx(elec, d)/m
        *(Y[elec]*(b/(1.0-exp(b)))+w->Y[elec]*(b/(1.0-exp(-b))));
}

scalar node::Jny(gaussian_doping *d)
{
    scalar le, he, lw, hw;
    scalar b=By(elec, d, le, he, lw, hw);
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    if (b==0.0) return K*T*uny(d)/Hy(elec, d)/m*(n->Y[elec]-Y[elec]);
}

```

```

    return K*T*uny(d)/Hy(elec,d)/m
    *{Y[elec]*(b/(1.0-exp(b)))+n->Y[elec]*(b/(1.0-exp(-b)))};
}

```

```

scalar node::qI(unsigned int side,gaussian_doping *d)

```

```

{
    scalar xy;
    xy=0.0;
    if ((side==north)||side==south) {
        if (e) xy+=(x-e->x)/2.0;
        if (w) xy+=(w->x-x)/2.0;
    } else if ((side==east)||side==west) {
        if (s) xy+=(y-s->y)/2.0;
        if (n) xy+=(n->y-y)/2.0;
    }
    switch (side) {
        case north: return s->Jy(d)*xy;
        case south: return -Jy(d)*xy;
        case west: return e->Jx(d)*xy;
        case east: return -Jx(d)*xy;
    }
    return 0.0;
}

```

```

scalar node::dJpx_dpMN(gaussian_doping *d)

```

```

{
    scalar ls,hs,ln,hn;
    scalar b=Bx(hole,d,ls,hs,ln,hn);
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    return K*T*upx(d)/Hx(hole,d)/m*b/(1.0-exp(-b));
}

```

```

scalar node::dJpx_dpMlN(gaussian_doping *d)

```

```

{
    scalar ls,hs,ln,hn;
    scalar b=Bx(hole,d,ls,hs,ln,hn);
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    return K*T*upx(d)/Hx(hole,d)/m*b/(1.0-exp(b));
}

```

```

scalar node::dJpy_dpMN(gaussian_doping *d)

```

```

{
    scalar le,he,lw,hw;
    scalar b=By(hole,d,le,he,lw,hw);
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    return K*T*upy(d)/Hy(hole,d)/m*b/(1.0-exp(-b));
}

```

```

scalar node::dJpy_dpMlN(gaussian_doping *d)

```

```

{
    scalar le,he,lw,hw;
    scalar b=By(hole,d,le,he,lw,hw);
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    return K*T*upy(d)/Hy(hole,d)/m*b/(1.0-exp(b));
}

```

```

scalar node::dJnx_dnMN(gaussian_doping *d)

```

```

{
    scalar ls,hs,ln,hn;
    scalar b=Bx(elec,d,ls,hs,ln,hn);
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    return K*T*unx(d)/Hx(elec,d)/m*b/(1.0-exp(b));
}

```

```

scalar node::dJnx_dnMlN(gaussian_doping *d)

```

```

{
    scalar ls,hs,ln,hn;
    scalar b=Bx(elec,d,ls,hs,ln,hn);
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    return K*T*unx(d)/Hx(elec,d)/m*b/(1.0-exp(-b));
}

```

```

scalar node::dJny_dnMN(gaussian_doping *d)

```

```

{
    scalar le,he,lw,hw;

```

```

    scalar b=By(elec,d,le,he,lw,hw);
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    return K*T*uny(d)/Hy(elec,d)/m*b/(1.0-exp(b));
}

scalar node::dJny_dnmN1(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    scalar b=By(elec,d,le,he,lw,hw);
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    return K*T*uny(d)/Hy(elec,d)/m*b/(1.0-exp(-b));
}

scalar node::dJpx_dvMN(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    scalar b=Bx(hole,d,ls,hs,ln,hn);
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    return q*upx(d)/hx()/m
        *(1.0/(1.0-exp(-b))*(1.0-(b*exp(-b))/(1.0-exp(-b)))*Y[hole]
        +1.0/(1.0-exp(b))*(1.0+(b*exp(b))/(1.0-exp(b)))*w->Y[hole]);
}

scalar node::dJpx_dvM1N(gaussian_doping *d)
{
    return -dJpx_dvMN(d);
}

scalar node::dJpx_dvMN0(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(hole,d,ls,hs,ln,hn);
    return hx()*upy(d)*Bz*hs/(ls+hs)/(hy()+s->hy())*dJpx_dvMN(d);
}

scalar node::dJpx_dvM1N0(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(hole,d,ls,hs,ln,hn);
    return hx()*upy(d)*Bz*ls/(ls+hs)/(hy()+s->hy())*dJpx_dvMN(d);
}

scalar node::dJpx_dvMN1(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(hole,d,ls,hs,ln,hn);
    return -hx()*upy(d)*Bz*hn/(ln+hn)/(hy()+s->hy())*dJpx_dvMN(d);
}

scalar node::dJpx_dvM1N1(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(hole,d,ls,hs,ln,hn);
    return -hx()*upy(d)*Bz*ln/(ln+hn)/(hy()+s->hy())*dJpx_dvMN(d);
}

scalar node::dJpy_dvMN(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    scalar b=By(hole,d,le,he,lw,hw);
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    return q*upy(d)/hy()/m
        *(1.0/(1.0-exp(-b))*(1.0-(b*exp(-b))/(1.0-exp(-b)))*Y[hole]
        +1.0/(1.0-exp(b))*(1.0+(b*exp(b))/(1.0-exp(b)))*n->Y[hole]);
}

scalar node::dJpy_dvMN1(gaussian_doping *d)
{
    return -dJpy_dvMN(d);
}

scalar node::dJpy_dvMON(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(hole,d,le,he,lw,hw);

```

```

    return -hy()*upx(d)*Bz*he/(1e+he)/(hx()+e->hx())*dJpy_dvMN(d);
}

scalar node::dJpy_dvMN1(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(hole,d,le,he,lw,hw);
    return -hy()*upx(d)*Bz*le/(1e+he)/(hx()+e->hx())*dJpy_dvMN(d);
}

scalar node::dJpy_dvMN1N(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(hole,d,le,he,lw,hw);
    return hy()*upx(d)*Bz*hw/(lw+hw)/(hx()+e->hx())*dJpy_dvMN(d);
}

scalar node::dJpy_dvMN1N1(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(hole,d,le,he,lw,hw);
    return hy()*upx(d)*Bz*lw/(lw+hw)/(hx()+e->hx())*dJpy_dvMN(d);
}

scalar node::dJnx_dvMN(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    scalar b=Bx(elec,d,ls,hs,ln,hn);
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    return q*unx(d)/hx()/m
        *(1.0/(1.0-exp(-b))*(1.0+(b*exp(-b))/(1.0-exp(-b)))*Y[elec]
        +1.0/(1.0-exp(-b))*(1.0-(b*exp(-b))/(1.0-exp(-b)))*w->Y[elec]);
}

scalar node::dJnx_dvMN1N(gaussian_doping *d)
{
    return -dJnx_dvMN(d);
}

scalar node::dJnx_dvMN0(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(elec,d,ls,hs,ln,hn);
    return hx()*uny(d)*Bz*hs/(ls+hs)*(hy()+s->hy())*dJnx_dvMN(d);
}

scalar node::dJnx_dvMN1N0(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(elec,d,ls,hs,ln,hn);
    return hx()*uny(d)*Bz*ls/(ls+hs)*(hy()+s->hy())*dJnx_dvMN(d);
}

scalar node::dJnx_dvMN1(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(elec,d,ls,hs,ln,hn);
    return -hx()*uny(d)*Bz*hn/(ln+hn)*(hy()+s->hy())*dJnx_dvMN(d);
}

scalar node::dJnx_dvMN1N1(gaussian_doping *d)
{
    scalar ls,hs,ln,hn;
    Bx(elec,d,ls,hs,ln,hn);
    return -hx()*uny(d)*Bz*ln/(ln+hn)*(hy()+s->hy())*dJnx_dvMN(d);
}

scalar node::dJny_dvMN(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    scalar b=By(elec,d,le,he,lw,hw);
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    return q*uny(d)/hy()/m
        *(1.0/(1.0-exp(-b))*(1.0+(b*exp(-b))/(1.0-exp(-b)))*Y[elec]
        +1.0/(1.0-exp(-b))*(1.0-(b*exp(-b))/(1.0-exp(-b)))*n->Y[elec]);
}

```

