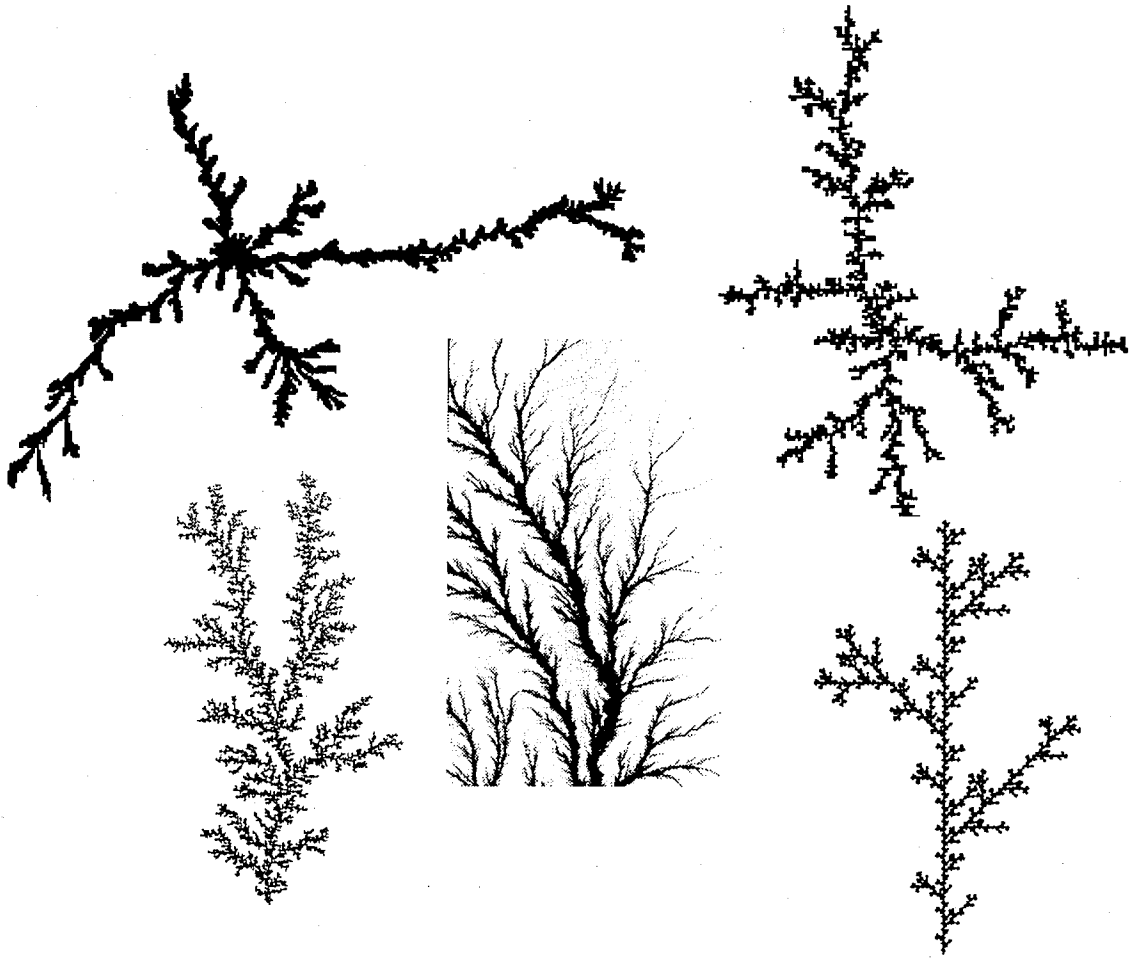
STOCHASTIC FRACTAL MODELLING OF DIELECTRIC DISCHARGES



by Geoff Stacey

A thesis presented to the University of Manitoba in

partial fulfillment of the requirements for the degree of

Master of Science

in the

Department of Electrical and Computer Engineering

Winnipeg, Manitoba

1994

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 accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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 propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-13515-2

Canada

Name _____

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 AND ELECTRICAL ENGINEERING

SUBJECT TERM

0544

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
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 0727
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
Sciences 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
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

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

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

SUJET | CODE DE SUJET | U·M·I

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Evaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-âtre social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hipitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures
Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÇres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Electronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie
Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et ther modynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systâmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

STOCHASTIC FRACTAL MODELLING OF DIELECTRIC DISCHARGES

BY

GEOFF STACEY

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

© 1994

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publications rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's permission.

ABSTRACT

Electrical discharge trees grow within solid dielectrics in the presence of strong electric fields. Stochastic fractal models effectively simulate their growth and structure.

This thesis presents several implementations of existing and new stochastic 2D models for such dielectric discharges. The dielectric field is calculated using the finite difference approximation to Laplace's equation with both Gauss-Seidel (successive relaxation) and Jacobi (simultaneous displacement) techniques on a square grid. These simulations have produced electrical trees on grids as large as 600 by 600 elements with up to 25000 pixels. Graphical rendering of the result includes field displays, data plotting, and real time animation.

A new dilation dimension algorithm was introduced and tested with the results compared to existing box counting and correlation dimension algorithms. The accuracy of this new algorithm is better than 0.3% on L-systems fractals as opposed to an accuracy of 2% with the other two methods. Extensions to the dilation dimension algorithm are also proposed giving results identical to those achieved with the correlation dimension algorithm on multifractals. Variations between fractal dimension measurements also show that multifractal measures are important for use on multifractal images.

Testing the process of image digitization, enhancement, and measuring fractal dimension indicates a 3% error is introduced on larger images of size 1024 by 1024 pixels. The error grows to more than 10% as the image size shrinks to 64 by 64 pixels.

Tuning a stochastic model to simulate both diffusion limited aggregation (DLA) and electrical trees show that the growth is related to the field gradient raised to the exponent 1.65 in both cases. A problem with the growth equation in the existing stochastic model is discussed and testing with a new model shows this problem to be minor. Improvements to the existing models are suggested such as employing a field gradient threshold and partial conduction of the simulated electrical trees.

ACKNOWLEDGEMENTS

I would like to thank all those people that contributed to this work. Special thanks to my supervisor, Dr. W. Kinsner, for suggesting this thesis topic and for also providing a feeling for the larger picture through formal and informal discussions.

I express my gratitude to all fellow members of the Signal and Image Compression Group at the University of Manitoba for their support and especially to Ken Ferens and Armein Langi who have born the brunt of my outbursts for the past two years.

Without the full support of my parents this work would have been much more difficult or impossible to complete which I gratefully acknowledge.

Partial support from the Natural Sciences and Engineering Research Council of Canada is also acknowledged.

TABLE OF CONTENTS

Page

ABSTRACT ii
ACKNOWLEDGEMENTS iii
TABLE OF CONTENTS iv
LIST OF FIGURES vii
LIST OF TABLES ix
LIST OF ABBREVIATIONS AND ACRONYMS x
1.0 INTRODUCTION 1
1.1 Problem Definition 1
1.2 Structure of Thesis 2
1.3 Summary 4
2.0 BACKGROUND 5
2.1 Theory and the Stochastic Model 5
2.2 Origins of Fractals and Fractal Dimension 8
2.3 Fractal Dimension 14
2.3.1 Simple Box Dimension 15
2.3.2 Dilation Dimension 18
2.3.3 Full Dilation Dimension 21
2.3.4 Correlation Dimension 25
2.4 Image Enhancement 26
2.5 L-systems Fractal Generation 27
2.6 Summary 28
3.0 SYSTEM DEFINITION 30
3.1 Overall System Structure 30
3.2 TEST Program Design and Implementation 31
3.3 TEST Program Menu Structure 33
3.3.1 Main Menu 33
3.3.2 File I/O Sub-menu 33
3.3.3 L-systems Sub-menu 34
3.3.4 Dimension Sub-menu 36
3.3.5 Picture Enhancement Sub-menu 37
3.3.6 Edge Detector Sub-menu 38
3.3.7 Simulation Sub-menu 39
3.4 Summary 41
4.0 SOFTWARE DESIGN AND IMPLEMENTATION 43
4.1 Stage One: Data Representation, File Input and Image Display 43
4.1.1 Internal Data Representation for Images 43
4.1.2 File I/O - TIFF File Reader 43
4.1.3 Design of Screen and User Interface 44

4.1.4 Display of Images 44
4.2 Stage Two: Standard Algorithm Implementation 45
4.2.1 High Pass Spatial Filter 46
4.2.2 Low Pass Spatial Filter 46
4.2.3 Median Filter 47
4.2.4 Image Negation Subroutine 47
4.2.5 Laplacian Edge Detectors 47
4.3 Stage Three: L-systems Generator and Test Fractal Generation 49
4.3.1 L-systems Generator 49
4.3.2 L-systems Test Fractals 53
4.3.2.1 (a) Koch Type Curve 1 55
4.3.2.2 (b) The True Koch Curve 55
4.3.2.3 (c) Koch Type Curve 3 56
4.3.2.4 (d) Koch 1.5 Type Curve 4 56
4.3.2.5 (e) Tree Type Curve 1 56
4.3.2.6 (f) Tree Type Curve 2 57
4.3.2.7 (g) Tree Type Curve 3 57
4.3.2.8 (h) Tree Type Curve 4 58
4.3.2.9 (i) Sierpinski Gasket 59
4.3.2.10 Koch Type Curve with Overlap 59
4.4 Stage Four: Dimension Analysis Subroutine Implementation 60
4.4.1 Simple Box Dimension Subroutine 61
4.4.2 Dilation Dimension Subroutine 62
4.4.3 Full Dilation Dimension Subroutine 64
4.4.4 Correlation Dimension Subroutine 65
4.5 Stage Five: Implementation of the Stochastic Model 67
4.6 Summary 75
5.0 EXPERIMENTAL RESULTS AND DISCUSSION 77
5.1 Dimension Algorithm Accuracy on L-systems Test Fractals 78
5.2 Dimension Algorithm Consistency on Stochastic Model Fractals 81
5.3 Dimension Error as a Function of Image Size Experiment 83
5.4 Dimension Error as a Function of Sub-image Size Experiment 84
5.5 Dimension Error as a Function of Image Threshold Experiment 86
5.6 Diffusion Limited Aggregate Experiment 88
5.7 Testing the Field Convergence using Gauss-Seidel Iteration 90
5.8 Handling the Differences Between L-systems, Simulated, and Actual Images 92
5.9 Tuning the Stochastic Model to Match Actual DLAs 93
5.10 Tuning the Stochastic Model to Match Actual Electrical Discharges 98
5.11 Summary 101
6.0 CONCLUSIONS AND RECOMMENDATIONS 103

6.1 Fractal Dimension Estimation 103
6.2 The Stochastic Model 104
6.3 Summary 106
REFERENCES 107
APPENDICES 110
A Data for Graphs used in Thesis 110
A1 Data for plot in Fig. 5.1 , page 84. 110
A2 Data for plot in Fig. 5.2 , page 86. 112
A3 Data for plot in Fig. 5.3 , page 87. 114
B Listing of Subroutines and Location 117
C Actual C Code Listings for Routines used in Thesis 119
C1 FILE: TEST.C 119
C2 FILE: DIMEN.C 136
C3 FILE: EDGES.C 185
C4 FILE: FULD.C 194
C5 FILE: PIC_DRAW.C 212
C6 FILE: ROUTINES.C 227
C7 FILE: SCREEN.C 242
C8 FILE: SIM.C 246
C9 FILE: TIF_READ.C 284
C10 FILE: PICTURE.H 295
C11 FILE: LMAP.H 296
C12 FILE: SCREEN.H 297
C13 FILE: MAKEFILE 297

LIST OF FIGURES

Figure		Page
2.1	Construction of the Cantor set.	9
2.2	Properties of the Hausdorff-Besicovitch dimension.	11
2.3	Illustration of the box counting dimension definitions.	16
2.4	Variations of the box counting dimensions.	17
2.5	Simple box dimension.	18
2.6	Dilation dimension.	21
3.1	Overall system structure	30
3.2	Computer display screen layout.	32
3.3	TEST program main menu structure.	34
3.4	File I/O sub-menu.	35
3.5	L-systems test fractal sub-menu.	36
3.6	Dimension calculation sub-menu.	37
3.7	Picture enhancement sub-menu.	38
3.8	Edge detector sub-menu.	39
3.9	Simulation sub-menu.	40
4.1	Pixel masks used for subroutines.	46
4.2	LENGINE flowchart.	53
4.3	L-systems test fractal subroutines.	54
4.4	L-systems generated test fractals.	60
4.5	Flowchart of box dimension algorithm.	62
4.6	Flowchart of dilation dimension algorithm.	64
4.7	Correlation dimension flowchart.	66
4.8	General flowchart of stochastic model subroutines.	69
4.9	General flowchart of subroutines for model subroutines.	70

4.10 Parallel tree field profile plot, Gauss-Seidel iteration. 72
4.11 Circular tree field profile plot, Gauss-Seidel iteration. 73
4.12 Parallel tree fractal, Gauss-Seidel iteration with box dimension. 74
4.13 Circular tree fractal, Gauss-Seidel iteration with dilation dimension. 74
4.14 Results from model for different values of h. 75
5.1 Dimension error (%) vs. image size. 84
5.2 Dimension error vs. sub-image plot. 86
5.3 Dimension error vs. image threshold. 87
5.4 DLA experimental setup. 89
5.5 Example of DLA created using copper sulphate solution. 90
5.6 Simulated and actual DLA images. 95
5.7 Simulation lattice showing corner and pocket growth sites. 96
5.8 Old and new stochastic model images. 98
5.9 Simulated and actual electrical discharges. 100

LIST OF TABLES

Table		Page
4.1	Outer pixel weights.	48
4.2	Center pixel weights and divisor for exponential masks.	48
4.3	Center pixel weights and divisor for gaussian masks.	49
5.1	Dimension measurements on the L-systems test fractals.	80
5.2	Full dilation dimension measurements.	82
5.3	Stochastic model dimension measurements.	83

LIST OF ABRBREVIATIONS AND ACRONYMS

C	"C" programming language.
D_C	Dimension measured using the correlation algorithm.
D_B	Dimension measured using the simple box counting algorithm.
D_D	Dimension measured using the dilation dimension algorithm.
D_{FD}	Dimension measured using the full dilation dimension algorithm.
D_S	Dimension measured using the similarity dimension algorithm.
DOS	Disk operating system for Intel based microprocessor systems.
DLA	Diffusion limited aggregate.
GIF	Graphics image file.
IBM PC	International Business Machines personal computer.
I/O	Input/Output.
L-systems	Lindenmayer systems method for generating fractals.
PICT	Picture file format.
TIFF	Tagged image file format.
VGA	Video graphics adapter.

CHAPTER 1
INTRODUCTION

1.1 Problem Definition

Incorporating stochastic processes into the mathematical modelling of physical events represents a departure from traditional thinking. Since the majority of such events behave stochastically, traditional solutions are limited. An inflated rubber balloon will follow a different random path as it deflates each time released no matter how carefully the initial conditions are controlled. Stochastic fractal modelling, as well as fractal or multifractal dimension measures can offer more information about the travels of the erratic balloon and other fundamentally stochastic events such as electrical discharges.

Electrical discharges in dielectrics or across surfaces eventually lead to the breakdown of the dielectric or surface. Modelling the growth of such events using stochastic models leads to a better understanding of this process. Using fractal dimension to measure the complexity of the discharge is an obvious next step since the electrical tree growth is best described as a fractal. This thesis looks at the existing stochastic model for discharges in dielectrics and applies techniques for measuring the dimensions of fractals to these trees for investigation of the composition of the discharge. The discharges and their simulated counterparts produced by the stochastic models are usually called Laplacian multifractals because their growth is best represented using Laplace's equation, and their structure exhibits more complexity than simple fractals [WiPi86, PiEW86].

The dimensions of various parts of the discharge will be measured to determine first if a structure exists that can be measured, and second, if this structure is duplicated using existing stochastic models simulating the electrical discharge growth. With the simulation results a better understanding of stochastic modeling and ultimately of electrical discharge trees is obtained.

Emphasis will be on developing a testing program to fully investigate and identify the lim-

itations of previous efforts on this subject and to provide a foundation for further investigation. Since the literature contains very little information pertaining to the estimation of errors, this area will be of special interest.

Before tackling the images from Lichtenberg experiments and actual photographs, the simpler images of diffusion limited aggregates (DLAs) will be studied. Does the model for DLAs produce images that are similar to that of actual DLAs grown in the lab, and are these images multifractal? After looking at this question, can the stochastic model be tuned to then create simulations of DLAs and electrical discharges that match actual experiments? The results will confirm or question the usefulness of the current stochastic model and possibly indicate areas for improvements.

1.2 Structure of Thesis

This thesis begins by looking into the published results in the area of stochastic modelling of multifractal phenomena such as DLAs and electrical discharges. The existing stochastic model is introduced in Chapter 2, with implementation details discussed later in Chapter 4, and limitations and suggested improvements being made in Chapter 5. The stochastic model cannot be studied in isolation however. Many supporting functions were also required for making measurements on the images created by the stochastic model and from experimental data.

After introducing the model, a brief introduction to fractals follows. This introduction will focus on the definitions of the Hausdorff s-dimensional measure and Hausdorff-Besicovitch dimension. The concept of fractals and objects with fractional dimension is in some ways counter intuitive and must be carefully considered. After introducing fractals, the techniques used for measuring fractal dimension in this thesis are detailed later in Chapter 2. The methods used in this thesis represent only a small subset of the many methods used for determining dimension [Falc90, Schr91]. The mechanics behind how this subset of the dimension measuring techniques are implemented is detailed in Chapter 4.

When analyzing images, image enhancement is an important preprocessing step towards

the final measurements. The routines implemented in this thesis are listed in Chapter 2, with programming details given in Chapter 4. Since all the images of actual fractals would need some form of enhancement, this was especially important.

Testing and verification of the analysis procedures implemented in this thesis required supporting images of known structure and dimension. The L-systems or Lindenmayer systems generated fractals were used as testing images to test the performance of many of the routines produced for this work [Lind68]. The L-systems method of generating fractals is introduced at the end of Chapter 2 with detailed description of each of the test fractals given in Chapter 4.

While the literature search for papers concerning the stochastic model proceeded, initial programming efforts to develop the basic data manipulation functions for the fractal testing program called `TEST` also commenced. This program ultimately represents the culmination of the authors efforts regarding the subject of analyzing and modelling Laplacian multifractals. Chapter 3 deals with the design of the structure of the `TEST` program and introduces all the routines and algorithms that were implemented.

After all the other stages were in place, investigation of the stochastic model and its properties could commence. Implementing the model was simple at this stage though investigating the convergence properties of the iterative field calculations was more involved. Measuring the dimension of the images obtained through simulation and comparison with actual experimental results was then possible. Experimentation to quantify the limitations of the measurement process being used was necessary to estimate the errors involved. These experiments and their results are given in Chapter 5.

The final stage of this thesis involves tuning the parameters of the stochastic model so that the simulated fractal growth dictated by the model and the actual experimental data for DLAs and electrical discharges matched. The limitations of the experiments and the model are then discussed. The results of this work will hopefully provide a better understanding of the limitations of the current stochastic model and a better understanding of actual discharges and electrical

treeing in dielectrics. The results of these experiments are also given in Chapter 5.

In Chapter 6 the overall conclusions are given for the experiments with recommendations for further research and direction offered.

The last several sections contain the references, data for the graphs, and the actual "C" code written and/or used for this thesis.

1.3 Summary

The objective of this thesis is to study the intricacies of stochastic fractal modelling in order to achieve a better understanding of such models and their usefulness when applied to discharges in solid dielectrics. Indirectly, this also involves studying the measurements of fractals. Then the process of tuning the model to real data will hopefully reveal limitations with the current model and lead to improvements with the model and the testing procedure.

CHAPTER 2
BACKGROUND

The background work for this thesis consisted of searching the literature for papers concerning the stochastic model being studied, learning about fractals and their properties, and duplicating the efforts of previous authors. Stochastic models incorporate randomness or probability in their structure in order to better simulate real world processes which exhibit non-deterministic behavior. Fractals are objects with certain mathematical properties which do not appear to "fit" into classical Euclidean spaces. This is covered in better detail after first introducing the stochastic model. Following this, an introduction to fractals and fractal sets is given. Then the algorithms used in this thesis to estimate a parameter called the fractal dimension is covered. We quickly noted a lack of detail in the reports on exactly how the previous authors calculated fractal dimension or how accurate these calculations were believed to be [NiPW84, StMe88, KuMa91, MeMa92]. The derivation of a new dimension measuring algorithm is also presented which capitalizes on the unique digital representation of images.

2.1 Theory and the Stochastic Model

Electrical treeing in dielectrics is a very complex phenomenon. Electrical trees are tracks left in the dielectric from the destructive action of electrons excited by a high intensity local electric field within the dielectric. The mobile electrons degrade the dielectric through chemical changes, creation of voids, and other mechanisms. The trees have poorer insulating properties compared to the parent dielectric material and essentially represent an electrical short circuit.

It is desired to have a simple growth model for these trees in order to investigate conditions governing their growth. Such a simple approach could be based on a stochastic model used to represent the growth of diffusion limited aggregates or DLAs. This model produces images which have a fractal appearance. The model uses a Laplacian field in the calculations with the results called Laplacian fractals [WiPi86]. The original model intuitively describes the growth of

what has become to be known as DLAs since the main mechanism for growth in this case is believed to be a diffusion process. The extensions to the model are hoped to create fractal images with the same properties as true measured electrical discharges [NiPW84]. Other techniques have been used to model electrical trees growth in dielectrics with reasonable success. Some models use the injection of random walking particles which stick to the growing fractal tree where the particles meet the tree during their walk [KuMa91]. A model based upon random walking particles does not involve an electrical field explicitly and does not lead to a better understanding of the interaction of the electrical field and subsequent electrical tree growth. Models of greater complexity trying to incorporate a better understanding of the underlying physics of the problem are now being studied [MeMa92].

Following the work introduced by Niemeyer, Pietronero, and Wiesmann [NiPW84], the same stochastic model was initially adopted for simulating discharge tree growth in a homogeneous dielectric. The simulation begins with the creation of a discrete representation of the electrode and field configuration being tested. The discrete field and electrodes are used to approximate the actual continuous electrical field and electrodes. The model (or expression that determines the mechanism for growth) is then used to decide where the next bond in the dielectric is broken. After each discrete stage of growth, the electrical field is disturbed and must be recalculated. The process of growth followed by recalculating the field is repeated until the resulting growth reaches from one electrode to the other. The resulting tree structure is also assumed to be perfectly conducting to simplify the field calculations.

Three electrode configurations for this original model were implemented and are detailed later in this thesis. The first, tree growth was initially simulated on arbitrary sizes of square lattice using parallel electrodes. This configuration was investigated previously by J. Fukai and N. Sano in 1987 [FuSa87]. The second configuration used a circular outer electrode with a point inner electrode as used by the original authors, also using various sizes of square lattice. The third configuration also used a circular outer electrode but also a circular inner electrode. For convention, the coordinate system adopted here is *(i,j)* where *i* is the discrete x position and *j* is the discrete y

position.

For the existing stochastic model, the expression governing the location where the extension to the fractal tree occurs is associated with the potential gradient at this location as given by

$$P(i, j \rightarrow i', j') = \frac{\phi^{\eta}_{i', j'}}{\Sigma\phi^{\eta}_{i', j'}} \tag{2.1}$$

where *(i,j)* is a point on the fractal growth, *(i',j')* is the new position immediately adjacent to it being considered, ϕ is the potential at a point, and η is an exponent modifying the contribution of this potential. The term *P(i,j→i',j')* becomes the probability the fractal will grow between these two points. Since the potential of the fractal is zero, ϕ also represents the gradient at point *(i',j')*.

Such fractals generated with η being 1 are accepted as modelling DLAs. Tree growth is restricted to a starting point set midway upon the bottom electrode for the case of the parallel plates and to the center electrodes in the circular configuration cases.

When the discharge tree is extended at a location, the field distribution in the lattice changes. Here, the second order finite difference approximation to the Laplace equation is used iteratively to determine the field potentials within the lattice. This is given by

$$\phi_{i, j} = \tfrac{1}{4}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}) \tag{2.2}$$

The field can be iterated using many techniques and several different lattice scanning directions, acceleration factors, and Gauss-Seidel or Jacobi type field calculation procedures [Acto70]. The simplest and usually adopted configuration was used for this work. The lattice is scanned and recalculated from the top left and across thus ending at the bottom right. It is noted that such a direction is not optimal for the implementation of the simulation with parallel electrodes. The use of acceleration factors might also be useful but can also lead to field instability. It appears that nobody is using acceleration factors to speed field convergence with current implementations of the stochastic model. Acceleration factors were not used for field calculations in this work but becomes a recommendation for future work (also studied under another

M.Sc. thesis in our research group).

Gauss-Seidel type field iterations simply use the results from previous field calculations immediately for subsequent calculations (i.e. the result for point *(i,j)* is immediately used in the calculation for point *(i+1,j)*). Jacobi field iterations recalculate a new field from only old data points obtained from the previous iteration. The Gauss-Seidel type of field calculation converges at rates approaching a thousand times faster than iterations using Jacobi field calculations. Gauss-Seidel type field calculations were used mostly for this work but both cases were implemented and tested for the parallel electrode simulation.

The top and bottom electrodes are treated as fixed *Dirichlet boundaries* with the edge effects on the left and right side of the lattice treated as *Neumann boundaries* for the parallel tree simulation. In this case the lattice gradient in the i or discrete x direction is set to zero on the two latter boundaries by using Eqs. (2.3.b) and (2.3.c). In the circular tree simulations, the only requirement is for the inner and outer electrodes to be treated as fixed Dirichlet boundaries. The following equations are the Neumann boundary conditions for the parallel electrode configuration.

$$\frac{\partial \phi}{\partial i} = 0 \tag{2.3.a}$$

$$\phi_{0,j} = \tfrac{1}{4}(\phi_{0,j+1} + \phi_{0,j-1} + 2\phi_{1,j}) \tag{2.3.b}$$

$$\phi_{jmax,j} = \tfrac{1}{4}(\phi_{jmax,j+1} + \phi_{jmax,j-1} + 2\phi_{jmax-1,j}) \tag{2.3.c}$$

Problems and limitations with this stochastic model are discussed later.

2.2 Origins of Fractals and Fractal Dimension

If one searches for the origin of fractals and fractal sets, one ultimately finds reference to the Cantor set. Introduced in 1883 by Georg Cantor, it was labelled then by many as simply a pathological set because in its construction there appeared a paradox that was unexplainable by existing set theory [Daub79].

The Cantor Set is a set containing an uncountably infinite number of points whose combined length is still zero. (Recall that a simple line segment also has an uncountably infinite number of points but does have length) The iterated construction of the Cantor set consists of the removal of the middle 1/3 of the line segments obtained from the previous iteration step. The construction begins with the interval from 0 to 1 and ends when the limit is taken where each line segment itself becomes a point. Each point in the Cantor set has infinitely many points around it so each point in the set is an accumulation point. There are also infinitely many of these points in the set. Such a set could not be explained using the set theory at that time. The construction of this set can be seen in Fig. 2.1.

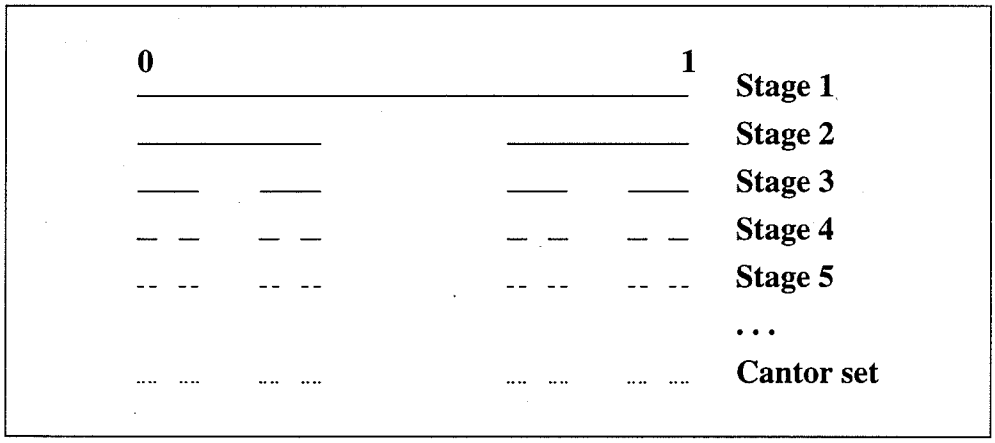


Fig. 2.1. Construction of the Cantor set.

The pathological Cantor set was not the only problem. Introduction of the Koch curve by Helge Von Koch in 1904 demonstrated that the classical notion of a set needed to be revised [Schr91]. Others soon provided many more examples. Explanations presented by Cantor were considered too radical and were dismissed. The theoretical work on set theory by Felix Hausdorff around the year 1919 provided an acceptable basis for fractal sets with fractional dimension. Further work by A.S. Besicovitch in 1929 improved the concepts and ideas for sets of fractional dimension [Falc90]. These sets were only recently called fractals by the much later works of Benoit Mandelbrot in 1979.

Defining what is and what is not a fractal is often given as a simple statement. A fractal is any set with Hausdorff-Besicovitch dimension strictly greater than its topological dimension [Mand82]. This usual definition may be too restrictive since this would exclude some sets such as the space filling Peano curves (which may be legitimate exclusion, dimension=2) and even possibly the Mandelbrot set (dimension=2) [PeJS92]. A fractal may be defined in somewhat looser fashion by saying any objects that are too irregular to fit into a traditional geometrical setting are fractals.

Measuring the dimension of fractals essentially involves testing the fractal at various scales. The Koch curve, for example, (image (b) in Fig. 4.4) is still a line with no area but appears to occupy more space than a simple line. Measuring how much area or volume the set appears to occupy at a given scale can be used to estimate the dimension of the set. This idea of covering sets of various radii measuring how much volume or area the fractal appears to occupy at a given scale is usual when using the morphological algorithms to estimate fractal dimension. Much of this theoretical work was based on extensions to set theory introduced by Hausdorff and Besicovitch.

Work on fractals would be incomplete without including and explaining the ideas of Hausdorff s-dimensional measure and Hausdorff-Besicovitch dimension. Since this is a fundamental beginning to the understanding of fractals, it is included in almost every book concerning the subject. Many other definitions for measuring the dimension of a set can be found and these may give different answers [Falc90, Schr91]. Following is an explanation of the definitions offered by Hausdorff and Besicovitch.

Consider an n-dimensional Euclidean space R^n. Let F be a subset of R^n with s a non-negative number. For any $\delta > 0$ the Hausdorff s-dimensional measure of a set F is defined as

$$H_\delta^s(F) = inf\left\{ \sum_{i=1}^{\infty} |U_i|^s : \{U_i\} \text{ is a } \delta\text{-cover of } F \right\} \tag{2.4}$$

This means that the sets U_i form the minimal δ-cover set of F with a maximum subset size of δ.

The size of this δ is the maximum distance (i.e. any metric such as Euclidean, Manhattan, or maximum (Chebychev) [PeJS92]) between any two points in each of the U_i. This definition means little to us until one investigates the behavior of s when the limit $\delta \rightarrow 0$ is taken. The limit is given as follows.

$$H^s(F) = \lim_{\delta \to 0} H^s_\delta(F) \tag{2.5}$$

This limit exists for any subset F of R^n with the limit being usually 0 or ∞. Again, the definition gives the s-dimensional measure of the set F for an arbitrary $s > 0$. What is really required is the critical value of s when the following conditions apply.

$$dim_H(F) = inf\{s : H^s(F) = 0\} = sup\{s : H^s(F) = \infty\} \tag{2.6}$$

The value of s satisfying the above conditions is called the Hausdorff dimension of F or Hausdorff-Besicovitch dimension because of the contributions to Hausdorff's work by A.S. Besicovitch.

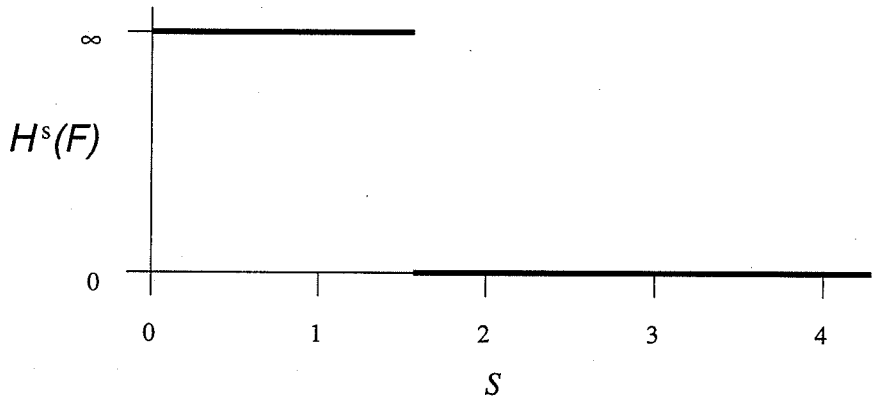


Fig. 2.2. Properties of the Hausdorff-Besicovitch dimension.

The Hausdorff-Besicovitch or fractal dimension is a careful balancing act. For values of s greater than the fractal dimension, the Hausdorff measure of F is zero. For values of s less than the fractal dimension, the Hausdorff measure of F is infinite. The s-dimensional measure at the critical value of s can be anything in between. This can be seen pictorially in Fig. 2.2. The formal

definition of fractal dimension is most awkward to apply to practical problems because it implies a search for a minimum amongst an infinite number of sets. This leads to the algorithms generally used for estimating this parameter. In some special cases however, the definition can be applied directly. The Hausdorff s-dimensional measure and its properties can be understood better by giving such an example.

Closer scrutiny of the Hausdorff s-dimensional measure of a set and its limit reveals its properties and the possibility of infinitely infinite numbers as proposed by G. Cantor. In some special instances the definition can be rewritten as follows

$$H^s(F) = c \cdot (f^s) \cdot (g^{s_T}) \tag{2.7}$$

where f^s represents the sum of all the infinitely small covering sets, g^{s_T} is the testing function and represents a zero to the testing exponent s_T, and c is a proportionality constant.

If $s_T=s$ then the two functions f and g cancel leaving the measure of the set in the s-dimensional space which is equal to the constant c above. This constant has properties of length of F if $s=s_T=1$, area of F if $s=s_T=2$, and volume of F if $s=s_T=3$.

This relationship is made clearer through an example. Take a cube of size 1.0 on each side and perform the calculation for H^s. Let $\delta=1/n$ in size, and use the Chebychev distance. Then

$$H^s_\delta(F) = inf\left\{\sum_{i=1}^{\infty} |U_i|^s\right\} = (n^3)\left(\frac{1}{n}\right)^{s_T} \tag{2.8}$$

The equality can be used since the construction is the optimal covering of the cube sought with sets consisting of smaller cubes representing the sets U_i. The maximum subset size of the $|U_i|$ is given by the length of the side of the cube and is 1/n. Taking the limit obtains

$$H^s(F) = \lim_{n\to\infty}(H^s_\delta(F)) = \lim_{n\to\infty}\left((n^3)\left(\frac{1}{n}\right)^{s_T}\right) = (\infty)^3(0_\delta)^{s_T}$$

$$H^s(F) = \begin{cases} 0 & \text{if } s_T > 3 \\ 1 & \text{if } s_T = 3 \\ \infty & \text{if } s_T < 3 \end{cases} \tag{2.9}$$

showing that indeed that numbers larger than infinity are present. Note however, the limit is in terms of a covering of sets of the embedding dimension 3 in this case. The Hausdorff definition uses a function of dimension equal to the embedding dimension to find the dimension of the set in question. By doing so, the Hausdorff s-dimensional measure of a set avoids having to define a relationship between the number of points in a line segment and the number of points in n-dimensional space. Cantor offered an explanation using the concept of transfinite numbers. This idea has not found much favor amongst mathematicians and has been relatively forgotten. For such constructions as above, volumes in infinitely many dimensions produces numbers that are infinitely infinite.

It is important to note that fractals necessarily have a fractional Hausdorff-Besicovitch dimension (accepting the first definition of a fractal given earlier). The Cantor set has an uncountably infinite number of members but no length, the Koch curve has infinite length but has no area, and a two dimensional fractal surface has infinite area but no volume. Clearly then it makes no sense to say the map area of England or Norway has fractal dimension since they have finite area (though difficult to measure accurately). We can say however, the coast of England or the coast of Norway has fractal dimension since the length of such depends upon the length of the measuring stick used and so is indeterminate in true linear measure. Since measured physical phenomena have true volume, true area or finite number of elements, care must be taken when looking at these sets through "fractal" glasses. Generally, consideration must be given to only the boundaries of real sets and limited ranges of scale when investigating their fractal properties. It should also be noted that with the Hausdorff-Besicovitch dimension, the true fractal characteristic of functions only appears when the limit is taken as $\delta \rightarrow 0$.

Two terms are often associated with fractals. These terms are *self similarity* and *self affinity*. If one magnifies a fractal that exhibits true self similarity, at some magnification value, the structure of the whole fractal reappears. In contrast, fractals exhibiting self affinity only appear similar or have similar properties at different scales or magnifications. The Cantor set is an example of a self similar fractal and the path of a particle undergoing Brownian motion is an

example of a self affine fractal [ZhBM90].

Fractal dimension has been measured with a variety of methods [Thei90]. The dimension algorithms have been classified into categories [Kins94, ZhBM90]. These are:

(i) morphological

(ii) entropy

(iii) variational

(iv) spectral

Of the various methods, those belonging to the morphological class appear best suited for the two dimensional white on black images being analyzed in this thesis. Since Hausdorff dimension is defined in terms of morphological covering sets, the morphological methods represent a direct approach to measuring dimension. The methods of simple box dimension, dilation dimension and full dilation dimension discussed later fall into this class of technique. The other techniques (entopy, variational and spectral) measure dimension indirectly. Spectral and variational methods might work well in finding the fractal dimension of a mountain slope but have problems with the white on black fractals being analyzed by this work [ZhBM90].

2.3 Fractal Dimension

The next focus of the investigation became measuring fractal dimension [Kins94]. There were some references to spectral or variational methods but these were detailed only for one dimensional fractional brownian motion or time series data [ZhBM90, MaSu93]. Much work involved modelling the fractal growth as a radial function and measuring the fractal dimension using a mass density or correlation function [NPW84, PiEW86, WiSa81]. Others have tested obtaining dimension using a gyration radius and have found the method works well on radially symmetric growths but not on other types of fractals [Fuji91]. This study also indicated that using morphological covers appeared to be a more accurate direction toward calculating dimension. Other authors used the dilation of what appeared to be circular and other sets but again the details

were missing or unclear [MaSu93]. This lead to the development of the dilation dimension routine detailed later on. Following is an explanation of the dimension algorithms implemented in this thesis.

2.3.1 Simple Box Dimension

Box dimension has many seemingly different but mathematically equivalent definitions. Definition (3) listed below I have called simple box dimension because it is easily implemented in software [Falc90, PeJS92]. The others require a search for a minimum or maximum amongst an infinite number of sets which is impractical using any computer algorithm. These definitions are depicted in Fig. 2.3.

Equivalent definitions of box dimension are:

(1) the least number of closed balls of radius δ that cover the fractal *F*.

(2) the least number of cubes with side δ that cover *F*.

(3) the number of δ-mesh cubes that intersect *F*.

(4) the least number of sets of diameter at most δ that cover *F*.

(5) the greatest number of disjoint balls of radius δ with centers in *F*.

The above definitions can be illustrated to offer the same measure by use of a simple diagram. In Fig. 2.4 the line labeled A is the line giving the true dimension of an object. The U and M lines show the variation of counting that is possible with Defn. (3) with varying mesh position and shrinking mesh size. The lower line L shows what happens to Defn. (5) when the size of the cover sets shrink. Definitions (1), (2), and (4) all fall close to the M line. Each of these definitions converges to the same value as the size of the covering sets decrease. The slope of the lines gives the measured box dimension of the object but clearly can be seen to only converge to the correct value at the limit when the cube, mesh or ball size shrinks to zero.

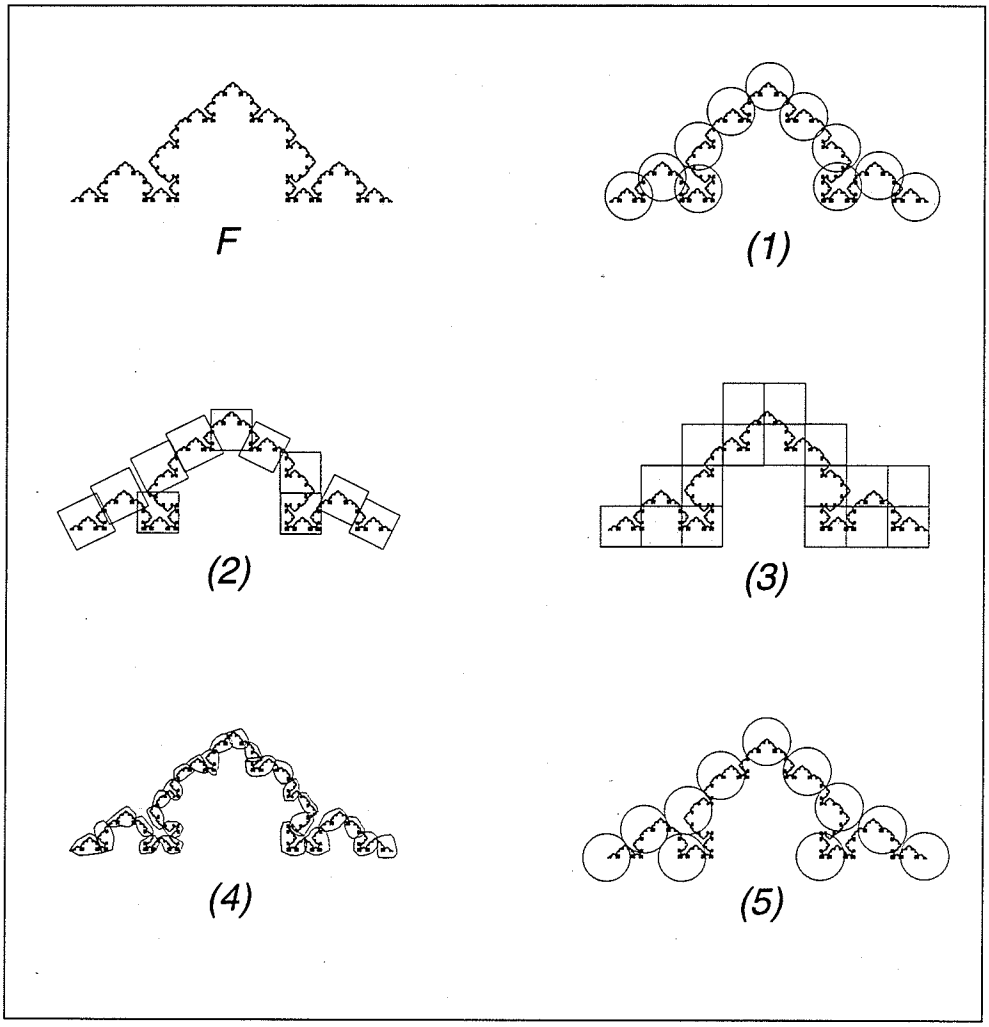


Fig. 2.3. Illustration of the box counting dimension definitions.

Definition (3) or the simple box dimension is a scheme for estimating the dimension of a black and white image. This algorithm uses a square δ-mesh grid to sample the fractal at different image scales. This idea of a mesh varying in size to test the fractal at different scales proves to be the easiest routine to write in software. A bonus is that the routine is surprisingly accurate on large images despite limitations.

The simple box counting algorithm however does not specify how to place the δ-mesh onto the image. As the box size grows, the possible positions where to place the δ-mesh grid also grows. Thus for a box size of 10 by 10 pixels on a 2D image, there are 100 possible locations for this grid. If the image was not a discretely sampled representation, the possibilities become

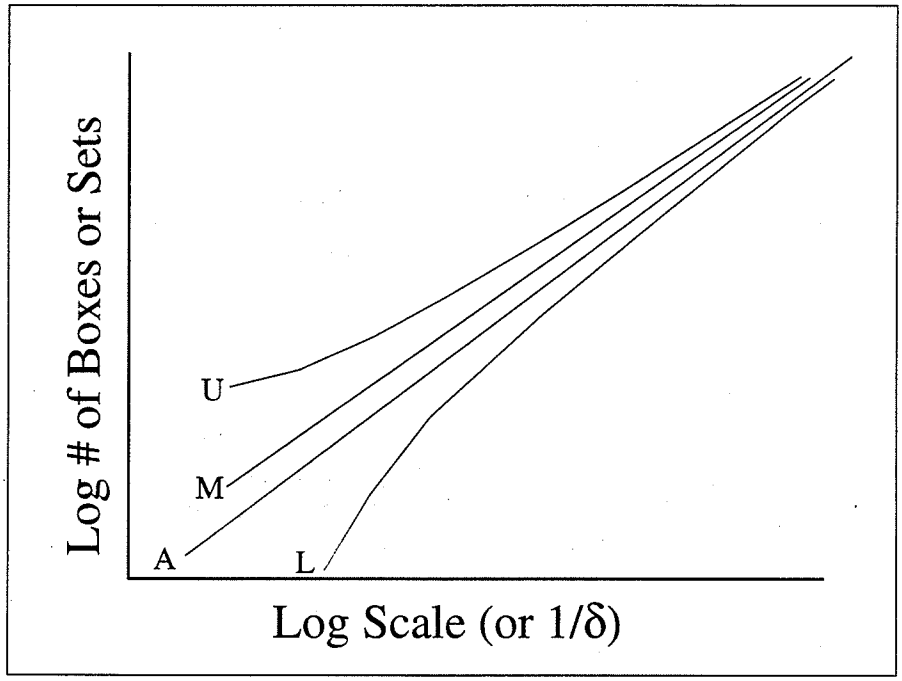


Fig. 2.4. Variations of the box counting dimensions.

infinite. Usually this definition is implemented by fixing the orientation of the δ-mesh at one point. This leads to satisfactory though definitely not optimal results. Using an arbitrary fixed point for grid positioning introduces what the author believes to be the major source of error with this algorithm.

The number of boxes in the δ-mesh containing at least one part of the image are counted at each of the box sizes chosen. The log of both the number of boxes counted and the scale at which they were counted is calculated. Scale is the same as 1/δ; i.e., one over the size of the covering sets. The results are then fitted using a least squares fit to a line, log scale vs. log count. Obviously, using smaller box sizes will ensure best results. The slope of this line gives the estimate of the dimension of the image. This process is depicted for two such meshes in Fig. 2.5.

The image or subimage being analyzed must be large enough so that the calculation is not affected by granularity inherent in using the larger box sizes. Also, the image must be digitized at

a resolution grainier than the smallest details in the original image so that counts using the smallest box sizes are not adversely affected.

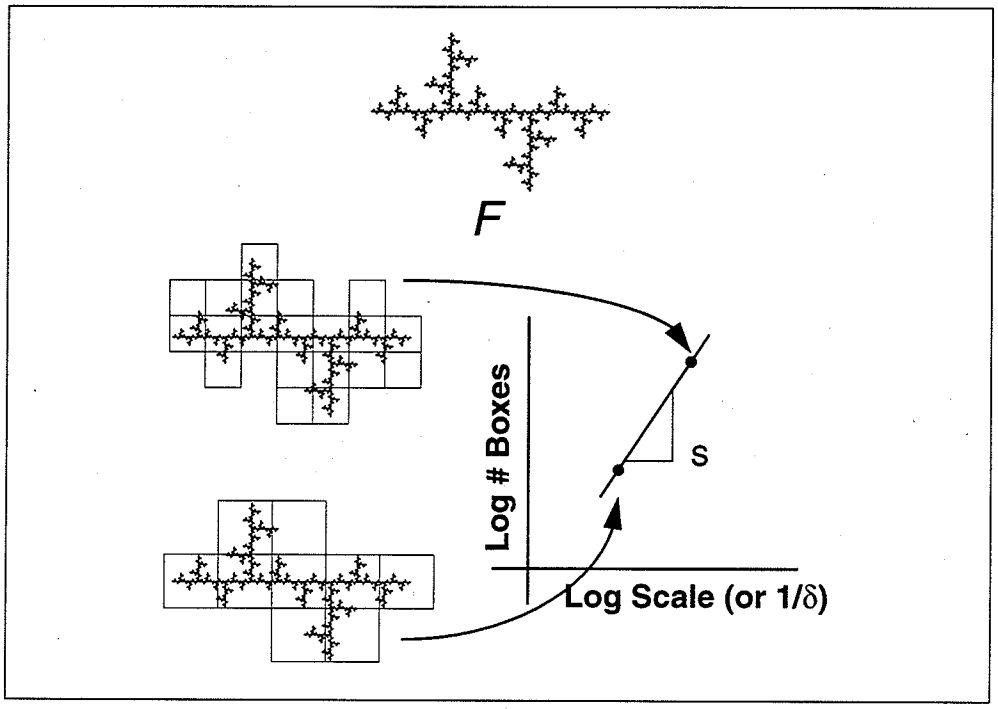


Fig. 2.5. Simple box dimension.

2.3.2 Dilation Dimension

Dilation dimension is an algorithm developed for this thesis to increase the accuracy of the dimension measurements being made on the images [KiSt94]. There are several references which use dilation to measure dimension but the authors do not cover the mathematical basis for its use or how they implement the routines [MaSu93].

The variation of area or volume of the Minkowski "sausage" formed when following a fractal curve with circular (2D) or spherical (3D) sets of various radii can be used to determine the dimension of the underlying fractal. This idea was introduced by H. Minkowski and G. Bouligand with the Minkowski-Bouligand dimension being named after them for their contribution to fractal sets [Kins94]. While trying to improve the accuracy of the simple box dimension routine, it occurred to the author that the square dilation of each pixel might function along similar lines as

boxes in the simple box dimension routine. Since a δ-mesh worked well for the simple box dimension algorithm, a cubic style of dilation might yield similar results. This variation of the Minkowski "sausage" idea proved to be correct and is covered next.

Each pixel in an image is best thought of as a square. This square represents testing of the fractal at a scale of one pixel. The pixels then represent a δ-cover of the fractal using small square sets. This idea is not limited to only two dimensional images however.

Two concepts are required for the derivation of the dilation dimension. First, the scaling property of a fractal with *s*-dimensional Hausdorff measure, H^s is recalled. Let us consider a metric space (R^n,d) where d is the Euclidean distance. Let F be a fractal such that if

$$F \subset R^n \text{ and } \lambda \geq 0 \text{ then}$$

$$H^s(\lambda F) \leq \lambda^s H^s(F) \tag{2.10}$$

where $\lambda F = \{ \lambda x : x \in F \}$, i.e. the set F is scaled by a factor λ.

Second, the idea that the fractals exhibit a power-law relationship is required. If F obeys a power-law relationship then the volume V (depending upon δ) needed to cover the fractal behaves as [Falc90]

$$V(\delta) \sim c\delta^{-s} \tag{2.11}$$

Formally, these two ideas are applied to estimate the dimension of a fractal.

The image of the fractal $F \subset R^n$ can be thought of as representing F with a collection of δ sets. Simply, the image can be viewed as a countable collection of Borel sets which cover F. Assume this set of pixels also represents the minimal δ-cover of F. We also assume that the scaling property of fractals also holds for this minimal δ-cover of F. Then the total volume V_1 of the δ-cover of F (volume in 2D is area) is given by

$$V_1 = \bigcup_{i=1}^{M} |U_i| = M\delta_1 = M \tag{2.12}$$

where the volume δ_1 is taken as unity at the pixel resolution of the image, M is the number of pixels in the image representing the sets U_i which are disjoint.

Then using the power law relationship for fractals to F (with the assumption that the fractal behaves with a power law relationship) obtains

$$V_k = \bigcup_{i=1}^{M} |\lambda_k U_i| = Mc\delta_k^{-s} \tag{2.13}$$

$$\text{with } c = \lambda_k^n \tag{2.14}$$

where n is the topological Euclidean dimension. Now applying the scaling property to δ_k^{-s} the following results

$$V_k = M\lambda_k^n\lambda_k^{-s}\delta_1 = M\lambda_k^n\lambda_k^{-s} \tag{2.15}$$

$$\text{or} \quad \frac{V_k}{\lambda_k^n} = M\lambda_k^{-s} \tag{2.16}$$

This can be seen as a dilation of each set δ_1. The dilated sets overlap giving a volume that increases proportional to the dimension of the underlying fractal. The cubic (square in 2D) style of dilation is easily implemented in software. The dilation dimension starts by dilating each image pixel into a second image buffer. Once the image has been dilated into the buffer, the area of the pixels set in the buffer is obtained.

Taking the logs of Eqs. (2.12) and (2.16), the next two equations are obtained

$$\log(V_1) = \log(M) \tag{2.17}$$

$$\log\left(\frac{V_k}{\lambda_k^n}\right) = \log(M) - s\log(\lambda_k) \tag{2.18}$$

Solving for s gives the dilation dimension D_D

$$D_D = s \cong \frac{\log(V_1) - \log\left(\frac{V_i}{\lambda_i^n}\right)}{\log(\lambda_i)} \tag{2.19}$$

This single approximation is insufficient for the accurate determination of the fractal dimension. As with the box counting dimension [PeJS92], if a number of data pairs are computed in the form

$$\left(\log(\lambda_k), -\log\left(\frac{V_k}{\lambda_k^n}\right)\right) \tag{2.20}$$

a least squares fit to a line should achieve higher accuracy. The results from testing this algorithm are presented in Chapter 5.

The image, Fig. 2.6, shows how the dilation dimension functions with a simple point set representing the fractal for clarity. The mechanics for implementation can be seen. The image is dilated into a cleared buffer whose size is increased by the size of the dilation in all directions. Each pixel is then dilated into the buffer as shown. Such a method only requires a dilation buffer with a size slightly larger than the original image size.

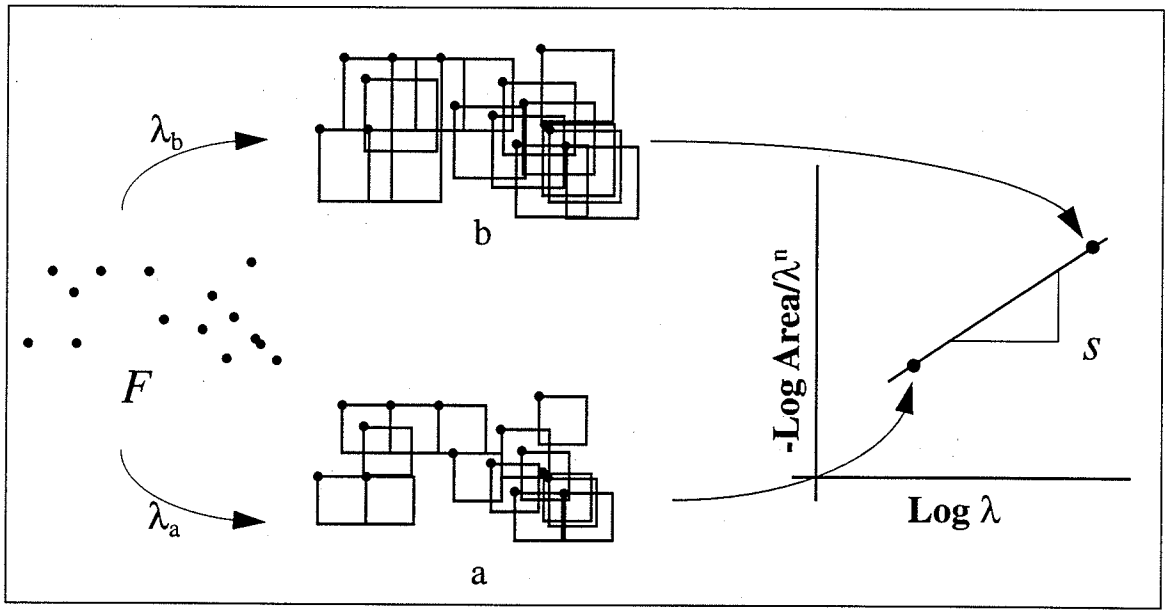


Fig. 2.6. Dilation dimension.

2.3.3 Full Dilation Dimension

The full dilation dimension subroutine is an extension of the dilation dimension subroutine noted above. This routine tests the image over an extended scale range of 256:1, and gives

the multifractal dimension as the greatest dimension obtained within this range as measured using a cubic polynomial fit to the dilation curve. This method is more general than the methods above.

The dimension of a multifractal image changes with the scale at which it is measured at or at different spatial locations. The case where dimension varies spatially is not looked at in great detail in this thesis. Giving the dimension of the fractal as a function of scale clearly characterizes the dimension better than a single dimension number. What is usual however, is to take the slope of the mass density function or correlation function at the position of greatest linearity and use this value to express the dimension of the fractal.

It must be explained that any finite fractal, of single dimension or otherwise, exhibits a variation of dimension if tested over a large enough range of scales. At the smaller scales the definition of the pixel dominates. For the L-systems generated test fractals (discussed later) this definition is ideal and truly represents a collection of covering sets. For the definitions of the pixels in scanned images or computer simulated images however this is not the case. At the other extreme, as the scale of testing grows larger and larger, the dimension of any finite fractal image will behave more and more like that of a point.

The full dilation procedure can be improved by noting that the fractals being tested are all finite. The dilation area overlaps the ends of the fractal giving a greater area than is desired. To eliminate the end or edge effects, the formula for dilation was changed to

$$\left(\log\left(\lambda_k\right), -\log\left(\left(\frac{V_k}{\lambda_k^n}\right) - 1\right)\right) \tag{2.21}$$

This change was only done for the full dilation dimension algorithm. The accuracy of both dilation algorithms is tested and described in Chapter 5.

Using polynomials to characterize the curves obtained by dilating the points in the fractal is the logical next step to characterizing the dimension of the fractal over as large a range as possible. Usually, the fitted function is derived as $y=P(x)$. But for the curves obtained through the dilation procedure, a polynomial of the form $x=P(y)$ is a more appropriate choice. At the

larger scales, the slope of the log area/dilation area vs. log dilation function tends toward zero. At the smaller scales below a pixel in size, the slope tends toward either zero (if the data is treated as discrete points), one (if line segments are used between the discrete data points), or n (if n-dimensional volumes are used to represent the pixels). In the first two cases, a curve fit using a third order polynomial of the form $x=P(y)$ would be appropriate.

Once a polynomial is fitted to the dilation curve data using a least squares fit, the maximum slope can be solved for mathematically within the range of dilation. The least squares fit also serves as a filter to even out any irregularities with the data. The multifractal dimension is then given as this value.

The full dilation dimension subroutine is an extension of the dilation dimension subroutine detailed before. With this routine however, the range of dilation and the values used for dilation were increased and changed. This routine uses the sixteen dilation values of 1,2,3,4,6,8,12,16,24,32,48,64,96,128,192, and 256 pixels. The memory requirements are much greater than the previous routines and special speed enhancements must be employed to increase the speed of calculation. The log-log data obtained using these values of dilation are fitted using a least squares fit to polynomials of first and third degree. The least squares fit is achieved using the Moore-Penrose generalized inverse method for the third degree polynomial [Usma87]. Since the size of the matrices used are small, the accuracy of this calculation was verified using a hand calculator.

The least squares solution of the form $x=P(y)$ is found using the Moore-Penrose generalized matrix inverse method for a cubic curve with equation given by

$$x = ay^3 + by^2 + cy + d \tag{2.22}$$

This is achieved in the following manner. First the matrix A below is formed using the y-values of the log-log data obtained through the dilation procedure.

$$A = \begin{bmatrix} y_1^3 & y_1^2 & y_1 & 1 \\ y_2^3 & y_2^2 & y_2 & 1 \\ & \cdots & & \\ y_{16}^3 & y_{16}^2 & y_{16} & 1 \end{bmatrix} \tag{2.23}$$

Then, the least squares solution to the cubic polynomial is found by performing the following matrix calculations

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = (A^T A)^{-1} A^T [x] \qquad \text{where} \qquad [x] = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{16} \end{bmatrix} \tag{2.24}$$

Once the coefficients to the polynomial are obtained, the maximum slope within the interval of dilation can be solved for. Since the polynomial is of the form $x=P(y)$, solution for the minimum of this equation leads to solution for the maximum slope desired. Solving for the minimum is achieved by differentiating the polynomial twice and setting the result to zero.

$$6ay + 2b = 0 \qquad y = \frac{-2b}{6a} \tag{2.25}$$

(Note: if one were to use the first differential of the cubic equation, the resulting parabola is the dimension vs. scale function desired by some to characterize the fractal totally. This also indicates the limitation of a cubic curve fit to the data.) Constraining this answer to within the data range gives the point where the minimum slope of the cubic curve occurs.

$$y_{min} = min\,(y_{16}, max\,(y_1, y)\,) \tag{2.26}$$

Plugging this value of y back into the polynomial gives the x-value where the minimum slope of the $x=P(y)$ curve or the maximum slope of the $y=F(x)$ curve occurs. This value of slope is then solved for with the equation

$$D_{FD} = s = \frac{1}{3ay_{min}^2 + 2by_{min} + c} \tag{2.27}$$

This value of slope gives the maximum dimension of the fractal within the data range. The accuracy of the cubic polynomial fit using end correction is presented in Chapter 5. Higher order polynomial fits could be used but would require numerical solution for the maximum slope within the data range.

2.3.4 Correlation Dimension

Correlation dimension is an improvement and is similar is some aspects to the mass dimension used by Niemeyer, Pietronero, and Wiesmann [Kins94, NiPW84, PiEW86, WiPi86]. Mass dimension measures the distribution of the fractal from a single center point. This center point must be entered or selected by the user before performing this calculation. Correlation dimension measures the distribution of the fractal from every point in the fractal. This method does not require user intervention to select the center of the fractal since all points in the fractal are used as centers. Such a calculation however is computationally intensive and takes many hours on larger fractals with over a hundred thousand pixels. Care must be also taken when calculating the correlation dimension using data points constrained to a discrete lattice.

The correlation dimension uses the density correlation function to measure the dimension of a fractal. The density is defined to be one for pixel positions occupied by the fractal and zero for vacant pixels. The correlation function for a fractal containing N pixels is given by

$$C(r) \equiv \frac{1}{N}\sum_{r'} \rho(r')\,\rho(r'+r) \tag{2.28}$$

where $\rho(r')$ is the density of a point at r' in the fractal and $\rho(r'+r)$ is the density of another point within a radius r. The sum counts the number of pixels that are part of the fractal within a given radius of each pixel in the fractal. The equation then essentially gives the average probability that a pixel within the radius r of any pixel in the fractal is also part of the fractal.

The above correlation function is a discrete approximation to the ensemble average correlation function [WiSa81]. Intuitively, looking from any point in a fractal, the probability that a neighborhood pixel belongs to the fractal must decrease as the distance from the point increases.

The correlation function measures this decrease in proportion of pixels in the fractal to those not. The mechanics behind determining the correlation dimension can be found in Chapter 4.

Finding the correlation dimension of the fractal then involves taking the slope of the $\log(C(r))$ function plotted against $\log(r)$ over the region of greatest linearity. The region of greatest linearity is generally from three pixels to about half the gyration radius. The gyration radius is found using the second moment about the center of gravity of the fractal. For a two dimensional image, once this slope is determined (which is negative), the correlation dimension D_c is determined with the following equation [WiSa81]

$$D_c = 2 + \alpha \quad \text{where } \alpha \text{ is the slope} \tag{2.29}$$

Implementing the above formulas must be done with care and is covered later.

2.4 Image Enhancement

Image processing plays a very important role when analyzing fractal images. Each image must be enhanced to increase the accuracy of the analysis. The input to the system is a simple picture containing the DLA or electrical discharge. After applying a regime of picture enhancement procedures, the picture can then be finally analyzed by the dimension estimation algorithms to obtain the data required. The limitations of each step along this path must be understood for the result to have merit.

After developing the programs to input, manipulate and display images, the standard techniques for enhancing images were implemented. Investigation of various algorithms for enhancing pictures has shown that several standard enhancement techniques can be used.

The algorithms looked at for enhancing pictures include the high pass spatial filter, low pass spatial filter, median spatial filter, several Laplacian and Gaussian edge detectors, negation, normalization and histogram plotting, and threshold detection. Of these, the edge detectors and threshold detection play the most important role in the analysis of the pictures. These routines are discussed in greater detail later.

2.5 L-systems Fractal Generation

One of the more popular methods for generating fractals is to use L-systems first introduced by A. Lindenmayer [Lind68]. Another popular method, iterated function systems (IFSs), generates beautiful fractals but they require much more computation time to obtain complete fractals. The methods used to calculate the dimension of iterated function system fractals requires identification of all the mappings used for construction. The approach using L-systems was adopted for generating the test fractals for this work. This generation technique is relatively simple and can be explained best with an example.

First, a string of starting symbols is chosen in a pattern. These symbols represent movement, turning, and branching commands. For example, the string "f++f++f++" can represent the movement commands of "forward one step (f), turn right 120 degrees (++), forward one step (f), turn right 120 degrees (++), forward one step (f), turn right 120 degrees (++)" which draws a triangle. The command turn left 60 degrees (-) is not in the initial string. Now defining a set of rules for replacing these symbols such as "+ to +, - to -, and f to f+f--f+f" one can transform this initial string. Each time the string is run through these set of rules a new and longer string is formed. At the same time, to maintain the same image size as the initial string, the step distance must be reduced. In this case the initial step size is multiplied by a factor 1/3 each time the rules are applied.

Each application of the rules to the string is called an iteration. Clearly, the string quickly grows in length to be very large. Generally, for display, it is sufficient to stop the iterations when the step size becomes an order of magnitude smaller than the resolution of the computer screen. After the string is created, it can be converted into an image. By having the computer draw lines on the computer screen following the directions given by the final string, an image can be drawn. In the example starting string and rules above, the image after many iterations becomes the Koch snowflake!

The above method for generating the "road map" string for the computer to follow takes a

lot of memory space. A simpler algorithm can use recursive programming at the expense of taking a little more time. By adding more commands and changing the rules a very large variety of fractals can be drawn. In addition, only one end point of each line segment need be drawn since the true fractal only appears when the number of iterations forces each line segment to be much less than one pixel in length.

The dimension of this type of fractal is known by how the fractal is generated and by ensuring the fractal is self avoiding. Self avoidance is a necessary criteria for the dimension of the image to equal the dimension of the underlying fractal. Herein lies the difficulty with both iterated function systems generated fractals as well as for L-systems generated fractals. Self avoidance can only be achieved by careful selection of the rules and mappings and checking the results. This problem is alleviated with the stochastic model being looked at by this thesis because self avoidance is inherently built into how this model functions.

The analysis program developed for this thesis also includes a fractal that is not self avoiding where its measured dimension does not equal its calculated dimension for this reason.

Since the fractals generated with L-systems have a known dimension, they are effective calibration sources to check the performance of the analysis algorithms. These images are also pure (mathematical) fractals; i.e. they exhibit the same dimensional characteristics at all scales and spatial locations. This is unlike multifractals which may have different characteristics at different scales and/or at different spatial locations. These pure fractals can also be printed and computer scanned to investigate sources of error when digitizing images. Much time was spent looking at errors introduced when measuring dimension and L-systems provided the necessary calibration images.

2.6 Summary

The study and application of stochastic models to the simulation of discharge trees formed in dielectrics through unknown breakdown mechanisms should lead to a better overall understanding of the phenomena. The trees have a random structure whose growth must be governed

primarily by the electrical field gradients present in the dielectric. Measurements using existing and new fractal dimension estimating techniques will allow investigation of the limitations of the current model and provide insight towards improvements. Verification of the accuracy of the routines used is accomplished using L-systems generated test fractals.

CHAPTER 3
SYSTEM DEFINITION

3.1 Overall System Structure

Studying stochastic fractal models requires a complete system for testing. The overall system requires input data from various sources, followed by image enhancement, level detection and finally measurement of the fractal dimension. The fractal dimension of the actual images is then compared to the measured fractal dimension of images produced by running the stochastic models being studied. Supporting functions such as test fractals, image display and magnification, data storage and retrieval, and dimension measurement algorithms are required. Most of these functions including the stochastic model can be incorporated into a single program with only basic input functions provided by external commercial programs. This structure can be seen in Fig. 3.1. Such a program structure has complete control over the data at all times.

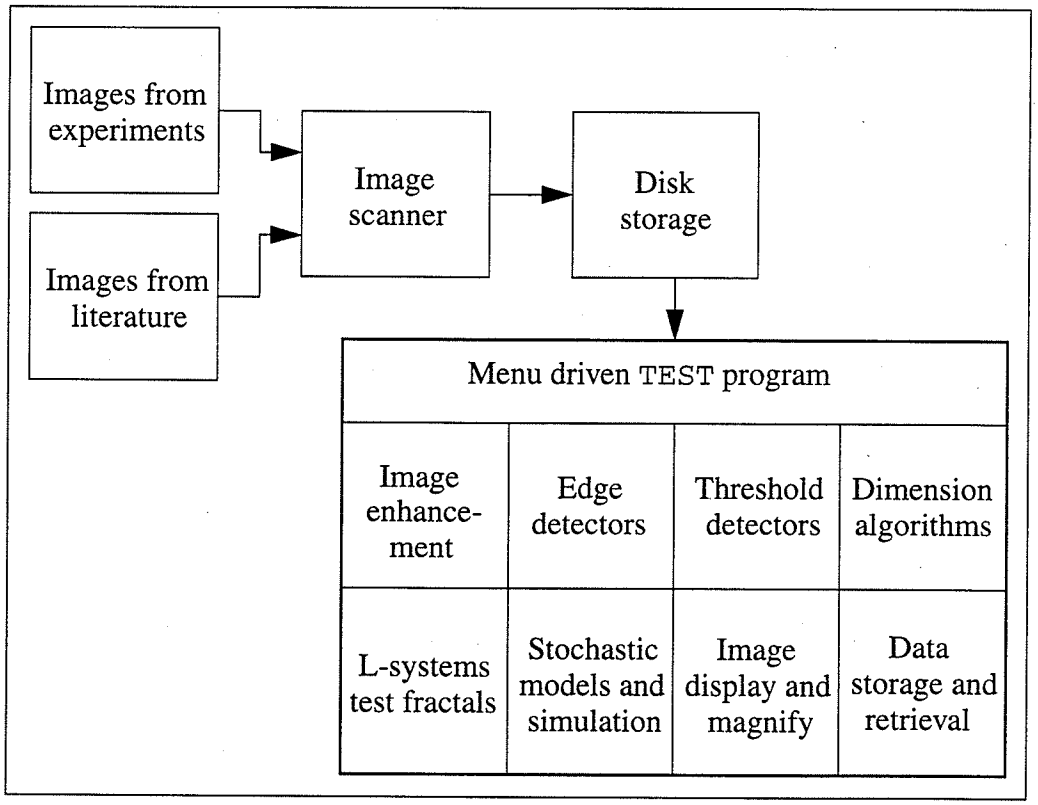


Fig. 3.1. Overall system structure

3.2 TEST Program Design and Implementation

The work presented in this thesis required writing many software subroutines to calculate, manipulate and display fractals and images. These routines were assembled into a simple menu driven shell program called TEST to allow the author to run the regime of procedures required for analysis. Since the papers read concerning this work did not include detailed explanations of how each of the authors performed their analysis, many routines had to be written in order to test their usefulness. Watcom "C" and the IBM PC type of computer was used for this work since the author owned a 386 based PC clone. The 32 bit Watcom "C" compiler was chosen because it could handle the large memory arrays required for this work.

The process began by selecting a storage format for all images. After some group debate, a subset of the TIFF 6.0 file format was decided upon for all file I/O. Once the storage format was decided upon, the TIFF file reader could be written. Since the TIFF 6.0 file format contains many features not required by this work, only a subset of the features were incorporated. All features required for the TIFF file reader to read uncompressed greyscale images from either the Macintosh, Sun workstation, or the IBM PC were implemented. This cross platform compatibility was the major reason for deciding upon the TIFF file format. It was found later however that most of the commercial programs written for the Macintosh and Sun were unable to read TIFF files from the IBM PC which negated much of this convenience.

Once images could be read from disk it was necessary to be able to display them. Routines for displaying and manipulating images were then written. The computer screen was partitioned into different regions each controlled by various parts of the TEST program. The first part of the screen decided upon was the image display area. It was decided that displaying two images on the IBM PC screen at a time in side by side format was the best approach. Then running the display at the PC's maximum resolution of 1024 by 768 pixels allowed the display of two images at 510 by 510 pixel size. Reserved under each of these image windows is a message area for messages and results.

Since the images being used were of unknown size, it became necessary to be able to display any image in these display windows. The display routines map the arbitrary image size in memory to these two windows. It was decided that the fractals would be displayed as white on a black background to give better image clarity. This coupled with giving brighter pixels priority when displaying images larger than 510 by 510 pixels would ensure details would not be lost on the screen. This property is only be for displaying images and would not affect calculations done on the original data preserved in memory. These routines were also useful when adding the image magnification subroutines later.

A screen plot subroutine was then required to plot the results of calculations. These screen plots are presented in the right image window and can also be captured from the screen and stored to a file if required. Below the two image windows a menu area for all system menus and some data results is constructed. Finally, below the menu area another text window is reserved for showing any errors or messages generated by the subroutines. The structure of the computer display screen is detailed in Fig. 3.2.

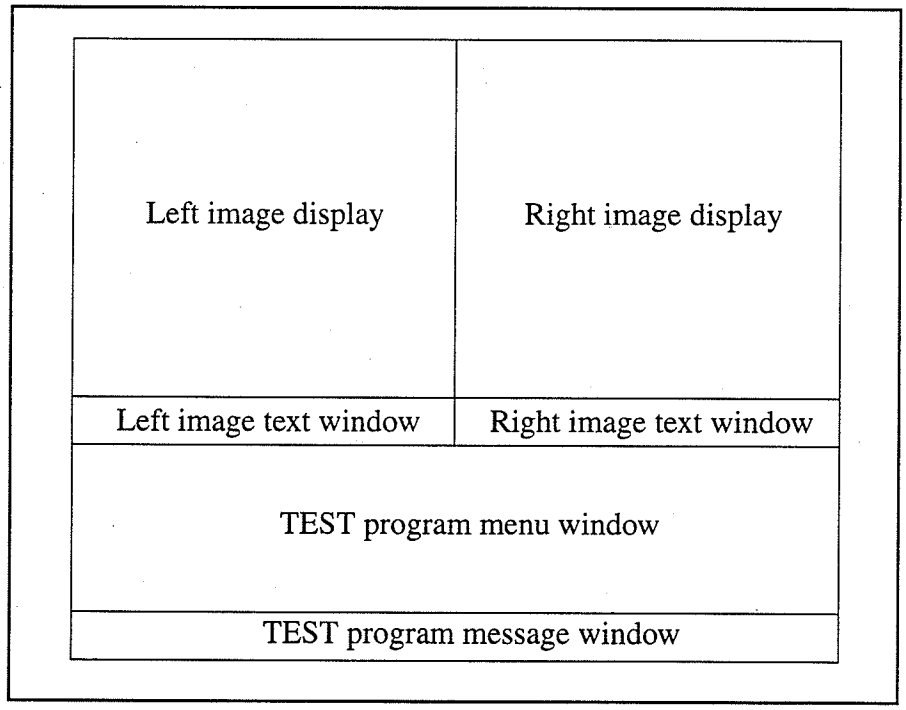


Fig. 3.2. Computer display screen layout.

When analyzing images many standard algorithms can be used. Many of these routines were written to allow testing their usefulness in the analysis. Edge detectors, low pass and high pass spatial filters, a median filter, upper and lower threshold detectors, an image negation routine, and histogram equalization and display routines were all to be incorporated into the shell program called TEST. These routines were tested using the standard test images of LENA and PEPPERS from the University of Manitoba image database on the Sun workstations.

3.3 TEST Program Menu Structure

Next is detailed the overall structure of the program TEST. The overall structure for the program TEST is a collection of subroutines called by user selection through a menu driven system. The entries in each of the menus is discussed next.

3.3.1 Main Menu

The main menu allows the user to select the function or sub menu required. The main menu is displayed upon execution of the program TEST and contains both sub-menus and often used functions. The draw memory to left or right screen displays the current memory buffer in the left or right window area discussed earlier. The functions save and recall memory to/from the buffer allows the user to save the current image in memory into a temporary memory buffer. This allows the intermediate results from image enhancements to be saved and restored before doing a destructive procedure. These routines save time recalling images from disk if the image is destroyed during the enhancement procedure. They prove very useful in long procedures. The structure of the main menu can be seen in Fig. 3.3. Other functions available from the main menu are File I/O, L-systems fractal generation, picture enhancement, dimension calculations, edge detectors, and simulation sub-menus. These sub-menus are discussed next.

3.3.2 File I/O Sub-menu

The file I/O menu includes the subroutines required to store and recall TIFF files to and from disk. The file input routine asks the user for the name of the file to be read into the global

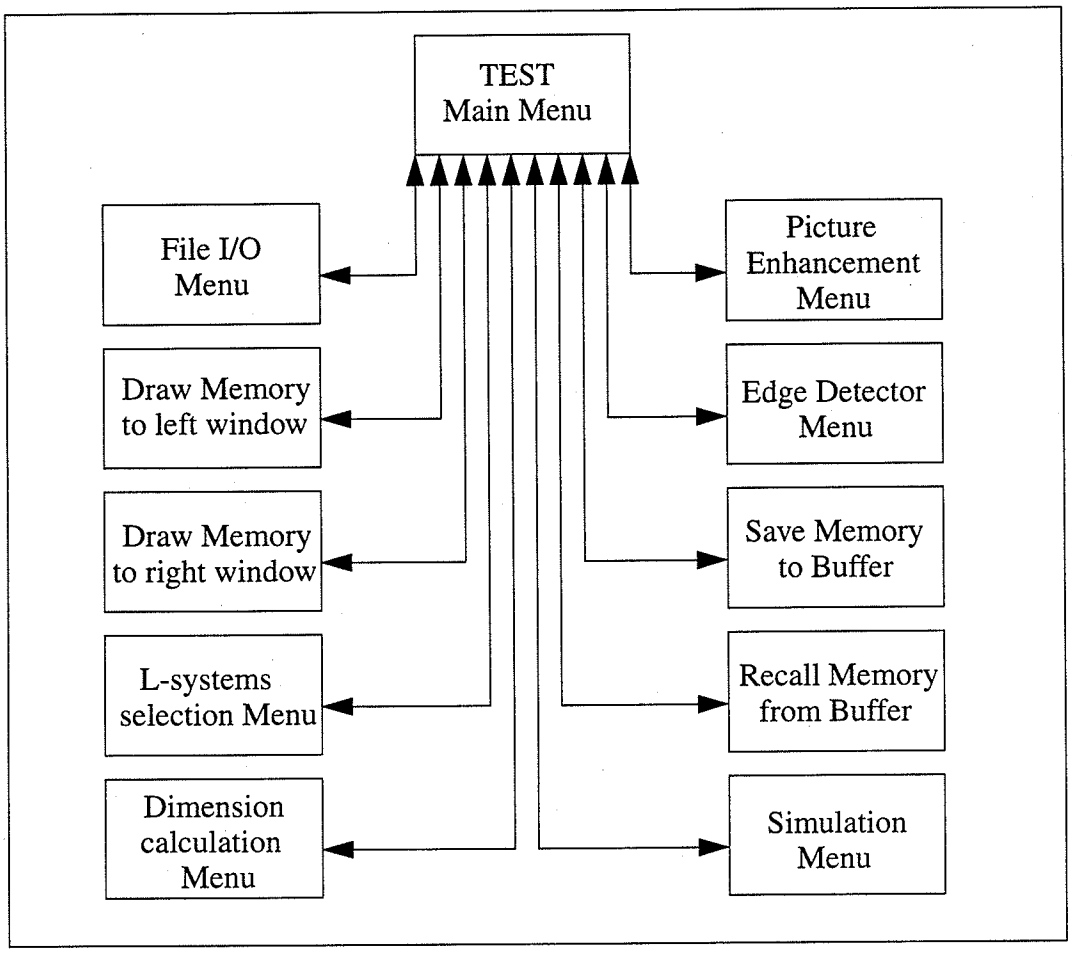


Fig. 3.3. TEST program main menu structure.