```

)

scalar node::dJny_dvMN1(gaussian_doping *d)
{
    return -dJny_dvMN(d);
}

scalar node::dJny_dvMON(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(elec,d,le,he,lw,hw);
    return -hy()*unx(d)*Bz*he/(le+he)/(hx()+e->hx())*dJny_dvMN(d);
}

scalar node::dJny_dvMON1(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(elec,d,le,he,lw,hw);
    return -hy()*unx(d)*Bz*le/(le+he)/(hx()+e->hx())*dJny_dvMN(d);
}

scalar node::dJny_dvM1N(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(elec,d,le,he,lw,hw);
    return hy()*unx(d)*Bz*hw/(lw+hw)/(hx()+e->hx())*dJny_dvMN(d);
}

scalar node::dJny_dvM1N1(gaussian_doping *d)
{
    scalar le,he,lw,hw;
    By(elec,d,le,he,lw,hw);
    return hy()*unx(d)*Bz*lw/(lw+hw)/(hx()+e->hx())*dJny_dvMN(d);
}

scalar node::ldJpx_dpMN(gaussian_doping *d)
{
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    return K*T*upx(d)/Hx(hole,d)/m;
}

scalar node::ldJpx_dpM1N(gaussian_doping *d)
{
    scalar m=(1.0+upx(d)*upx(d)*Bz*Bz);
    return -K*T*upx(d)/Hx(hole,d)/m;
}

scalar node::ldJpy_dpMN(gaussian_doping *d)
{
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    return K*T*upy(d)/Hy(hole,d)/m;
}

scalar node::ldJpy_dpM1N(gaussian_doping *d)
{
    scalar m=(1.0+upy(d)*upy(d)*Bz*Bz);
    return -K*T*upy(d)/Hy(hole,d)/m;
}

scalar node::ldJnx_dnMN(gaussian_doping *d)
{
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    return -K*T*unx(d)/Hx(elec,d)/m;
}

scalar node::ldJnx_dnM1N(gaussian_doping *d)
{
    scalar m=(1.0+unx(d)*unx(d)*Bz*Bz);
    return K*T*unx(d)/Hx(elec,d)/m;
}

scalar node::ldJny_dnMN(gaussian_doping *d)
{
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    return -K*T*uny(d)/Hy(elec,d)/m;
}

```

```

}

scalar node::ldJny_dnMN1(gaussian_doping *d)
{
    scalar m=(1.0+uny(d)*uny(d)*Bz*Bz);
    return K*T*uny(d)/Hy(elec,d)/m;
}

scalar node::ldJpx_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpx_dvMN1N(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpx_dvMN0(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpx_dvMN1N0(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpx_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpx_dvMN1N1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpy_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpy_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpy_dvMN0N(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpy_dvMN0N1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJpy_dvMN1N1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJnx_dvMN(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJnx_dvMN1N(gaussian_doping *d)
{

```



```

    return 0.0;
}

scalar node::ldJnx_dvMN0(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJnx_dvMN10(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJnx_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJnx_dvMN11(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN0N(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN0N1(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN1N(gaussian_doping *d)
{
    return 0.0;
}

scalar node::ldJny_dvMN1N1(gaussian_doping *d)
{
    return 0.0;
}

// these are incremental portions of the Matrix equations
// relating this node with adjacent nodes.
// the function nNt produces the NORTH matrix elements which
// depend on the values of the node north of this node
// the function eNt produces the NORTH matrix elements which
// depend on the values of the node east of this node
// the function wNt produces the NORTH matrix elements which
// depend on the values of the node west of this node
// the NORTH matrix relates the values of the node north of this
// node to this node.
matrix node::nNt(gaussian_doping *d)
{
    matrix t;
    t[0][0]=1.0/q/hyp()*dJpy_dpMN1(d);
    t[0][1]=0.0;
    t[0][2]=1.0/q/hyp()*dJpy_dvMN1(d);
    t[1][0]=0.0;
    t[1][1]=1.0/q/hyp()*dJny_dnMN1(d);
    t[1][2]=1.0/q/hyp()*dJny_dvMN1(d);
    t[2][0]=0.0;
    t[2][1]=0.0;
    t[2][2]=1.0/hyp()/hy();
}

```

```

    return t;
}

matrix node::eNt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hxp()*e->dJpx_dvMN1(d);
    t[1][2]=-1.0/q/hxp()*e->dJnx_dvMN1(d);
    return t;
}

matrix node::wNt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hxp()*dJpx_dvMN1(d);
    t[1][2]=1.0/q/hxp()*dJnx_dvMN1(d);
    return t;
}

// the lnNt etc... functions return the same matrix
// elements as nNt etc... , but when the potential
// at the respective node is the same as this node.
// so leNt would be called when the potential of the
// node east to this one is the same.
matrix node::lnNt(gaussian_doping *d)
{
    matrix t;
    t[0][0]=1.0/q/hyp()*ldJpy_dpmN1(d);
    t[0][1]=0.0;
    t[0][2]=1.0/q/hyp()*ldJpy_dvMN1(d);
    t[1][0]=0.0;
    t[1][1]=1.0/q/hyp()*ldJny_dnmN1(d);
    t[1][2]=1.0/q/hyp()*ldJny_dvMN1(d);
    t[2][0]=0.0;
    t[2][1]=0.0;
    t[2][2]=1.0/hyp()/hy();
    return t;
}

matrix node::leNt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hxp()*e->ldJpx_dvMN1(d);
    t[1][2]=-1.0/q/hxp()*e->ldJnx_dvMN1(d);
    return t;
}

matrix node::lwNt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hxp()*ldJpx_dvMN1(d);
    t[1][2]=1.0/q/hxp()*ldJnx_dvMN1(d);
    return t;
}

matrix node::sSt(gaussian_doping *d)
{
    matrix t;
    t[0][0]=-1.0/q/hyp()*s->dJpy_dpmN(d);
    t[0][1]=0.0;
    t[0][2]=-1.0/q/hyp()*s->dJpy_dvMN(d);
    t[1][0]=0.0;
    t[1][1]=-1.0/q/hyp()*s->dJny_dnmN(d);
    t[1][2]=-1.0/q/hyp()*s->dJny_dvMN(d);
    t[2][0]=0.0;
    t[2][1]=0.0;
    t[2][2]=1.0/hyp()/s->hy();
    return t;
}

matrix node::eSt(gaussian_doping *d)
{

```

```

matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*e->dJpx_dvM1N0(d);
t[1][2]=-1.0/q/hyp()*e->dJnx_dvM1N0(d);
return t;
}

matrix node::wSt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*dJpx_dvMNO(d);
t[1][2]=1.0/q/hyp()*dJnx_dvMNO(d);
return t;
}

matrix node::lsSt(gaussian_doping *d)
{
matrix t;
t[0][0]=-1.0/q/hyp()*s->ldJpy_dpMN(d);
t[0][1]=0.0;
t[0][2]=-1.0/q/hyp()*s->ldJpy_dvMN(d);
t[1][0]=0.0;
t[1][1]=-1.0/q/hyp()*s->ldJny_dnMN(d);
t[1][2]=-1.0/q/hyp()*s->ldJny_dvMN(d);
t[2][0]=0.0;
t[2][1]=0.0;
t[2][2]=1.0/hyp()/s->hy();
return t;
}

matrix node::leSt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*e->ldJpx_dvM1N0(d);
t[1][2]=-1.0/q/hyp()*e->ldJnx_dvM1N0(d);
return t;
}

matrix node::lwSt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*ldJpx_dvMNO(d);
t[1][2]=1.0/q/hyp()*ldJnx_dvMNO(d);
return t;
}

matrix node::eEt(gaussian_doping *d)
{
matrix t;
t[0][0]=-1.0/q/hyp()*e->dJpx_dpMN(d);
t[0][1]=0.0;
t[0][2]=-1.0/q/hyp()*e->dJpx_dvMN(d);
t[1][0]=0.0;
t[1][1]=-1.0/q/hyp()*e->dJnx_dnMN(d);
t[1][2]=-1.0/q/hyp()*e->dJnx_dvMN(d);
t[2][0]=0.0;
t[2][1]=0.0;
t[2][2]=1.0/hyp()/e->hx();
return t;
}

matrix node::sEt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*s->dJpy_dvMON1(d);
t[1][2]=-1.0/q/hyp()*s->dJny_dvMON1(d);
return t;
}

matrix node::nEt(gaussian_doping *d)
{

```

```

matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*dJpy_dvMON(d);
t[1][2]=1.0/q/hyp()*dJny_dvMON(d);
return t;
}

matrix node::leEt(gaussian_doping *d)
{
matrix t;
t[0][0]=-1.0/q/hxp()*e->ldJpx_dpMN(d);
t[0][1]=0.0;
t[0][2]=-1.0/q/hxp()*e->ldJpx_dvMN(d);
t[1][0]=0.0;
t[1][1]=-1.0/q/hxp()*e->ldJnx_dnMN(d);
t[1][2]=-1.0/q/hxp()*e->ldJnx_dvMN(d);
t[2][0]=0.0;
t[2][1]=0.0;
t[2][2]=1.0/hxp()/e->hx();
return t;
}

matrix node::lsEt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*s->ldJpy_dvMON1(d);
t[1][2]=-1.0/q/hyp()*s->ldJny_dvMON1(d);
return t;
}

matrix node::lnEt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*ldJpy_dvMON(d);
t[1][2]=1.0/q/hyp()*ldJny_dvMON(d);
return t;
}

matrix node::wWt(gaussian_doping *d)
{
matrix t;
t[0][0]=1.0/q/hxp()*dJpx_dpM1N(d);
t[0][1]=0.0;
t[0][2]=1.0/q/hxp()*dJpx_dvM1N(d);
t[1][0]=0.0;
t[1][1]=1.0/q/hxp()*dJnx_dnM1N(d);
t[1][2]=1.0/q/hxp()*dJnx_dvM1N(d);
t[2][0]=0.0;
t[2][1]=0.0;
t[2][2]=1.0/hxp()*hx();
return t;
}

matrix node::sWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*s->dJpy_dvM1N1(d);
t[1][2]=-1.0/q/hyp()*s->dJny_dvM1N1(d);
return t;
}

matrix node::nWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*dJpy_dvM1N(d);
t[1][2]=1.0/q/hyp()*dJny_dvM1N(d);
return t;
}

matrix node::lwWt(gaussian_doping *d)
{

```

```

matrix t;
t[0][0]=1.0/q/hxp()*ldJpx_dpM1N(d);
t[0][1]=0.0;
t[0][2]=1.0/q/hxp()*ldJpx_dvM1N(d);
t[1][0]=0.0;
t[1][1]=1.0/q/hxp()*ldJnx_dnM1N(d);
t[1][2]=1.0/q/hxp()*ldJnx_dvM1N(d);
t[2][0]=0.0;
t[2][1]=0.0;
t[2][2]=1.0/hxp()/hx();
return t;
)

matrix node::lsWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*s->ldJpy_dvM1N1(d);
t[1][2]=-1.0/q/hyp()*s->ldJny_dvM1N1(d);
return t;
}

matrix node::lnWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*ldJpy_dvM1N(d);
t[1][2]=1.0/q/hyp()*ldJny_dvM1N(d);
return t;
}

matrix node::wNWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hxp()*dJpx_dvM1N1(d);
t[1][2]=1.0/q/hxp()*dJnx_dvM1N1(d);
return t;
}

matrix node::nNWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*dJpy_dvM1N1(d);
t[1][2]=1.0/q/hyp()*dJny_dvM1N1(d);
return t;
}

matrix node::lwNWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hxp()*ldJpx_dvM1N1(d);
t[1][2]=1.0/q/hxp()*ldJnx_dvM1N1(d);
return t;
}

matrix node::lnNWt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=1.0/q/hyp()*ldJpy_dvM1N1(d);
t[1][2]=1.0/q/hyp()*ldJny_dvM1N1(d);
return t;
}

matrix node::eNEt(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hxp()*e->dJpx_dvM1N1(d);
t[1][2]=-1.0/q/hxp()*e->dJnx_dvM1N1(d);
return t;
}

```

```

matrix node::nNEt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hyp()*dJpy_dvMON1(d);
    t[1][2]=1.0/q/hyp()*dJny_dvMON1(d);
    return t;
}

matrix node::leNEt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hxp()*e->ldJpx_dvMN1(d);
    t[1][2]=-1.0/q/hxp()*e->ldJnx_dvMN1(d);
    return t;
}

matrix node::lnNEt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hyp()*ldJpy_dvMON1(d);
    t[1][2]=1.0/q/hyp()*ldJny_dvMON1(d);
    return t;
}

matrix node::wSWt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hxp()*dJpx_dvM1N0(d);
    t[1][2]=1.0/q/hxp()*dJnx_dvM1N0(d);
    return t;
}

matrix node::sSWt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hyp()*s->dJpy_dvM1N(d);
    t[1][2]=-1.0/q/hyp()*s->dJny_dvM1N(d);
    return t;
}

matrix node::lwSWt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=1.0/q/hxp()*ldJpx_dvM1N0(d);
    t[1][2]=1.0/q/hxp()*ldJnx_dvM1N0(d);
    return t;
}

matrix node::lsSWt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hyp()*s->ldJpy_dvM1N(d);
    t[1][2]=-1.0/q/hyp()*s->ldJny_dvM1N(d);
    return t;
}

matrix node::eSEt(gaussian_doping *d)
{
    matrix t;
    t=0.0;
    t[0][2]=-1.0/q/hxp()*e->dJpx_dvMN0(d);
    t[1][2]=-1.0/q/hxp()*e->dJnx_dvMN0(d);
    return t;
}

matrix node::sSEt(gaussian_doping *d)
{

```

```

matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*s->dJpy_dvMON(d);
t[1][2]=-1.0/q/hyp()*s->dJny_dvMON(d);
return t;
}

matrix node::leSet(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp()*e->ldJpx_dvMON(d);
t[1][2]=-1.0/q/hyp()*e->ldJnx_dvMON(d);
return t;
}

matrix node::lsSet(gaussian_doping *d)
{
matrix t;
t=0.0;
t[0][2]=-1.0/q/hyp() ->ldJpy_dvMON(d);
t[1][2]=-1.0/q/hyp()*s->ldJny_dvMON(d);
return t;
}

// the matrix and vector returned by SSt an. Rt are
// used to calculate the values at floating boundary nodes.
matrix node::SSt(unsigned int dir)
{
scalar E0,v1,p1,n1;
matrix t;
switch (dir) {
case north: v1=n->Y[volt];p1=n->Y[hole];n1=n->Y[elec]; break;
case south: v1=s->Y[volt];p1=s->Y[hole];n1=s->Y[elec]; break;
case east: v1=e->Y[volt];p1=e->Y[hole];n1=e->Y[elec]; break;
case west: v1=w->Y[volt];p1=w->Y[hole];n1=w->Y[elec]; break;
}
E0=exp(theta*(v1-Y[volt]));
t[0][0]=(E0*p1+n1)/E0/(p1+n1);
t[0][1]=-(1.0-E0)*p1/(p1+n1);
t[0][2]=0.0;
t[1][0]=(1.0-E0)*n1/E0/(p1+n1);
t[1][1]=(E0*p1+n1)/(p1+n1);
t[1][2]=0.0;
t[2][0]=(1.0-E0)/q*K*T/E0/(p1+n1);
t[2][1]=(1.0-E0)/q*K*T/(p1+n1);
t[2][2]=1.0;
return t;
}

vector node::Rt(unsigned int dir)
{
scalar v1,p1,n1,D1;
vector t;
switch (dir) {
case north: v1=n->Y[volt];p1=n->Y[hole];n1=n->Y[elec];D1=n->D; break;
case south: v1=s->Y[volt];p1=s->Y[hole];n1=s->Y[elec];D1=s->D; break;
case east: v1=e->Y[volt];p1=e->Y[hole];n1=e->Y[elec];D1=e->D; break;
case west: v1=w->Y[volt];p1=w->Y[hole];n1=w->Y[elec];D1=w->D; break;
}
t[0]=-p1/(p1+n1)*(D1-D);
t[1]=n1/(p1+n1)*(D1-D);
t[2]=v1-Y[volt]+(D1-D)/theta/(p1+n1);
return t;
}

// this calculates the 1D newton matrices for initial guesses
void node::calc_1D_newton_matrices(unsigned int dir,gaussian_doping *d)
{
scalar dJpyMN0dpMN,dJpyMN0dvMN,dJnyMN0dnMN,dJnyMN0dvMN;
scalar dJpyMNdpMN,dJpyMNdvMN,dJnyMNdnMN,dJnyMNdvMN;
scalar dJpxMN0dpMN,dJpxMN0dvMN,dJnxMN0dnMN,dJnxMN0dvMN;
scalar dJpxMNdpMN,dJpxMNdvMN,dJnxMNdnMN,dJnxMNdvMN;
scalar dJpxMN0dvMN,dJpyMN0dvMN0,dJnxMN0dvMN0,dJnyMN0dvMN0;
scalar ll,lh,h1,hh;
vector R;
matrix SS;

```