PICTURE "C" structure used for all images. Two file output routines are provided. The first writes the image in the global PICTURE structure to a file named by the user. The second takes the top two display windows on the computer screen and dumps the image to a file named by the user. To allow identification of the file, two text routines are also included to read and write to a character string that is included when writing files to disk. This menu can be seen in Fig. 3.4.

3.3.3 L-systems Sub-menu

Next, routines to produce fractals with known dimension were required. Investigation indicated the simplest method to accurately and quickly create these fractals was with L-systems. The self similarity dimension can be calculated for these fractals and this calculation is given later

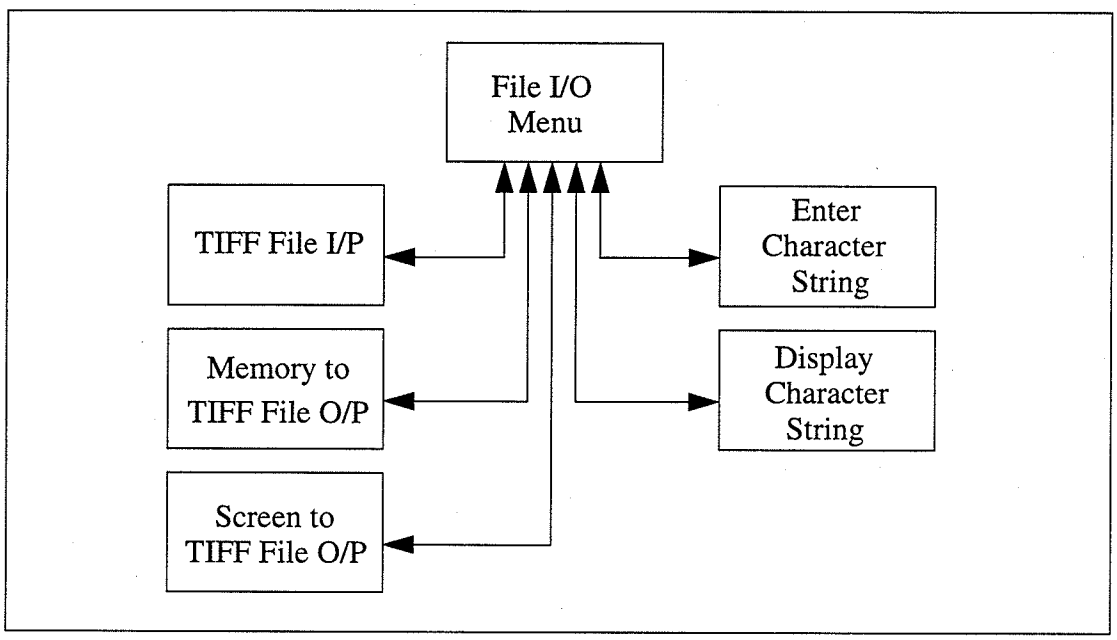


Fig. 3.4. File I/O sub-menu.

in Chapter 4 [Edga90]. A large selection of L-systems fractals are required for subsequent testing so this menu is reserved for these functions. The L-systems selection menu has ten selections. The first nine are the L-systems created test fractals used extensively in this thesis to measure and test the characteristics of the analysis programs. They are all self avoiding fractals. The tenth entry is a test fractal that is not self avoiding. The measured dimension of this fractal does not equal its calculated value. For this reason, this fractal is included as a reminder that care must always be exercised when studying fractals. The L-systems menu is detailed in Fig. 3.5.

The most important part of writing a computer program is testing and verifying its operation and accuracy. This becomes the paradox "Which comes first: the chicken or the egg" problem in this case. Test fractals are required to verify that the dimension calculating routines work and the dimension calculating routines are required to verify that the fractal generation routines work. It helps that this area is relatively simple and checking intermediate results produced by each of the test fractal subroutines was sufficient to verify correct operation.

The L-systems generator subroutine called `LENGINE` was created to generate all the test fractals. This routine must be written to minimize errors introduced by computer round-off and

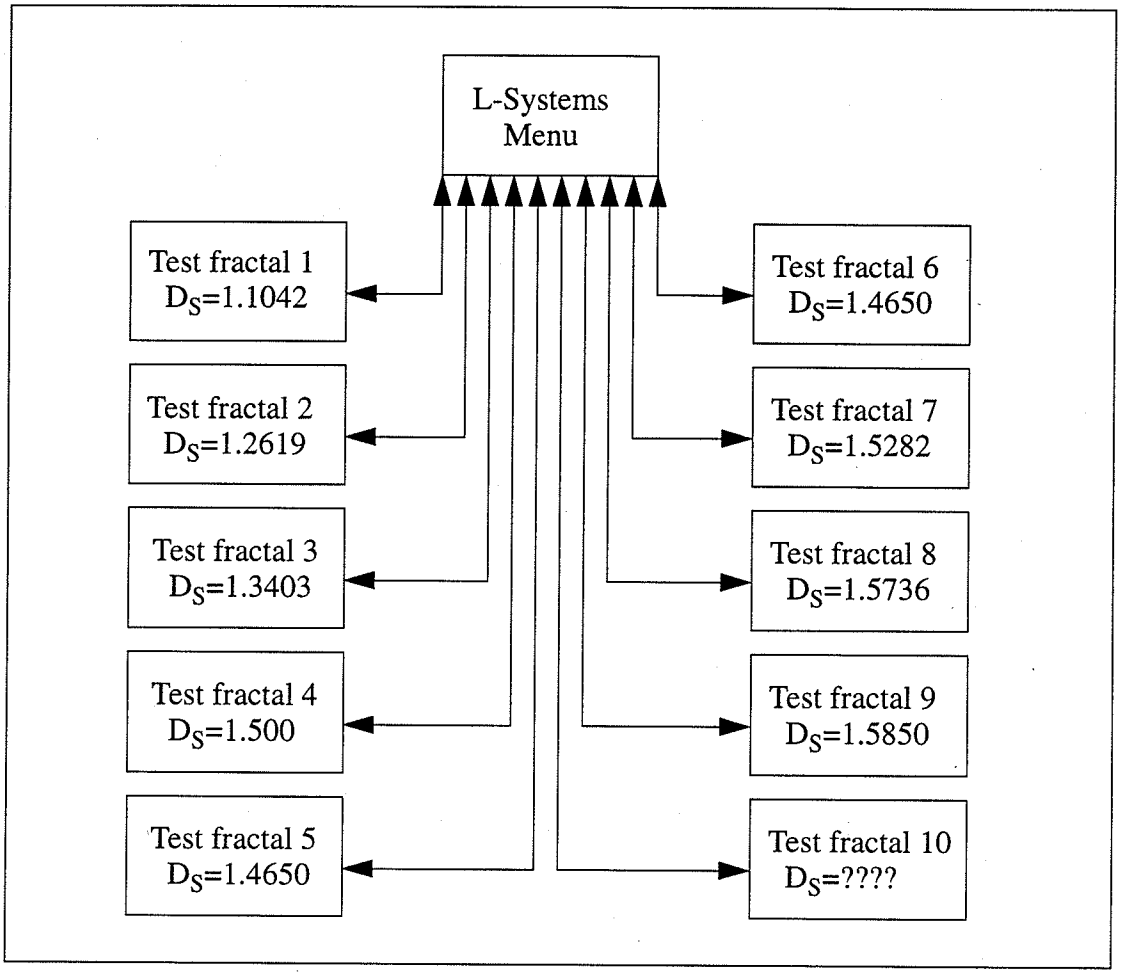


Fig. 3.5. L-systems test fractal sub-menu.

truncation of numbers. Once the generator subroutine was complete, each L-system routine used to generate the selected test fractals could be written and tested.

These routines were to be written using the global `PICTURE` structure for output of the results. Thus all the other routines such as the display and storage routines could be used to inspect the results. Details of these routines is covered in Chapter 4.

3.3.4 Dimension Sub-menu

The analysis used four routines for calculating dimension. Details for the simple box dimension, dilation dimension, full dilation dimension, and correlation dimension subroutines are covered in Chapter 4. The subroutines simple box dimension, dilation dimension, full dilation

dimension, correlation dimension calculating routines plus an image reduction algorithm are all found under this menu. These routines calculate the dimension of the image in the global PICTURE structure or reduce the fractal size by using a user selected testing mesh. The REDUCE routine tests the fractal at a scale giving a tested image smaller than the original. This routine is discussed later as well. Graphical results are then presented in the right screen window with data results given in the menu window area on the screen. The dimension is given below the graph of the data points used for the calculation of the dimension. This shows the operator how the data behaves. This menu can be found on Fig. 3.6.

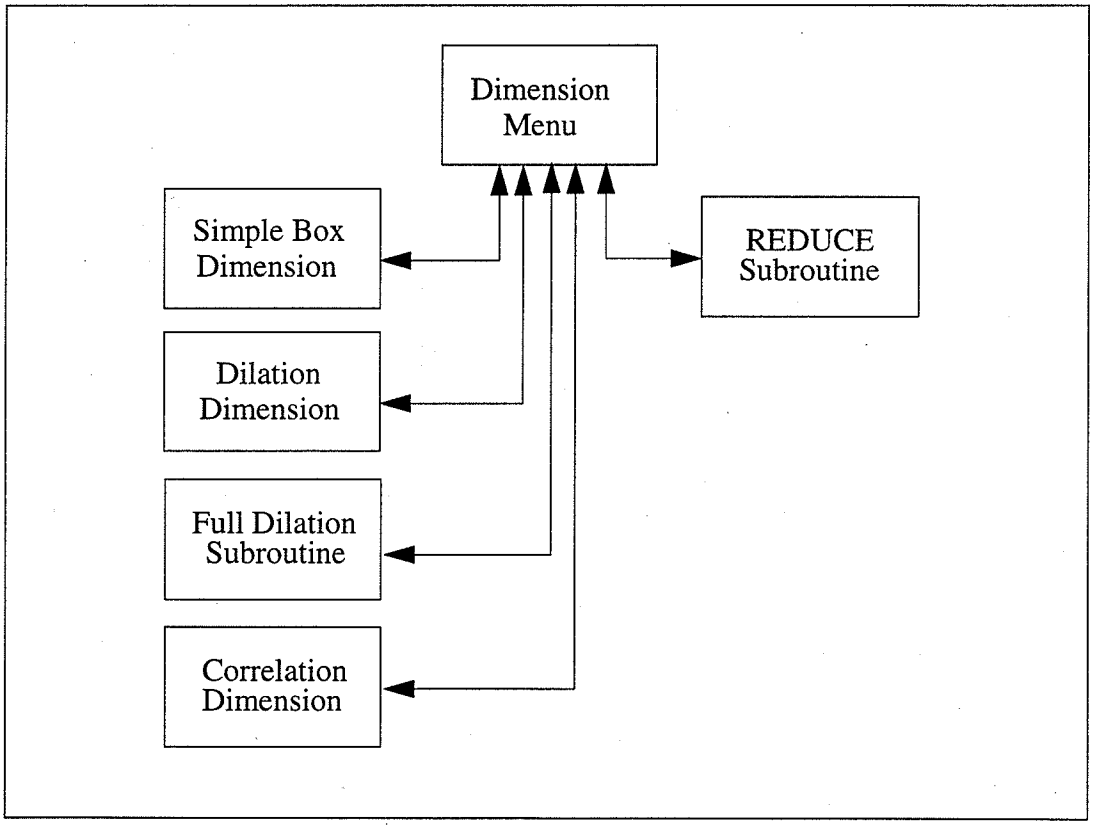


Fig. 3.6. Dimension calculation sub-menu.

3.3.5 Picture Enhancement Sub-menu

The picture enhancement sub-menu contains the routines to enhance the image. These routines include negation, histogram plot, histogram normalize, low pass, high pass, and median

filters, plus zero cut and maximum cut threshold detectors. Each routine works on the global PICTURE structure and returns the results there. This menu is seen below in Fig. 3.7. Also included under this menu is the sub-image selection subroutine. This routine allows magnification and selection of parts of the full image.

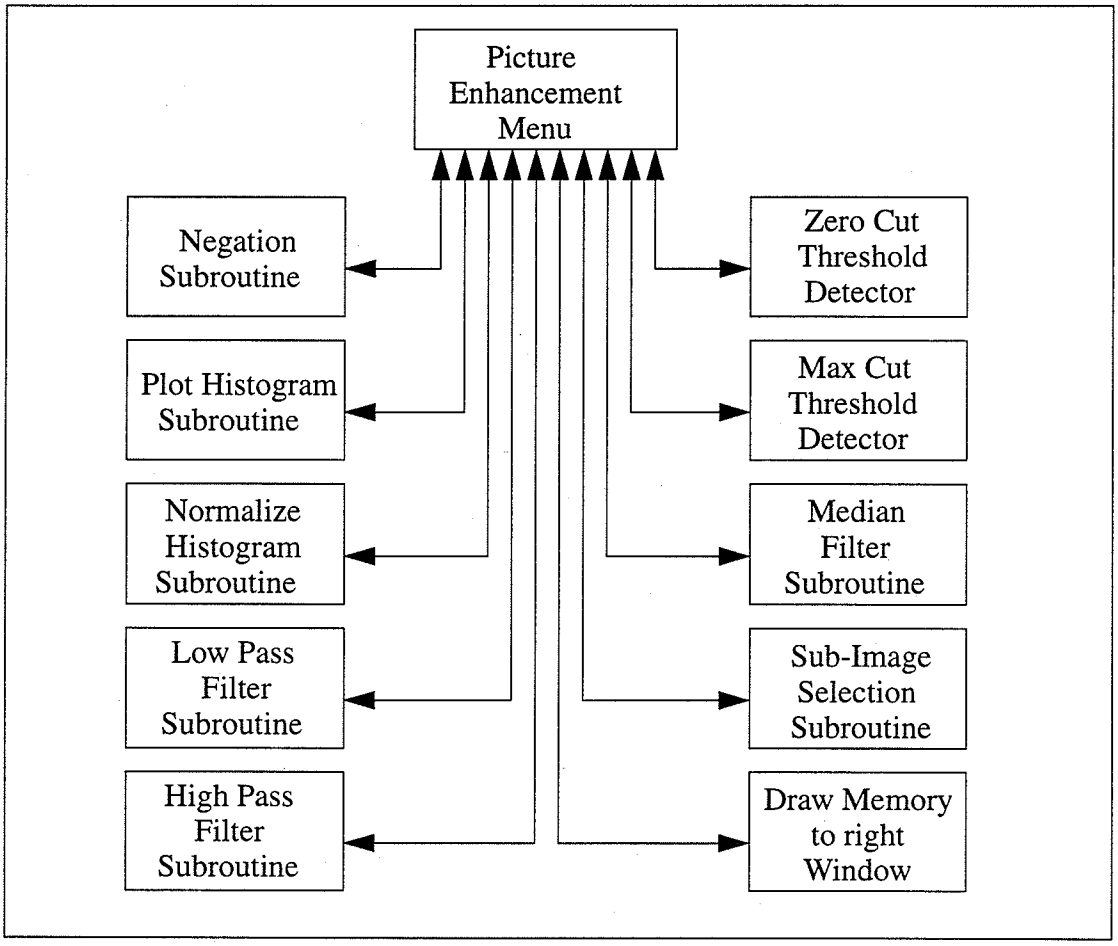


Fig. 3.7. Picture enhancement sub-menu.

3.3.6 Edge Detector Sub-menu

The Edge detector sub-menu contains all the edge detector subroutines. Generally, the larger the pixel mask used in calculating, the better the results. The routines use weights for the mask derived from exponential and gaussian distribution functions. The implementation details are given in Chapter 4 and the menu is detailed in Fig. 3.8.

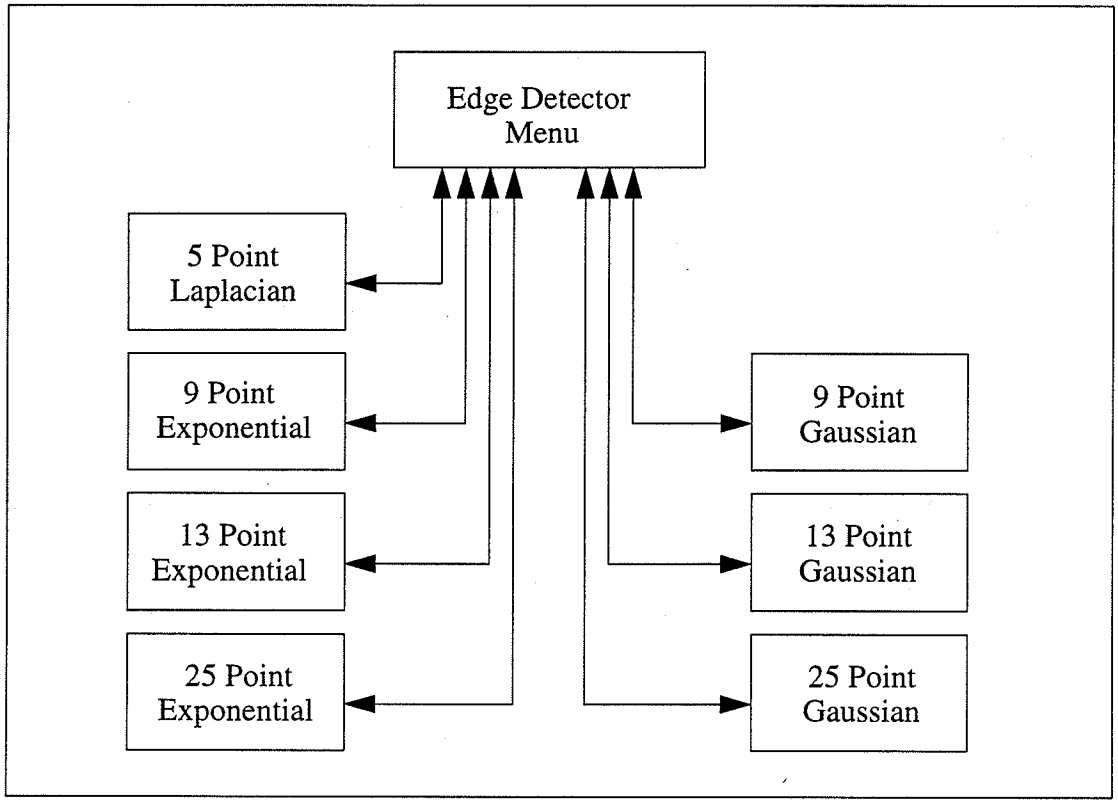


Fig. 3.8. Edge detector sub-menu.

3.3.7 Simulation Sub-menu

The simulation sub-menu contains the routines implementing the original and new stochastic models. Under this sub-menu is also included routines to test the convergence of parallel and circular fields, an animation routine, a circular field calculation routine, and a routine to set the number of local and global field iterations. The routines implementing the model are detailed later. The animation subroutine uses the greyscale value of the simulated fractal to animate in real time the growth of the simulated electrical discharge or DLA. The routine `ITERSET` sets the number of local and global iterations to be used by the models. This menu is detailed as follows in Fig. 3.9.

The stochastic model is relatively simple in concept but requires enormous amounts of computer time to perform the field calculations. Three implementations of the model were

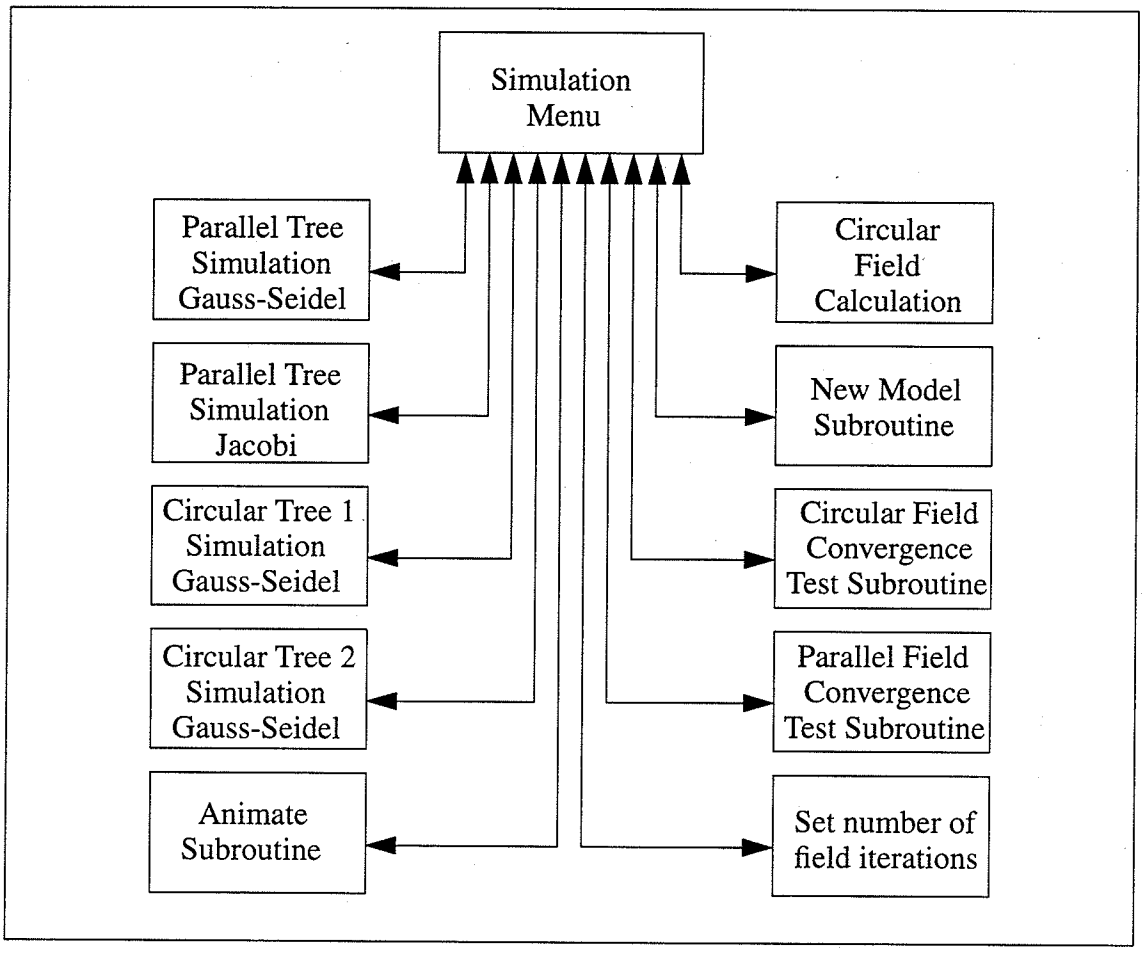


Fig. 3.9. Simulation sub-menu.

decided upon. The parallel tree simulation was introduced first since the concepts were better explained by Fukai and Sano and this configuration seemed easier at the time. This configuration was implemented using both Gauss-Seidel and Jacobi field iterations. Next, the circular field with point inner electrode configuration was to be implemented. This implementation was perfected by Niemeyer, Pietronero and Wiesmann before the work by Fukai and Sano. The last configuration was for circular inner and circular outer electrodes to match the configuration for experimental data obtained from another experimenters testing.

Producing the subroutines for these implementations of the stochastic model required producing common subroutines for field and probability calculations. The output of each of the model subroutines was to be through the global `PICTURE` structure allowing further analysis of

the results by the other routines written. Display of the field profile was deemed important. The field profile was to be displayed after the addition of each 100 pixels in the image. In order to preserve roughly when each pixel was added to the image, the intensity of the pixel added was related to the time when it was added. This only requires the user to crudely estimate the number of iterations the generation requires for completion.

The size of the images to be produced was to be made a variable so the user can select how large an image is to be created. This allows testing and verification with small images which take only tens of minutes to create. The program could also be asked to create large images as well which could take several days to complete. The larger images were to be used for analysis.

The subroutine `ANIMATE` was added to animate the growth of the simulated fractals in real time. The actual growth of the fractal takes many hours for a large image. Animating the growth at a much faster rate allows the dynamics of the growth to be observed. This routine uses the intensity of the image added during the growth as the time factor.

Also under this sub-menu is a special routine to calculate the field around an object. A fractal contained in memory can be placed into a circular field and the field iterated by the selected number of iterations to view how the field lines fall upon the image. Neat effect but not a very useful routine.

Two routines were also added to test the field convergence properties on arbitrary sizes of lattice. One iterates the parallel electrode and field case and the other the circular electrode case. Both use the Gauss-Seidel method for iterating the field calculations.

Finally, a routine called `ITERSET` was added to allow changing the default number of local and global field iterations used by the models. For simulations when the model exponent is large, more field iterations are required to keep the overall field error small.

3.4 Summary

A simple menu driven program encompassing a number of image enhancement, detection,

and creation algorithms forms the basic framework for the software implementation. By keeping all these routines together with the implementations of the stochastic model, full control over the data can be maintained simplifying the analysis procedures and reducing overall analysis time. Verification procedures for each algorithm must be carefully considered and speed enhancements made where possible.

CHAPTER 4
SOFTWARE DESIGN AND IMPLEMENTATION

Implementation of the program TEST was completed in stages. Since the author owns an IBM PC 386 clone computer, this platform was the obvious choice for programming. Also clear were the memory requirements needed to work with images up to 1200 by 1200 pixels. This size was desired so that an image of this size could be transported without problems on a floppy disk. Not many programming software packages capable of using large arrays in memory are available. The author decided to use the Watcom "C" compiler since it comes with a royalty free DOS extender providing support for large data arrays. The initial stage of software design dealt with the mechanics of inputting and displaying images. This stage is detailed as follows.

4.1 Stage One: Data Representation, File Input and Image Display

4.1.1 Internal Data Representation for Images

For speed and flexibility, the input image which can be of arbitrary size (within memory limits) must be stored as an array in memory. A data structure named `PICTURE` was designed to hold a pointer to this image as well as variables containing the discrete x and y sizes of the image. All images to be analyzed were to be 8 bit per pixel or 256 greyscale. This is more than sufficient since after all analysis is complete, the image is reduced to a white fractal on black background image. This number of greyscales allows enough flexibility during image processing.

4.1.2 File I/O - TIFF File Reader

The choice of image format to follow when reading and writing image files was a difficult one. There are many competing formats to choose from. This is also not a stagnant problem since these formats are changing continuously. The more popular image file formats are PICT (mainly on Apple computers), RASTER (mainly on SUN computers), GIF (mainly on IBM PC's), and TIFF (mainly on IBM PC's). Of these choices the file format for TIFF was the best docu-

mented and has support on all three types of computers used for this thesis, i.e. the SUN workstations, the APPLE Macintosh and the IBM PC.

Implementation of only a subset of the TIFF 6.0 standard was necessary for this thesis. The TIFF file reader was written so that should any feature needed (but not implemented) could be easily added. This was accomplished by adding entries into the `PICTURE` structure holding the internal representation of the image. Then program design simply followed the TIFF file format specification.

4.1.3 Design of Screen and User Interface

Since the program `TEST` was to be primarily a tool to investigate fractals and their properties, the display and user interface were desired to be simple but functional. The screen was divided into three functional areas. The top two thirds of the screen were devoted to the display of two images side by side, each with a small text window underneath. Under these two areas is the third area devoted to displaying menus and results. Under this third area is another small text window for display of general system and other messages. Side by side display of images was important to allow results of image enhancements to be compared subjectively. The computer screen resolution chosen was 1024x768x256 colors to allow maximum resolution for the images.

4.1.4 Display of Images

The source for images was expected to be primarily from a greyscale hand scanner. The images from this source would not be a standard size such as 512x512 or 256x256 pixels. This meant the display routines must be as flexible as possible. Since the display windows were now defined and were 510x510 pixels, all images stored in memory needed to be mapped to this area.

The display routines must enlarge images that are smaller than the display area or shrink images that are larger. The enlargement routine was obvious but the shrinking routine was not. The routine to shrink large images to the smaller view area was written to preserve details in a white on black image. The shrinking routine represents a many to one mapping from the image in

memory to the screen. In order that details are not lost in this mapping, the greatest intensity of all pixels mapped into each display window pixel is displayed. In this way, shot type noise and continuous curves will always appear correctly in the display window.

Such display routines will of course distort images which are not one to one in aspect ratio. This is unimportant since image displaying was only to allow the user to monitor what was happening to the actual image preserved correctly in memory.

Another minor problem occurs because of the limitations of the IBM PC Super VGA graphics adapter. Although 256 colors can be displayed at one time on most IBM PC's, only 64 unique greyscale levels can be represented. During program initialization, the color map of the system is altered by adding specific colors for borders and graphs plus the 64 greyscale values possible with the standard graphics adapter. The values between 192 and 255 are used for the greyscale palette thus requiring the display programs to map to these values.

4.2 Stage Two: Standard Algorithm Implementation

After the initial stages of the software implementation were in place and debugged, algorithms for enhancing the images were introduced and tested one by one. Since at this stage no good images of fractals were available (these came later), the standard images of `LENA` and `PEPPERS` from the U. of M. image database were used. Each routine worked on the internally represented image and was designed to be as modular and independent as possible. This allowed routines to be tried and removed if they served no purpose.

Having introduced these routines earlier, further detail on each routine and its function can be described. The pixel masks used in most of the routines are given below in Fig. 4.1 . The numbers 5,9,13,25 denote the number of pixels in the mask or mask size. Following is a description of

the enhancement routines implemented in software.

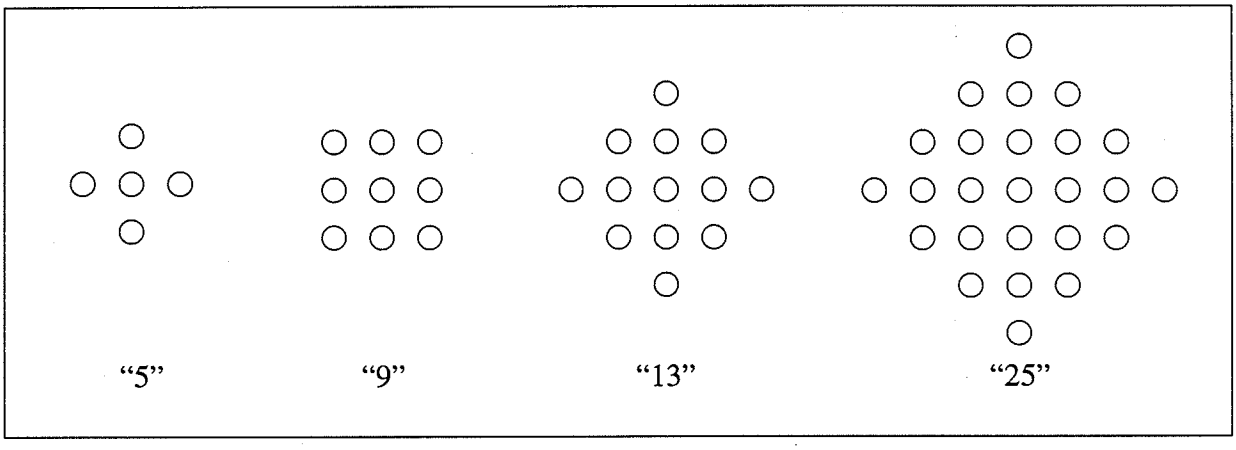


Fig. 4.1. Pixel masks used for subroutines.

4.2.1 High Pass Spatial Filter

The high pass spatial filter uses a five pixel mask for operation. This filter enhances the intensity gradient in the image but does not affect areas in the image that do not have an intensity gradient. The intensity of the center pixel in the mask is multiplied by a weight of five and the four outlying pixel intensities are then subtracted to get the result. The resulting intensity is then moved to the resulting output image. The mask is floated over the image until each pixel is recalculated. A temporary image buffer is used to hold the results and is transferred back to the original buffer when complete. The edges of the image are calculated slightly different to try to reduce the edge effects. This filter has not been found to be of much use in this work.

4.2.2 Low Pass Spatial Filter

The low pass spatial filter was implemented with the nine pixel mask size. The routine simply adds all the intensity values of the pixels in the mask together and takes the average. This average value is then given as the center pixel intensity in the resulting output image. The mask is floated over the image and the results held in a temporary buffer until complete when it overwrites the original image.

Filtering the image can also be accomplished using a two dimensional fast fourier transform but this method is not as flexible when it comes to image size.

4.2.3 Median Filter

The median filter technique is a non-linear filter technique that removes shot type noise (also called salt and pepper noise) in a picture. The more popular low pass spatial filter does not do as well on this unique type of noise. Thus the median filter will be very useful when the digitized image contains this type of noise.

The median filter takes the pixel and its immediate area (a total of 9 pixels) and puts the intensities into an array. This array is then sorted. The new output pixel intensity is selected as the middle value of the sorted array. A temporary buffer is used to hold the results until calculations are complete when the result replaces the original image. This is a non-linear filter and performs very well.

4.2.4 Image Negation Subroutine

The image negation subroutine was the simplest and most useful routine to write. The intensity values in each image are 8-bit or have a value between 0 and 255. To take the negative of the image the routine simply takes the original intensity value and subtracts it from 255. The result then becomes the resulting intensity in the output image. Since the images are best displayed on the computer screen as white on black and printing of images best as black on white, this routine was used extensively between displaying and printing the images. The calculation is done in place without the need of a temporary buffer.

4.2.5 Laplacian Edge Detectors

The Laplacian edge detectors use a pixel mask which is floated over the image during the calculation of the edges. They function by enhancing the intensity gradient. The intensity derivative around the center pixel is approximated using either the rate of decay of an exponential or gaussian curve.

The exponential and the gaussian are debated to be the best choices on philosophical grounds. The exponential is related to the way light is measured and how the eye is believed to function. The gaussian distribution is used in signal analysis and comes from probability theory.

All the mask sized detailed earlier were implemented for the edge detectors. The value of each pixel in the mask is multiplied by the weights given in Table 4.1 . The sum of all the weighted pixels in the mask is then divided by the divisor to maintain the same range for all cases. The pixel intensity in the output image is then given this value. The edges of the image the mask cannot fully reach are simply set to zero. This causes the least impact on later analysis for this application. In this fashion, each pixel in the output image is processed by shifting the mask across the input image. A temporary buffer is employed to hold the results until completion when the new image replaces the old.

The equations given below are used for the calculation of the pixel weights.

Exponential: Gaussian:

$$y = e^{-2x} \qquad y = \frac{1}{\sqrt{2\pi}} e^{(-x^2)/2} \tag{4.1}$$

Table 4.1. Outer pixel weights.

Pixel Distance (from center)	Exponential Value	Exponential Weight	Gaussian Value	Gaussian Weight
1.0	0.1353	-1000	0.2420	-1000
1.414	0.0591	-437	0.1468	-607
2.0	0.0183	-135	0.0540	-223
2.236	0.0114	-84	0.0328	-136
3.0	0.0025	-18	0.0044	-18

Table 4.2. Center pixel weights and divisor for exponential masks.

Mask Size	5	9	13	25
Center Weight	4000	5748	6288	7032
Divisor	400	575	629	703

Table 4.3. Center pixel weights and divisor for gaussian masks.

Mask Size	5	9	13	25
Center Weight	4000	6428	7320	8480
Divisor	400	643	732	848

4.3 Stage Three: L-systems Generator and Test Fractal Generation

4.3.1 L-systems Generator

The L-systems generator subroutine went through many revisions to improve accuracy. The routine uses three software data stacks for saving length, branch and direction information when executing the recursive algorithm used. To improve the accuracy of the position summations, software extensions to the position counters were employed to increase the dynamic range of the summations. Since the type of error this algorithm will suffer from most is numerical truncation, the error in generation can be calculated by calculating the distance between the calculated end position of the generation and the actual position obtained at the end of the generation. The error was observed to be less than less than one-one thousandth of a pixel even when 10^8 line segments were used in generation.

The question of how many steps or how many iterations are required in order to produce a true representation of the fractal must be asked. This is a careful balancing act. If more iterations and subsequently more line segments were used, the resulting image will contain more pixels and be a better representation than the same fractal created with fewer iterations. But also, the level of error introduced into the image will be greater due to limitations of the computer. The time required to produce these images goes up exponentially as well.

When generating the test fractals, how many pixels were used in construction is displayed after completion of the generation. Watching how this number varies with the number of iterations as well as watching how the dimension of the image changes with iteration level led to the number of 10^8 line segments being used for all test images. It is important to note that the effect of this type of error is difficult to predict.

The commands implemented in the L-systems generator interpreter are given below.

1) + increase angle by angle 0

2) - decrease angle by angle 0

3) d increase angle by angle 1

4) e decrease angle by angle 1

5) f forward or first rule set

6) l left hand or first rule set

7) r right hand or second rule set

8) b branch fractal here

9) c continue from last branch

10) s short segment = length * mult2

These commands are not the only commands that could be implemented. More commands would allow a greater number of unique fractals to be generated.

The most important problem is ensuring that the fractals do not cross during generation. If the fractal overlaps then its measured fractal dimension will not equal its calculated fractal dimension. Ensuring that the fractal does not overlap consisted of magnifying the fractal after generation to see if the branches overlap and modifying the rules if necessary. This is a severe limitation for L-systems and for iterated function systems generators as well. Adding stochastic properties to these fractals by randomizing the changes in angle makes ensuring self avoidance impossible. This is one reason for the use of the stochastic model used for simulating diffusion limited aggregates or for simulating electrical discharges.

Each of the fractal generation subroutines simply allocates memory for the data stacks, initializes the rule sets and angles required (using a "C" structure) and calls the L-systems generator interpreter. The interpreter then calls itself recursively based upon the number of iterations

required. The number of iterations required is entered by the user so that how the fractal converges can also be tested. If the user does not enter an iteration value, the computer enters the default value listed on the screen. The error in generation is given to the user at after the generation is complete so the user has a feeling how accurate the generation was.

One of the mathematical methods for determining the dimension of a fractal is to identify all the transformations used in the fractals construction and use these to calculate the similarity dimension D_S [Edga90]. The dimension of the L-systems generated test fractals can be calculated by solving for s in a simple equation. This equation uses the scaling and number of line segments added at each stage of the generation procedure to obtain the similarity dimension for the fractal. For these self similar fractals the similarity dimension equals the Hausdorff-Besicovitch dimension. For some of the test images, the similarity dimension can be solved for analytically. For the others, numerical solution is required. The author used the symbolic math program

MAPLE to solve the equations requiring numerical solution. The solution of the dimensions of the test fractals follows.

a) $2\left(\frac{1}{3}\right)^s + 2\left(\frac{1}{3\sqrt{2}}\right)^s = 1$ s = 1.1042

b) $4\left(\frac{1}{3}\right)^s = 1 \quad s = \frac{\log(4)}{\log(3)}$ s = 1.2619

c) $3\left(\frac{1}{3}\right)^s + 2\left(\frac{1}{4}\right)^s = 1$ s = 1.3403

d) $8\left(\frac{1}{4}\right)^s = 1 \quad s = \frac{\log(8)}{\log(4)}$ s = 1.500

e) $5\left(\frac{1}{3}\right)^s = 1 \quad s = \frac{\log(5)}{\log(3)}$ s = 1.4650

f) $5\left(\frac{1}{3}\right)^s = 1 \quad s = \frac{\log(5)}{\log(3)}$ s = 1.4650

g) $6\left(\frac{1}{4}\right)^s + 4\left(\frac{0.7}{4}\right)^s = 1$ s = 1.5282

h) $6\left(\frac{1}{4}\right)^s + 8\left(\frac{0.52}{4}\right)^s = 1$ s = 1.5736

i) $3\left(\frac{1}{2}\right)^s = 1 \quad s = \frac{\log(3)}{\log(2)}$ s = 1.5850

The L-systems generator subroutine called LENGINE is a recursive routine. The flow-chart for this routine can be found in the Fig. 4.2 . The routine is called with four variables. These variables are the command string, the iteration depth, a pointer to a PICTURE structure, and a pointer to an LMAP structure. The subroutine then creates the image in memory following the rules and variables in the LMAP structure. The displacement commands are the f,l,r,and s commands detailed earlier. These commands invoke the LENGINE routine recursively with an iteration number one less than the current value if the iteration level is greater than zero. All the other commands are executed the same at all values of iteration number. This method is surprisingly fast due to efficient subroutine calling provided by the "C" language.

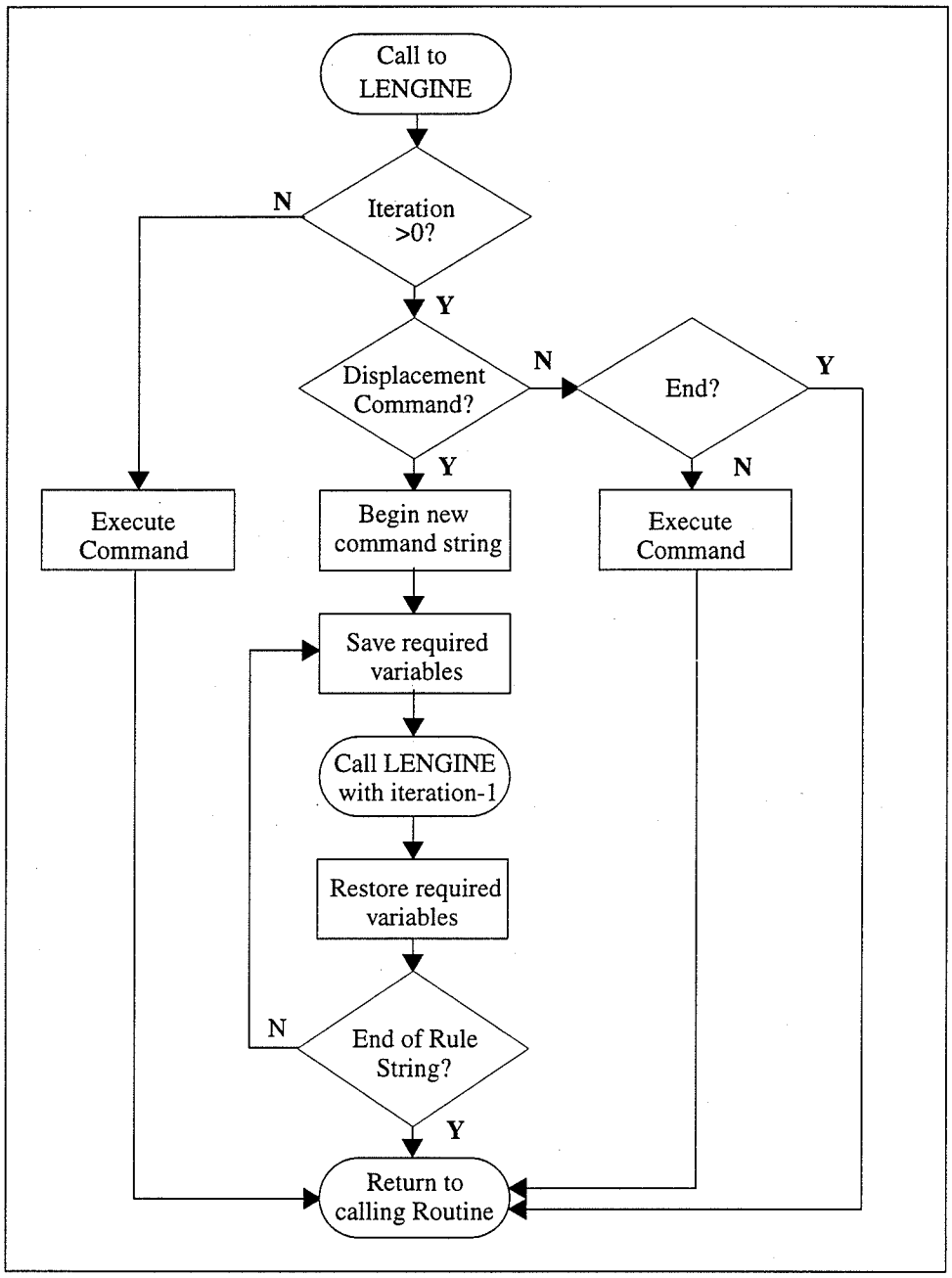


Fig. 4.2. LENGINE flowchart.

4.3.2 L-systems Test Fractals

Each of the L-systems generated test fractals was written as a separate subroutine thus following the philosophy of keeping the program as modular as possible. The size of each of the images created was fixed at 1024 by 1024 pixels. The actual fractal size was chosen to be 1000

pixels from starting point to end point so that it would be surrounded with some empty space. Ten fractal subroutines were written; nine of which were used as the test fractals and the tenth given as an example why self avoidance in generation is so important. The next sections cover the rules and scaling factors used to create each of these ten fractals. The importance of these fractals cannot be overstated. With the images created by these subroutines, the performance of many of the other routines could be verified.

The procedure used in each of the fractal generating subroutines is to allocate memory for the image, initialize the software stack pointers and the angle arrays, initialize the L-systems `LMAP` structure with the rule and scaling data, and finally calling the L-systems generator subroutine to produce the image. Following is a flow chart describing the operation of each of the L-systems fractal generation subroutines. After the subroutine flowchart are the actual rules and scales used for each fractal.

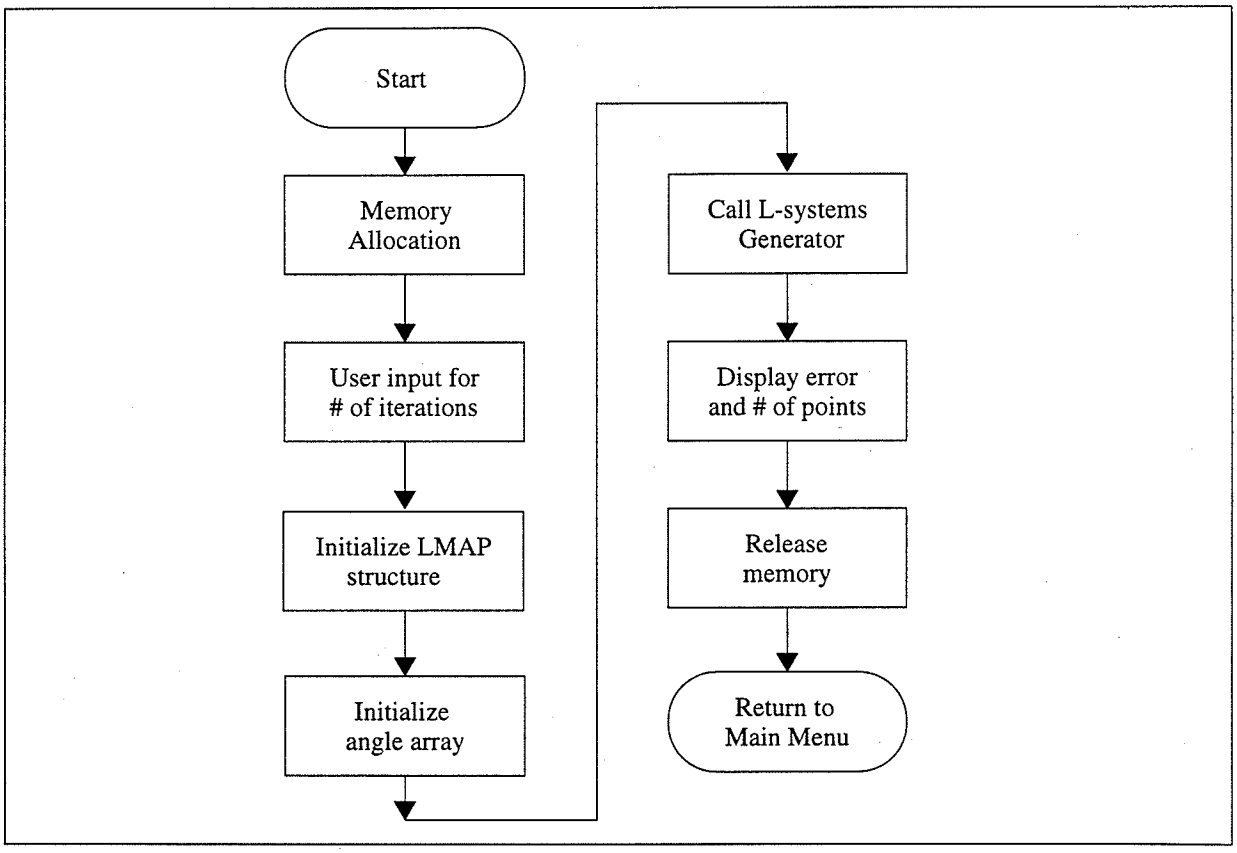


Fig. 4.3. L-systems test fractal subroutines.

4.3.2.1 (a) Koch Type Curve 1

The first and lowest dimension image of the L-System created fractals uses a similar construction to the famous true Koch Curve covered next. The difference in the construction of this curve lies in the sizes and angles of each of the legs in the construction. The parameters used for the construction of this curve are as follows:

Rule String 0: `f+s--s+f`

Angle 0: 45 degrees

Multiplier 1: 1/3

Multiplier 2: sqrt(1/2)

Starting Length: 1000.0

Starting Position: (10,512)

With these parameters, calling the L-systems generator will create test fractal (a) displayed in Fig. 4.4 .

4.3.2.2 (b) The True Koch Curve

This curve follows the original construction introduced by Helge Von Koch in 1904. This curve is simpler than the previous curve and its construction consists of the following instructions:

Rule String 0: `f+f--f+f`

Angle 0: 60 degrees

Multiplier 1: 1/3

Starting Length: 1000.0

Starting Position: (10,512)

The True Koch Curve fractal is created when called with these parameters.

4.3.2.3 *(c) Koch Type Curve 3*

The Koch type curves are the simplest curves to create since the curve is essentially a single twisted line with no branching. This type of construction is still, however, very flexible. This curve uses five line segments in construction as opposed to four for the previous two curves. The rules used by the L-systems interpreter are as follows:

Rule String 0:	`f-s+f+s-f`
Angle 0:	-90 degrees
Multiplier 1:	1/3
Multiplier 2:	3/4
Starting Length:	1000.0
Starting Position:	(10,512)

4.3.2.4 *(d) Koch 1.5 Type Curve 4*

This is another standard curve found in many books. It consists of eight line segments inserted per segment per iteration. The line in the limit becomes a fuzzy blur indicating the complexity of this curve. This is also a very simple curve to create. The rules used for creation are as follows:

Rule String 0:	`f+f-f-ff+f+f-f`
Angle 0:	-90 degrees
Multiplier 1:	1/4
Starting Length:	1000.0
Starting Position:	(10,512)

4.3.2.5 *(e) Tree Type Curve 1*

The tree type curves represent a more complex family of L-System curves that incorporate branching as well as multiple scaling multipliers. This curve uses only one scaling multiplier and is the simplest of the tree type curves. This curve can also be found in the book "Fractals for the Classroom" [PeJS92]. The rules for construction are as follows:

Rule String 0:	`fb+fcfb-fcf`
Angle 0:	22.5 degrees
Starting angle:	-90 degrees
Multiplier 1:	1/3
Starting Length:	1000.0
Starting Position:	(512,1010)

4.3.2.6 (f) Tree Type Curve 2

This curve can obviously be generated in several ways. If the Koch type continuous curve type construction were employed, the calculated dimension would not equal the dimension of the image since parts of the image would overlap. By utilizing the branching rules in the L-System interpreter however, the construction does not overlap thus ensuring the calculated dimension and the image dimension will be equal. The rules for this fractal are as follows:

Rule String 0:	`fb-fcfb+fcf`
Angle 0:	90degrees
Starting angle:	-90 degrees
Multiplier 1:	1/3
Starting Length:	1000.0
Starting Position:	(512,1010)

4.3.2.7 (g) Tree Type Curve 3

This fractal is an example of how complex fractals created using multiple scaling values and branching can be. Notice that symmetry is not required when constructing the fractal. The following rules and scaling factors apply to this fractal.

Rule String 0:	`fb-fscfb+sb+scscfb-fcf`
Angle 0:	45degrees
Starting angle:	-90 degrees
Multiplier 1:	1/4
Multiplier 2:	0.70
Starting Length:	1000.0
Starting Position:	(512,1010)

4.3.2.8 (h) Tree Type Curve 4

As the construction becomes more complex, it becomes more and more difficult to ensure that the branches in the fractal do not overlap. This tree represents the most complex fractal the author could produce that did not overlap. Although fractals of higher dimension are definitely possible, their construction becomes much more difficult since self avoidance must be ensured by careful rule selection. The following rules apply for this fractal tree.

Rule String 0:	`fb+sfcb-sfcfb+sscb-sscfb+scb-scf`
Angle 0:	55degrees
Starting angle:	-90 degrees
Multiplier 1:	1/4
Multiplier 2:	0.520
Starting Length:	1000.0
Starting Position:	(512,1010)

4.3.2.9 *(i) Sierpinski Gasket*

This fractal is a standard fractal illustrated in nearly every textbook and has the highest dimension of all the test fractals in this group. The author has tried but failed to develop a fractal with greater dimension. The implementation of this fractal was with the Sierpinski Arrowhead construction technique. This construction uses two rule strings thus setting the mechanics of construction apart from all the other fractals created in this group.

Rule String 0:	`-l+r+l-`
Rule String 1:	`+r-l-r+`
Angle 0:	-60degrees
Multiplier 1:	1/2
Starting Length:	1000.0
Starting Position:	(10,950)

4.3.2.10 *Koch Type Curve with Overlap*

This curve was included in the program TEST as an example of why self avoidance of the fractal construction is required to ensure that the measured and calculated dimensions of the fractal match. The construction of this curve is very similar to curve (c) detailed above.

Rule String 0:	`f-f+f+f-f`
Angle 0:	-90 degrees
Multiplier 1:	1/3
Starting Length:	1000.0
Starting Position:	(10,512)

With the test fractals fully described above, their actual appearance can be viewed below in Fig. 4.4 . The importance of these test fractals cannot be over emphasized. They have served as test functions to test and verify most of the algorithms used in this thesis.

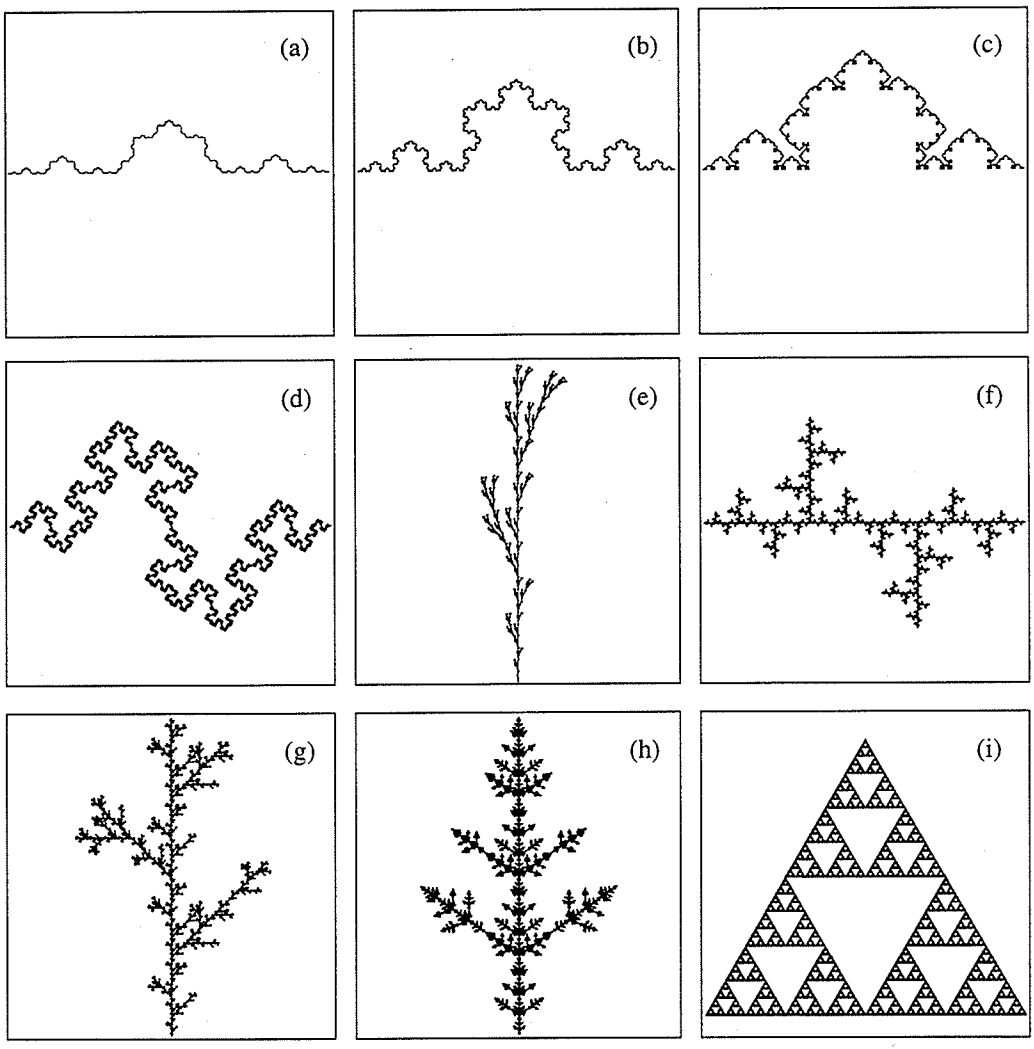


Fig. 4.4. L-systems generated test fractals.

4.4 Stage Four: Dimension Analysis Subroutine Implementation

Once some of the fractal images created by the L-systems generator were available, the routines for calculating dimension could be written. Initially, it was unclear how accurate the creation of the images needed to be or how sensitive the dimension calculating algorithms were. It soon became apparent however that much more accuracy was required with the fractal images

than was suspected. This lead to changes both to the L-systems generator and to the dimension calculating routines in order to minimize the effects of round-off and truncation inherent with all computer calculations. Next, the implementation of the dimension analysis routines is described.

4.4.1 Simple Box Dimension Subroutine

The picture enhancement routines ultimately end up with a white on black image. This processed image is used by the simple box dimension calculation routine. The preprocessed image is then broken down into boxes of 1x1, 2x2, 3x3, 5x5, 6x6, 10x10, and 15x15 pixel boxes. Thus the image is being tested over a 15:1 scale range. A larger scale range may yield more accurate results for large images but accuracy on small images would suffer. If the image is small, edge effects will also affect the accuracy of the routine. The 15:1 scale range was decided upon as a good compromise between accuracy on large images (where a greater range is desired) and small images (where a smaller range is more useful).

The box dimension routine essentially counts the number of δ-mesh boxes that intersect the fractal. In order that the same area is used for calculation at each box size, the size of the fractal is rounded down in the discrete x and y directions to a multiple of 30 pixels. To allow the user to monitor the progress of the calculations, the number of boxes counted and the size of boxes is displayed. Once the boxes are counted and the log is taken, the results are fitted using the method of least squares to a line. The slope of this line is the dimension and is displayed to the user. A plot of the data is also given in the right screen window as well. The flow chart for the box dimen-

sion algorithm can be seen in Fig. 4.5 .

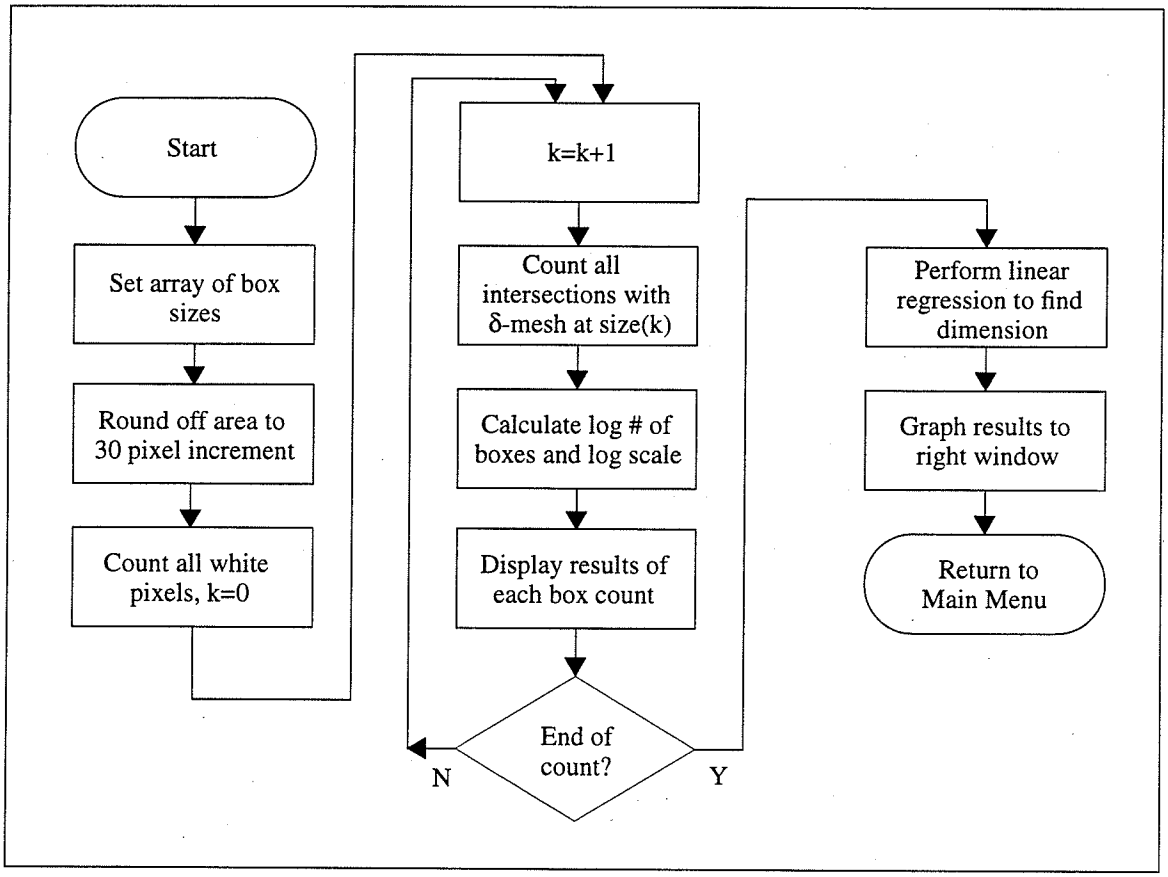


Fig. 4.5. Flowchart of box dimension algorithm.

4.4.2 Dilation Dimension Subroutine

The dilation dimension subroutine is very similar to that of the box dimension subroutine. Implementing the dilation dimension routine is only a little more difficult than implementing the box dimension algorithm. The dilation routine only requires a dilation buffer of one largest dilation size minus one larger than the area of the original image being processed in both the x and y directions. For the test fractals, the actual dilation buffer used was 1024+15 =1039x1039 pixels. The buffer is cleared before each dilation (not necessary! but only takes a little time).

To facilitate ease of programming, the dilation sizes selected are entered into an dilation size array called `dilationsize`. Next the image size is rounded off or truncated to a multiple of 30 pixels in both the discrete x and y directions. This ensures that the area of the image being

processed is the same for all the dilation values. Next comes the dilation and counting part of the routine.

The first stage simply counts the number of white pixels in the rounded image size since no dilation is required for the first step. The images analyzed in Fig. 4.4 were created as white fractals on a black background simply because this looked better when displaying them on the computer screen. For each subsequent stage, each pixel in the rounded image size is dilated into the buffer by the dilation size. The size of dilation is recalled from the `dilationsize` array mentioned earlier. This array simplifies coding the dilation loop. Dilation is from the location (i,j) position of the white pixel in the image to the (i+dilationsize,j+dilationsize) position in the dilation buffer. Dilating to the right and down from the pixel position includes an offset which is common to all the pixels and therefore is of no consequence.

The number of pixels is found by simply counting the number of white pixels in the dilation buffer. The log of the number of pixels divided by the area of the dilation and the log of the dilation size gives each data point. In similar fashion to the simple box dimension implementation, the intermediate results are flashed to the user as the counting progresses with the final result again graphed in the right screen window. The L-systems generated test fractals were used for verification of the dimension routines. The dimension is then found by the slope of the least

squares fit to the data points thus obtained to a line using the standard linear regression formula. The flow chart in Fig. 4.5 shows the logic flow for the dilation dimension routine.

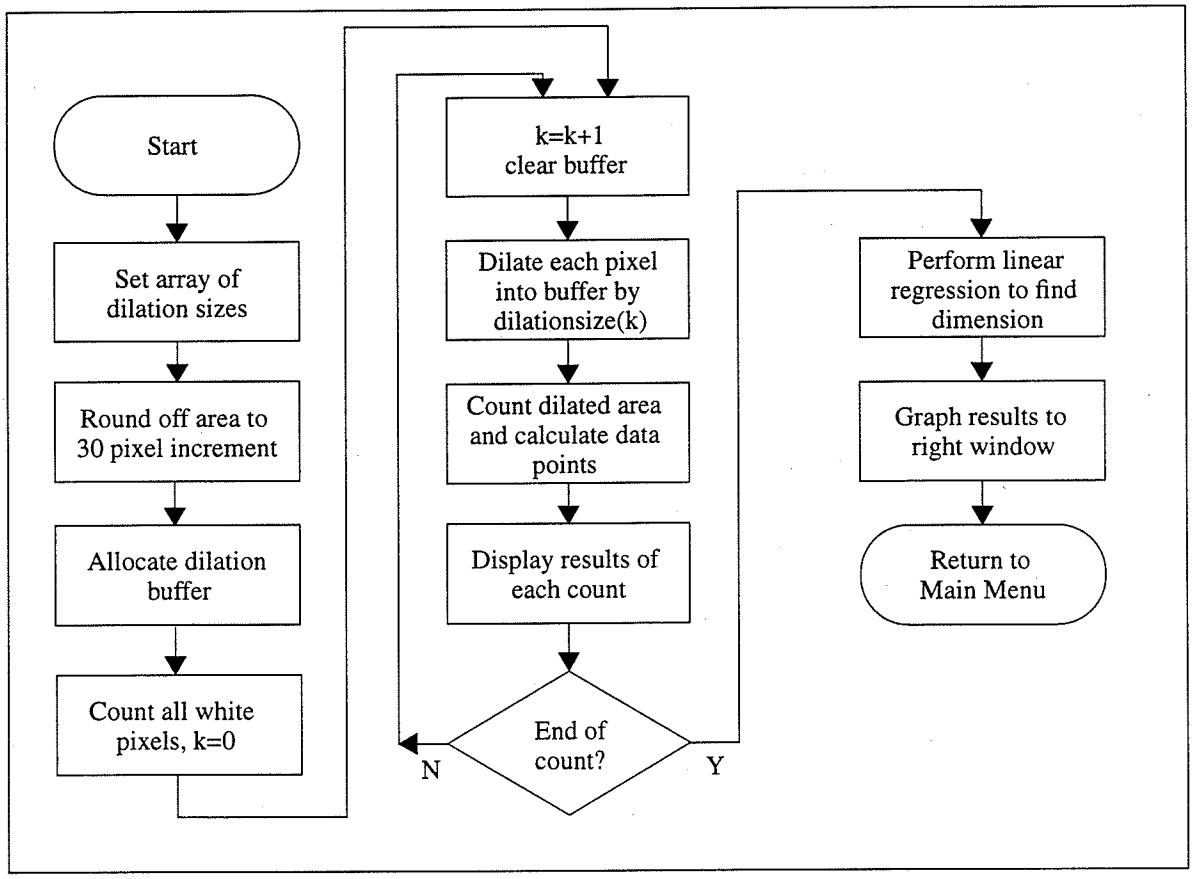


Fig. 4.6. Flowchart of dilation dimension algorithm.

4.4.3 Full Dilation Dimension Subroutine

Next, the full dilation dimension subroutine was written. This routine is an extension of the dilation dimension subroutine mentioned above. The dilation values of 1,2,3,4,6,8, 12,16,24,32,48,64,96,128,192, and 256 were used. These values gave data points that were evenly spaced along the full dilation dimension curve. End correction was employed with this routine. The dilation buffer is also larger by the image size plus 255 in each direction.

Once the data for the dilation curve was obtained, it was plotted to the right screen window for observation. This data was then fitted to a cubic polynomial of the form x=P(y) using the Moore-Penrose generalized matrix inverse method of least squares fit and by using the standard

linear regression formula. The cubic polynomial with the critical value of dimension (slope) and point of maximum dimension were all displayed in the menu area of the screen. Verification of the routine was performed by comparing the numbers produced during dilation to the same numbers produced using the dilation dimension subroutine and by tracing the program and calculating values in the matrices with a hand calculator. Matrix inversion was performed using the matrix inversion routines from "Numerical Recipes in C" used successfully in past work.

4.4.4 Correlation Dimension Subroutine

Finally, the correlation dimension algorithm was implemented and tested. The correlation dimension algorithm measures the dimension by testing the distribution of the fractal from each point in the fractal. This is unlike an implementation of the mass dimension which must be given the center of the fractal to start from and only measures the distribution of the fractal from this point. The mass dimension is only useful for fractals which are radially self affine.

Implementing the correlation dimension begins with calculating two arrays. The first array gives the total number of pixels within a given radius. This array is calculated in two steps. The first step calculates the radius from a reference pixel and increments a counter array. The count of all pixels at each radius value is formed. This array is then integrated in place to obtain the total number of pixels within a given radius array.

In similar fashion, the number of fractal pixels at each radius is then determined. This is accomplished by first creating two position arrays holding the position of each pixel in the fractal. The distance from each pixel to each pixel is determined and the count of all the pixels with a Euclidean distance r from each other is determined. This count array is then integrated to obtain the total number of pixels with distance r from every other pixel. This array is then normalized by dividing by N. Then the array is divided by the first array to obtain the correlation

function desired. Symmetry is used to reduce computation time for the calculation of these arrays. The flowchart for this routine is seen in Fig. 4.7 , page 66.

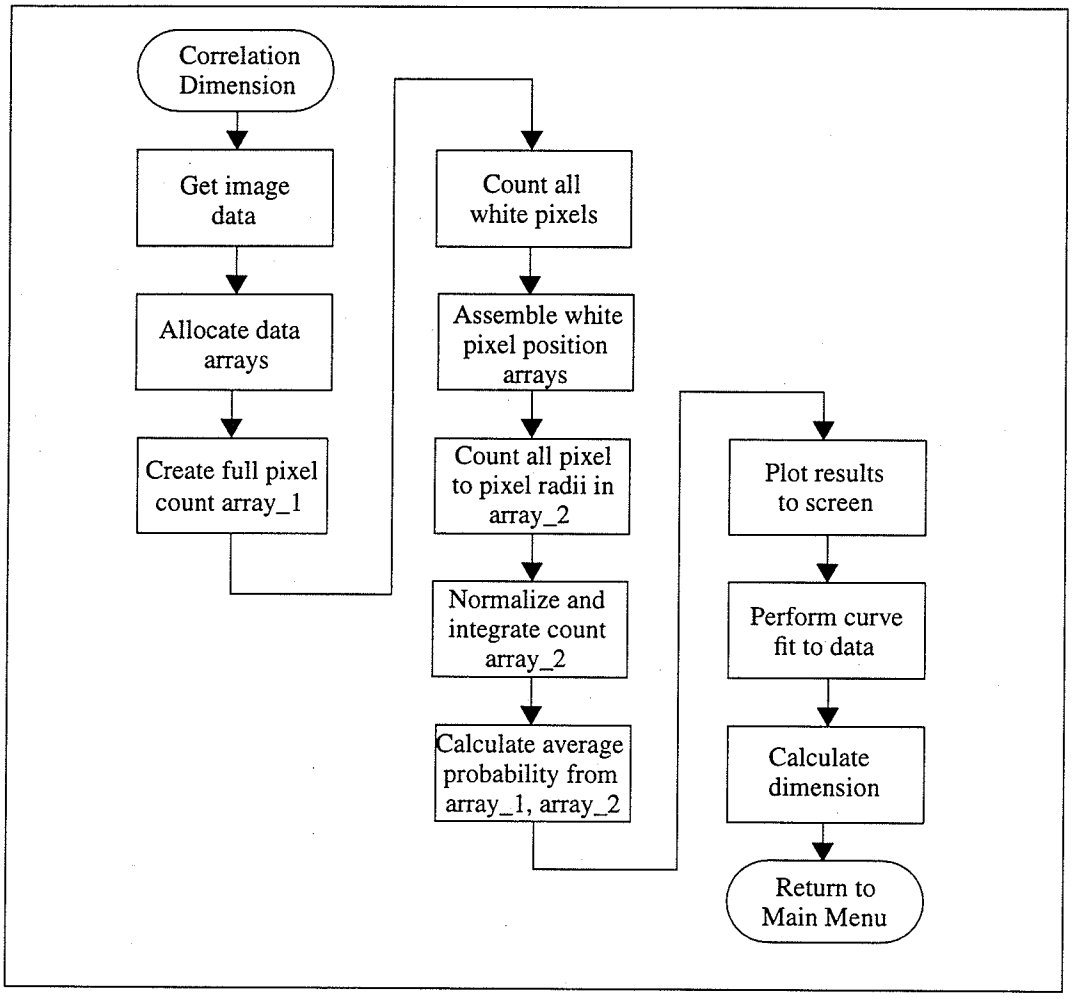


Fig. 4.7. Correlation dimension flowchart.

From the correlation function array, the output data array is determined by selecting the same points as used for the full dilation dimension subroutine. The log of the probability a point is occupied at radius 1,2,3,4,6,8,12,16,24,32,48,64,96,128,192 and 256 verses the log of the radius is then plotted on the right window screen. The data is then curve fitted using a least squares curve fit to both linear (using a linear regression formula) and cubic polynomials (using the Moore-Penrose method used also for the full dilation dimension subroutine). By using the same points as used for the full dilation dimension subroutine, the relative accuracies of these two routines could be better compared.

The linear curve fit is done to the points at radius 3,4,6,8,12,16,24,32,48, and 96 pixels. These points comprise the most linear region on fractals with sizes greater than about 250x250 pixels. The cubic curve fit is done with all points. The resulting dimension is calculated for both the linear and cubic curve fitted data. Verification of this routine was accomplished by comparing the results with the results from previous authors [WiSa81].

4.5 Stage Five: Implementation of the Stochastic Model

Once the analysis tools were in place, implementation of the stochastic model could proceed. The case of simulated fractal growth between two parallel plates was implemented first. This is not the simplest case but was the most clearly documented by the authors J. Fukai and N. Sano in their paper [FuSa87]. Two implementations, one using Gauss-Seidel field iterations and the other using Jacobi field iterations, were produced. The Jacobi field iteration implementation was used to check if any strange effects were happening with the Gauss-Seidel case.

Once the parallel electrode cases were implemented, the stochastic model could be implemented for the circular electrode cases. Three implementations of the circular field simulation were then implemented. The first implementation used a circular outer electrode and a point inner electrode. The second implementation used a circular inner electrode as well as circular outer electrode. The third implementation used an improved model for governing branch growth in a circular outer electrode, point inner electrode configuration. In Fig. 4.8 , the general structure of these subroutines can be seen. The sub-tasks for Gauss-Seidel field iterations and calculating the branch extensions were separate subroutines common to all simulations except the parallel tree Jacobi iteration. This simulation required its own field calculation routine.

Each of these routines was written allowing user selection of the simulation parameters including image size, model exponent, and the number of local and global field iterations. Generally, when testing and verifying the routines, smaller lattices were chosen to increase calculation speed. With an IBM PC machine and 8 megabytes of memory, a lattice size of 750 by 750 pixels is possible. Images of this size however would take many days on a 486 based machine to per-

form the simulation. Normally, images within a lattice size of 500 by 500 pixels were generated for the most important testing procedures. To improve speed of calculation as much as possible, the programming included the case where the model exponent was one as a special condition.

Verification of the output can be done by animating the results and studying the way the fractal grows. Also, the simulated fractal can be magnified to study the structure of the simulation more closely.

The structure of the general routines for calculating the point where extensions to the fractal occur and how the field is iterated is given in Fig. 4.8 . The location where the stochastic model adds a new branch to the simulated fractal is calculated in two stages.

In the primary stage, the sum of all the gradients around each of the points in the fractal are added. This is accomplished as follows. The sum is first reset to zero. Then the four nearest neighbor voltages around each of the points in the fractal are added to the sum. These voltages represent the gradient around the point in the fractal since the fractal is given a zero voltage. All the gradients around all the points in the fractal are added to the summation.

In the second stage, the location where the extension to the fractal is determined. The sum in the primary stage is multiplied by a random number between zero and one. The sum is restarted summing the gradients around each of the fractal points as in the primary stage. The process ends when the new sum equals or exceeds the random number times the old sum value. The location where this occurs is where the fractal is extended. The subroutine sets the new point to zero and returns with its location.