```

Ynp=Y;

if ((dir==north)|| (dir==south)) {
  if (By(elec,d,ll,lh,h1,hh)!=0.0) {
    N=nNt(d);
    dJpyMNdpmN=dJpy_dpMN(d);
    dJpyMNdvMN=dJpy_dvMN(d);
    dJnyMNdnMN=dJny_dnMN(d);
    dJnyMNdvMN=dJny_dvMN(d);
  } else {
    N=lnNt(d);
    dJpyMNdpmN=ldJpy_dpMN(d);
    dJpyMNdvMN=ldJpy_dvMN(d);
    dJnyMNdnMN=ldJny_dnMN(d);
    dJnyMNdvMN=ldJny_dvMN(d);
  }
  if (s->By(elec,d,ll,lh,h1,hh)!=0.0) {
    S=sSt(d);
    dJpyMN0dpmN=s->dJpy_dpMN1(d);
    dJpyMN0dvMN=s->dJpy_dvMN1(d);
    dJnyMN0dnMN=s->dJny_dnMN1(d);
    dJnyMN0dvMN=s->dJny_dvMN1(d);
  } else {
    S=lsSt(d);
    dJpyMN0dpmN=s->ldJpy_dpMN1(d);
    dJpyMN0dvMN=s->ldJpy_dvMN1(d);
    dJnyMN0dnMN=s->ldJny_dnMN1(d);
    dJnyMN0dvMN=s->ldJny_dvMN1(d);
  }
  dJpyMN0dvMN0=-dJpyMN0dvMN;
  dJnyMN0dvMN0=-dJnyMN0dvMN;
  C[0][0]=+1.0/q/hyp()*(dJpyMNdpmN-dJpyMN0dpmN)
  +dU_dpMN();
  C[0][1]= dU_dnMN();
  C[0][2]=+1.0/q/hyp()*(dJpyMNdvMN-dJpyMN0dvMN);
  C[1][0]=-dU_dpMN();
  C[1][1]=+1.0/q/hyp()*(dJnyMNdnMN-dJnyMN0dnMN)
  -dU_dnMN();
  C[1][2]=+1.0/q/hyp()*(dJnyMNdvMN-dJnyMN0dvMN);
  C[2][0]= q/es;
  C[2][1]=-q/es;
  C[2][2]=-1.0/hyp()*(1.0/hy()+1.0/s->hy());
  F[0]=-U()+dU_dpMN()*Y[hole]+dU_dnMN()*Y[elec]
  -1.0/q/hyp()*dJpyMN0dvMN0*(s->Y[volt]-Y[volt])
  +1.0/q/hyp()*dJpyMNdvMN*(Y[volt]-n->Y[volt]);
  F[1]= U()+-dU_dpMN()*Y[hole]-dU_dnMN()*Y[elec]
  -1.0/q/hyp()*dJnyMN0dvMN0*(s->Y[volt]-Y[volt])
  +1.0/q/hyp()*dJnyMNdvMN*(Y[volt]-n->Y[volt]);
  F[2]=-q/es*D;
  if (n->flag&floating_boundary) {
    SS=SSt(north);
    R=Rt(north);
    n->C=SS;
    n->F=R;
    C+=N*SS;
    F-=N*R;
    N=0.0;
  }
  if (s->flag&floating_boundary) {
    SS=SSt(south);
    R=Rt(south);
    s->C=SS;
    s->F=R;
    C+=S*SS;
    F-=S*R;
    S=0.0;
  }
} else { // dir==east or dir==west
  if (Bx(elec,d,ll,lh,h1,hh)!=0.0) {
    W=wWt(d);
    dJpxMNdpmN=dJpx_dpMN(d);
    dJpxMNdvMN=dJpx_dvMN(d);
    dJnxMNdnMN=dJnx_dnMN(d);
    dJnxMNdvMN=dJnx_dvMN(d);
  }
}

```



```

    ) else (
        w=1wnt(d);
        dJpxMNdpmN=1dJpx_dpMN(d);
        dJpxMNdvmN=1dJpx_dvmN(d);
        dJnxMNdnmN=1dJnx_dnmN(d);
        dJnxMNdvmN=1dJnx_dvmN(d);
    )
    if (e->Bx(elec,d,ll,1h,h1,hh)!=0.0) {
        E=eEt(d);
        dJpxMNdpmN=e->dJpx_dpM1N(d);
        dJpxMNdvmN=e->dJpx_dvm1N(d);
        dJnxMNdnmN=e->dJnx_dnm1N(d);
        dJnxMNdvmN=e->dJnx_dvm1N(d);
    } else (
        E=1eEt(d);
        dJpxMNdpmN=e->1dJpx_dpM1N(d);
        dJpxMNdvmN=e->1dJpx_dvm1N(d);
        dJnxMNdnmN=e->1dJnx_dnm1N(d);
        dJnxMNdvmN=e->1dJnx_dvm1N(d);
    )
    dJpxMNdvmN=-dJpxMNdvmN;
    dJnxMNdvmN=-dJnxMNdvmN;
    C[0][0]= 1.0/q/hxp()* (dJpxMNdpmN-dJpxMNdpmN)
    +dU_dpMN();
    C[0][1]= dU_dnmN();
    C[0][2]= 1.0/q/hxp()* (dJpxMNdvmN-dJpxMNdvmN);
    C[1][0]=-dU_dpMN();
    C[1][1]= 1.0/q/hxp()* (dJnxMNdnmN-dJnxMNdnmN)
    -dU_dnmN();
    C[1][2]= 1.0/q/hxp()* (dJnxMNdvmN-dJnxMNdvmN);
    C[2][0]= q/es;
    C[2][1]=-q/es;
    C[2][2]=-1.0/hxp()* (1.0/hx()+1.0/e->hx());
    F[0]=-U()+dU_dpMN()*Y[hole]+dU_dnmN()*Y[elec]
    -1.0/q/hxp()*dJpxMNdvmN*(e->Y[volt]-Y[volt]);
    +1.0/q/hxp()*dJpxMNdvmN*(Y[volt]-w->Y[volt]);
    F[1]= U()-dU_dpMN()*Y[hole]-dU_dnmN()*Y[elec]
    -1.0/q/hxp()*dJnxMNdvmN*(e->Y[volt]-Y[volt]);
    +1.0/q/hxp()*dJnxMNdvmN*(Y[volt]-w->Y[volt]);
    F[2]=-q/es*D;
    if (e->flag&floating_boundary) {
        SS=SSt(east);
        R=Rt(east);
        e->C=SS;
        e->F=R;
        C+=E*SS;
        F+=E*R;
        E=0.0;
    }
    if (w->flag&floating_boundary) {
        SS=SSt(west);
        R=Rt(west);
        w->C=SS;
        w->F=R;
        C+=W*SS;
        F+=W*R;
        W=0.0;
    }
}
}
// this calculates the 2D newton matrices. It calls the respective
// normal or limit functions depending on adjacent potentials
void node::calc_newton_matrices(gaussian_doping *d,int initial_value)
{
    scalar dJpyMN0dpmN,dJpyMN0dvmN,dJnyMN0dnmN,dJnyMN0dvmN;
    scalar dJpyMNdpMN,dJpyMNdvMN,dJnyMNdnmN,dJnyMNdvMN;
    scalar dJpxMN0dpmN,dJpxMN0dvmN,dJnxMN0dnmN,dJnxMN0dvmN;
    scalar dJpxMNdpMN,dJpxMNdvMN,dJnxMNdnmN,dJnxMNdvMN;
    scalar dJpxMN0dvMON,dJpyMN0dvMNO,dJnxMN0dvMON,dJnyMN0dvMNO;
    scalar dJpxMNdvM1N1,dJpxMNdvM1N0,dJpxMNdvM1,dJpxMNdvM0;
    scalar dJpxMN0dvM0N1,dJpxMN0dvM0N0,dJpxMN0dvM0N1,dJpxMN0dvM0N0;
    scalar dJpyMN0dvM0N1,dJpyMN0dvM0N0,dJpyMN0dvM0N1,dJpyMN0dvM0N0;
    scalar dJpyMN0dvM1N,dJpyMN0dvM0N,dJpyMN0dvM1N0,dJpyMN0dvM0N0;
    scalar dJnxMN0dvM1N1,dJnxMN0dvM1N0,dJnxMN0dvM1,dJnxMN0dvM0;
    scalar dJnxMN0dvM0N1,dJnxMN0dvM0N0,dJnxMN0dvM0N1,dJnxMN0dvM0N0;
}

```

```

scalar dJnyMNdvmON1,dJnyMNdvm1N1,dJnyMNdvmON,dJnyMNdvm1N;
scalar dJnyMN0dvM1N,dJnyMN0dvMON,dJnyMN0dvM1N0,dJnyMN0dvMON0;
scalar ll, lh, hl, hh;
vector R;
matrix SS;

```