The field calculated using Gauss-Seidel type field calculations used the same local field iteration technique performed by Fukai and Sano [FuSa87]. Gauss-Seidel iterations simply use the intermediate results calculated immediately for subsequent calculations. This type of algorithm is not useful on large vector processing multiprocessor systems since the results obtained from a previous step are required for subsequent steps. This type of computation is only useful on a single processor system. After each extension to the discharge tree, the local region (+/-25 pix-

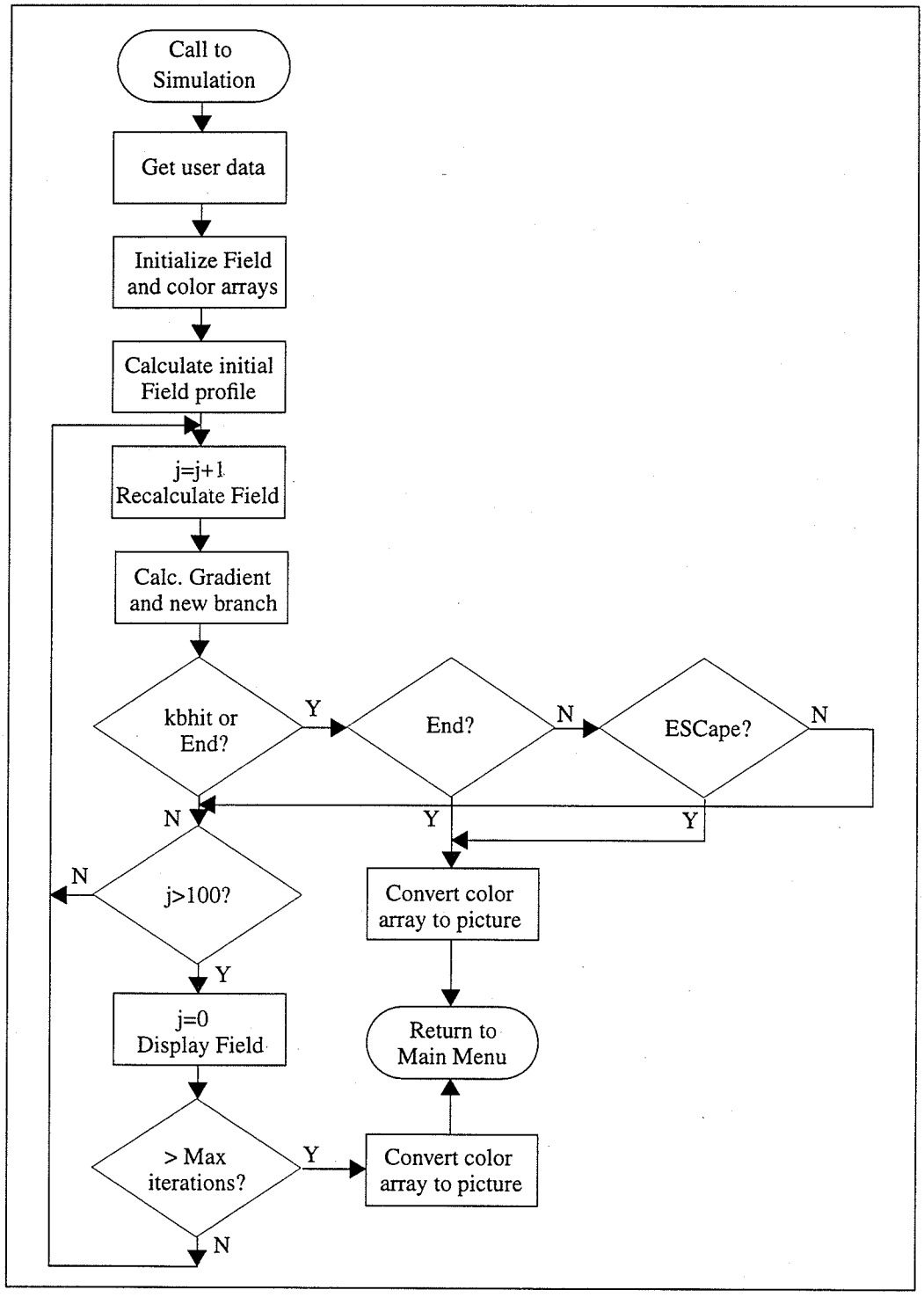


Fig. 4.8. General flowchart of stochastic model subroutines.

els in both x and y directions) was iterated 50 times before doing a global recalculation of 8 iterations. The number of iterations for local region allows the disturbance to propagate to the

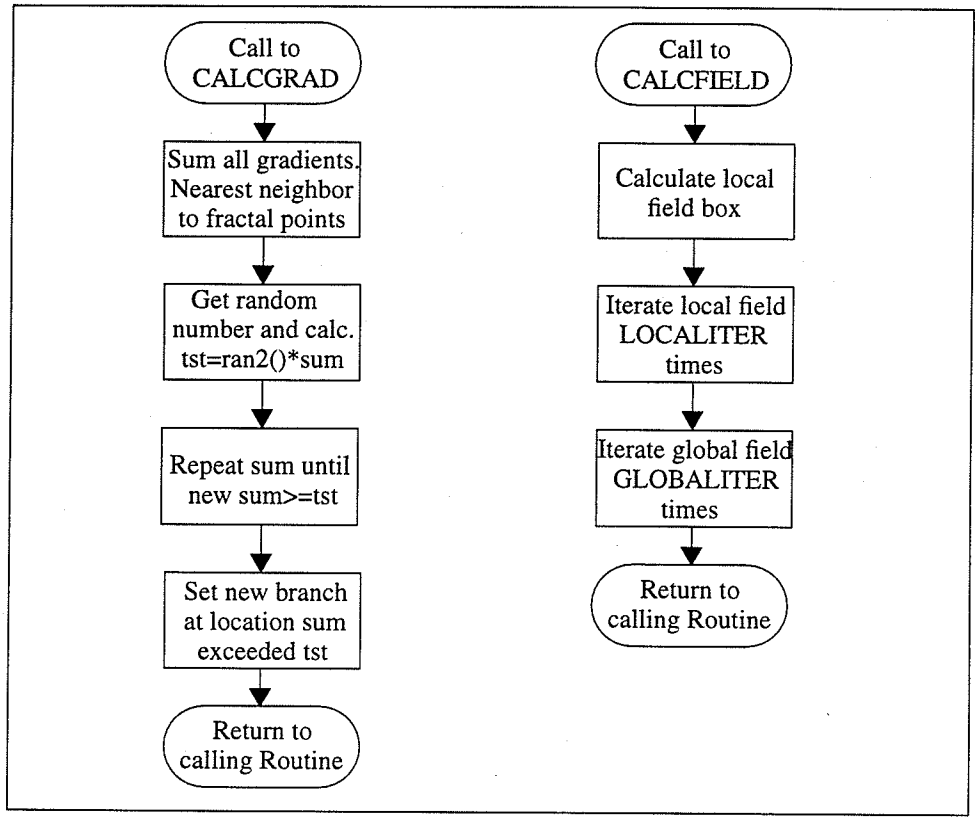


Fig. 4.9. General flowchart of subroutines for model subroutines.

local boundary and back to the center once. These default values for the number of local and global field iterations can be reset by the user to any positive even value allowing tests to be performed with greater or lesser field accuracies. Increasing the number of field iterations naturally increases the amount of time required for the simulation as well. The convergence properties of the fields will be covered in greater detail in a later section.

Field calculations used unsigned integer arithmetic and set the top parallel electrode (or outer electrode, circular cases) to a value of 10^9 and the bottom electrode to 0 (or inner electrode, circular cases). Integers carefully used do not affect the accuracy of the field calculations but do provide a much better performance from the computer. The field potential in the lattice was initialized to the known steady state value which is simply a linear or log type distribution given by

$$\phi(i,j) = (10^9 \times (j - j_{max})) \qquad \text{parallel electrodes} \tag{4.2}$$

$$\phi(r) = \frac{\log(r) - \log(A)}{\log(B) + \log(A)} \times 10^9 \qquad \text{circular electrodes} \tag{4.3}$$

The stochastic model uses a random number generator to determine where the next bond is broken or pixel is added. Since the performance of the standard number generator usually implemented with most "C" compilers is known to be limited, the "RAN2" random number generator from the book "Numerical Recipes in C" was implemented and tested. Each simulation requires up to fifty thousand random numbers so the random number generator was tested for two hundred thousand points. There was no observable skew in the distribution of the 'RAN2' number sequence.

To check that no strange effects were taking place, the parallel tree simulation was also repeated using Jacobi iteration. The rate of convergence for Gauss-Seidel is known to be much faster (and possibly unstable) than Jacobi field calculations. Although there was a marked difference in the field profile, the resulting tree structure was very similar between these two methods.

To provide more information about how the simulated discharge grows, the intensity of the pixels added to the tree depends upon when they were added. Initial pixels are given the intensity "white" while progressively later pixels are given darker and darker shades of grey. This addition to the stochastic model allows information of how the fractal grows to be saved in the representation of the image. This time value was used by an animation routine to animate the growth of the image in seconds as compared to many hours for the actual growth. This allowed the user to study the growth dynamically.

Once the simulated discharge reaches from the bottom electrode to the top or from the inside to the outside electrode, the simulation stops. (Actually, the simulation was stopped when the sum of all the gradients around a pixel in the fractal exceeded 10^9. This stops the simulation about two steps before complete punch through.)

During the simulation process, the field profile is displayed on the computer screen so the user knows how far the simulation has progressed. Shown in Fig. 4.10 and Fig. 4.11 are exam-

ples of these field images.

The first image shows the field profile for a parallel tree generation test where the image size was 500 by 500 pixels. The top plate was maintained at an integer value of 10^9 and the bottom plate maintained at an integer value of 0. The left and right edges of the image were treated as Neumann boundaries explained earlier. The image profile is set so that each band from white to black represents one tenth of the applied potential.

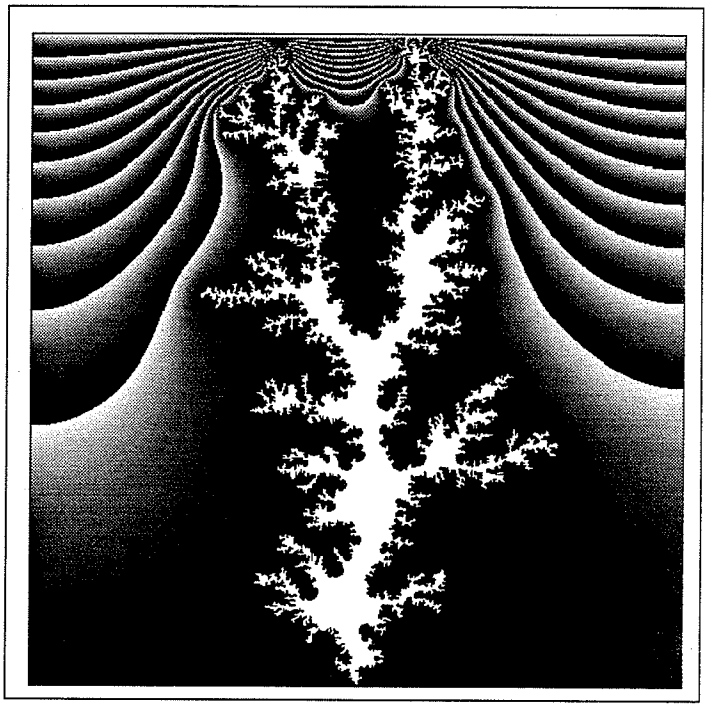


Fig. 4.10. Parallel tree field profile plot, Gauss-Seidel iteration.

In similar fashion, the field plot for the circular tree simulation example is again calculated using a 500 by 500 pixel lattice. The outer ring is treated as a Dirichlet boundary and held at an integer value of 10^9 as with the previous case. The center of the circle is given a seed point from which the fractal grows. The field profile is again set so that the white to black bands represent

one tenth the potential on the outer ring.

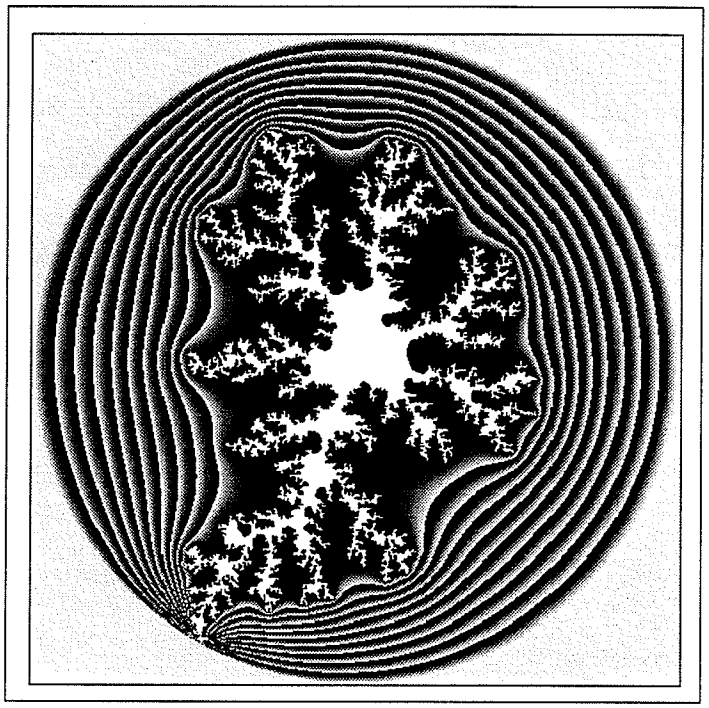


Fig. 4.11. Circular tree field profile plot, Gauss-Seidel iteration.

After the simulation is completed, the actual fractal image can be displayed. Note that the field images do not show the actual fractal but the zero potential surfaces which includes the effects of faraday caging (and the limits of integers). The actual fractals contained in the field profiles above can be seen in Fig. 4.12 and Fig. 4.13 . The development of this size of fractal takes many hours on a 486 based computer. In order to watch how the fractal develops in a much shorter period, the image can be animated by the test program after the image has been generated.

Once the image has been obtained, estimating the dimension was performed using the simple box dimension, dilation dimension, full dilation dimension, and correlation dimension algorithms. The greyscale image is converted to a white on black image before this analysis. The white on black processed image is used by the dimension calculation routines.

Both dimension plot figures were obtained by plotting the results on the screen and then

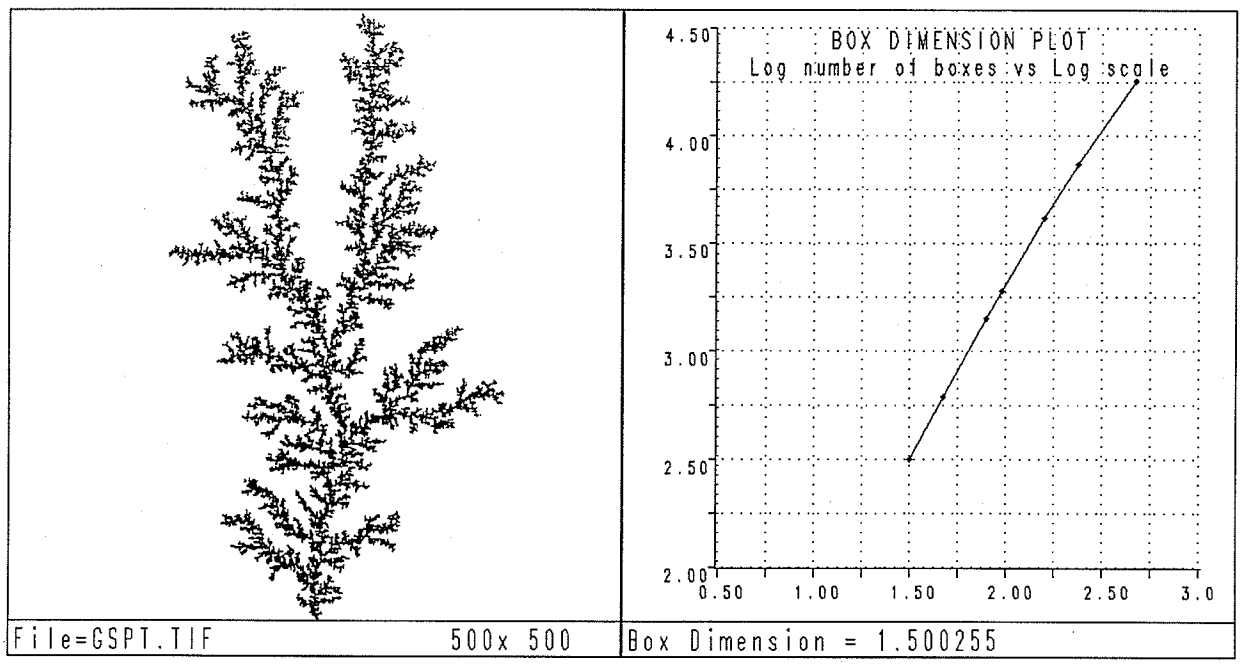


Fig. 4.12. Parallel tree fractal, Gauss-Seidel iteration with box dimension.

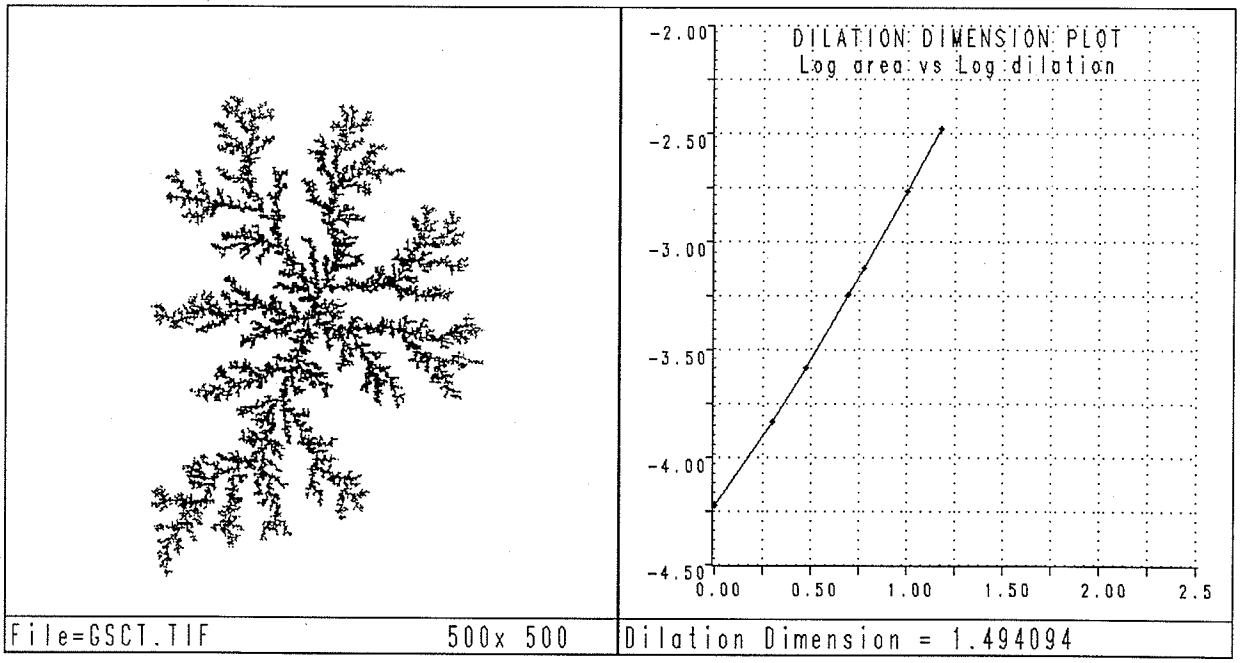


Fig. 4.13. Circular tree fractal, Gauss-Seidel iteration with dilation dimension.

capturing the screen to a file. To further show how changes in the exponent η affects the character of the fractals created by the original stochastic model, a collage of images showing the range of

image complexity possible is given below.

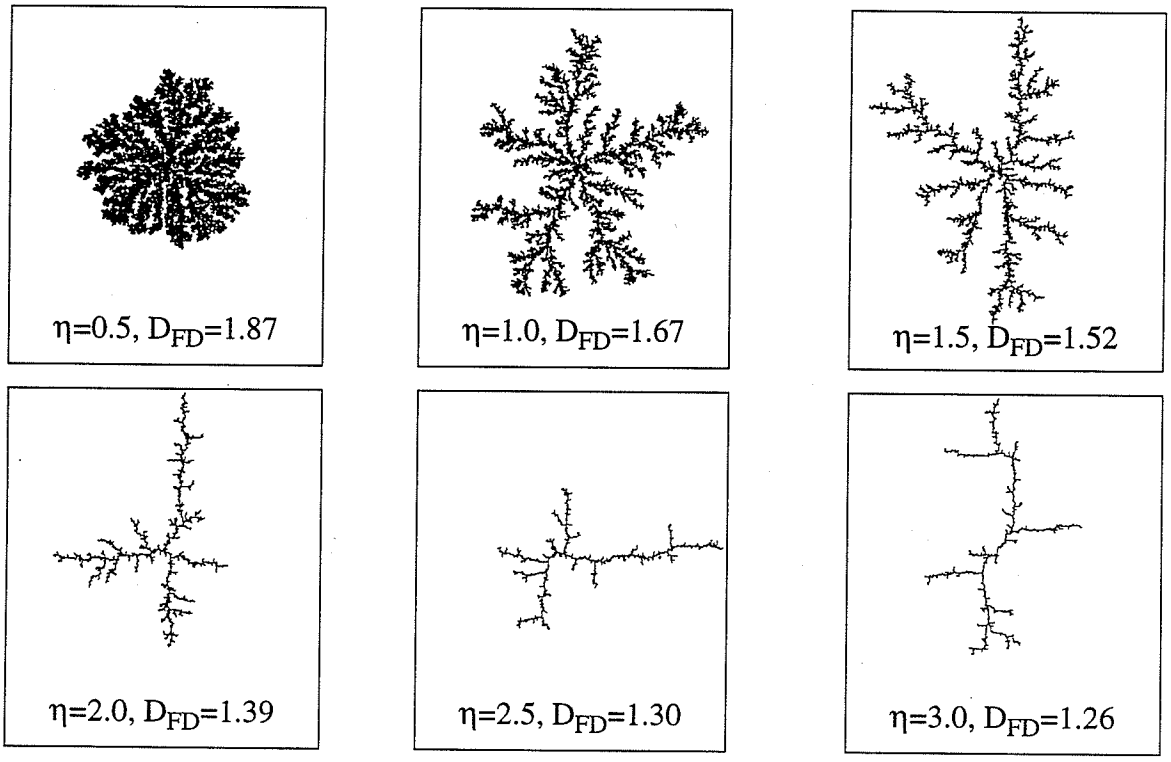


Fig. 4.14. Results from model for different values of η.

4.6 Summary

The implementation of the program TEST proceeded in stages. The initial decisions for data representation, display and storage would affect all subsequent programming. The choice to use the TIFF file representation for storage ensures compatibility with other programs. Following the implementation of the file input and output routines, the image display routines were finalized.

Once the basic methods for handling the data were established, implementation and verification of many standard image enhancement algorithms commenced. Some of these routines acted as supporting functions in the final fractal analysis procedures.

After completing the standard image enhancement procedures, the next stage of development was implementation of the L-systems test fractals. These images were especially important

for testing and verification of the subsequent dimension estimation algorithms added next.

The dimension analysis procedures provide the algorithms required for characterizing the structure of the fractals created by the stochastic models and from real images. The performance of these routines was tested using the L-systems test fractals implemented earlier.

The last stage of development was implementing the variations of the stochastic model. Three basic configurations were decided upon. Testing the variations of the model's parameter η quickly shows the range of tuning possible with the current stochastic model. With all algorithms and procedures in place the experimental testing could then follow.

CHAPTER 5
EXPERIMENTAL RESULTS AND DISCUSSION

After the testing program TEST was written and verified, the limitations of scanning, enhancing, and analyzing fractal images could be explored. Initial experiments were run to identify limitations with the testing procedures being used. The experimental process takes images of fractals, digitizes them to their computer representation as a bit mapped image, enhances the digital representation while preserving the fractal parameters of the image, followed by image analysis to identify these fractal parameters.

The first experiments were run on the L-systems test fractals to determine the limits of the dimension analysis programs. In order to test and identify the structure of the fractals obtained during simulation and experimentation, the following experiments again used the test fractals to test the limitations and the subsequent errors introduced when taking and analyzing subsets of the images. Since the dimension of the test fractals is the same at all scales, an error band for the dimension measurements could be established as a function of image or sub-image size. In similar fashion, the value to be used for the threshold detector could also be obtained. The usefulness of the error bands obtained is limited since that only a small number of images were used to obtain them. However it does give a good indication of their magnitude.

The scanning or digitization of the images is an important step in the analysis. A 256 level greyscale hand scanner was used to digitize the pictures obtained experimentally. The hand scanner was used because of convenience. Tests were run to verify the quality of the hand scanned images were as good as a flatbed scanner would produce. Details appeared to be better preserved with the hand scanner than with the flatbed scanner tested. The hand scanner also had a calibration procedure that ensured all scanned images had an identical intensity reference. All images were obtained under such calibrated conditions. The next few experiments were run to identify the limitations of the image digitization procedure. These experiments were followed by

experiments generating actual DLAs and finally experiments to tune the model to DLAs and to a Lichtenberg figure. The following sections detail each experiment and their conclusions.

5.1 Dimension Algorithm Accuracy on L-systems Test Fractals

In this section the accuracies of the dimension estimation routines are determined experimentally using the L-systems test fractals described earlier. The L-systems generated test fractals are pure (mathematical) fractals with only one underlying dimension. Any wavering or curving of the data obtained with any of the dimension algorithms is an indication of inaccuracy or limitations of that algorithm. The test images used were 1024x1024 pixels in size generated using the L-systems generator described fully in Chapter 4. These fractals can be seen in Fig. 4.4 . The iteration depth was selected to give approximately 10^8 segments in each image.

The box and dilation dimension routines were tested using box and dilation sizes of 1,2,3,5,6,10, and 15 pixels. The selection of these values was made to allow testing later on parts of the whole image to investigate how accuracy suffers when analyzing parts of sub-images. The full dilation dimension routine uses dilation values of 1,2,3,4, 6,8,12,16,24,32,48,64,96,128,192, and 256 pixels. The correlation dimension routine also uses these values for radius but only the data values for 3,4,6,8,12,24,32,48,64, and 96 pixel radius were only used when calculating the dimension.

The results in Table 5.4 indicate the best algorithm to use for measuring the dimensions of pure (mathematical) fractals is the dilation dimension. The accuracy on the nine test fractals was better than 0.3%. The data points obtained with the dilation dimension algorithm appear to also follow a smooth curve. The curve indicates that a systematic error is present in the algorithm that might be correctable. (see Fig. 4.13 , page 74) Adding the edge correction used in the full dilation dimension routine made the accuracy worse so was not included in the dilation dimension algorithm. The reasonable accuracy of this algorithm indicates that the approximations made in its derivation were valid. By modelling the fractal with a power-law relationship, the results followed. The author believes measuring fractal dimension is the measurement of the power-law

relationship of the function at various scales thus the derivation is not restrictive in this fashion. This algorithm took approximately 30 seconds on a 486-33 MHz machine to calculate the dimension of the most complicated fractal tested; the Sierpinski gasket. The error produced by the dilation dimension routine is low however only on pure (mathematical) fractals. Using the dilation dimension on multifractals produces large errors since the dimension of the multifractal is usually not determined at the smallest scales of the image.

The box dimension algorithm is next in accuracy giving an accuracy of approximately 2%. This algorithm was the fastest taking only about 7 seconds to compute the dimension of the Sierpinski gasket. Noted with these results is the slight random wavering of the data points when plotted on the computer screen. This wavering is believed to be caused by the testing grid lining up onto the fractal better or worse at different scales. This could be reduced or eliminated by looking at all possible mesh positions at each mesh size and using the smallest result. This may in fact make this algorithm better than the dilation dimension algorithm though would add complexity and a severe speed penalty.

The correlation dimension routine (intuitively at least) appears to test the fractal well at all scales using the probability the fractal exists within a given radius of a point in the fractal. The routine is very slow taking over 8 hours to compute the dimension of the Sierpinski gasket. It also appeared to be the most inaccurate routine tested with an accuracy of around 2%. The data points did not appear to follow any consistent curve or line thus the error does not appear to be systematic as with the dilation dimension routine. It is believed by the author that the accuracy of the correlation algorithm is hampered severely by the boundaries of the fractal where the probability function naturally decreases. Comparison of all the algorithms is still valid however since each is given the same data. The correlation routine did however generate reasonably smooth curves on the fractals generated with the stochastic model. This is described in the next section.

Clearly the performance of these three routines is disappointing. The dilation dimension algorithm created an incorrect but smooth curved function for dimension. The simple box dimen-

sion curve wavered indicating dimension variation at different scales. The data obtained with the correlation dimension appeared to be inconsistent. Most of the dimension error can be attributed to the inaccuracy of the dimension estimating algorithms but some must be attributed to the inaccuracies of the L-systems images as well. A majority of the images created by the L-systems generator had not converged to the final image so are missing some pixels. However; to generate more accurate images would require more iterations and this would introduce more round-off error in the resulting image. One solution would be to rewrite the routines in assembler using the 80 bit number representation possible with the 386/486 coprocessors. This would significantly reduce the error compared to using the double number (64 bit) representation possible within "C". Better images would however only increase the accuracy of the dilation routine since the errors with the other routines do not appear to be systematic.

The results in the literature for the correlation dimension fail to identify the accuracy of the algorithm but only its consistency mainly on stochastic fractals. It is important to identify why the correlation dimension does not produce consistent results on pure (mathematical) fractals. It may be an indication that this routine is severely limited by the boundaries of the finite fractals being tested.

Table 5.4. Dimension measurements on the L-systems test fractals.

Fractal	Calculated Dimension	Iteration Depth	Box Dimension	Error %	Dilation Dimension	Error %	Correlation Dimension	Error %
a	1.1042	13	1.0799	-2.20	1.1071	0.26	1.0805	-2.15
b	1.2619	13	1.2551	-0.54	1.2622	0.02	1.2434	-1.47
c	1.3403	11	1.3268	-1.01	1.3403	0.00	1.3113	-2.16
d	1.5000	9	1.4906	-0.63	1.4967	-0.22	1.4870	-0.87
e	1.4650	11	1.4551	-0.68	1.4622	-0.19	1.4434	-1.47
f	1.4650	11	1.4661	0.08	1.4617	-0.23	1.4636	0.10
g	1.5282	8	1.5198	-0.55	1.5251	-0.20	1.5276	0.04
h	1.5736	7	1.5719	-0.11	1.5723	-0.08	1.5884	-0.93
i	1.5850	17	1.5749	-0.64	1.5815	-0.22	1.5777	-0.46

5.2 Dimension Algorithm Consistency on Stochastic Model Fractals

As mentioned earlier, fractals whose growth is governed by a Laplacian field have been called Laplacian multifractals. They are called multifractal because the dimension of these fractals appears to differ with the scale at which these fractals are observed. Characterizing the fractal with the maximum dimension observed or with a function would afford more information about the fractal than a single number obtained at a random location on the data curve.

The standard morphological techniques can be extended to look at this variation. The dilation dimension routine was extended to cover a larger range of scales and this new routine is called the full dilation dimension algorithm. The range of the box dimension routine was not extended here for testing on multifractals. This is due to the nature of the box counting algorithm. The box counting routine counts fewer and fewer boxes the larger the boxes get thus is very susceptible to small changes at very large box sizes with even large images. This test checks the performance of the full dilation dimension algorithm and then compares the results with the correlation dimension algorithm.

The dilation dimension routine was extended to measure the dimension over a 256:1 range of scales in the full dilation dimension procedure. In addition, a cubic curve was least squares fit to the data for the full dilation algorithm. The maximum slope achieved by this curve within the data range was then given as the multifractal dimension of the fractal. Such an extension needed to be tested on the L-system test fractals to ensure its validity. The results in Fig. 5.5 show that such an extension (though not as accurate as the dilation dimension) is reasonable, giving an accuracy on the test fractals of better than 1%.

The end or edge correction added to this routine (refer to (2.21) in Chapter 2) effectively halved the error resulting with this routine. The reason it worked for the full dilation dimension routine and not for the ordinary dilation routine is the cubic curve fit. The dilation routine has the same data curvature as the full dilation routine but over a smaller range. The error caused by not including the end or edge correction on the dilation dimension happens to be in a direction that

reduces the error of the linear curve fit. Should a correction for the data curvature be found these two routines would give the same results.

Table 5.5. Full dilation dimension measurements.

Fractal	Calculated Dimension	Full Dilation Dimension	Error%
a	1.1042	1.1073	0.07
b	1.2619	1.2594	0.35
c	1.3403	1.3212	0.43
d	1.5000	1.4845	0.36
e	1.4650	1.4287	0.94
f	1.4650	1.4486	0.20
g	1.5282	1.4965	0.44
h	1.5736	1.5474	0.43
i	1.5850	1.5745	0.20

Now that the accuracy of the both the full dilation dimension and correlation dimension routines has been established, the routines could be used on images produced by the stochastic models. The stochastic model introduced by Niemeyer, Pietronero, and Wiesmann [NiPW84] was used to generate five images of size 500x500 pixels containing roughly 15 to 20 thousand points. The data curves using both dimension routines were very similar with significant curvature near the smaller scales.

Measuring the multifractal dimension with the full dilation dimension subroutine and the cubic curve fit of the 500 by 500 pixel fractals produced by the stochastic model with η=1 yields a dimension of 1.68 +/- 0.01 over five tests (see Table 5.6). This result agrees with the values obtained using the correlation dimension and given by others [WiSa81].

Testing with the correlation dimension yielded yet another surprise. Using a cubic polynomial with all the data to obtain the dimension yielded poorer results than a simple linear regression over the radius range of 3 to 96 pixels. Results with the cubic curve fit appeared inconsistent

with each other.

Table 5.6. Stochastic model dimension measurements.

Stochastic Fractal	Full dilation dimension	Correlation dimension
a	1.6773	1.6614
b	1.6825	1.6765
c	1.6803	1.6732
d	1.6994	1.6914
e	1.6884	1.6764

The correlation dimension has been given as one of the best methods for determining the dimension of a fractal or multifractal. The results from testing the correlation dimension subroutine on the L-systems generated test fractals seem to question this. The error as measured on the test fractals is inconsistent and the plots of the data were far from smooth as expected.

The full dilation dimension routine however achieves reasonable accuracy on the test fractals and very similar results to the correlation dimension routine on the stochastic multifractals. From these results, the full dilation dimension routine appears superior for measuring the dimension of both pure (mathematical) fractals and multifractals. The two dilation routines still exhibit a systematic error causing the data to curve towards a lower slope and therefore smaller dimension at the larger scales. If a correction formula could be produced to counteract this effect (which appears to be a function of the fractals dimension) these routines would be even more accurate.

5.3 Dimension Error as a Function of Image Size Experiment

Naturally, analysis using large images will produce better results than the same analysis using small images. This experiment checks how image size affects the error of the box and dilation dimension calculating subroutines. This test takes each of the test fractals, "reduces" them in dyadic fashion (by 2), and tests the dimension at the new image size. The REDUCE subroutine overlays the fractal in the image with a δ-mesh of user selected size and tests the existence of the fractal at this new scale. This is essentially how the L-systems fractals are formed in the first

place so the character of the test images is preserved at the new scale.

The maximum dimension error of the fractal at each of the new image sizes is then taken for the error bands and this is plotted in the graph Fig. 5.1 . For large images the dilation dimension routine is vastly superior. When the image size reduces, naturally the error from the dimension routines will increase. The error produced by both routines is very similar for image sizes of 64 by 64 pixels. This graph is important since the simulation images being generated were generally 500 by 500 pixels. This shows that the dilation dimension routine can be expected to yield an accuracy of about +/- 0.5% for this image size at the smaller scales in the image.

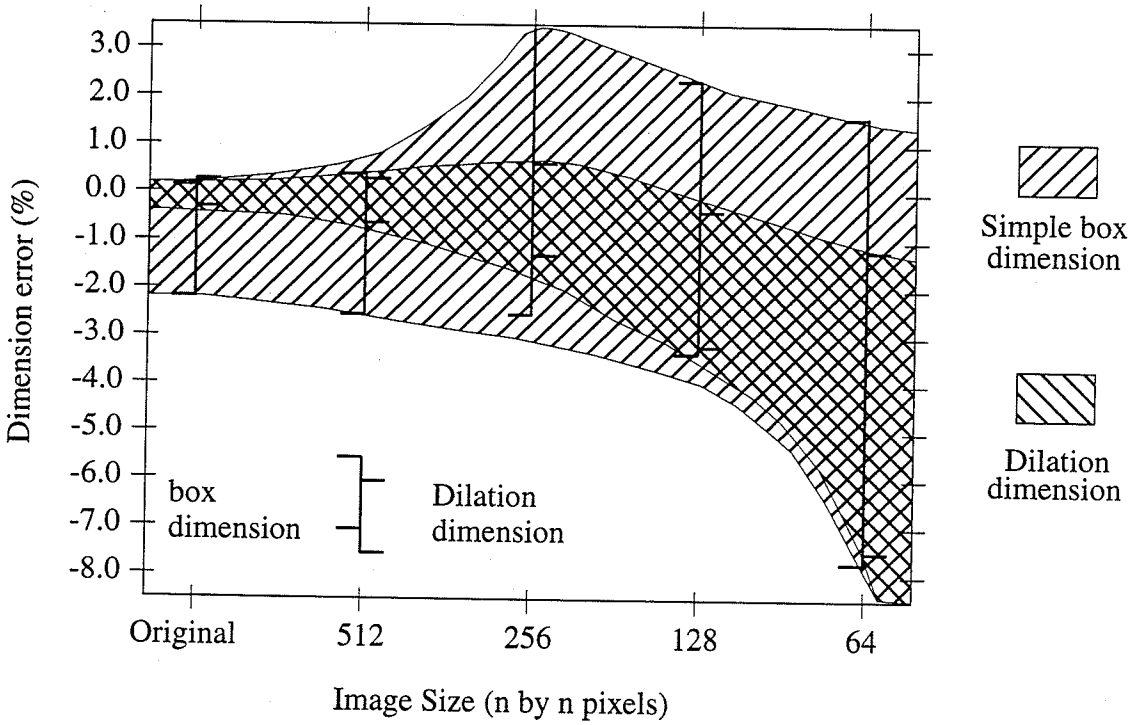


Fig. 5.1. Dimension error (%) vs. image size.

5.4 Dimension Error as a Function of Sub-image Size Experiment

One objective of this thesis is to explore the structure of Laplacian multifractals. The error introduced by the selection and analysis of parts of the total image was explored through this experiment. Investigating the errors introduced by measuring arbitrary pieces of full images will show the inaccuracies introduced by the boundaries of these sub-images.

This experiment takes five random sub-images from each of the nine L-system test fractals at each of four sub-image sizes. The fractal dimension is calculated with the simple box and dilation dimension routines for each of the sub-images. The five results at each image size for each fractal are averaged. The error bars on the plot (Fig. 5.2 , page 86) is then the maximum and minimum error values obtained at that sub-image size over all the fractal images.

Performing such a test on the L-systems test fractals (which have only one dimension) shows the limitations of measuring spatial variations of fractal dimension. Indeed, initial measurements of fractal dimension on images created by the stochastic model indicated the center of these fractals had higher dimension than the whole. The results of this experiment showed those measurements were within the range to be expected. Further testing on the multifractal images created by the stochastic model did not show any significant spatial variation of fractal dimension. These results are also limited by the relatively small sizes of the images created by the stochastic model and the limitations of the dimension estimation routines. One interesting observance is that the errors for the box counting dimension and the dilation dimension appear to have opposite trends. It would appear that an average of both readings would lead to better overall accuracy for small sections of images.

The plot in Fig. 5.2 shows the results of this experiment.

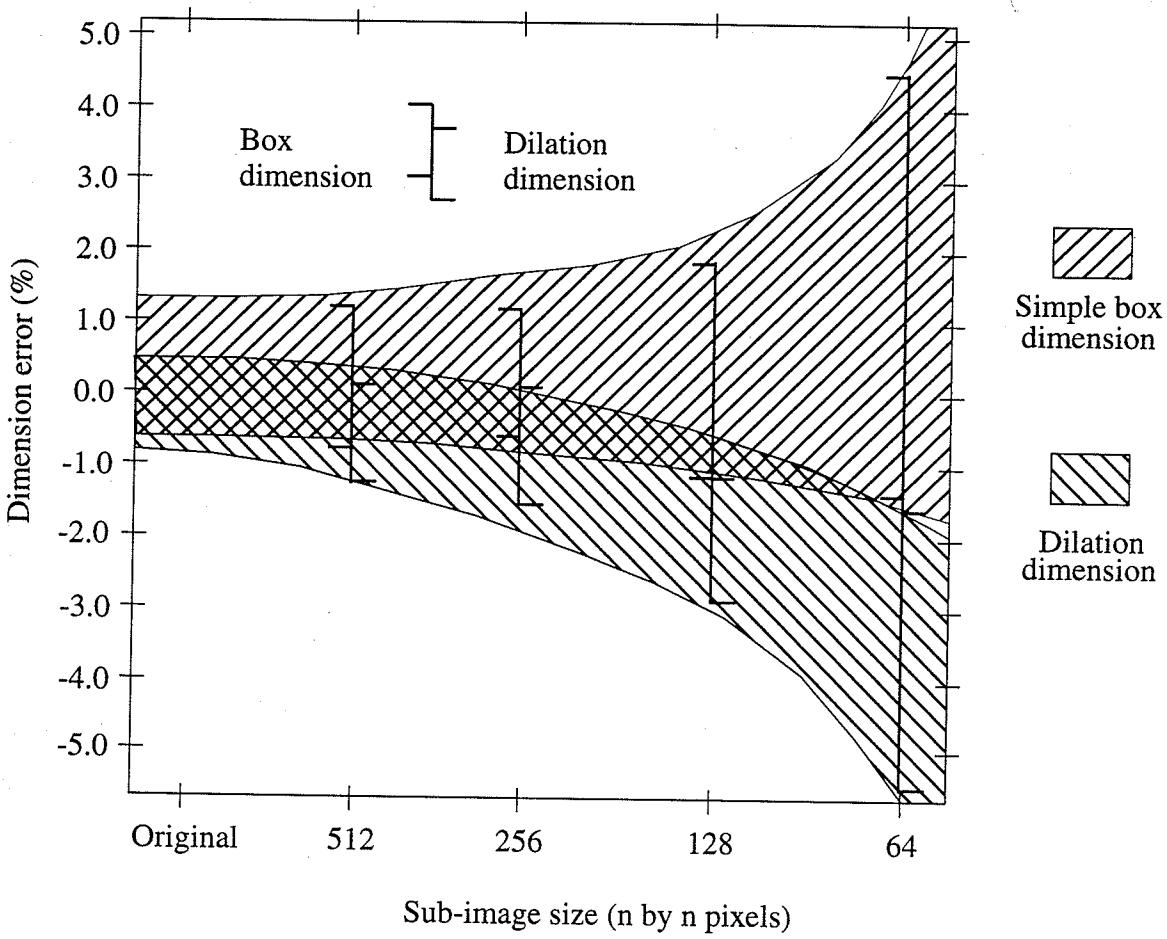


Fig. 5.2. Dimension error vs. sub-image plot.