```
Ynp=Y;
```

```

N=S=E=W=NW=NE=SW=SE=0.0;
if (By{elec,d,ll,lh,hl,hh}!=0.0) {
  N+=nNt(d);
  E+=nEt(d);
  W+=nWt(d);
  NW+=nNWt(d);
  NE+=nNEt(d);
  dJpyMNdpMN=dJpy_dpMN(d);
  dJpyMNdvMN=dJpy_dvMN(d);
  dJnyMNdnMN=dJny_dnMN(d);
  dJnyMNdvmN=dJny_dvMN(d);
  dJpyMNdvmON1=dJpy_dvMON1(d);
  dJpyMNdvm1N1=dJpy_dvM1N1(d);
  dJpyMNdvmON=dJpy_dvMON(d);
  dJpyMNdvm1N=dJpy_dvM1N(d);
  dJnyMNdvmON1=dJny_dvMON1(d);
  dJnyMNdvm1N1=dJny_dvM1N1(d);
  dJnyMNdvmON=dJny_dvMON(d);
  dJnyMNdvm1N=dJny_dvM1N(d);
} else {
  N+=lnNt(d);
  E+=lnEt(d);
  W+=lnWt(d);
  NW+=lnNWt(d);
  NE+=lnNEt(d);
  dJpyMNdpMN=ldJpy_dpMN(d);
  dJpyMNdvMN=ldJpy_dvMN(d);
  dJnyMNdnMN=ldJny_dnMN(d);
  dJnyMNdvmN=ldJny_dvMN(d);
  dJpyMNdvmON1=ldJpy_dvMON1(d);
  dJpyMNdvm1N1=ldJpy_dvM1N1(d);
  dJpyMNdvmON=ldJpy_dvMON(d);
  dJpyMNdvm1N=ldJpy_dvM1N(d);
  dJnyMNdvmON1=ldJny_dvMON1(d);
  dJnyMNdvm1N1=ldJny_dvM1N1(d);
  dJnyMNdvmON=ldJny_dvMON(d);
  dJnyMNdvm1N=ldJny_dvM1N(d);
}
if (s->By{elec,d,ll,lh,hl,hh,'!=0.0'}) {
  S+=sSt(d);
  E+=sEt(d);
  W+=sWt(d);
  SW+=sSWt(d);
  SE+=sSEt(d);
  dJpyMN0dpMN=s->dJpy_dpMN1(d);
  dJpyMN0dvMN=s->dJpy_dvMN1(d);
  dJnyMN0dnMN=s->dJny_dnMN1(d);
  dJnyMN0dvMN=s->dJny_dvMN1(d);
  dJpyMN0dvMON=s->dJpy_dvMON1(d);
  dJpyMN0dvM1N=s->dJpy_dvM1N1(d);
  dJpyMN0dvMON0=s->dJpy_dvMON(d);
  dJpyMN0dvM1N0=s->dJpy_dvM1N(d);
  dJnyMN0dvMON=s->dJny_dvMON1(d);
  dJnyMN0dvM1N=s->dJny_dvM1N1(d);
  dJnyMN0dvMON0=s->dJny_dvMON(d);
  dJnyMN0dvM1N0=s->dJny_dvM1N(d);
} else {
  S+=lsSt(d);
  E+=lsEt(d);
  W+=lsWt(d);
  SW+=lsSWt(d);
  SE+=lsSEt(d);
  dJpyMN0dpMN=s->ldJpy_dpMN1(d);
  dJpyMN0dvMN=s->ldJpy_dvMN1(d);
  dJnyMN0dnMN=s->ldJny_dnMN1(d);
  dJnyMN0dvMN=s->ldJny_dvMN1(d);
  dJpyMN0dvMON=s->ldJpy_dvMON1(d);

```

```

dJpyMNOdvM1N=s->ldJpy_dvM1N1(d);
dJpyMNOdvM0N0=s->ldJpy_dvM0N(d);
dJpyMNOdvM1N0=s->ldJpy_dvM1N(d);
dJnyMNOdvM0N=s->ldJny_dvM0N1(d);
dJnyMNOdvM1N=s->ldJny_dvM1N1(d);
dJnyMNOdvM0N0=s->ldJny_dvM0N(d);
dJnyMNOdvM1N0=s->ldJny_dvM1N(d);
)
if (Bx(elec,d,ll,h,h1,hh)!=0.0) {
W+=wWt(d);
N+=wNt(d);
S+=wSt(d);
NW+=wNWt(d);
SW+=wSWt(d);
dJpxMNDpMN=dJpx_dpMN(d);
dJpxMNdvmN=dJpx_dvMN(d);
dJnxMNdnmN=dJnx_dnmN(d);
dJnxMNdvmN=dJnx_dvMN(d);
dJpxMNdvm1N1=dJpx_dvM1N1(d);
dJpxMNdvm1N0=dJpx_dvM1N0(d);
dJpxMNdvmN1=dJpx_dvMN1(d);
dJpxMNdvmN0=dJpx_dvMN0(d);
dJnxMNdvm1N1=dJnx_dvM1N1(d);
dJnxMNdvm1N0=dJnx_dvM1N0(d);
dJnxMNdvmN1=dJnx_dvMN1(d);
dJnxMNdvmN0=dJnx_dvMN0(d);
) else {
W+=lwWt(d);
N+=lwNt(d);
S+=lwSt(d);
NW+=lwNWt(d);
SW+=lwSWt(d);
dJpxMNDpMN=ldJpx_dpMN(d);
dJpxMNdvmN=ldJpx_dvMN(d);
dJnxMNdnmN=ldJnx_dnmN(d);
dJnxMNdvmN=ldJnx_dvMN(d);
dJpxMNdvm1N1=ldJpx_dvM1N1(d);
dJpxMNdvm1N0=ldJpx_dvM1N0(d);
dJpxMNdvmN1=ldJpx_dvMN1(d);
dJpxMNdvmN0=ldJpx_dvMN0(d);
dJnxMNdvm1N1=ldJnx_dvM1N1(d);
dJnxMNdvm1N0=ldJnx_dvM1N0(d);
dJnxMNdvmN1=ldJnx_dvMN1(d);
dJnxMNdvmN0=ldJnx_dvMN0(d);
)
if (e->Bx(elec,d,ll,h,h1,hh)!=0.0) {
E+=eEt(d);
N+=eNt(d);
S+=eSt(d);
NE+=eNEt(d);
SE+=eSEt(d);
dJpxMNDpMN=e->dJpx_dpMN(d);
dJpxMNdvmN=e->dJpx_dvMN(d);
dJnxMNdnmN=e->dJnx_dnmN(d);
dJnxMNdvmN=e->dJnx_dvMN(d);
dJpxMNdvmN1=e->dJpx_dvMN1(d);
dJpxMNdvmN0=e->dJpx_dvMN0(d);
dJpxMNdvmN1=e->dJpx_dvMN1(d);
dJpxMNdvmN0=e->dJpx_dvMN0(d);
dJnxMNdvmN1=e->dJnx_dvMN1(d);
dJnxMNdvmN0=e->dJnx_dvMN0(d);
dJnxMNdvmN1=e->dJnx_dvMN1(d);
dJnxMNdvmN0=e->dJnx_dvMN0(d);
) else {
E+=leEt(d);
N+=leNt(d);
S+=leSt(d);
NE+=leNEt(d);
SE+=leSEt(d);
dJpxMNDpMN=e->ldJpx_dpMN(d);
dJpxMNdvmN=e->ldJpx_dvMN(d);
dJnxMNdnmN=e->ldJnx_dnmN(d);
dJnxMNdvmN=e->ldJnx_dvMN(d);
dJpxMNdvmN1=e->ldJpx_dvMN1(d);
dJpxMNdvmN0=e->ldJpx_dvMN0(d);

```

```

dJpxMONdvMON1=e->ldJpx_dvMN1(d);
dJpxMONdvMON0=e->ldJpx_dvMN0(d);
dJnxMONdvMN1=e->ldJnx_dvMN1(d);
dJnxMONdvMN0=e->ldJnx_dvMN0(d);
dJnyMONdvMON1=e->ldJny_dvMN1(d);
dJnyMONdvMON0=e->ldJny_dvMN0(d);
)
dJpxMONdvMON=-dJpxMONdvMN;
dJpyMONdvMON0=-dJpyMONdvMN;
dJnxMONdvMON=-dJnxMONdvMN;
dJnyMONdvMON0=-dJnyMONdvMN;
C[0][0]= 1.0/q/hxp()* (dJpxMONdpMN-dJpxMONdpMN)
+1.0/q/hyp()* (dJpyMONdpMN-dJpyMONdpMN)
+du_dpMN();
C[0][1]= du_dnMN();
C[0][2]= 1.0/q/lxp()* (dJpxMONdvMN-dJpxMONdvMN)
+1.0/q/hyp()* (dJpyMONdvMN-dJpyMONdvMN);
C[1][0]=-du_dpMN();
C[1][1]= 1.0/q/hxp()* (dJnxMONdnMN-dJnxMONdnMN)
+1.0/q/hyp()* (dJnyMONdnMN-dJnyMONdnMN)
-du_dnMN();
C[1][2]= 1.0/q/hxp()* (dJnxMONdvMN-dJnxMONdvMN)
+1.0/q/hyp()* (dJnyMONdvMN-dJnyMONdvMN);
C[2][0]= q/es;
C[2][1]=-q/es;
C[2][2]=-1.0/hxp()* (1.0/hx()+1.0/e->hx())
-1.0/hyp()* (1.0/hy()+1.0/s->hy());
F[0]=-U()+du_dpMN()*Y[hole]+du_dnMN()*Y[elec]
-1.0/q/hxp()*dJpxMONdvMON*(e->Y[volt]-Y[volt])
+1.0/q/hxp()*dJpxMONdvMN*(Y[volt]-w->Y[volt])
-1.0/q/hyp()*dJpyMONdvMON0*(s->Y[volt]-Y[volt])
+1.0/q/hyp()*dJpyMONdvMN*(Y[volt]-n->Y[volt]);
F[1]= U()-du_dpMN()*Y[hole]-du_dnMN()*Y[elec]
-1.0/q/hxp()*dJnxMONdvMON*(e->Y[volt]-Y[volt])
+1.0/q/hxp()*dJnxMONdvMN*(Y[volt]-w->Y[volt])
-1.0/q/hyp()*dJnyMONdvMON0*(s->Y[volt]-Y[volt])
+1.0/q/hyp()*dJnyMONdvMN*(Y[volt]-n->Y[volt]);
F[2]=-q/es*D;