5.5 Dimension Error as a Function of Image Threshold Experiment

All images obtained through experimentation must go through a digitization, enhancement and thresholding procedure to obtain the best digital representation of the fractal possible. The optimum value of threshold to be used by the threshold detector is a critical step in this process. This experiment investigates the properties of the threshold detector and identifies the optimum threshold to be used in the final stage of the image enhancement procedure.

In the plot Fig. 5.3 the effects of processing scanned images can be seen. This plot was created by first printing the test fractals at 150 dots per inch and scanning the results back into the

computer at 200 dots per inch. The scanned images were greyscale so that the test program could have control over where the threshold detector was set. Each image after scanning is a black fractal on white background image. This is then negated to create a white fractal on black background image. The image was then normalized to occupy the full range of black to white intensity variation using the auto-scaling routine.

Various thresholds were then applied before calculating the dimension. The dimension results were then plotted showing how the detection threshold changes the measured dimension. The error bands show the variation of the measurements over all the nine test fractals. Clearly, this plot shows this is the major source of error when processing images introducing inaccuracies of approximately 2%. Reading the graph puts the optimum threshold for the detector at a grey-

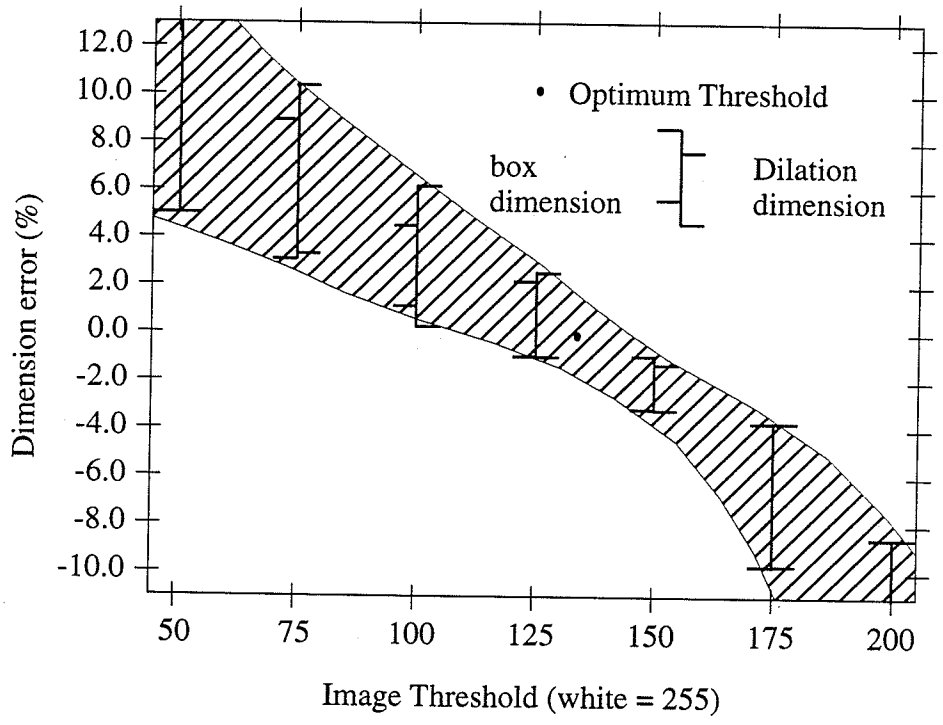


Fig. 5.3. Dimension error vs. image threshold.

scale value of 131. This value was used in subsequent experiments and testing.

5.6 Diffusion Limited Aggregate Experiment

The first step in tuning the stochastic model was to check how closely the model created DLA images matched images of actual DLAs. Since only a few good images could be found in the literature actual experimentation was required to provide the necessary data. It was also thought that the printing process may have lost much of the finer details in the image as well. The usual method for creating DLAs was to use zinc electrodes and an aqueous solution of zinc sulfate; both items not easily obtainable.

The actual process of creating these DLAs is simple so the experiment was duplicated using copper electrodes and a saturated solution of copper sulphate. These materials are readily available at local stores and the resulting fractals appeared identical to the images found in the literature.

Materials used in the experiment included a copper sheet of thickness 0.025 inches, copper sulfate chemical, tap water, accetate sheets, a 5.0 volt power supply, a light table to increase contrast, and a camera to photograph the results. Construction of the experiment was rudimentary.

A hole of diameter 3.5 inches was put into the copper sheet with a circular cutter on a drill press. To this sheet was soldered a wire then tied to the positive terminal of the power supply. An accetate sheet was used above and below this outer copper electrode. Copper sulfate solution was puddled into the middle of the electrode before carefully placing on the top accetate sheet. A small hole placed in the top sheet allowed the negative center electrode to contact the solution between the sheets. Surface tension helped remove all bubbles and keep the solution between the accetate sheets. The copper DLA was grown until the edges came close to the outer electrode. The DLA at this stage was generally several inches across. The light table under the growth increased the contrast of the copper colored DLA so that the finer details would be captured by the camera. An Olympus OM-1 camera with split prism focussing screen and external focussing screen magnifier was used to capture the image. A close-up lens enabled the image to be captured with as much detail as possible. The configuration can be seen in Fig. 5.4 , page 89.

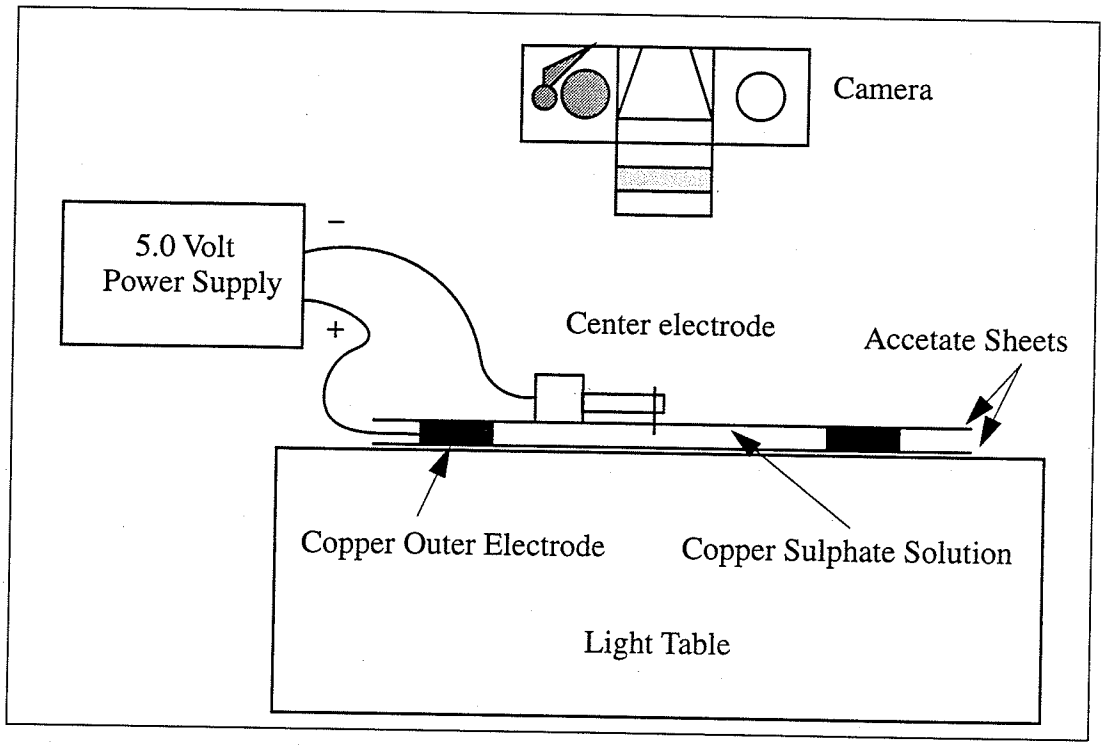


Fig. 5.4. DLA experimental setup.

Once the pictures of the DLAs were obtained, they could be digitized using the calibrated 256 greyscale hand scanner and stored on disk. These images were similar in structure to those observed in the literature. It was noted when comparing the experimental images with those from the literature that none of the finer detail appeared lost in the printing process. After the images were collected, they could be enhanced using the auto-scaling subroutine, negation subroutine and the threshold detector to obtain the white on black images required for analysis.

The auto-scaling subroutine normalizes the image so that the results from previous experiments would be applicable. Negation converts the black on white background picture to a white on black background image used for analysis. Applying the threshold detector using the optimum threshold of 131 obtained earlier converts the image to a true black and white image suitable for the dimension analysis subroutines.

An example of one of these DLAs can be seen in Fig. 5.5 , page 90. Clearly, this image does not seem to match that of the simulated DLAs generated with the stochastic model (see Fig.

5.8). The model depicts growth of a fractal with only a diffusion limited mechanism. The real world rarely exhibits such perfect examples. It is not surprising that actual experimentation creates fractals with more than one mechanism dictating its growth. Later, the stochastic model is tuned to try to find out the value of η that best represents the growth of the DLAs obtained in this experiment.

The structure of the copper DLA growths appears to be very consistent. The branches are similar in width and complexity over all the image. This possibly indicates inadequacies with the

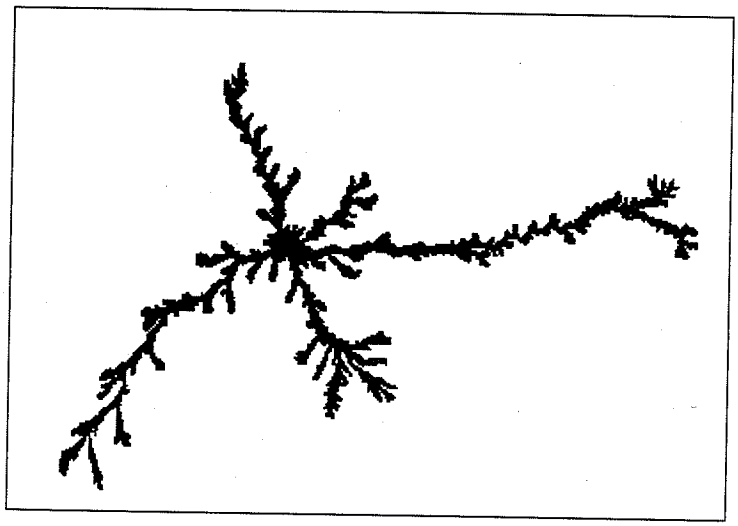


Fig. 5.5. Example of DLA created using copper sulphate solution.

current model. Incorporation of a voltage gradient threshold under which growth would not occur could help produce such consistent branch width in the simulated DLAs. Later, tuning the model to match these DLAs will show good results apart from this difference.

5.7 Testing the Field Convergence using Gauss-Seidel Iteration

Understanding the properties and limitations of the field calculations is important in selecting how many iterations are required for reasonable results. The Gauss-Seidel field calculations are known to be much faster than Jacobi type field calculations and were used almost exclusively for the simulations.

The process of using finite difference equations for field calculations requires an approximation to Laplaces equation. The standard second order approximation detailed earlier was used for simplicity. This type of calculation coupled with using answers obtained immediately for subsequent calculations (the Gauss-Seidel method) is simple, reasonably fast and stable.

To test how fast the field converges using global iterations, two subroutines were written and run on various sizes of lattice. The parallel electrode field convergence test sets the top electrode to 10^9 and the bottom electrode to zero. All the other values in between were set to a small non-zero value. The field is then iterated until the maximum difference between iterations becomes less than a small value. On the screen is displayed the number of iterations required with a decrease in the field difference of a factor of ten. Thus the test yields a list of numbers giving the number of field iterations required to obtain a given rate of change of the field potentials or field error rate.

The circular field convergence test uses a circular outer electrode and point inner electrode. The potential of the outer electrode is fixed at 10^9 and the inner electrode is fixed at zero. All the other values are set to a small non-zero value. The field is iterated and the same data is listed on screen as for the previous test.

The data listed on screen for both the field convergence tests indicates how many iterations are required to obtain a given approximation to the field potentials. It was surprising that field convergence is clearly slower for the circular electrode case. This in retrospect is understandable since there is only one fixed point for the inner electrode in this case. Although this is a static convergence test (no growing fractal to change the field), the convergence tests still show the relaxation type properties of such field calculations.

When running the actual stochastic model, the number of global field iterations recommended by Niemeyer was between 5 and 50 when simulating electrical discharges [NiPW84]. With 8 global iterations and the 50 local field iterations recommended by Fukai [FuSa87] a field error rate of approximately 10^{-4} per unit was observed when η=1. Using the stochastic model

with $\eta>1$ to simulate electrical discharges, the number of global and local iterations needs to be increased to maintain this same field error rate. Larger numbers of global and local field iterations are required for larger values of η since the fractal grows radially in size much more rapidly. In conclusion, maintaining a field error rate below 10^{-4} per unit appears reasonable and is desired when running the stochastic model.

5.8 Handling the Differences Between L-systems, Simulated, and Actual Images

The pixels in the images created using the L-systems generator subroutine, the pixels created by the stochastic model, and the pixels in the scanned images each have different definitions or meaning. The pixels in the L-systems images indicate that the fractal curve touched the pixel, no matter how slightly. The pixels in the images created using the current stochastic model represent branch growth of the simulated fractal. The pixels of scanned images represent the convolution and integration of a (gaussian?) weighted mask intersecting the true image.

Reconciling such different definitions of the pixels from these three sources is not a trivial one. After testing, the author believes the best method of handling the difference in the definitions between scanned images, simulated images and the L-systems images is to ignore details at the absolute size of the pixel for the images from simulated experiments or by using multifractal measures.

In all cases, this can be achieved by testing the image at a scale several times larger than a single pixel. The larger the testing δ-mesh used, the less important the difference between the pixel definitions becomes. The subroutine `REDUCE` was written to perform this task.

The subroutine `REDUCE` overlays the image with a user selected δ-mesh and tests the existence of the image in this mesh. The resulting image thus created is smaller by the factor of the mesh size in each direction. It was found that such an operation on the L-systems images does not change the structure of the image. This is not surprising given testing at scale is how these images are created. Such an operation however on the stochastic model created images makes big differences in the calculated dimension using the simple box dimension or dilation dimension

methods. The results understandably tend toward those obtained using the multifractal measures of the full dilation and correlation dimension subroutines.

Images created by the stochastic model and from actual experiments appear to be radically different even though their dimensions calculate to being very similar. By performing a reduction of each of these images thus removing the importance of the pixel definitions the dimensions can be better compared. This requires images several times larger so that after the testing at scale reduction, the images still contain enough data for accurate dimension estimation.

Using multifractal measures to estimate the dimension of the simulated and actual fractals should yield values that can be better compared. The multifractal nature of fractals created by the stochastic model and those obtained through experiment appears similar however if only the boundary of the experimentally obtained images is used for analysis. If the interior of actual experimental images from DLAs or electrical discharges is included when estimating the fractal dimension, the true area present will skew the results toward the full dimension of 2 at the smaller image scales. This problem was identified during the next experiment.

5.9 Tuning the Stochastic Model to Match Actual DLAs

With the images of DLAs obtained through experiment as reference, the model could be tuned to create images that matched in dimension. During this process, it was found that only the boundary of the actual DLA image can be considered when measuring fractal dimension. In retrospect, this is obvious since mathematically speaking the fractal pictures cannot have any area if the dimension is to be fractional. The interior of the actual DLA has dimension 2 and must be removed before calculating the boundary dimension of these images. Such is the inherent problem with using the mathematical concept of a fractal on real world data which cannot have true fractal characteristics at all scales. The dimension of the experimental DLA images drops significantly when only the boundary is considered.

The work by Shen and Caston has shown that the infinite symmetric exponential filter is optimal for detection of edges in images [ShCa92]. The largest exponentially weighted edge

detector programmed was the 25 point exponential edge detector and is the best approximation to this. This detector was used to detect the edges in the DLA images.

The clearest images of DLAs obtained through experiment were analyzed using the following procedure. The scanned image was first negated to form a white on black image. Then the image was normalized using the auto-scale subroutine. Since shot noise was not a problem with these images, the median filter was not required. The resulting image was then threshold detected using the optimum threshold of 131 obtained with an earlier experiment. The result was then edge detected using the 25 point exponential edge detector to obtain the edges of the DLA image. These edges were then thresholded at a value of 50 using the threshold detector to obtain the white edges used for calculating dimension. The dimension of the image could then be calculated.

Since the dilation dimension subroutine has consistently shown less error, this routine was then used to measure the dimension of the enhanced DLA images and for tuning the model. Using this routine which measured the dimension at the smallest scales present in the image is not a big limitation. The same routine is used throughout the tuning process and effectively matches the smaller features created by the model to the smaller features in the experimental DLA images. The boundary of the actual DLAs obtained through experiment was found to average D_D=1.32 +/ - 0.01 for six experimental DLA images. The dilation dimension routine and not the full dilation dimension routine was used for this because the dilation dimension routine produced more consistent results for the dimension of the DLA images than the full dilation dimension routine provided.

Many images were created with the stochastic model with η=1. All of these images had dimensions (using both box and dilation dimension routines) of 1.5. This indicates diffusion limited aggregation is not occurring with the above experimental procedure and/or the model is incorrect! The existing stochastic model could now be tuned to obtain images more closely

resembling the experimental results.

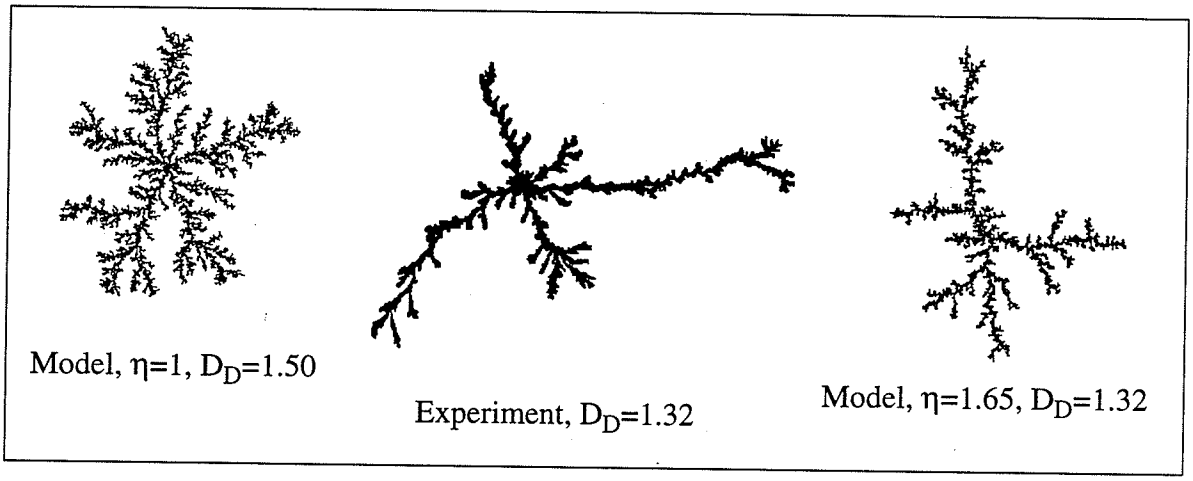


Fig. 5.6. Simulated and actual DLA images.

When tuning the stochastic model to produce better approximations to the experimental DLAs, simple interpolation between trials was used. Interpolation was performed between the first two trials where η=1 and η=3. The dimension at η=1 was 1.50 and the dimension at η=3 was 1.15. The numbers of η for the initial trials were guesses. The successive trials using interpolation were η=2.0, η=1.75, η=1.70, and finally η=1.65.

It was noted at that when η was greater than one, fewer pixels were used in the simulations. Consequently, the field error was significantly greater as well (indicated by the program during simulation). To decrease the field error during simulation for the final tuning steps of the model, the number of local field iterations was increased to 100 and the number of global field iterations was increased to 12. This decreased the field error rate to below 10^{-4}. The results of tuning the original stochastic model gave η=1.65 to produce fractals with a dilation dimension of 1.32. Several tests with η=1.65 indicated that the tuning was reasonably accurate. The results shown in Fig. 5.6 show the tuning achieved results closely matching the actual experimental data. One cannot achieve a match with the thickness of the actual experimental fractal since the thickness is probably due to another fundamental mechanism not being modeled.

Clearly the results for tuning the model more closely match the results from the experiment. The fact that η was not equal to 1.0 as expected must be explained. Either the exper-

iment is not totally a diffusion limited process and/or the model is too simple or limited.

The model introduced by Niemeyer, Pietronero and Wiesmann looks at the growth of the simulated fractal as the growth of branches. Closer inspection of the mechanics of the model reveals that inside corners and pockets formed during simulated growth are given double or triple the probability of growth compared to the growth at the leading edges of the fractal. Although the corners and pockets formed during simulated fractal growth naturally have a smaller potential gradient associated with them, to assume branch growth is equally likely from all associated neighbors is incorrect. Such an assumption will clearly lead to simulated fractals having a more dense

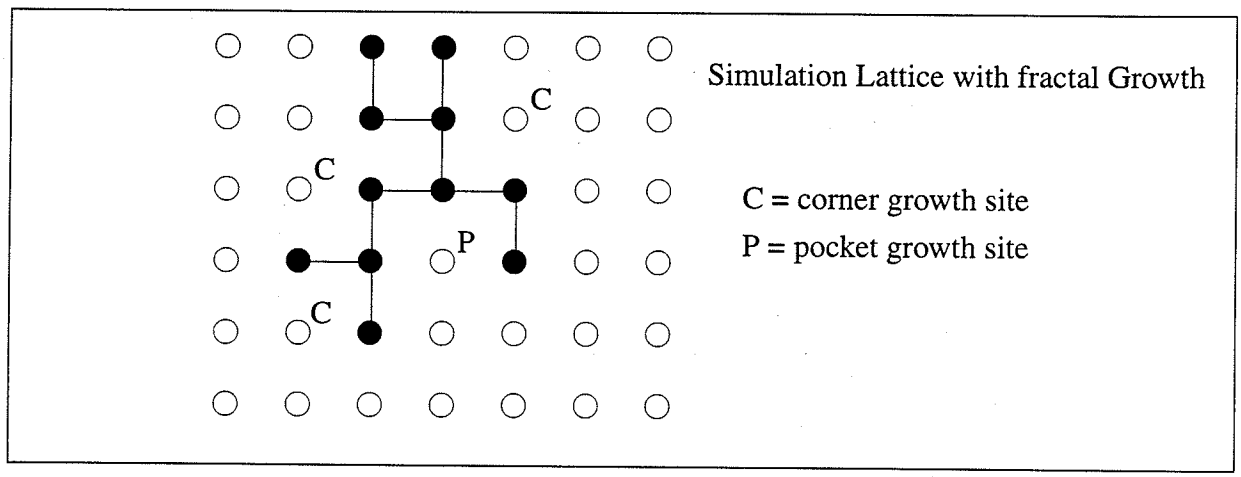


Fig. 5.7. Simulation lattice showing corner and pocket growth sites.

and subsequently higher dimension than what should happen. The corners and pockets formed in the lattice during growth are depicted in Fig. 5.7 .

To reflect what we intuitively feel should happen at the inside corners and pockets formed during simulated fractal growth, a change to the model is necessary. Instead of looking at fractal growth as branch growth from a location on in the fractal to a free adjoining location based upon gradient (which invites counting corners twice or pockets three times), simulated growth should look at the growth as additions of particles at the free locations nearest the fractal based upon the gradient at these free locations. Then the locations of the corners and pockets would only be considered once for each stage of the fractal growth process.

Naturally, such a change would involve a much different algorithm for calculating where the growth site would be. The model should be changed from the current equation to the new equation given below.

$$P(i, j \rightarrow i', j') = \frac{\phi_{i', j'}^{\eta}}{\Sigma \phi_{i', j'}^{\eta}} \qquad \text{current model} \qquad (5.1)$$

$$P(i', j') = \frac{\phi_{i', j'}^{\eta}}{\Sigma \phi_{i', j'}^{\eta}} \quad \text{where } (i', j') \text{ is adjacent to } F \qquad \text{new model} \qquad (5.2)$$

All other parts of the models implementation can be left as is. With this change, the dilation dimension of the resulting simulations drops but insignificantly. Tuning the new model in the same fashion yields a value for η of 1.50. Thus removing the overlap of branch additions does not change the results radically.

If the simulated fractals are looked at using the full dilation dimension subroutine, both models produce the same multifractal dimension. It appears that differences in dimension between the new and original models only occurs at the smallest scales close to the size of a single pixel. Thus the global or large scale characteristics of the simulated fractals from both the original and new models are equivalent. This can be seen by comparing the two images in Fig. 5.8 . The maximum dimension of the simulated fractals occurs at a dilation size of 15 pixels on the 500

by 500 pixel simulations run.

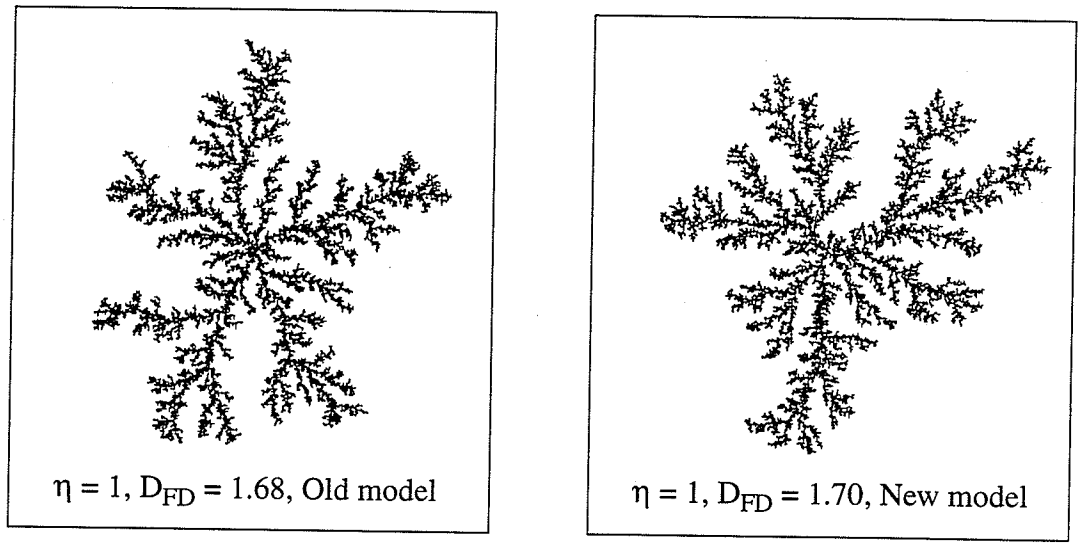


Fig. 5.8. Old and new stochastic model images.

Does the actual experiment exhibit only a diffusion limited process? Clearly it does not since the fractal growths are fat and not thin as the model (current or new) predicts. This is not surprising when one imagines of how the actual growths form. Instead of single particles attaching to the growing aggregate, many millions of molecules diffuse to the points of highest gradient or probability. These molecules are charged ions that will interact with each other as they approach and attach to the growing fractal growth. Consideration of this process is not being done by the current or new model but might be effectively mimicked if the gradient threshold mechanism mentioned earlier was introduced into the model. This idea also provides some justification for considering only the boundary of these experimentally created aggregates.

5.10 Tuning the Stochastic Model to Match Actual Electrical Discharges

This experiment used the same procedure for tuning as the previous experiment for DLAs. The image used for this test is a lichtenberg figure obtained from the literature [GuSt91]. The image was scanned using a hand scanner and the result stored to disk. The quality of this image was not as good as that obtained for the experimental DLAs. There is unfortunately only one image as well.

The image enhancement procedure detailed for the previous experiment was again used. This begins with inverting or negating the image to obtain the white fractal on black background required. Then normalization of the pixel intensities in the image follows with final thresholding at the optimal intensity value of 131. Thus the total fractal image results. The edges are subsequently detected using the 25 point exponential edge detector to remove the interior of the fractal as before. The image obtained from the edge detector is again thresholded at 50 giving the white edges of the fractal on the black background.

Following the enhancement procedure, the fractal dimension was again estimated using the dilation dimension subroutine. The dimension of the lichtenberg figure measured 1.31. This is surprisingly close to the average value of 1.32 obtained for the six DLAs in the previous experiment. Since the model has produced parallel trees with similar dimensions to the circular trees, (Fig. 4.13 , page 74 and Fig. 4.12 , page 74), the result from tuning in the previous experiment (η=1.65) was used as the starting point for this experiment.

The results shown in Fig. 5.9 , page 100 show this initial starting point of η=1.65 is sufficient to tune the stochastic model to the lichtenberg figure. In the previous experiment, the tuning appeared reasonable. One can look at those results and agree the tuned model and the actual DLA data had similar characteristics.

When comparing the results from the tuned model to the lichtenberg figure one would have difficulty saying the results were similar. Why?

First note the biggest difference appears to be that the angles between the branches in the lichtenberg figure and that in either of the model images does not match. This could be explained if the field distribution in the lichtenberg figure was not the same as for the simulated images created by the stochastic model. Recall that in the model simulated images, the fractals are assumed to be perfectly conducting. If this is not the case in reality then the branches of the lichtenberg figure do not as effectively shield each other thus leading to a different field profile. This altered field profile would allow branching to occur at angles less than that observed with the current sto-

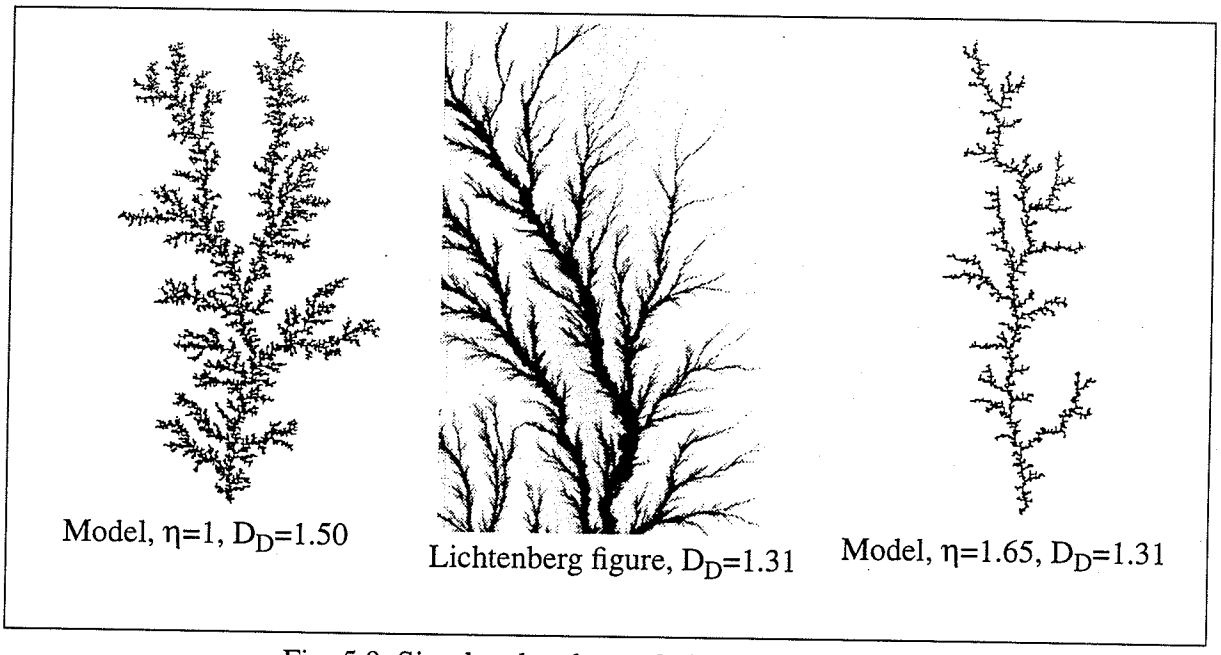


Fig. 5.9. Simulated and actual electrical discharges.

chastic model images. This could also explain why the number or density of branches does not seem to match either.

Another problem occurs when using the dilation dimension routine as the measure for tuning the model in both tuning experiments. The dilation dimension gives the dimension of the fractal at the smaller scales of the image. Thus tuning the model at the smaller scales will give correct results at the smaller scales but not necessarily at the larger scales. In the DLA tuning experiment the data from all the DLA images appeared to be most consistent at the smaller scales. This made choosing the dilation dimension as the comparison measure the logical choice. In this experiment only one image was available so the choice whether to use the dilation or full dilation dimension measures was not as clear. The choice of the dilation dimension was made to be consistent with the previous experiment.

The conclusion drawn with this experiment is that making the simulated fractal perfectly conducting is a big restriction with the current model. The computational requirements and program complexity would unfortunately be much greater if the simulated fractal were allowed to be only partially conducting. Such a development would also help in removing the restriction that

the starting point of the simulated fractals must be one of the electrodes. Removing this restriction would allow the possibility of simulated fractal growth starting from non-homogeneous areas in the dielectric media.

5.11 Summary

Experimentation started with testing the accuracy of the simple box, correlation and new dilation dimension routines on the L-systems generated test fractals. Next, the experiments checked the limitations of using the new full dilation dimension routine. Finally, to show the usefulness of the full dilation dimension routine, a comparison of results on stochastic model fractals with the correlation dimension and full dilation dimension routines was given.

In order to show how image or sub-image size affects the accuracy of the dimension routines, the L-systems images were scaled or cropped and tested using the simple box and dilation dimension routines. This shows how large an image needs to be to achieve a given dimension accuracy.

When an experimental image is digitized from a picture or negative, the result is usually greyscale. In order to perform a calculation of the dimension, the image must be enhanced and then thresholded into a black and white image. The threshold experiment begins by taking the L-systems test fractals and printing them onto paper with a laser printer. The image is then scanned back into the computer as a greyscale image. The effects of varying thresholds on the resulting dimension was then tested. This tested the limitations of the testing procedure on this source of data.

Next, a simple experiment was run to generate some actual DLAs using copper electrodes. Images of DLAs in books appeared to differ from the results from the stochastic model and it was thought that this was due to limits in printing the image in the book. The experimental DLAs were very similar to the images in the books and this lead to discussion of the possible reasons.

The structure of Laplacian fractals is discussed using some of the results and discoveries made in the previous experiments. Accuracy limitations of the dimension estimating routines does not allow investigation of the spatial variations of dimension on the current images.

The final series of experiments test the procedure of running and tuning the model to simulate both DLAs and Lichtenberg figures. Several limitations were noted and areas for improvement suggested.

This ends the experimental work of this thesis. Two major areas or directions for further research were identified. These are further investigation into the errors and limitations of dimension estimating routines and the creation of new stochastic models to improve the match between actual DLAs or electrical discharges and their simulated counterparts.

CHAPTER 6
CONCLUSIONS AND RECOMMENDATIONS

The experimentation with the L-systems test fractals, fractal dimension algorithms, and stochastic models has shown limitations in each area. Theirfore, the conclusions drawn follow under two sections; Fractal Dimension Estimation and The Stochastic Model.

6.1 Fractal Dimension Estimation

The overall accuracy of the dimension algorithms was found to be reasonable on large images with the new dilation dimension algorithm giving the least error of 0.3% on the L-systems test fractal images. These images are special conditions, however, since they are pure (mathematical) fractals with only one underlying dimension. Such accuracy does not transfer to measurements on multifractal images. The new full dilation dimension algorithm whose use is proposed for both types of fractals gives the next smallest error of approximately 1%. The error with these two algorithms also appears to be systematic and it is believed further investigation and research into this area could improve the results. These algorithms also represent the major contribution of this thesis.

Results from testing the simple box dimension and correlation dimension shows the accuracy of these routines to be only 2%. The errors with these routines does not appear to be systematic and further research into these algorithms may not be as fruitful. The correlation and full dilation dimension routines used on images created with the existing stochastic model gives identical results within the errors noted with these algorithms. This indicates that the full dilation dimension routine is useful on multifractals as well as fractals with only one underlying dimension.

A greyscale image must be converted to a black and white image before calculating the dimension. Choosing the optimal threshold is very important to minimize errors. The optimal threshold for an 8-bit greyscale image was identified through experiment to be 131. After nor-

malizing the image this value was used to threshold the image. The error introduced at this threshold is approximately 2% when converting the image to a white fractal on a black background.

Naturally, image size is very important when calculating its dimension. Smaller images contain less data and will produce greater errors. The accuracy of the dilation dimension algorithm falls from about 0.3% on a 1024x1024 pixel image to 8% on images of size 64x64 pixels when considering image size alone. It is recommended that images of size 512x512 pixels or larger be used when calculating dimension. With images this size and larger, the limitations of the image enhancement procedure becomes the limiting factor to dimension accuracy.

Another problem occurs when performing dimension calculations on parts of an image. Cropping a fractal image introduces boundary effects that are different from the effects of smaller images. The effects of such image cropping appear opposite for the dilation and simple box dimension routines making averages of these results more accurate than the results from either.

Currently, estimating fractal dimension is severely limited. Fractal dimension algorithm accuracy is very sensitive to image quality, image size, and truncation or boundary effects. Estimating fractal dimension on all types of fractals to better than one percent is clearly difficult. Image processing becomes the largest source of error introducing approximately a two percent error. The best multifractal dimension method (full dilation dimension) used here is was also only good to one percent giving a total error band of three percent. Further research into image enhancement procedures would provide the best return on investment for improving the estimation of dimension. This coupled with research into limiting or compensating for the boundary effects on fractal dimension measurements becomes one recommendation for future work.

6.2 The Stochastic Model

Growing diffusion limited aggregates or DLAs with copper and copper sulphate solution has shown that only the boundary of such growths or any real world fractal image can be considered when calculating dimension or when simulating such growths with the current stochastic

model. The true thickness present in the DLA or fractal image pushes the dimension results close to the full dimension of 2 as expected for objects with true area. The current stochastic model does not incorporate any branch thickening mechanism and can only match the fractal aspects of the boundaries of true DLAs.

Multifractals exhibit a variation of dimension with image scale or spatial location. Some of this variation can be attributed to the definition of the pixel used and some to the true area occupied by real fractal images. No evidence was found that images produced by the stochastic model had any spatial variation of dimension though such results were limited by the inaccuracy of the dimension routines on small sub-images.