F[0]+= 1.0/q/hxp()*dJpxMONdvMN1*n->w->Y[volt]
+1.0/q/hxp()*dJpxMONdvMN0*s->w->Y[volt]
+1.0/q/hxp()*dJpxMONdvMN1*n->Y[volt]
+1.0/q/hxp()*dJpxMONdvMN0*s->Y[volt]
-1.0/q/hxp()*dJpxMONdvMN1*n->Y[volt]
-1.0/q/hxp()*dJpxMONdvMN0*s->Y[volt]
-1.0/q/hxp()*dJpxMONdvMON1*n->e->Y[volt]
-1.0/q/hxp()*dJpxMONdvMON0*s->e->Y[volt]
+1.0/q/hyp()*dJpyMONdvMON1*n->e->Y[volt]
+1.0/q/hyp()*dJpyMONdvMN1*n->w->Y[volt]
+1.0/q/hyp()*dJpyMONdvMON*e->Y[volt]
+1.0/q/hyp()*dJpyMONdvMN1*w->Y[volt]
-1.0/q/hyp()*dJpyMONdvMON*e->Y[volt]
-1.0/q/hyp()*dJpyMONdvMN1*w->Y[volt]
-1.0/q/hyp()*dJpyMONdvMON0*s->e->Y[volt]
-1.0/q/hyp()*dJpyMONdvMN0*s->w->Y[volt];
F[1]+= 1.0/q/hxp()*dJnxMONdvMN1*n->w->Y[volt]
+1.0/q/hxp()*dJnxMONdvMN0*s->w->Y[volt]
+1.0/q/hxp()*dJnxMONdvMN1*n->Y[volt]
+1.0/q/hxp()*dJnxMONdvMN0*s->Y[volt]
-1.0/q/hxp()*dJnxMONdvMN1*n->Y[volt]
-1.0/q/hxp()*dJnxMONdvMN0*s->Y[volt]
-1.0/q/hxp()*dJnxMONdvMON1*n->e->Y[volt]
-1.0/q/hxp()*dJnxMONdvMON0*s->e->Y[volt]
+1.0/q/hyp()*dJnyMONdvMON1*n->e->Y[volt]
+1.0/q/hyp()*dJnyMONdvMN1*n->w->Y[volt]
+1.0/q/hyp()*dJnyMONdvMON*e->Y[volt]
+1.0/q/hyp()*dJnyMONdvMN1*w->Y[volt]
-1.0/q/hyp()*dJnyMONdvMON*e->Y[volt]
-1.0/q/hyp()*dJnyMONdvMN1*w->Y[volt]
-1.0/q/hyp()*dJnyMONdvMON0*s->e->Y[volt]
-1.0/q/hyp()*dJnyMONdvMN0*s->w->Y[volt];

if ((n->flag&floating_boundary)&&(!n->inside_corner)) {
    if (!initial_value) {
        if (e->flag&floating_boundary) {

```

```

        ll=fabs(n->e->Y[volt]-e->Y[volt]);
        lh=fabs(n->e->Y[volt]-n->Y[volt]);
        if (ll==lh) ll=lh=1.0;
        hh=ll+lh;ll/=hh;lh/=hh;
        N+=NE*ll;
        E+=NE*lh;
        NE=0.0;
    } else if (!(e->flag&contacted_boundary)) {
        E+=NE*SS;
        F-=NE*R;
        NE=0.0;
    }
    if (w->flag&floating_boundary) {
        ll=fabs(n->w->Y[volt]-w->Y[volt]);
        lh=fabs(n->w->Y[volt]-n->Y[volt]);
        if (ll==lh) ll=lh=1.0;
        hh=ll+lh;ll/=hh;lh/=hh;
        N+=NW*ll;
        W+=NW*lh;
        NW=0.0;
    } else if (!(w->flag&contacted_boundary)) {
        W+=NW*SS;
        F-=NW*R;
        NW=0.0;
    }
    )
    SS=SSt(north);
    R=Rt(north);
    n->C=SS;
    n->F=R;
    if (!initial_value) {
        C+=N*SS;
        F-=N*R;
        N=0.0;
    }
    )
    if ((s->flag&floating_boundary)&&!(s->inside_corner())) {
        if (!initial_value) {
            if (e->flag&floating_boundary) {
                ll=fabs(s->e->Y[volt]-e->Y[volt]);
                lh=fabs(s->e->Y[volt]-s->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                S+=SE*ll;
                E+=SE*lh;
                SE=0.0;
            } else if (!(e->flag&contacted_boundary)) {
                E+=SE*SS;
                F-=SE*R;
                SE=0.0;
            }
            if (w->flag&floating_boundary) {
                ll=fabs(s->w->Y[volt]-w->Y[volt]);
                lh=fabs(s->w->Y[volt]-s->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                S+=SW*ll;
                W+=SW*lh;
                SW=0.0;
            } else if (!(w->flag&contacted_boundary)) {
                W+=SW*SS;
                F-=SW*R;
                SW=0.0;
            }
        }
        SS=SSt(south);
        R=Rt(south);
        s->C=SS;
        s->F=R;
        if (!initial_value) {
            C+=S*SS;
            F-=S*R;
            S=0.0;
        }
    }
    )
}

```

```

    if ((e->flag&floating_boundary)&&(!e->inside_corner())) {
        if (!initial_value) {
            if (n->flag&floating_boundary) {
                ll=fabs(n->e->Y[volt]-e->Y[volt]);
                lh=fabs(n->e->Y[volt]-n->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                N+=NE*ll;
                E+=NE*lh;
                NE=0.0;
            } else if (!(n->flag&contacted_boundary)) {
                N+=NE*SS;
                F-=NE*R;
                NE=0.0;
            }
            if (s->flag&floating_boundary) {
                ll=fabs(s->e->Y[volt]-e->Y[volt]);
                lh=fabs(s->e->Y[volt]-s->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                S+=SE*ll;
                E+=SE*lh;
                SE=0.0;
            } else if (!(s->flag&contacted_boundary)) {
                S+=SE*SS;
                F-=SE*R;
                SE=0.0;
            }
        }
        SS=SSt(east);
        R=Rt(east);
        e->C=SS;
        e->F=R;
        if (!initial_value) {
            C+=E*SS;
            F-=E*R;
            E=0.0;
        }
    }
    if ((w->flag&floating_boundary)&&(!w->inside_corner())) {
        if (!initial_value) {
            if (n->flag&floating_boundary) {
                ll=fabs(n->w->Y[volt]-w->Y[volt]);
                lh=fabs(n->w->Y[volt]-n->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                N+=NW*ll;
                W+=NW*lh;
                NW=0.0;
            } else if (!(n->flag&contacted_boundary)) {
                N+=NW*SS;
                F-=NW*R;
                NW=0.0;
            }
            if (s->flag&floating_boundary) {
                ll=fabs(s->w->Y[volt]-w->Y[volt]);
                lh=fabs(s->w->Y[volt]-s->Y[volt]);
                if (ll==lh) ll=lh=1.0;
                hh=ll+lh;ll/=hh;lh/=hh;
                S+=SW*ll;
                W+=SW*lh;
                SW=0.0;
            } else if (!(s->flag&contacted_boundary)) {
                S+=SW*SS;
                F-=SW*R;
                SW=0.0;
            }
        }
        SS=SSt(west);
        R=Rt(west);
        w->C=SS;
        w->F=R;
        if (!initial_value) {
            C+=W*SS;
            F-=W*R;
        }
    }

```