Choosing the number of field iterations to use when running the stochastic model is a judgement decision. Combining 8 global iterations with 50 local field iterations produced field error rates of 10^{-4} per unit when the model exponent η was 1. These values are within the range recommended by others. Maintaining this or better field error rate is felt necessary when tuning the model to match DLA growths or Lichtenberg figures. Introducing acceleration factors into the field calculations would produce large savings of computer time by reducing the number of field iterations necessary to achieve this field accuracy.

Tuning the stochastic model to the exterior boundary of experimentally created DLAs yielded a value for η of 1.65. With this value of η the images created with the model and that from experiment had identical fractal dimensions. The structure of both simulated and actual DLA images also appears to be a good match. Here, introduction of a field gradient threshold is recommended. Below such a threshold growth would not occur. A threshold mechanism might produce simulated fractals with greater branch thickness thus better matching the growth of actual DLAs.

Performing the same model tuning experiment with a Lichtenberg figure yielded the same value of 1.65 for η. Here however, the structure of the Lichtenberg figure and the simulated discharge image and the did not appear to match as well as for the previous tuning experiment.

Assuming the electrical discharge tree is perfectly conducting is possibly the major cause of difference here. Limiting the conduction of the fractals produced by the model is recommended. With a new model incorporating partially conducting fractals, many limitations with the current stochastic model could be removed. Currently, simulated growth is limited to one of the electrodes. A model incorporating limited conduction of the fractal growth would be much more complicated and slower to run but would allow greater flexibility for tuning and matching real stochastic events.

6.3 Summary

Estimating the fractal dimension of images is time consuming and inaccurate. The full dilation dimension algorithm introduced in this thesis appears to give better results than the simple box dimension or correlation dimension algorithms, two commonly used methods for estimating the fractal dimension of pure (mathematical) fractals. The full dilation dimension algorithm has obvious limitations that might be removed with further research. There also appears to be a one to one relationship between the full dilation and correlation dimension algorithms when estimating a single dimension of multifractals. This suggests that differences of results between fractal dimension algorithms occurs because each method emphasizes features at different scales.

Current stochastic fractal models are limited in scope. Tuning the existing model to DLAs and to a Lichtenberg figure met with mixed results. In the case of the DLAs, the current model fails to create the thick branches present in the actual DLAs. The actual DLAs created in thesis also exhibited more than a simple diffusion limited process complicating analysis. The structure of the Lichtenberg figure was not matched with the tuned model either. Extensions to the current model such as a growth threshold and limited conduction of the simulated fractal growth are recommendations.

The study of fractals and fractal models is a relatively new field. Further research in any direction is bound to find interesting and relevant details towards a fuller understanding of fractals both in theory and in nature.

REFERENCES

[Acto70] Acton, F.S., "Numerical Methods that Work", Harper and Row, New York, N.Y. 1970.

[Baxs84] Baxes,G. A., "Digital Image Processing, A Practical Primer", Prentice-Hall, NJ, 1984.

[Daub79] Dauben, J. W., "Georg Cantor, His mathematics and Philosophy of the Infinite", Princeton University Press, Princeton, NJ, 1979.

[DiSF92] Dissado,L.A., Sweeney,P.J., Fothergill, J.C., "Field Dependence of Electrical Field Growth", *Sixth International Conference on Dielectric Materials, Measurements and Applications*, IEE Conference Publications n363, Sept. 1992, pp. 13-16.

[Edga90] Edgar, G.A., "Measure, Topology, and Fractal Geometry", Springer-Verlag, New York Inc, 1990.

[Falc92] Falconer,K., "The Geometry of Fractal Sets", University Press, Cambridge, Great Britain, 1992.

[Falc90] Falconer,K., "Fractal Geometry, Mathematical Foundations and Applications", John Wiley & Sons, West Sussex, England, 1990.

[Fuji91] Fujimori, S., "Fractal Analysis of Tree Figures", *Proceedings of the 3rd International Conference on Properties and Applications of Dielectric Materials*, Tokyo, Japan, 1991, pp. 131-134.

[FuSa87] Fukai, J., and Sano N., "Tree Growth Simulation associated with Dielectric Breakdown", *IEEE 1987 Report, Conference On Electrical Insulation and Dielectric Phenomena*, 1987, pp. 432-439.

[GuSt91] Guyon, E., Stanley, H.E., "Fractal Forms", Elsevier Science Publishers B.V. and Palais de la Decouverte, 1991.

[Kins94] Kinsner, W., "Fractal Dimensions: Morphological, entropy, spectral and variance classes", *Technical Report DEL94-4*, Dept. of Electrical and Computer Eng., University of Manitoba, May 31, 1994, 146 pp.

[KiSt93] Kinsner, W., Stacey, G., "Computer modelling of Laplacian fractals in dielectric discharge phenomena", *Journal of Mathimatical & Computer Modelling*, ICMCM'93, Berkeley, CA, July 26-29,1993, in press.

[KiSK94] Kinsner, W., Stacey, G., Kuffel, E., "Multifractals in dielectric discharge phenomena", *7th International Symposium on Gaseous Dielectrics*, Oak Ridge National

Labs, TN, April 3-4, 1994, in press.

[KiSt94] Kinsner, W., Stacey, G., "Dilation Fractal Dimension for Laplacian Multifractals", *Proc. Canadian Conference on Electrical and Computer Engineering*, (Halifax N.S.; Sept 25-28,1994), IEEE Cat. No. 94TH8023, 1994,pp. 401-404.

[KuMa91] Kudo, K. Maruyama, S., "Studies on the Fractal Properties of Electrical Trees", *Proceedings of the 3rd International Conference on Properties and Applications of Dielectric Materials*, New York, IEEE, 1991, pp. 135-138.

[Lind68] Lindenmeyer, A., "Mathematical models for cellular interaction in development", *Parts 1 and 2, Journal of Theoretical Biology 18*, 1968, pp. 280-315.

[Mand82] Mandelbrot, B.B. "The Fractal Geometry of Nature", W.H.Freeman and Co., New York, 1982.

[MaSu93] Maragos, P., Sun, F., "Measuring the Fractal Dimension of Signals: Morphological Covers and Iterative Optimization", *IEEE Transactions of Signal Processing*, Vol 41, No. 1, January 1993, pp. 108-121.

[MeMa92] Mee, C., Macinlay, R., "Parameters for Tree Growth in Cable Insulation", *Sixth International Conference on Dielectric Materials, Measurements and Applications*, IEE Conference Publications n363, Sept. 1992, pp. 334-337.

[NiPW84] Niemeyer, L., Pietronero, L., and Wiesmann, H.J., "Fractal Dimension of Dielectric Breakdown", *Physical Review Letters*, volume 52, number 12, 19 March 1984, pp. 1033-1036.

[PiEW86] Pietronero, L., Evertsz, C., Wiesmann, H.J., "Scaling Properties of Growing Zone and Capacity of Laplacian Fractals", *Fractals in Physics*, Elsevier Science Publishers B.V., 1986, pp. 159-163.

[PeJS92] Peitgen, H.O. , Jürgens, H., Saupe, D. , "Fractals for the Classroom", Springer-Verlag, New York, 1992.

[Satp86] Satpathy, S., "Dielectric Breakdown in Three Dimensions", *Fractals in Physics*, Elsevier Science Publishers B.V., 1986, pp. 173-176.

[ShCa92] Shen, J., Caston, S., "An Optimal Linear Operator for Step Edge Detection", *Graphical Models and Image Processing*, Vol 54, No. 2, March 1992, pp. 112-133.

[Schr91] Schroeder, M., Fractals, "Chaos, Power Laws, Minutes from an Infinite Paradise", W.H. Freeman and Company, New York, 1991.

[StMe88] Stanley,H.E., Meakin, P., "Multifractal phenomena in physics and chemistry", *Nature*, vol 335, Sept. 1988, pp. 405-409.

[Thei90] Theiler, J., "Estimating fractal dimension", *Journal of the Optical Society of America A Optics and Image Science*, Vol 7, No 6, June 1990, pp. 1055-1073.

[WiPi86] Wiesmann, H.J., Pietronero, L., "Properties of Laplacian Fractals for Dielectric Breakdown in 2 and 3 Dimensions", *Fractals in Physics*, Elsevier Science Publishers B.V., 1986, pp. 151-157.

[WiSa81] Witten, T.A., Sander, L.M., "Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon", *Pysical Review Letters*, volume 47, number 19, Nov. 1981, pp. 1400-1405.

[Usma87] Usmani, R.A., "Applied Linear Algebra", Marcel Dekker, Inc., New York, New York, 1987, pp. 84-97.

[ZhBM90] Zhang, P. , Barab, H., Martinez, A. , "Fractal Dimension Estimation of Fractional Brownian Motion", *Southeastcon '90 Proceedings*, IEEE New York, N.Y., Vol. 3, 1990, pp. 934-939.

APPENDICES

A Data for Graphs used in Thesis

A1 Data for plot in Fig. 5.1 , page 84.

Experiment to find the dimension error vs image size relationship. The 9 test fractals are decimated in size to check how image size affects the dimension calculating routine error.

Box Dimension:

Fractal #	Dimension	1024x1024	error %	512x512	error %
a	1.1042	1.0799	-2.2007	1.0835	-1.8747
b	1.2619	1.2551	-0.5389	1.2617	-0.0158
c	1.3403	1.3268	-1.0072	1.3049	-2.6412
d	1.5000	1.4906	-0.6267	1.4895	-0.7000
e	1.4650	1.4551	-0.6758	1.4543	-0.7304
f	1.4650	1.4661	0.0751	1.4550	-0.6826
g	1.5282	1.5198	-0.5497	1.5170	-0.7329
h	1.5736	1.5719	-0.1080	1.5791	0.3495
i	1.5850	1.5749	-0.6372	1.5791	-0.3722
Max error:			0.08		0.35
Min error:			-2.20		-2.64

Fractal #	Dimension	256x256	error %	64x64	error %
a	1.1042	1.1558	4.6731	1.1171	1.1683
b	1.2619	1.2886	2.1159	1.3002	3.0351
c	1.3403	1.3480	0.5745	1.3145	-1.9249
d	1.5000	1.5194	1.2933	1.5259	1.7267
e	1.4650	1.4610	-0.2730	1.4615	-0.2389
f	1.4650	1.4523	-0.8669	1.4474	-1.2014
g	1.5282	1.5294	0.0785	1.5097	-1.2106
h	1.5736	1.5892	0.9914	1.5646	-0.5719
i	1.5850	1.5957	0.6751	1.6041	1.2050
Max error:			4.67		3.04
Min error:			-0.87		-1.92

Fractal #	Dimension	64x64	error %
a	1.1042	1.0280	-6.9009
b	1.2619	1.2011	-4.8181

c	1.3403	1.2358	-7.7968
d	1.5000	1.4921	-0.5267
e	1.4650	1.3775	-5.9727
f	1.4650	1.4557	-0.6348
g	1.5282	1.4630	-4.2665
h	1.5736	1.5176	-3.5587
i	1.5850	1.6114	1.6656
Max error:			1.67
Min error:			-7.80

Dilation Dimension:

Fractal #	Dimension	1024x1024	error %	512x512	error %
a	1.1042	1.1071	0.2626	1.1053	0.0996
b	1.2619	1.2622	0.0238	1.2642	0.1823
c	1.3403	1.3403	0.0000	1.3336	-0.4999
d	1.5000	1.4967	-0.2200	1.4945	-0.3667
e	1.4650	1.4622	-0.1911	1.4595	-0.3754
f	1.4650	1.4617	-0.2253	1.4538	-0.7645
g	1.5282	1.5251	-0.2029	1.5209	-0.4777
h	1.5736	1.5723	-0.0826	1.5678	-0.3686
i	1.5850	1.5815	-0.2208	1.5812	-0.2397
Max error:			0.26		0.18
Min error:			-0.23		-0.76

Fractal #	Dimension	256x256	error %	128x128	error %
a	1.1042	1.1112	0.6339	1.0995	-0.4256
b	1.2619	1.2544	-0.5943	1.2683	0.5072
c	1.3403	1.3284	-0.8879	1.3074	-2.4547
d	1.5000	1.4878	-0.8133	1.4792	-1.3867
e	1.4650	1.4461	-1.2901	1.4193	-3.1195
f	1.4650	1.4584	-0.4505	1.4464	-1.2696
g	1.5282	1.5103	-1.1713	1.4845	-2.8596
h	1.5736	1.5617	-0.7562	1.5343	-2.4975
i	1.5850	1.5782	-0.4290	1.5680	-1.0726
Max error:			0.63		0.51
Min error:			-1.29		-3.12

Fractal #	Dimension	64x64	error %
a	1.1042	1.0912	-1.1773

b	1.2619	1.2265	-2.8053
c	1.3403	1.2614	-5.8867
d	1.5000	1.4479	-3.4733
e	1.4650	1.3534	-7.6177
f	1.4650	1.4024	-4.2730
g	1.5282	1.4333	-6.2099
h	1.5736	1.4907	-5.2682
i	1.5850	1.5419	-2.7192
Max error:			-1.18
Min error:			-7.62

A2 Data for plot in Fig. 5.2 , page 86.

This sheet contains the raw data for the plot of dimension error vs. sub image size. Each fractal has 5 sub images selected at given sizes. The results are then averaged to give a percent error for both dimension routines. B = box dimension, D = dilation dimension

a) LSA 1.1042

B512	D512	B256	D256	B128	D128	B64	D64
1.1043	1.1042	1.0956	1.0960	1.0921	1.0779	1.1271	1.0482
1.1341	1.1073	1.1359	1.1001	1.0965	1.0701	1.1236	1.0553
1.1143	1.1071	1.1054	1.1035	1.1090	1.0869	1.0652	1.0613
1.1215	1.1060	1.0827	1.1023	1.1293	1.0723	1.0257	1.0668
1.1116	1.1062	1.1128	1.0975	1.0795	1.0772	1.1118	1.0568
1.17%	0.18%	0.21%	-0.39%	-0.26%	-2.47%	-1.22%	-4.21%

b) LSB 1.2619

B512	D512	B256	D256	B128	D128	B64	D64
1.2900	1.2605	1.2682	1.2490	1.2333	1.2435	1.2747	1.2371
1.2563	1.2572	1.2520	1.2534	1.2973	1.2383	1.3069	1.2270
1.2538	1.2601	1.2490	1.2496	1.2470	1.2498	1.2095	1.2102
1.2505	1.2485	1.2805	1.2544	1.2202	1.2402	1.2506	1.2239
1.2526	1.2563	1.2804	1.2505	1.2418	1.2359	1.2666	1.2187
-0.10%	-0.43%	0.33%	-0.83%	-1.11%	-1.61%	-0.02%	-3.05%

c) LSC 1.3403

B512	D512	B256	D256	B128	D128	B64	D64
1.3491	1.3393	1.3817	1.3350	1.3863	1.3475	1.3832	1.3305

1.3287	1.3423	1.3702	1.3253	1.3687	1.3281	1.3584	1.3309
1.3485	1.3451	1.3491	1.3173	1.3616	1.3118	1.5321	1.3394
1.3443	1.3422	1.3414	1.3335	1.3819	1.3124	1.3624	1.3021
1.3608	1.3389	1.3404	1.3125	1.3317	1.3008	1.3647	1.3079
0.45%	0.09%	1.21% -	1.16%	1.92%	-1.51%	4.47%	-1.35%

d) LSD 1.5000

B512	D512	B256	D256	B128	D128	B64	D64
1.4971	1.4885	1.4978	1.4888	1.4783	1.4764	1.5280	1.4641
1.5023	1.4930	1.4866	1.4886	1.5060	1.4864	1.5438	1.4574
1.4933	1.4872	1.5249	1.4828	1.5331	1.4812	1.5750	1.4639
1.5029	1.4933	1.4910	1.4887	1.5000	1.4841	1.5801	1.4588
1.4951	1.4858	1.4841	1.4824	1.5148	1.4788	1.5422	1.4692
-0.12%	-0.70%	-0.21%	-0.92%	0.43%	-1.24%	3.59%	-2.49%

e) LSE 1.4649

B512	D512	B256	D256	B128	D128	B64	D64
1.4527	1.4560	1.4902	1.4453	1.4241	1.4000	1.4587	1.4129
1.4562	1.4623	1.4499	1.4513	1.4988	1.4395	1.4864	1.4493
1.4457	1.4587	1.4615	1.4570	1.4722	1.4301	1.5219	1.4198
1.4563	1.4607	1.4544	1.4427	1.4621	1.4332	1.4622	1.3828
1.4521	1.4550	1.4535	1.4535	1.4335	1.4254	1.4663	1.3996
-0.84%	-0.43%	-0.20%	-1.02%	-0.46%	-2.68%	0.97%	-3.55%

f) LSF 1.4649

B512	D512	B256	D256	B128	D128	B64	D64
1.4672	1.4608	1.4709	1.4400	1.4605	1.4368	1.5658	1.4400
1.4632	1.4614	1.4741	1.4521	1.5047	1.4536	1.5267	1.4058
1.4691	1.4608	1.4751	1.4592	1.4885	1.4410	1.5303	1.4423
1.4758	1.4620	1.4630	1.4460	1.4783	1.4411	1.4783	1.4311
1.4627	1.4589	1.4715	1.4553	1.4924	1.4542	1.4484	1.4232
0.18%	-0.28%	0.41%	-0.98%	1.36% -	1.34%	3.07% -	2.49%

g) LSG 1.5282

B512	D512	B256	D256	B128	D128	B64	D64
1.5191	1.5201	1.5294	1.5127	1.5368	1.5123	1.5547	1.4963

1.5264	1.5197	1.5384	1.5130	1.5470	1.5120	1.5671	1.4883
1.5464	1.5250	1.5350	1.5194	1.5642	1.5139	1.5553	1.4567
1.5231	1.5248	1.5381	1.5219	1.5784	1.5135	1.5808	1.4914
1.5200	1.5269	1.5278	1.5174	1.5381	1.5037	1.5267	1.4729
-0.08%	-0.32%	0.36%	-0.74%	1.62%	-1.12%	1.88%	-3.08%

h) LSH 1.5736

B512	D512	B256	D256	B128	D128	B64	D64
1.5639	1.5628	1.5645	1.5680	1.5739	1.5494	1.6704	1.5772
1.5868	1.5644	1.5856	1.5735	1.6252	1.5770	1.6736	1.5856
1.5746	1.5734	1.5862	1.5733	1.5655	1.5297	1.6292	1.5221
1.5713	1.5637	1.5845	1.5710	1.6257	1.5552	1.5667	1.4468
1.5592	1.5629	1.5700	1.5656	1.6267	1.5696	1.6893	1.5537
-0.16%	-0.52%	0.29%	-0.21%	1.89%	-1.11%	4.59%	-2.32%

i) LSI 1.5849

B512	D512	B256	D256	B128	D128	B64	D64
1.5748	1.5780	1.5733	1.5657	1.5820	1.5146	1.5802	1.4821
1.5822	1.5743	1.5791	1.5458	1.5821	1.5416	1.6154	1.5177
1.5774	1.5709	1.5716	1.5634	1.5723	1.5587	1.5667	1.4854
1.5848	1.5680	1.5686	1.5674	1.6181	1.5466	1.5769	1.4806
1.5886	1.5746	1.5997	1.5525	1.6054	1.5305	1.5837	1.5333
-0.21%	-0.74%	-0.41%	-1.64%	0.45%	-2.93%	-0.02%	-5.37%

A3 Data for plot in Fig. 5.3 , page 87.

This sheet contains the raw data for checking the threshold vs. dimension. Each image is scanned, negated and equalized to fill the available black to white data range.

B.D. = Box Dimension, D.D. = Dilation Dimension

a) LSA 1.1042

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.252	1.264	13.39%	14.47%
75	1.205	1.221	9.13%	10.58%
100	1.165	1.177	5.51%	6.59%
125	1.130	1.137	2.34%	2.97%

150	1.074	1.078	-2.74%	-2.37%
175	0.994	1.000	-9.98%	-9.44%
200	0.859	0.865	-22.21%	-21.66%

b) LSB 1.2619

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.394	1.398	10.47%	10.79%
75	1.354	1.361	7.30%	7.85%
100	1.318	1.324	4.45%	4.92%
125	1.280	1.286	1.43%	1.91%
150	1.228	1.238	-2.69%	-1.89%
175	1.158	1.172	-8.23%	-7.12%
200	1.036	1.057	-17.90%	-16.24%

c) LSC 1.3403

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.461	1.462	9.01%	9.08%
75	1.436	1.432	7.14%	6.84%
100	1.405	1.400	4.83%	4.45%
125	1.370	1.367	2.22%	1.99%
150	1.330	1.327	-0.77%	-0.99%
175	1.279	1.273	-4.57%	-5.02%
200	1.174	1.169	-12.41%	-12.78%

d) LSD 1.5000

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.582	1.581	5.47%	5.40%
75	1.556	1.555	3.73%	3.67%
100	1.530	1.528	2.00%	1.87%
125	1.500	1.500	0.00%	0.00%
150	1.464	1.466	-2.40%	-2.27%
175	1.420	1.421	-5.33%	-5.27%
200	1.336	1.338	-10.93%	-10.80%

e) LSE 1.4649

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.542	1.542	5.26%	5.26%
75	1.511	1.514	3.15%	3.35%
100	1.483	1.484	1.24%	1.30%
125	1.455	1.455	-0.68%	-0.68%

150	1.421	1.422	-3.00%	-2.93%
175	1.372	1.373	-6.34%	-6.27%
200	1.286	1.288	-12.21%	-12.08%

f) LSF 1.4649

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.561	1.562	6.56%	6.63%
75	1.532	1.536	4.58%	4.85%
100	1.502	1.508	2.53%	2.94%
125	1.475	1.479	0.69%	0.96%
150	1.437	1.444	-1.90%	-1.43%
175	1.383	1.391	-5.59%	-5.04%
200	1.280	1.287	-12.62%	-12.14%

g) LSG 1.5282

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.605	1.608	5.03%	5.22%
75	1.579	1.583	3.32%	3.59%
100	1.551	1.537	1.49%	0.58%
125	1.522	1.528	-0.41%	-0.01%
150	1.490	1.496	-2.50%	-2.11%
175	1.443	1.447	-5.58%	-5.31%
200	1.340	1.344	-12.32%	-12.05%

h) LSH 1.5736

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.655	1.656	5.17%	5.24%
75	1.632	1.633	3.71%	3.77%
100	1.608	1.611	2.19%	2.38%
125	1.584	1.587	0.66%	0.85%
150	1.554	1.558	-1.25%	-0.99%
175	1.518	1.520	-3.53%	-3.41%
200	1.439	1.440	-8.55%	-8.49%

i) LSI 1.5849

Thr.	B.D.	D.D.	B.D. err	D.D. err
50	1.673	1.676	5.56%	5.75%
75	1.649	1.651	4.04%	4.17%
100	1.622	1.625	2.34%	2.53%
125	1.589	1.592	0.26%	0.45%

150	1.550	1.553	-2.20%	-2.01%
175	1.488	1.492	-6.11%	-5.86%
200	1.353	1.355	-14.63%	-14.51%

B Listing of Subroutines and Location

Next includes all the listings of all the programs written for this thesis. There are many subroutines distributed among several files. In order to facilitate location of the various routines, a short list of the subroutines is given first to identify in which file they reside and then their function.

FILE: TEST.C

This file contains the shell routine which calls all the other subroutines.

FILE: DIMEN.C

BOXDIMENSION - Calculates the box dimension of the picture in memory
DILATIONDIM - Calculates the dilation dimension of the picture in memory.
LKOCH - Creates the stadard KOCH curve or test fractal b) curve.
LMKOCH - Creates the test fractal c) curve.
LMKOCH2 - Creates the test fractal with overlap and questionable dimension.
LMKOCH3 - Creates the test fractal a) curve.
LKOCH15 - Creates the standard KOCH type 1.5 curve or test fractal d).
LTREE1 - Creates the test fractal e) curve.
LTREE2 - Creates the test fractal f) curve.
LTREE3 - Creates the test fractal g) curve.
LTREE4 - Creates the test fractal h) curve.
LSGASKET - Creates the standard Sierpinski Gasket or test fractal i).
LENGINE - Recursive L-Systems generator subroutine.

FILE: EDGES.C

LAPLACE5PT - Edge detector using 5 point mask.
EXP9PT - Edge detector using 9 point mask, exponential weights.
EXP13PT - Edge detector using 13 point mask, exponential weights.
EXP25PT - Edge detector using 25 point mask, exponential weights.
GAUSS9PT - Edge detector using 9 point mask, gaussian weights.
GAUSS13PT - Edge detector using 13 point mask, gaussian weights.
GAUSS25PT - Edge detector using 25 point mask, gaussian weights.

FILE: FULD.C

FULDILATION - Full Dilation Dimension multifractal measurement subroutine.

lubksb - Numerical Recipes in C matrix solving subroutine.
lubcmp - Numerical Recipes in C matrix solving subroutine.
free_vector - Numerical Recipes in C subroutine.
nrerror - Numerical Recipes in C subroutine.
*vector - Numerical Recipes in C subroutine.
MASSCORR - Correlation Dimension multifractal measurement subroutine.

FILE: PIC_DRAW.C

HIGHPASS - High pass filter using 5 point mask.
PIC_DRAW - Subroutine to draw memory to screen windows.
LINREG - Linear regression routine calculates slope of data given.
DRAW_BORDER - Subroutine to initialize graphics palette and screen borders.
DRAW_MESSAGE - Subroutine to draw messages to screen message windows.
DRAW_MENU - Subroutine to draw the menus to the screen.
GET_MESSAGE - Subroutine to get message from the user.
PLOT_HISTOGRAM - Plots the intensity histogram to the left screen window.
NEGATE_IMAGE - Negates the image in memory to create negative image.
AUTO_SCALE - Subroutine normalizes the intensity histogram.
LOW_PASS - Low pass filter using 9 point mask.

FILE: ROUTINES.C

ZERO_CUT - Threshold detector takes all values below to zero.
MAX_CUT - Threshold detector takes all values above to maximum.
SAVE - Subroutine allocates buffer and saves current image into it.
RECALL - Subroutine takes saved image and puts it into current image array.
MEDIAN - Median filter subroutine to remove shot type noise from image.
picsrt - Bubble sort routine used by the median filter routine.
GETSUBIMAGE - Subroutine allowing selection and cutting of sub-images.
DRAW_BOX - Subroutine used by above routine to draw selection box.
TIFF_WRITE - Subroutine to write memory or dump screen to TIFF file.

FILE: SCREEN.C

DATASET - Sets the data into SCREEN structure for screen graph.
TITLES - Sets the titles to be drawn onto the screen when displaying screen graph.
DRAWBORDER - Draws the border for the screen graph.
DRAWDATA - Draws the data onto the screen graph.

FILE: SIM.C

PARTREESIM1 - Creates a parallel tree using Gauss-Seidel field iterations.
PARTREESIM2 - Creates a parallel tree using Jacobi field iterations.
CIRTREESIM1 - Creates a circ. tree using G.S. field iterations, point initiator.
CIRTREESIM2 - Creates a circ. tree using G.S. field iterations, circular initiator.
CIRTREESIM3 - New model, circular tree, G.S. field iterations, point initiator.
PARFIELD - Convergence test for parallel electrode configuration.
CIRFIELD - Convergence test for circular electrode configuration, point electrode.
CIRFIELDCAL - Embeds image in circular electrode and calculates field.

CALCFIELD1 - Calculates the field using Gauss-Seidel type iterations.
CALCFIELD2 - Calculates the field using Jacobi type iterations.
CALCGRAD - Calculates the gradient and extends the fractal stochastically.
CALCGRAD2 - Calc. the gradient using the new model and extends the fractal stochastically.
CONVERTPIC1 - Converts laplacian field array to display image.
CONVERTPIC2 - Converts simulated fractal array to image in memory.
REDUCE - Performs a test at scale reduction of the image.
ran2 - Modified "Numerical recipes in C" random number generator routine.
ANIMATE - Subroutine to animate greyscale image in memory onto screen.
sort2 - Modified "Numerical Recipes in C" routine to sort data for ANIMATE.
ITERSET - Subroutine to set the number of iterations used for all simulations.

FILE: TIF_READ.C

TIFF_READ - Reads TIFF file from disk into memory and PICTURE structure.
SWAP2 - Swaps two bytes for little endian to big endian file conversions.
SWAP4 - Swaps four bytes for little endian to big endian file conversions.
SWAPIFD - Swaps the IFD information to reflect processor format.
GET_DATA - Subroutine to locate and read blocks of data from TIFF files.

FILE: PICTURE.H - Contains the definition for the PICTURE structure used for all images.

FILE: LMAP.H - Contains the LMAP structure used for all L-Systems subroutines.

FILE: SCREEN.H - Contains the SCREEN structure used for all screen plotting routines.