```

        W=0.0;
    }
}

```

10.13 Node1.cpp

```

// Copyright 1995, Brian Gieschen
#include <math.h>
#include "node.h"
#ifdef NULL
#define NULL 0x0
#endif
// this calculates the slor doing elimination in the forward
// direction.
void node::calc_slor(unsigned int dir)
{
    Ft=F;
    Ct=C;
    if (dir&north_travel) {
        if (dir&east_include) Ft-=E*(e->Y)+NE*(n->e->Y)+SE*(s->e->Y);
        if (dir&west_include) Ft-=W*(w->Y)+NW*(n->w->Y)+SW*(s->w->Y);
        if (dir&end_of_travel) Ft-=N*(n->Y);
        if (!(dir&start_of_travel)) {
            Ct-=S*{(s->Ct).inv()*{(s->N)}};
            Ft-=S*{(s->Ct).inv()*{(s->Ft)}};
        } else Ft-=S*(s->Y);
    }
    else if (dir&south_travel) {
        if (dir&east_include) Ft-=E*(e->Y)+NE*(n->e->Y)+SE*(s->e->Y);
        if (dir&west_include) Ft-=W*(w->Y)+NW*(n->w->Y)+SW*(s->w->Y);
        if (dir&end_of_travel) Ft-=S*(s->Y);
        if (!(dir&start_of_travel)) {
            Ct-=N*{(n->Ct).inv()*{(n->S)}};
            Ft-=N*{(n->Ct).inv()*{(n->Ft)}};
        } else Ft-=N*(n->Y);
    }
    else if (dir&east_travel) {
        if (dir&north_include) Ft-=N*(n->Y)+NE*(n->e->Y)+NW*(n->w->Y);
        if (dir&south_include) Ft-=S*(s->Y)+SE*(s->e->Y)+SW*(s->w->Y);
        if (dir&end_of_travel) Ft-=E*(e->Y);
        if (!(dir&start_of_travel)) {
            Ct-=W*{(w->Ct).inv()*{(w->E)}};
            Ft-=W*{(w->Ct).inv()*{(w->Ft)}};
        } else Ft-=W*(w->Y);
    }
    else if (dir&west_travel) {
        if (dir&north_include) Ft-=N*(n->Y)+NE*(n->e->Y)+NW*(n->w->Y);
        if (dir&south_include) Ft-=S*(s->Y)+SE*(s->e->Y)+SW*(s->w->Y);
        if (dir&end_of_travel) Ft-=W*(w->Y);
        if (!(dir&start_of_travel)) {
            Ct-=E*{(e->Ct).inv()*{(e->W)}};
            Ft-=E*{(e->Ct).inv()*{(e->Ft)}};
        } else Ft-=E*(e->Y);
    }
}
// this does the back substitution for the forward SLOR.
void node::back_sub(unsigned int dir)
{
    if (!(dir&start_of_travel)) {
        if (dir&north_travel) Ft-=S*(s->Y);
        else if (dir&south_travel) Ft-=N*(n->Y);
        else if (dir&east_travel) Ft-=W*(w->Y);
        else if (dir&west_travel) Ft-=E*(e->Y);
    }
    Yip=Y;
    Y=Ct.inv()*Ft;
}
// this calculates the updated boundary values, it also takes into account
// special cases such as inside corners and outside corners.
void node::calc_boundary(unsigned int isstart)
{
    scalar l,h;
    if (flag&floating_boundary) {
        Yip=Y;
    }
}

```

```

// north edge
if ((!(n)&&(s)&&(e)&&(w))) {
    Y=C*(s->Y)+F;
// south edge
} else if ((!(n)&&(s)&&(e)&&(w))) {
    Y=C*(n->Y)+F;
// east edge
} else if ((!(n)&&(s)&&(e)&&(w))) {
    Y=C*(w->Y)+F;
// west edge
} else if ((!(n)&&(s)&&(e)&&(w))) {
    Y=C*(e->Y)+F;
// north east corner (outer)
} else if ((!(n)&&(s)&&(e)&&(w))) {
    if (isstart) {
l=fabs(Yip[volt]-w->Y[volt]);h=fabs(Yip[volt]-s->Y[volt]);if (l==h) l=h=1.0;
Y=((s->C*s->w->Y+s->F)*l+(w->C*w->s->Y+w->F)*h)/(l+h);
    } else {
l=fabs(Yip[volt]-e->Y[volt]);h=fabs(Yip[volt]-s->Yip[volt]);if (l==h) l=h=1.0;
Y=(s->Y*l+(e->C*e->s->Y+e->F)*h)/(l+h);
    }
// north west corner (outer)
} else if ((!(n)&&(s)&&(e)&&(w))) {
    if (isstart) {
l=fabs(Yip[volt]-s->Y[volt]);h=fabs(Yip[volt]-e->Y[volt]);if (l==h) l=h=1.0;
Y=((e->C*e->s->Y+e->F)*l+(s->C*s->e->Y+s->F)*h)/(l+h);
    } else {
l=fabs(Yip[volt]-s->Y[volt]);h=fabs(Yip[volt]-e->Yip[volt]);if (l==h) l=h=1.0;
Y=(e->Y*l+(s->C*s->e->Y+s->F)*h)/(l+h);
    }
// south east corner (outer)
} else if ((!(n)&&(s)&&(e)&&(w))) {
    if (isstart) {
l=fabs(Yip[volt]-n->Y[volt]);h=fabs(Yip[volt]-w->Y[volt]);if (l==h) l=h=1.0;
Y=((w->C*w->n->Y+w->F)*l+(n->C*n->w->Y+n->F)*h)/(l+h);
    } else {
l=fabs(Yip[volt]-n->Y[volt]);h=fabs(Yip[volt]-w->Yip[volt]);if (l==h) l=h=1.0;
Y=(w->Y*l+(n->C*n->w->Y+n->F)*h)/(l+h);
    }
// south west corner (outer)
} else if ((!(n)&&(s)&&(e)&&(w))) {
    if (isstart) {
l=fabs(Yip[volt]-e->Y[volt]);h=fabs(Yip[volt]-n->Y[volt]);if (l==h) l=h=1.0;
Y=((n->C*n->e->Y+n->F)*l+(e->C*e->n->Y+e->F)*h)/(l+h);
    } else {
l=fabs(Yip[volt]-e->Y[volt]);h=fabs(Yip[volt]-n->Yip[volt]);if (l==h) l=h=1.0;
Y=(n->Y*l+(e->C*e->n->Y+e->F)*h)/(l+h);
    }
// inner corner
} else if ((!(n)&&(s)&&(e)&&(w))) {
// north east
    if ((!(n->is_inside())&&(e->is_inside()))) {
l=fabs(Yip[volt]-w->Yip[volt]);h=fabs(Yip[volt]-s->Yip[volt]);if (l==h) l=h=1.0;
Y=(s->Y*l+w->Y*h)/(l+h);
// north west
    } else if ((!(n->is_inside())&&(w->is_inside()))) {
l=fabs(Yip[volt]-e->Yip[volt]);h=fabs(Yip[volt]-s->Yip[volt]);if (l==h) l=h=1.0;
Y=(s->Y*l+e->Y*h)/(l+h);
// south east
    } else if ((!(s->is_inside())&&(e->is_inside()))) {
l=fabs(Yip[volt]-w->Yip[volt]);h=fabs(Yip[volt]-n->Yip[volt]);if (l==h) l=h=1.0;
Y=(n->Y*l+w->Y*h)/(l+h);
// south west
    } else if ((!(s->is_inside())&&(w->is_inside()))) {
l=fabs(Yip[volt]-e->Yip[volt]);h=fabs(Yip[volt]-n->Yip[volt]);if (l==h) l=h=1.0;
Y=(n->Y*l+e->Y*h)/(l+h);
    }
}
}
}
// this used to do the corner calculations, I just keep it for posterity
// Do NOT use this function.
void node::calc_corner()
{
    scalar l,h;

```



```

Yip=Y;
if (flag&floating_boundary) {
    if ((n==NULL)&&(e==NULL)&&
        (s->flag&floating_boundary)&&(w->flag&floating_boundary)) {
        l=fabs(Yip[volt]-w->Yip[volt]);
        h=fabs(Yip[volt]-s->Yip[volt]);
        Y=(s->Y*l+w->Y*h)/(l+h);
    } else if ((n==NULL)&&(w==NULL)&&
        (s->flag&floating_boundary)&&(e->flag&floating_boundary)) {
        l=fabs(Yip[volt]-e->Yip[volt]);
        h=fabs(Yip[volt]-s->Yip[volt]);
        Y=(s->Y*l+e->Y*h)/(l+h);
    } else if ((s==NULL)&&(e==NULL)&&
        (n->flag&floating_boundary)&&(w->flag&floating_boundary)) {
        l=fabs(Yip[volt]-w->Yip[volt]);
        h=fabs(Yip[volt]-n->Yip[volt]);
        Y=(n->Y*l+w->Y*h)/(l+h);
    } else if ((s==NULL)&&(w==NULL)&&
        (n->flag&floating_boundary)&&(e->flag&floating_boundary)) {
        l=fabs(Yip[volt]-e->Yip[volt]);
        h=fabs(Yip[volt]-n->Yip[volt]);
        Y=(n->Y*l+e->Y*h)/(l+h);
    }
}
}
// this is a hook for adding simulation end detection.
void node::update_newton()
{
    // this a test for slor validity
    void node::slor_test_setup(unsigned int f)
    // 0x01 - north, 0x02 - south
    // 0x04 - east, 0x08 - west
    // 0x10 - C, 0x20 - F
    // 0x40 - Y(1.5,.75,1.5), 0x80 - Y(.75,0,.75)
    {
        N=S=E=W=C=0.0; Y=F=0.0;
        if ((f&0x01)>0) {
            N[0][0]=2.0;N[0][1]=1.0;N[0][2]=0.0;
            N[1][0]=1.0;N[1][1]=2.0;N[1][2]=1.0;
            N[2][0]=0.0;N[2][1]=1.0;N[2][2]=2.0;
        }
        if ((f&0x02)>0) {
            S[0][0]=2.0;S[0][1]=1.0;S[0][2]=0.0;
            S[1][0]=1.0;S[1][1]=2.0;S[1][2]=1.0;
            S[2][0]=0.0;S[2][1]=1.0;S[2][2]=2.0;
        }
        if ((f&0x04)>0) {
            E[0][0]=2.0;E[0][1]=1.0;E[0][2]=0.0;
            E[1][0]=1.0;E[1][1]=2.0;E[1][2]=1.0;
            E[2][0]=0.0;E[2][1]=1.0;E[2][2]=2.0;
        }
        if ((f&0x08)>0) {
            W[0][0]=2.0;W[0][1]=1.0;W[0][2]=0.0;
            W[1][0]=1.0;W[1][1]=2.0;W[1][2]=1.0;
            W[2][0]=0.0;W[2][1]=1.0;W[2][2]=2.0;
        }
        if ((f&0x10)>0) {
            C[0][0]=4.0;C[0][1]=1.0;C[0][2]=0.0;
            C[1][0]=1.0;C[1][1]=4.0;C[1][2]=1.0;
            C[2][0]=0.0;C[2][1]=1.0;C[2][2]=4.0;
        }
        if ((f&0x20)>0) {
            F[0]=12.0;F[1]=12.0;F[2]=12.0;
        }
        if ((f&0x40)>0) {
            Y[0]=1.5;Y[1]=0.75;Y[2]=1.5;
        }
        else if ((f&0x80)>0) {
            Y[0]=0.75;Y[1]=0.0;Y[2]=0.75;
        }
    }
}

```

10.14 Fileio.cpp

```

// Copyright 1995, Brian Gieschen
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include "vector.h"
#include "matrix.h"
#include "gauss.h"
#include "node.h"
#include "region.h"
#include "thesis1.h"
// these functions use Borland C++ v3 streaming classes.
// most C++ have very similar io-streaming classes for handling
// file, stdin, stdout
// these functions store their information in text files
// so the output is human readable and modifyable, but be
// careful when modifying any sim file.

// this is the write operator for a vector to a stream
ostream& operator <<(ostream& s,vector& v)
{
    s<<"("<<setprecision(20)<<v[0]<<" "<<setprecision(20)<<v[1];
    s<<" "<<setprecision(20)<<v[2]<<")";
    return s;
}
// this is the read operator for a vector from a stream
istream& operator >>(istream& s,vector& v)
{
    char c;
    s>>c>>v[0]>>v[1]>>v[2]>>c;
    return s;
}
// this is the write operator for a matrix, notice it uses
// the vector operator to write each row
ostream& operator <<(ostream& s,matrix& m)
{
    s<<"("<<m[0]<<" "<<m[1]<<" "<<m[2]<<")";
    return s;
}
// this is the read operator for a matrix
istream& operator >>(istream& s,matrix& m)
{
    char c;
    s>>c>>m[0]>>c>>m[1]>>c>>m[2]>>c;
    return s;
}
// this is a write operator for a node
ostream& operator <<(ostream& s,node& n)
{
    s<<"("<<setprecision(10)<<n.x<<" "<<setprecision(10)<<n.y<<")";
    s<<setprecision(20)<<n.D<<" "<<n.flag<<n.Y;
    return s;
}
// this is a read operator for a node
istream& operator >>(istream& s,node& n)
{
    char c;
    s>>c>>n.x>>c>>n.y>>c>>n.D>>c>>n.flag>>n.Y;
    return s;
}
// this is a write operator for a gaussian_doping
ostream& operator <<(ostream& s,gaussian_doping& d)
{
    s<<"("<<d.dir<<" "<<setprecision(15)<<d.Ne<<" "<<
    s<<setprecision(15)<<d.Nb<<" "<<setprecision(15)<<d.ew<<" "<<
    s<<setprecision(15)<<d.xw<<" "<<setprecision(15)<<d.yw<<" "<<
    s<<setprecision(15)<<d.xc<<" "<<setprecision(15)<<d.yc<<")";
    return s;
}
// read for a gaussian doping
istream& operator >>(istream& s,gaussian_doping& d)
{
    char c;
    s>>c>>d.dir>>c>>d.Ne>>c>>d.Nb>>c>>d.ew>>c>>d.xw>>c>>d.yw>>c>>d.xc>>c>>d.yc>>c;
    return s;
}

```

```

// write for a region
ostream& operator <<(ostream& s,region& r)
{
    unsigned int i,j;
    s<<"["<<r.M<<" "<<r.N<<" "<<r.slor_horizontal<<" ";
    s<<r.from_low_coord<<" "<<(*r.dop)<<"\n";
    for(i=0;i<r.M;i++) {
        for(j=0;j<r.N;j++) {
            s<<(*r.m[i+j*r.M])<<"\n";
        }
    }
    s<<"\n";
    return s;
}

// read for a region
istream& operator >>(istream& s,region& r)
{
    unsigned int i,j,tM,tN;
    char c;
    s>>c>>tM>>c>>tN>>c>>r.slor_horizontal>>c>>r.from_low_coord>>c;
    if (r.dop!=NULL) delete r.dop;
    r.dop=new gaussian_doping();
    s>>(*r.dop)>>c;
    if ((r.M!=tM)|| (r.N!=tN)) {
        for(i=0;i<r.M;i++)
            for(j=0;j<r.N;j++)
                if (r.m[i+j*r.M]!=NULL) delete r.m[i+j*r.M];
        if (r.m!=NULL) free(r.m);
        r.M=tM; r.N=tN;
        r.m=(node**)malloc(sizeof(node*)*tM*tN);
        for(i=0;i<tM;i++)
            for(j=0;j<tN;j++)
                r.m[i+j*tM]=new node();
        for(i=0;i<tM;i++)
            for(j=0;j<tN;j++)
                r.m[i+j*tM]->link((j!=tN-1?r.m[i+(j+1)*tM]:NULL),
                                (j!=0 ?r.m[i+(j-1)*tM]:NULL),
                                (i!=0 ?r.m[i-1+j*tM] :NULL),
                                (i!=tM-1?r.m[i+1+j*tM] :NULL));
    }
    for(i=0;i<r.M;i++) {
        for(j=0;j<r.N;j++) {
            s>>(*r.m[i+j*r.M]);
        }
    }
    s>>c;
    return s;
}

// write for the main simulation object
ostream& operator <<(ostream& s,main& m)
{
    s<<"["<<setprecision(20)<<m.ve<<" "<<setprecision(20)<<m.vi;
    s<<" "<<setprecision(20)<<m.vc<<" "<<setprecision(20)<<m.vb<<" "<<setprecision(20)<<m.bz<<"\n";
    s<<(*m.rs)<<(*m.ri)<<(*m.re)<<(*m.rc)<<(*m.rb);
    return s;
}

// read for the main simulation object
istream& operator >>(istream& s,main& m)
{
    char c;
    region *r[4];
    s>>c>>m.ve>>c>>m.vi>>c>>m.vc>>c>>m.vb>>c;
    if (c=='\n') s>>m.bz>>c;
    else m.bz=0.0;
    if (m.rs) delete m.rs;
    if (m.ri) delete m.ri;
    if (m.re) delete m.re;
    if (m.rc) delete m.rc;
    if (m.rb) delete m.rb;
    m.rs=new region();
    m.ri=new region();
    m.re=new region();
    m.rc=new region();
}

```

```

m.rb=new region();
s>>(*{m.rs})>>(*{m.ri})>>(*{m.re})>>(*{m.rc})>>(*{m.rb});
// have to link up nodes between regions yet
r[0]=m.re;r[1]=m.ri;r[2]=m.rc;r[3]=m.rb;
m.rs->link(4,r);
return s;
}

```

10.15 Thesis1.rc

```

// Copyright 1995, Brian Gieschen
#include "...\\apps\\bc3\\include\\_defs.h"
#include "...\\apps\\bc3\\owl\\include\\owlrc.h"
COMMAND MENU
BEGIN
POPUP "&File"
BEGIN
MENUITEM "&Open", 2
MENUITEM SEPARATOR
MENUITEM "&Print", 10
END

POPUP "&Simulate"
BEGIN
MENUITEM "&Init", 11
MENUITEM "&Start", 1
END

POPUP "&Graph"
BEGIN
MENUITEM "&Hole Density", 3, CHECKED
MENUITEM "&Electron Density", 4, CHECKED
MENUITEM "&Potential", 5, CHECKED
MENUITEM "Current Density &X direction", 6
MENUITEM "Current Density &Y direction", 7
MENUITEM "E&lectric Field X direction", 8
MENUITEM "Electric &Field Y direction", 9
END

END
#include "...\\apps\\bc3\\owl\\include\\filedial.dlg"

```

10.16 Thesis1.def

```

; Copyright 1995, Brian Gieschen
NAME THESIS1
DESCRIPTION '2D DC-Steady State Lorentz Simulator'
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE 32768
STACKSIZE 16384

```

END

1 7 - 1 / 1 - 9 | 5

FIN