C Actual C Code Listings for Routines used in Thesis

C1 FILE: TEST.C

```
/*=====================================================================*/
/*                              Test program                           */
/*=====================================================================*/

/*===========================Include files=============================*/

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <dos.h>
#include <graph.h>
#include <string.h>
#include <conio.h>
#include <i86.h>
#include "picture.h"


/*                              function prototypes                    */
```

```
extern void DRAW_BORDER(void);
extern int  DRAW_MENU(int menu);
extern void DRAW_MESSAGE(int side,char *msg);
extern int  GET_MESSAGE(int side, char *file);
extern void TIFF_READ(char *file);
extern void TIFF_WRITE(char *file, int mode);
extern int  LKOCH(void);
extern int  LMKOCH(void);
extern int  LMKOCH2(void);
extern int  LMKOCH3(void);
extern int  LSGASKET(void);
extern int  LKOCH15(void);
extern int  LTREE1(void);
extern int  LTREE2(void);
extern int  LTREE3(void);
extern int  LTREE4(void);
extern void BOXDIMENSION(float *dimension);
extern void CUBEDIMENSION(float *dimension);
extern void BOXDIMEN2(float *dimension);
extern void BOXDIMEN3(float *dimension);
extern void MASSDIMEN(float *dimension,struct PICTURE *PIC);
extern void DILATIONDIM(float *dimension,struct PICTURE *PIC);
extern void FULDILATION(float *dimension,struct PICTURE *PIC);
extern void DILATIONDIM2(float *dimension,struct PICTURE *PIC);
extern void NEGATE_IMAGE(void);
extern void PLOT_HISTOGRAM(void);
extern void AUTO_SCALE(void);
extern void LOW_PASS(void);
extern void HIGH_PASS(void);
extern int  ZERO_CUT(void);
extern int  MAX_CUT(void);
extern void MEDIAN(void);
extern void LAPLACE5PT(void);
extern void EXP9PT(void);
extern void EXP13PT(void);
extern void EXP25PT(void);
extern void GAUSS13PT(void);
extern void GAUSS9PT(void);
extern void GAUSS25PT(void);
extern void SAVE(void);
extern void RECALL(void);
extern int  GETSUBIMAGE(void);
extern void PIC_DRAW(int side,struct PICTURE *DISP);
extern int PARTREESIM1(struct PICTURE *PIC);
extern int PARTREESIM2(struct PICTURE *PIC);
extern int CIRTREESIM1(struct PICTURE *PIC);
extern int CIRTREESIM2(struct PICTURE *PIC);
extern int CIRTREESIM3(struct PICTURE *PIC);
extern void ANIMATE(void);
extern int CIRFIELDCAL(struct PICTURE *PICT);
extern int REDUCE(struct PICTURE *PICT);
extern int PARFIELD(struct PICTURE *PICT);
extern int CIRFIELD(struct PICTURE *PICT);
extern void SETITER(void);
```

```
extern void MASSCORR(float *dimension,struct PICTURE *PIC);
extern void GEN3D(void);

struct PICTURE PIC;
char *imgptr;                    /* array for putimage command          */
int imflag=0;                    /* flag for subimage routines          */
char msg1[60];                   /* global display message              */
long localiter = 50;             /* must be even!!                      */
long globaliter = 8;             /* must be even!!                      */

/*              Mouse code, used for screen position                     */

int right_button = 0;            /* global mouse variables              */
int left_button = 0;
int mouse_event = 0;
int mouse_code = 0;
int mouse_cx = 0;
int mouse_dx = 0;
int installed = 0;
struct SREGS sregs;
union REGS inregs, outregs;
int (far *function_ptr)();

#pragma off (check_stack)
void _loadds far click_handler (int max, int mcx, int mdx)
{
#pragma aux click_handler parm [EAX] [ECX] [EDX]
   mouse_event = 1;
   mouse_code = max;
   mouse_cx = mcx;
   mouse_dx = mdx;
   if( mouse_code & 8 ) right_button = 1;
   if( mouse_code & 2 ) left_button = 1;
}
#pragma on (check_stack)

/*=========================================================================*/
/*                              Main routine                               */
/*=========================================================================*/

void main()
{
   struct PICTURE *DISP=&PIC;
   char in_file[40],out_file[40],msg2[60];
   long c,d,image;
   float dimension;

/*              Mouse code, used for cursor positioning                  */

   segread(&sregs);

/*              check for mouse driver                                   */

   inregs.w.ax = 0;
```

```
   int386 (0x33, &inregs, &outregs);
   installed = (outregs.w.ax == -1);

/*              install click watcher                                   */

   if( installed )
   {

/*              set vertical resolution to 768                          */

       inregs.w.ax = 0x0008;
       inregs.w.cx = 0x0000;
       inregs.w.dx = 0x0BFC;
       int386( 0x33, &inregs, &outregs );

/*              set horizontal resolution to 1024                       */

       inregs.w.ax = 0x0007;
       inregs.w.cx = 0x0000;
       inregs.w.dx = 0x0FFC;
       int386( 0x33, &inregs, &outregs );

/*              set mickeys to pixel ratio                              */

       inregs.w.ax = 0x000F;
       inregs.w.cx = 0x0002;
       inregs.w.dx = 0x0004;
       int386( 0x33, &inregs, &outregs );

/*              install click watcher                                   */

       inregs.w.ax = 0xC;
       inregs.w.cx = 0x0002 + 0x0008;
       function_ptr = click_handler;
       inregs.x.edx = FP_OFF( function_ptr );
       sregs.es     = FP_SEG( function_ptr );
       int386x( 0x33, &inregs, &outregs, &sregs );
   }

/*              Start of program                                        */

   DRAW_BORDER();
   c=0;
   PIC.valid_pic=0;                    /* start, no picture in memory  */
   image=_imagesize(1,1,510,510);      /* reserve buffer for screen    */
   imgptr=(char *)malloc(image);       /* screen buffer for speed      */
   PIC.message[0]='\0';                /* clear                        */

   while (c!=999)
   {
      c=DRAW_MENU(0);
      switch (c)
      {
    case 1 :                       /* File selection read/write    */
```

```
 while (c!=998)              /* File selection menu          */
 {
    c=DRAW_MENU(6);
    switch (c)
    {
  case 61 :
  DRAW_MESSAGE(2,"Enter TIF file name for input :");
  d=GET_MESSAGE(2,in_file);
  if(d==27) break;
  PIC.message[0]='\0';
  TIFF_READ(in_file);
  if(PIC.file_error!=0) {
   sprintf(msg2,"File error #%3d occurred. Press key to"
        " continue.",PIC.file_error);
   DRAW_MESSAGE(2,msg2);
   d=getch();
   PIC.valid_pic=0;
   break;
  }
  else {
   sprintf(msg1,"File=%-21s %4dx%4d",in_file,
      PIC.imagewidth,PIC.imagelength);
   DRAW_MESSAGE(2,PIC.message);
   delay(2000);
   PIC.valid_pic=1;
  }
  break;

  case 62 :
  DRAW_MESSAGE(2,"Enter TIF file name for output :");
  d=GET_MESSAGE(2,out_file);
  if(d==27) break;
  TIFF_WRITE(out_file,0);
  break;

  case 63 :
  DRAW_MESSAGE(2,"Enter TIF file name for output :");
  d=GET_MESSAGE(2,out_file);
  if(d==27) break;
  TIFF_WRITE(out_file,1);
  break;

  case 64 :
  DRAW_MESSAGE(2,"Enter string (max 39 chars):");
  d=GET_MESSAGE(2,PIC.message);
  break;

  case 65 :
  DRAW_MESSAGE(2,PIC.message);
  d=getch();
  break;
    }
}
break;
```

```
    case 2 :                          /* place memory onto left screen */
       if (PIC.valid_pic==1) {
        PIC_DRAW(0,DISP);
        DRAW_MESSAGE(0,msg1);
       }
       else {
        DRAW_MESSAGE(
        2,"No picture in memory to display.  Press key to continue");
        d=getch();                          /* wait for key press   */
       }
       break;

    case 3 :                          /* place memory onto right screen */
       if (PIC.valid_pic==1) {
        PIC_DRAW(1,DISP);
        DRAW_MESSAGE(1,msg1);
       }
       else {
        DRAW_MESSAGE(2,"No picture in memory to display.  Press key to con-
tinue");
        d=getch();                /* wait for key press           */
       }
       break;

    case 4:
       c=DRAW_MENU(1);              /* L-systems creation menu      */
       switch (c)
       {

        case 11:
        d=LMKOCH3();
        if(d==0)
        {
         DRAW_MESSAGE(2,"Error allocating memory."
              " Press key to continue");
         d=getch();                  /* wait for key press   */
        }
        else
        {
        sprintf(msg1,"D=1.1042 Koch curve 1, %4dx%4d",
           PIC.imagewidth,PIC.imagelength);
        }
        break;

        case 12:                   /* create Koch curve in memory  */
        d=LKOCH();
        if(d==0)
        {
         DRAW_MESSAGE(2,"Error allocating memory."
              " Press key to continue");
         d=getch();                  /* wait for key press   */
        }
        else
```

```
 {
 sprintf(msg1,"D=1.2619 The Koch curve, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 13:
 d=LMKOCH();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.3403 Koch curve 3, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 14:
 d=LKOCH15();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.5000 Koch curve 4, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 15:
 d=LTREE1();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.4650 Tree curve 1, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 16:
 d=LTREE2();
 if(d==0)
```

```
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.4650 Tree curve 2, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 17:
 d=LTREE3();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.5282 Tree curve 3, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 18:
 d=LTREE4();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.5736 Tree curve 4, %4dx%4d",
    PIC.imagewidth,PIC.imagelength);
 }
 break;

 case 19:
 d=LSGASKET();
 if(d==0)
 {
  DRAW_MESSAGE(2,"Error allocating memory."
       " Press key to continue");
  d=getch();                    /* wait for key press    */
 }
 else
 {
 sprintf(msg1,"D=1.5850 Sierpinski Arrowhead,%4dx%4d",
    PIC.imagewidth,PIC.imagelength);
```

```
      }
      break;

      case 10:
      d=LMKOCH2();
      if(d==0)
      {
       DRAW_MESSAGE(2,"Error allocating memory."
            " Press key to continue");
       d=getch();                  /* wait for key press    */
      }
      else
      {
      sprintf(msg1,"D?=1.4650 Koch type curve, %4dx%4d",
         PIC.imagewidth,PIC.imagelength);
      }
      break;

    }
    break;

case 5 :                          /* calculate dimension menu      */
    c=DRAW_MENU(4);               /* dimension calculation menu    */
    switch (c)
    {
     case 41:                /* calculate box dimension       */
     if(PIC.valid_pic==0)
     {
      DRAW_MESSAGE(2,"No picture in memory.  Press key to continue");
      d=getch();       /* wait for key press            */
     }
     else
     {
     BOXDIMENSION(&dimension);
     sprintf(msg2,"Box Dimension = %f",dimension);
     DRAW_MESSAGE(1,msg2);
     }
     break;

     case 42:
     if(PIC.valid_pic==0)
     {
      DRAW_MESSAGE(2,"No picture in memory.  Press key to continue");
      d=getch();               /* wait for key press    */
     }
     else
     {
       DILATIONDIM(&dimension,&PIC);
       sprintf(msg2,"Dilation Dimension = %f",dimension);
       DRAW_MESSAGE(1,msg2);
     }
     break;

     case 43:                /* calculate box dimension       */
```

```
if(PIC.valid_pic==0)
{
 DRAW_MESSAGE(2,"No picture in memory.  Press key to continue");
 d=getch();      /* wait for key press           */
}
else
{
BOXDIMEN3(&dimension);
sprintf(msg2,"Better Box Dimension = %f",dimension);
DRAW_MESSAGE(1,msg2);
}
break;

case 44:
if(PIC.valid_pic==0)
{
 DRAW_MESSAGE(2,"No picture in memory.  Press key to continue");
 d=getch();              /* wait for key press    */
}
else
{
  FULDILATION(&dimension,&PIC);
  sprintf(msg2,"Full Dilation Dimension = %f",dimension);
  DRAW_MESSAGE(1,msg2);
}
break;

case 45:
if(PIC.valid_pic==0)
{
 DRAW_MESSAGE(2,"No picture in memory.  Press key to continue");
 d=getch();              /* wait for key press    */
}
else
{
  MASSCORR(&dimension,&PIC);
  sprintf(msg2,"Correlation Dimension = %f",dimension);
  DRAW_MESSAGE(1,msg2);
}
break;

case 46:
d=REDUCE(&PIC);
if(d==0)
{
 DRAW_MESSAGE(2,"Error allocating memory."
      " Press key to continue");
 d=getch();                 /* wait for key press    */
}
else
{
sprintf(msg1,"Reduced Fractal, %4dx%4d",
   PIC.imagewidth,PIC.imagelength);
}
```

```
     break;

     case 47:
     GEN3D();
     break;

    }
    break;

case 6 :
    while (c!=998)
    {
     c=DRAW_MENU(2);        /* Picture Enhancement Menu      */
     switch (c)
     {
     case 21:
      if (PIC.valid_pic==1)
      {
      NEGATE_IMAGE();
      }
      else {
      DRAW_MESSAGE(
      2,"No picture in memory to negate.  Press key to continue");
      d=getch();       /* wait for key press            */
      }
      break;

     case 22 :
      if (PIC.valid_pic==1)
      {
      PLOT_HISTOGRAM();
      DRAW_MESSAGE(1,"Pixel Intensity Histogram");
      }
      else
      {
      DRAW_MESSAGE(
      2,"No picture in memory to plot.  Press key to continue");
      d=getch();       /* wait for key press            */
      }
      break;

    case 23 :
      if (PIC.valid_pic==1)
      {
      AUTO_SCALE();
      }
      else
      {
      DRAW_MESSAGE(
      2,"No picture in memory to scale.  Press key to continue");
      d=getch();       /* wait for key press            */
      }
      break;
```

```
 case 24 :
  if (PIC.valid_pic==1)
  {
  LOW_PASS();
  }
  else
  {
  DRAW_MESSAGE(
  2,"No picture in memory to filter.  Press key to continue");
  d=getch();      /* wait for key press           */
  }
  break;

 case 25 :
  if (PIC.valid_pic==1)
  {
  HIGH_PASS();
  }
  else
  {
  DRAW_MESSAGE(
  2,"No picture in memory to filter.  Press key to continue");
  d=getch();      /* wait for key press           */
  }
  break;

 case 26 :
  if (PIC.valid_pic==1)
  {
  d=ZERO_CUT();
  }
  else
  {
  DRAW_MESSAGE(
  2,"No picture in memory to cut.  Press key to continue");
  d=getch();      /* wait for key press           */
  }
  break;

 case 27 :
  if (PIC.valid_pic==1)
  {
  d=MAX_CUT();
  }
  else
  {
  DRAW_MESSAGE(
  2,"No picture in memory to cut.  Press key to continue");
  d=getch();      /* wait for key press           */
  }
  break;

 case 28 :
  if (PIC.valid_pic==1)
```

```
        {
        MEDIAN();
        }
        else
        {
        DRAW_MESSAGE(
        2,"No picture in memory to filter.  Press key to continue");
        d=getch();      /* wait for key press        */
        }
        break;

       case 29 :
        if (PIC.valid_pic==1) {
        PIC_DRAW(0,DISP);
        DRAW_MESSAGE(0,msg1);
        d=GETSUBIMAGE();
        if(d==0)
        {
           DRAW_MESSAGE(2,"Error allocating memory."
                " Press key to continue");
         d=getch();          /* wait for key press    */
        }
        }
        else {
        DRAW_MESSAGE(
        2,"No picture in memory to display.  Press key to continue");
        d=getch();          /* wait for key press    */
        }
        break;

       case 20 :                /* place memory onto right screen */
        if (PIC.valid_pic==1) {
           PIC_DRAW(1,DISP);
           DRAW_MESSAGE(1,msg1);
        }
        else {
        DRAW_MESSAGE(
        2,"No picture in memory to display.  Press key to continue");
        d=getch();          /* wait for key press       */
        }
        break;

      }
     }
     break;

case 7 :
     c=DRAW_MENU(3);             /* Edge Detection Selection     */
     switch (c)
     {
      case 31 :
      if (PIC.valid_pic==1)
      {
       LAPLACE5PT();
```

```
  sprintf(msg1,"%-15s LP5PT %4dx%4d",in_file,
     PIC.imagewidth,PIC.imagelength);
 }
 else {
  DRAW_MESSAGE(
  2,"No picture in memory to detect.  Press key to continue");
  d=getch();       /* wait for key press           */
 }
 break;

 case 32 :
 if (PIC.valid_pic==1)
 {
  EXP9PT();
  sprintf(msg1,"%-15s EXP9PT %4dx%4d",in_file,
     PIC.imagewidth,PIC.imagelength);
 }
 else {
  DRAW_MESSAGE(
  2,"No picture in memory to detect.  Press key to continue");
  d=getch();       /* wait for key press           */
 }
 break;

 case 33 :
 if (PIC.valid_pic==1)
 {
  EXP13PT();
  sprintf(msg1,"%-14s EXP13PT %4dx%4d",in_file,
     PIC.imagewidth,PIC.imagelength);
 }
 else {
  DRAW_MESSAGE(
  2,"No picture in memory to detect.  Press key to continue");
  d=getch();       /* wait for key press           */
 }
 break;

 case 34 :
 if (PIC.valid_pic==1)
 {
  EXP25PT();
  sprintf(msg1,"%-14s EXP25PT %4dx%4d",in_file,
     PIC.imagewidth,PIC.imagelength);
 }
 else {
  DRAW_MESSAGE(
  2,"No picture in memory to detect.  Press key to continue");
  d=getch();       /* wait for key press           */
 }
 break;

 case 35 :
 if (PIC.valid_pic==1)
```

```
      {
       GAUSS9PT();
       sprintf(msg1,"%-15s GAUSS9PT %4dx%4d",in_file,
          PIC.imagewidth,PIC.imagelength);
      }
      else {
       DRAW_MESSAGE(
       2,"No picture in memory to detect.  Press key to continue");
       d=getch();       /* wait for key press           */
      }
      break;

      case 36 :
      if (PIC.valid_pic==1)
      {
       GAUSS13PT();
       sprintf(msg1,"%-15s GAUSS13PT %4dx%4d",in_file,
          PIC.imagewidth,PIC.imagelength);
      }
      else {
       DRAW_MESSAGE(
       2,"No picture in memory to detect.  Press key to continue");
       d=getch();       /* wait for key press           */
      }
      break;

      case 37 :
      if (PIC.valid_pic==1)
      {
       GAUSS25PT();
       sprintf(msg1,"%-15s GAUSS25PT %4dx%4d",in_file,
          PIC.imagewidth,PIC.imagelength);
      }
      else {
       DRAW_MESSAGE(
       2,"No picture in memory to detect.  Press key to continue");
       d=getch();       /* wait for key press           */
      }
      break;

    }
    break;

case 8 :                          /* save memory into save buffer */
    if (PIC.valid_pic==1)
    {
     SAVE();
     strcpy(PIC.savestr,msg1);
    }
    else {
     DRAW_MESSAGE(
     2,"No picture in memory to save.  Press key to continue");
     d=getch();                      /* wait for key press    */
    }
```

```
    break;

case 9 :                          /* recall memory into save buffer */
    if (PIC.save==1)
    {
     RECALL();
     strcpy(msg1,PIC.savestr);
    }
    else {
     DRAW_MESSAGE(
     2,"No picture in memory to recall.  Press key to continue");
     d=getch();                     /* wait for key press    */
    }
    break;

case 0 :
    c=DRAW_MENU(7);                /* Simulation  Menu Selection    */
    switch (c)
    {
     case 71:
     d=PARTREESIM1(&PIC);
     if(d==0)
     {
      DRAW_MESSAGE(2,"Error allocating memory."
           " Press key to continue");
      d=getch();                    /* wait for key press    */
     }
     else
     {
     sprintf(msg1,"Parallel Tree Sim. (G.S.), %4dx%4d",
        PIC.imagewidth,PIC.imagelength);
     }
     break;

     case 72:
     d=PARTREESIM2(&PIC);
     if(d==0)
     {
      DRAW_MESSAGE(2,"Error allocating memory."
           " Press key to continue");
      d=getch();                    /* wait for key press    */
     }
     else
     {
     sprintf(msg1,"Parallel Tree Sim. (J.), %4dx%4d",
        PIC.imagewidth,PIC.imagelength);
     }
     break;

     case 73:
     d=CIRTREESIM1(&PIC);
     if(d==0)
     {
      DRAW_MESSAGE(2,"Error allocating memory."
```

```
        " Press key to continue");
   d=getch();                    /* wait for key press    */
  }
  else
  {
  sprintf(msg1,"Circular Tree Sim. (G.S.), %4dx%4d",
     PIC.imagewidth,PIC.imagelength);
  }
  break;

  case 74:
  d=CIRTREESIM2(&PIC);
  if(d==0)
  {
   DRAW_MESSAGE(2,"Error allocating memory."
        " Press key to continue");
   d=getch();                    /* wait for key press    */
  }
  else
  {
  sprintf(msg1,"Circular Tree Sim. (G.S.), %4dx%4d",
     PIC.imagewidth,PIC.imagelength);
  }
  break;

  case 75 :
  ANIMATE();
  break;

  case 76:
  d=CIRFIELDCAL(&PIC);
  if(d==0)
  {
   DRAW_MESSAGE(2,"Error allocating memory."
        " Press key to continue");
   d=getch();                    /* wait for key press    */
  }
  else
  {
  sprintf(msg1,"Circular Field Calc. (G.S.), %4dx%4d",
     PIC.imagewidth,PIC.imagelength);
  }
  break;

  case 77:
  d=CIRTREESIM3(&PIC);
  if(d==0)
  {
   DRAW_MESSAGE(2,"Error allocating memory."
        " Press key to continue");
   d=getch();                    /* wait for key press    */
  }
  else
  {
```

```
        sprintf(msg1,"Circular Tree Sim. (G.S.), %4dx%4d",
           PIC.imagewidth,PIC.imagelength);
        }
        break;

        case 78:
        d=PARFIELD(&PIC);
        if(d==0)
        {
         DRAW_MESSAGE(2,"Error allocating memory."
              " Press key to continue");
         d=getch();                  /* wait for key press    */
        }
        break;

        case 79:
        d=CIRFIELD(&PIC);
        if(d==0)
        {
         DRAW_MESSAGE(2,"Error allocating memory."
              " Press key to continue");
         d=getch();                  /* wait for key press    */
        }
        break;

        case 70:
        SETITER();
        break;
        }
        break;

     }
    }

/*              check installation again (to clear watcher)              */

     inregs.w.ax = 0;
     int386( 0x33, &inregs, &outregs );

/*       free up picture and reset video mode back to normal             */

   free(imgptr);
   _setvideomode( _DEFAULTMODE);
   printf("Done\n");
}
```

C2 FILE: DIMEN.C

```
/*                     Dimension calculating routines                    */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
```

```
#include <malloc.h>
#include <string.h>
#include <dos.h>
#include <i86.h>
#include <conio.h>
#include "picture.h"
#include "screen.h"
#include "lmap.h"

extern struct PICTURE PIC;

void BOXDIMENSION(float *dimension);
void BOXDIMEN2(float *dimension);
void BOXDIMEN3(float *dimension);
void MASSDIMEN(float *dimension,struct PICTURE *PIC);
void DILATIONDIM(float *dimension);
void DILATIONDIM2(float *dimension);
int LKOCH(void);
int LMKOCH(void);
int LMKOCH2(void);
int LMKOCH3(void);
int LKOCH15(void);
int LTREE4(void);
int LTREE3(void);
int LTREE2(void);
int LTREE1(void);
int LSGASKET(void);
void LENGINE(struct LMAP *RULES, struct PICTURE *PIC, int iteration,
             int command);
extern void DATASET(int n,float *data,int color,struct SCREEN *PLT);
extern void TITLES(char *st1,char *st2,struct SCREEN *PLT);
extern void DRAWBORDER(struct SCREEN *PLT);
extern void DRAWDATA(struct SCREEN *PLT);
extern void DRAW_MESSAGE(int location,char *msg);
extern void PIC_DRAW(int side,struct PICTURE *DISP);
extern float LINREG(int n,float *data);
extern void SCREENSET(int xst,int yst,int xsp,int ysp,float xmn,float ymn,
               float xmx,float ymx,int typ,struct SCREEN *SCRN);

/*-------------------------------------------------------------------*/
/*                              BOXDIMENSION                         */
/*      This routine takes the image in memory, calculates the       */
/*      Box Dimension and returns the result.  The routine works     */
/*      by counting the boxes in the picture that have full white    */
/*      pixels in them for selected box sizes.  Note, that all picture */
/*      enhancement must be completed and all edges defined before   */
/*      using this routine!                                          */
/*         The results are then fitted to a log/log line using linear */
/*      regression on the data.                                      */
/*-------------------------------------------------------------------*/

void BOXDIMENSION(float *dimension)
{
   long h,i,j,k,m1,m2,ix,iy,isz,ofst1,ofst2;
```

```
    long icx,icy,xoffset,boxsize[7];
    float data[16];
    double tmp1,tmp2,tmp3,tmp4,n;
    char c,msg[60],st1[60],st2[60];
    char *imgptr;
    struct SCREEN PLT;

    imgptr=PIC.pic_buf;                    /* locate picture              */
    ix=PIC.imagewidth;                     /* start at full resolution    */
    iy=PIC.imagelength;                    /* set y variable              */
    boxsize[0]=1;
    boxsize[1]=2;  boxsize[2]=3;
    boxsize[3]=5;  boxsize[4]=6;
    boxsize[5]=10; boxsize[6]=15;


/*      To eliminate edge effects, ensure same area used each iteration */

    icx=(ix/30)*30;                        /* ensure size divisible by 30 */
    icy=(iy/30)*30;                        /* ensure size divisible by 30 */

/*                Stage 1, count all white pixels in picture            */

    h=0;                                   /* pixel counter               */
    for(i=0;i<icy;i++)                     /* y loop                      */
    {
      for(j=0;j<icx;j++)                   /* x loop                      */
      {
        ofst1=i*ix+j;                      /* offset into picture         */
        if(imgptr[ofst1]==255) h++;        /* count all white pixels      */
      }
    }
    data[1]=(float)log10((double)(h));
    data[0]=(float)log10((double)((icx+icy)/(2)));
    sprintf(msg,"%d boxes at %dx%d box size counted",h,1,1);
    DRAW_MESSAGE(2,msg);

/*       Stage 2, count all 2,3,5,6,10,15  boxes with white pixels      */

    for(i=1;i<7;i++) {
      isz=boxsize[i];                      /* size of box                 */
      h=0;                                 /* pixel counter               */
      for(j=0;j<icy;j+=isz) {
        for(k=0;k<icx;k+=isz) {
          ofst1=j*ix+k;                    /* offset start of box         */
          c=imgptr[ofst1];                 /* get first value in box      */
            for(m1=0;m1<isz;m1++) {        /* get each row in box         */
              ofst2=ofst1+m1*ix;           /* get start of each row       */
              for(m2=0;m2<isz;m2++) {      /* get each pixel in box       */
                c=max(c,imgptr[ofst2+m2]); /* find greatest pixel         */
              }
            }
            if(c==255) h++;                /* count all white pixels      */
        }
```

```
        }
        data[(i)*2+1]=(float)log10((double)(h));
        data[(i)*2]=(float)log10((double)((icx+icy)/(isz*2)));
        sprintf(msg,"%d boxes at %dx%d box size counted",h,isz,isz);
        DRAW_MESSAGE(2,msg);
    }

/*                      calculate dimension from numbers                    */
/*                           by linear regression                           */

    n=7.0;
    *dimension=LINREG(n,data);

/*      Plot results to right half of screen for display                    */

    xoffset=513;
    _setviewport(xoffset,1,xoffset+509,510);     /* set viewport           */
    _clearscreen(_GVIEWPORT);                    /* clear screen           */
    tmp2=((float)((int)(data[13]*2.0)))/2.0;     /* set min scale          */
    tmp3=((float)((int)(data[0]*2.0))+1.0)/2.0;
    tmp4=((float)((int)(data[1]*2.0))+1.0)/2.0;  /* set max scale          */
    tmp1=tmp2-tmp4+tmp3;                         /* ensure 1:1 scale       */
    strcpy(st1,"BOX DIMENSION PLOT");
    strcpy(st2,"Log number of boxes vs Log scale");

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
    DATASET(14,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(14,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);

    return;
}

/*-------------------------------------------------------------------------*/
/*                          BETTER BOXDIMENSION                            */
/*      This routine takes the image in memory, calculates the             */
/*      Box Dimension and returns the result.  The routine works           */
/*      by counting the boxes in the picture that have full white          */
/*      pixels in them for selected box sizes.  Note, that all picture     */
/*      enhancement must be completed and all edges defined before         */
/*      using this routine!  The enhancement added to this routine is      */
/*      to first locate the beginning of the fractal along both x and y    */
/*      directions to force the fractal to lie in the smallest large       */
/*      box possible.                                                      */
/*         The results are then fitted to a log/log line using linear      */
/*      regression on the data.                                            */
/*-------------------------------------------------------------------------*/

void BOXDIMEN3(float *dimension)
```

```
{
   long h,i,j,k,m1,m2,ix,iy,br,isz,ofst1,ofst2;
   long icx,icy,xoffset,xbegin,ybegin,boxsize[7];
   float data[16];
   double tmp1,tmp2,tmp3,tmp4,n;
   char c,msg[60],st1[60],st2[60];
   char *imgptr;
   struct SCREEN PLT;

   imgptr=PIC.pic_buf;                    /* locate picture              */
   ix=PIC.imagewidth;                     /* start at full resolution    */
   iy=PIC.imagelength;                    /* set y variable              */
   boxsize[0]=1;
   boxsize[1]=2;  boxsize[2]=3;
   boxsize[3]=5;  boxsize[4]=6;
   boxsize[5]=10; boxsize[6]=15;

/*      Find smallest box to enclose entire fractal                      */

   k=ix*iy;                               /* find size of image array    */
   i=0;
   while (i<k && imgptr[i]!=255) i++;     /* find first white pixel      */
   ybegin=i/ix;                           /* save as offset              */
   br=0;
   for(i=0;i<ix;i++)                      /* x loop                      */
   {
      for(j=ybegin;j<iy;j++)              /* y loop                      */
      {
         ofst1=j*ix+i;                    /* calculate offset into picture */
         if(imgptr[ofst1]==255)
         {
            xbegin=i;                     /* save as offset              */
            br=1;
         }
         if(br==1) break;
      }
      if(br==1) break;
   }

/*      To eliminate edge effects, ensure same area used each iteration */

   icx=((ix-xbegin)/30)*30;               /* ensure size divisible by 30 */
   icy=((iy-ybegin)/30)*30;               /* ensure size divisible by 30 */

/*                Stage 1, count all white pixels in picture             */

   h=0;                                   /* pixel counter               */
   for(i=ybegin;i<icy;i++)                /* y loop                      */
   {
     for(j=xbegin;j<icx;j++)              /* x loop                      */
     {
       ofst1=i*ix+j;                      /* offset into picture         */
       if(imgptr[ofst1]==255) h++;        /* count all white pixels      */
     }
```

```
    }
    data[1]=(float)log10((double)(h));
    data[0]=(float)log10((double)((icx+icy)/(2)));
    sprintf(msg,"%d boxes at %dx%d box size counted",h,1,1);
    DRAW_MESSAGE(2,msg);

/*       Stage 2, count all 2,3,5,6,10,15  boxes with white pixels      */

    for(i=1;i<7;i++) {
      isz=boxsize[i];                    /* size of box                  */
      h=0;                               /* pixel counter                */
      for(j=ybegin;j<icy;j+=isz) {
        for(k=xbegin;k<icx;k+=isz) {
          ofst1=j*ix+k;                  /* offset start of box          */
          c=imgptr[ofst1];               /* get first value in box       */
            for(m1=0;m1<isz;m1++) {      /* get each row in box          */
              ofst2=ofst1+m1*ix;         /* get start of each row        */
              for(m2=0;m2<isz;m2++) {    /* get each pixel in box        */
                c=max(c,imgptr[ofst2+m2]); /* find greatest pixel        */
              }
            }
            if(c==255) h++;              /* count all white pixels       */
        }
      }
      data[(i)*2+1]=(float)log10((double)(h));
      data[(i)*2]=(float)log10((double)((icx+icy)/(isz*2)));
      sprintf(msg,"%d boxes at %dx%d box size counted",h,isz,isz);
      DRAW_MESSAGE(2,msg);
    }

/*                     calculate dimension from numbers                  */
/*                          by linear regression                         */

    n=7.0;
    *dimension=LINREG(n,data);

/*       Plot results to right half of screen for display               */

    xoffset=513;
    _setviewport(xoffset,1,xoffset+509,510);     /* set viewport        */
    _clearscreen(_GVIEWPORT);                    /* clear screen        */
    tmp2=((float)((int)(data[13]*2.0)))/2.0;     /* set min scale       */
    tmp3=((float)((int)(data[0]*2.0))+1.0)/2.0;
    tmp4=((float)((int)(data[1]*2.0))+1.0)/2.0;  /* set max scale       */
    tmp1=tmp2-tmp4+tmp3;                         /* ensure 1:1 scale    */
    strcpy(st1,"BOX DIMENSION PLOT");
    strcpy(st2,"Log number of boxes vs Log scale");

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
    DATASET(14,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);
```

```
    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(14,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);

    return;
}

/*-------------------------------------------------------------------------*/
/*                              BOXDIMEN2                                  */
/*      This routine is an extension of the standard box counting          */
/*      routine.  The Hausdorff-Besicovitch dimension requires the         */
/*      smallest finite covering of closed ball sets at various            */
/*      scales to calculate dimension of a fractal set.                    */
/*         This definition is intractable because the search for the       */
/*      smallest covering is impossible for large sets.  This routine      */
/*      uses a simple algorithm to cover the image with a smaller          */
/*      covering of square "balls" than with the standard box dimension    */
/*      algorithm.                                                         */
/*-------------------------------------------------------------------------*/

void BOXDIMEN2(float *dimension)
{
   long i,j,k,ii,jj,kk,ll,sizem,offset,imln,imwd,br;
   long cnt,toffset,icx,icy,n,xvl;
   short tst;
   float data[16],tmp1,tmp2,tmp3,tmp4;
   char msg[60],st1[60],st2[60];
   unsigned char *imgptr;
   struct SCREEN PLT;

   imgptr=PIC.pic_buf;                  /* locate picture                */
   imwd=PIC.imagewidth;                 /* start at full resolution      */
   imln=PIC.imagelength;                /* set y variable                */

/*      To eliminate edge effects, ensure same area used each iteration */

   icx=(imwd/30)*30;                    /* ensure size divisible by 30   */
   icy=(imln/30)*30;                    /* ensure size divisible by 30   */

/*                Stage 1, count all white pixels in picture            */

   cnt=0;                               /* pixel counter                 */
   for(i=0;i<icy;i++)                   /* y loop                        */
   {
     for(j=0;j<icx;j++)                 /* x loop                        */
     {
       offset=i*icx+j;                  /* offset into picture           */
       if(imgptr[offset]==255) cnt++;   /* count all white pixels        */
     }
   }
   data[1]=(float)log10((double)(cnt));
   data[0]=(float)log10((double)(icx));
   sprintf(msg,"%d boxes at %dx%d box size counted",cnt,1,1);
   DRAW_MESSAGE(2,msg);
```

```
/*              Stage 2,3,4,5,6, try to count smaller covering.          */
/*      Note that the right and bottom edges and bottom right corner     */
/*      require special handling to ensure calculations are consistant.  */

tst=256;                                  /* destructive counting         */
for(i=2;i<7;i++)                          /* Main counting loop           */
{
 tst--;                                   /* decrement here is clearer!   */
 cnt=0;                                   /* initialize counter           */
 sizem=i;                                 /* get size of box              */

/*       Bulk of picture excluding edges is processed here                */

 for(j=0;j<=(icy-sizem);j++)              /* step along y-axis            */
 {
  for(k=(sizem-1);k<=(icx-sizem);k++)     /* step along x-axis            */
  {
   offset=j*imwd+k;                       /* calculate initial offset     */
   br=0;                                  /* set loop break off           */
   if(imgptr[offset]==tst)                /* image pixel found            */
   {
    cnt++;                                /* count this box               */
    for(ii=(1-sizem);(ii<=0 && br==0);ii++) /* scan x-direction           */
    {
     for(jj=0;(jj<sizem && br==0);jj++)   /* scan y-direction             */
     {
      toffset=offset+ii+jj*imwd;
      if(imgptr[toffset]==tst)
      {
       br=1;                              /* break loops if pixel found   */
       xvl=ii;                            /* save x location              */
      }
     }
    }
    for(kk=0;kk<sizem;kk++)               /* y direction                  */
    {
     for(ll=xvl;ll<(xvl+sizem);ll++)      /* x direction                  */
     {
      toffset=offset+ll+kk*imwd;
      if(imgptr[toffset]==tst) imgptr[toffset]-=1; /* set mask pixels     */
     }
    }
   }
  }
 }

/*      The left edge of the image is processed here                      */

 for(j=0;j<=(icy-sizem);j++)                  /* scan along y-axis        */
 {
  offset=j*imwd;                              /* address pixel            */
  for(k=0;k<sizem;k++)                        /* scan along x-axis        */
  {
```

```
    toffset=offset+k;                              /* address examined pixel */
    if(imgptr[toffset]==tst)
    {
     cnt++;                                         /* count another box    */
     for(jj=0;jj<sizem;jj++)                        /* y direction          */
     {
      for(kk=0;kk<sizem;kk++)                       /* x direction          */
      {
       toffset=offset+jj*imwd+kk;
       if(imgptr[toffset]==tst) imgptr[toffset]--;  /* set mask pixels      */
      }
     }                                              /* end of box set loop  */
    }
   }
  }                                               /* scan along side complete */

/*      The bottom edge of the image is processed here                       */

  for(j=0;j<=(icx-sizem);j++)                       /* scan along x-axis    */
  {
   offset=(icy-sizem)*imwd+j;                       /* address pixel        */
   for(k=0;k<sizem;k++)                             /* scan along y-axis    */
   {
    toffset=offset+k*imwd;                         /* address examined pixel */
    if(imgptr[toffset]==tst)
    {
     cnt++;                                         /* count another box    */
     for(jj=0;jj<sizem;jj++)                        /* x direction          */
     {
      for(kk=0;kk<sizem;kk++)                       /* y direction          */
      {
       toffset=offset+kk*imwd+jj;
       if(imgptr[toffset]==tst) imgptr[toffset]--;  /* set mask pixels      */
      }
     }                                              /* end of box set loop  */
    }
   }
  }                                               /* scan along bottom complete */

/*      The right edge of the image is processed here                        */

  for(j=0;j<=(icy-sizem);j++)                       /* scan along y-axis    */
  {
   offset=j*imwd+icx-sizem;                         /* address pixel        */
   for(k=0;k<sizem;k++)                             /* scan along x-axis    */
   {
    toffset=offset+k;                              /* address examined pixel */
    if(imgptr[toffset]==tst)
    {
     cnt++;                                         /* count another box    */
     for(jj=0;jj<sizem;jj++)                        /* y direction          */
     {
      for(kk=0;kk<sizem;kk++)                       /* x direction          */
      {
```

```
      toffset=offset+jj*imwd+kk;
      if(imgptr[toffset]==tst) imgptr[toffset]--;  /* set mask pixels   */
     }
    }                                              /* end of box set loop  */
   }
  }
 }                                               /* scan along side complete */

/*        Scan last corner for pixels in set                                 */

 br=0;                                            /* reuse variable       */
 offset=(icy-sizem)*imwd+(icx-sizem);           /* first pixel in last box */
 for(jj=0;jj<sizem;jj++)                          /* y loop               */
 {
  for(kk=0;kk<sizem;kk++)                         /* x loop               */
  {
   toffset=offset+kk+jj*imwd;                     /* calculate offset     */
   if(imgptr[toffset]==tst)
   {
    imgptr[toffset]--;                            /* reset all set pixels */
    br=1;                                         /* signal for increment */
   }
  }
 }
 if(br==1) cnt++;                          /* count last box if required   */

/*      Now complete counting minimal covering.  Note that this is not   */
/*      the absolute minimum!!! but is much closer to the definition     */
/*      than the regular Box Dimension algorithm.                        */

 data[(i-1)*2+1]=(float)(log10((double)(cnt)));  /* convert to log value */
 data[(i-1)*2]=(float)(log10((double)(icx/sizem)));     /* save scale    */
 sprintf(msg,"%d boxes at %dx%d box size counted",cnt,sizem,sizem);
 DRAW_MESSAGE(2,msg);

}

/*                      calculate dimension from numbers                  */
/*                           by linear regression                         */

 n=6.0;
 *dimension=LINREG(n,data);

/*      Plot results to right half of screen for display                  */

   _setviewport(513,1,1022,510);                  /* set viewport         */
   _clearscreen(_GVIEWPORT);                      /* clear screen         */
   tmp2=((float)((int)(data[11]*2.0)))/2.0;       /* set min scale        */
   tmp3=((float)((int)(data[0]*2.0))+1.0)/2.0;
   tmp4=((float)((int)(data[1]*2.0))+1.0)/2.0;    /* set max scale        */
   tmp1=tmp2-tmp4+tmp3;                           /* ensure 1:1 scale     */
   strcpy(st1,"BBOX DIMENSION PLOT");
   strcpy(st2,"Log number of boxes vs Log scale");
   SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
```

```
    DATASET(12,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(12,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);
    return;
}

/*-------------------------------------------------------------------------*/
/*                            DILATIONDIM                                  */
/*      This routine takes the image in memory, creates a working          */
/*      buffer, and dilates each point in the image into the buffer.       */
/*      The buffer is then counted for set pixels.  The size of the        */
/*      dilation and the area the dilated picture covers gives the         */
/*      data required for calculating the dimension of the image.          */
/*         The results are then fitted to a log/log line using linear      */
/*      regression on the data.  The dilation here is with squares.        */
/*-------------------------------------------------------------------------*/

void DILATIONDIM(float *dimension)
{
   long h,i,j,k,ix,iy,isz,ofst1,ofst2,ofst3,buffersize;
   long icx,icy,ii,jj,xoffset,bufx,dilationsize[7];
   float data[16];
   double tmp1,tmp2,tmp3,tmp4,n;
   char c,msg[60],st1[60],st2[60];
   char *imgptr,*bufptr;
   struct SCREEN PLT;

   imgptr=PIC.pic_buf;                  /* locate picture              */
   ix=PIC.imagewidth;                   /* start at full resolution    */
   iy=PIC.imagelength;                  /* set y variable              */
   dilationsize[0]=1;
   dilationsize[1]=2;  dilationsize[2]=3;
   dilationsize[3]=5;  dilationsize[4]=6;
   dilationsize[5]=10; dilationsize[6]=15;


/*      To eliminate edge effects, ensure same area used each iteration */

   icx=(ix/30)*30;                      /* ensure size divisible by 30  */
   icy=(iy/30)*30;                      /* ensure size divisible by 30  */

/*      allocate memory for dilation buffer                             */

   bufx=icx+15;
   buffersize=(bufx)*(icy+15);          /* size of dilation buffer      */
  if((bufptr=(char *)malloc(buffersize))==NULL) {  /* memory allocation error
*/
      DRAW_MESSAGE(2,"memory allocation error");
      return;                           /* return with error flag       */
```

```
    }

/*                Stage 1, count all white pixels in picture                */

    h=0;                                    /* pixel counter                 */
    for(i=0;i<icy;i++)                      /* y loop                        */
    {
      for(j=0;j<icx;j++)                    /* x loop                        */
      {
        ofst1=i*ix+j;                       /* offset into picture           */
        if(imgptr[ofst1]==255) h++;         /* count all white pixels        */
      }
    }
    data[1]=-(float)log10((double)(h));
    data[0]=0.0;
    sprintf(msg,"%d pixels at %dx%d dilation",h,1,1);
    DRAW_MESSAGE(2,msg);

/*       Stage 2, dilate all 2,3,5,6,10,15  boxes with white pixels         */

    for(i=1;i<7;i++) {
      memset(bufptr,0,buffersize);          /* clear memory                  */
      isz=dilationsize[i];                  /* size of dilation              */
      h=0;                                  /* pixel counter                 */
      for(j=0;j<icy;j++)
      {
        for(k=0;k<icx;k++)
        {
          ofst1=j*ix+k;                     /* offset start of box           */
          c=imgptr[ofst1];                  /* get first value in box        */
          if(c==255)
          {
             ofst2=j*bufx+k;                /* offset into dilation buffer   */
             for(ii=0;ii<isz;ii++)          /* x dilation loop               */
             {
                ofst3=ii*bufx+ofst2;
                for(jj=0;jj<isz;jj++)       /* y dilation loop              */
                {
                   bufptr[ofst3+jj]=255;    /* dilate point                 */
                }
             }
          }
        }
      }
      h=0;
      for(k=0;k<buffersize;k++)
      {
         if(bufptr[k]==255) h++;            /* integrate dilated area       */
      }

      data[(i)*2+1]=-(float)log10((double)(h)/(isz*isz));
      data[(i)*2]=(float)log10((double)(isz));
      sprintf(msg,"%d pixels at %dx%d dilation",h,isz,isz);
      DRAW_MESSAGE(2,msg);
```

```
    }

 /*                      calculate dimension from numbers                  */
 /*                           by linear regression                         */

    n=7.0;
    *dimension=LINREG(n,data);

 /*      Plot results to right half of screen for display                  */

    xoffset=513;
    _setviewport(xoffset,1,xoffset+509,510);     /* set viewport          */
    _clearscreen(_GVIEWPORT);                    /* clear screen          */
    tmp1=0.0;                                    /* always zero!          */
    tmp2=(float)((int)(data[1]*2.0-1.0))/2.0;    /* set min scale         */
    tmp4=(float)((int)(data[13]*2.0))/2.0;       /* set max scale         */
    tmp3=tmp4-tmp2;                              /* set 1:1 scale         */
    strcpy(st1,"DILATION DIMENSION PLOT");
    strcpy(st2,"Log area vs Log dilation");

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
    DATASET(14,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(14,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);

    free(bufptr);                          /* free buffer used          */
    return;
}

/*-------------------------------------------------------------------------*/
/*                            DILATIONDIM2                                 */
/*      This routine takes the image in memory, creates a working          */
/*      buffer, and dilates each point in the image into the buffer.       */
/*      The buffer is then counted for set pixels.  The size of the        */
/*      dilation and the area the dilated picture covers gives the         */
/*      data required for calculating the dimension of the image.          */
/*         The results are then fitted to a log/log line using linear      */
/*      regression on the data.  The dilation in this case is circular.    */
/*-------------------------------------------------------------------------*/

void DILATIONDIM2(float *dimension)
{
   long h,i,j,k,ix,iy,ofst1,ofst2,buffersize;
   long icx,icy,xoffset,bufx;
   long x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16;
   long x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31,x32;
   long x33,x34,x35,x36,x37,x38,x39,x40,x41,x42,x43,x44,x45,x46,x47,x48;
   long x49,x50,x51,x52,x53,x54,x55,x56,x57;
   float data[16];
```

```
    double tmp1,tmp2,tmp3,tmp4,n;
    char c,msg[60],st1[60],st2[60];
    char *imgptr,*bufptr;
    struct SCREEN PLT;

    imgptr=PIC.pic_buf;                    /* locate picture              */
    ix=PIC.imagewidth;                     /* start at full resolution    */
    iy=PIC.imagelength;                    /* set y variable              */

/*      To eliminate edge effects, ensure same area used each iteration */

    icx=(ix/30)*30;                        /* ensure size divisible by 30  */
    icy=(iy/30)*30;                        /* ensure size divisible by 30  */

/*      allocate memory for dilation buffer                               */

    bufx=icx+15;
    buffersize=(bufx)*(icy+15);            /* size of dilation buffer      */
   if((bufptr=(char *)malloc(buffersize))==NULL) {  /* memory allocation error
*/
       DRAW_MESSAGE(2,"memory allocation error");
       return;                             /* return with error flag       */
    }

/*                Stage 1, count all white pixels in picture              */

   h=0;                                    /* pixel counter                */
   for(i=0;i<icy;i++)                      /* y loop                       */
   {
     for(j=0;j<icx;j++)                    /* x loop                       */
     {
       ofst1=i*ix+j;                       /* offset into picture          */
       if(imgptr[ofst1]==255) h++;         /* count all white pixels       */
     }
   }
   data[1]=(float)log10((double)(h));
   data[0]=(float)log10((double)((icx+icy)/(2)));
   sprintf(msg,"%d pixels at %dx%d dilation",h,1,1);
   DRAW_MESSAGE(2,msg);

/*                    Stage 2, 4 pixel dilation                           */

                       x1=0;     x2=1;
                       x3=bufx;  x4=bufx+1;

  memset(bufptr,0,buffersize);             /* clear memory                 */
  for(i=0;i<icy;i++)                       /* y loop                       */
  {
    for(j=0;j<icx;j++)                     /* x loop                       */
    {
      ofst1=i*ix+j;                        /* offset into picture          */
      if(imgptr[ofst1]==255)
      {
         ofst2=i*bufx+j;                   /* offset into dilation buffer  */
```

```
            bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
            bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255; /* dilate point   */
         }
       }
     }
     h=0;
     for(k=0;k<buffersize;k++)
     {
        if(bufptr[k]==255) h++;           /* integrate dilated area       */
     }
     data[3]=(float)log10((double)h/4.0);
     data[2]=(float)log10((double)(icx+icy)/(sqrt(4.0)*2.0));
     sprintf(msg,"%d pixels at 4 pixel dilation",h);
     DRAW_MESSAGE(2,msg);

 /*                    Stage 3, 5 pixel dilation                          */

                              x1=1;
                    x2=bufx-1;  x3=bufx;  x4=bufx+1;
                              x5=bufx*2;

    memset(bufptr,0,buffersize);          /* clear memory                 */
    for(i=0;i<icy;i++)                    /* y loop                       */
    {
      for(j=0;j<icx;j++)                  /* x loop                       */
      {
        ofst1=i*ix+j;                     /* offset into picture          */
        if(imgptr[ofst1]==255)
        {
           ofst2=i*bufx+j;                /* offset into dilation buffer  */
           bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
           bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255;
           bufptr[ofst2+x5]=255;          /* dilate point                 */
        }
      }
    }
    h=0;
    for(k=0;k<buffersize;k++)
    {
       if(bufptr[k]==255) h++;            /* integrate dilated area       */
    }
    data[5]=(float)log10((double)h/5.0);
    data[4]=(float)log10((double)(icx+icy)/(sqrt(5.0)*2.0));
    sprintf(msg,"%d pixels at 5 pixel dilation",h);
    DRAW_MESSAGE(2,msg);

 /*                    Stage 4, 12 pixel dilation                         */

                      x1=1;          x2=2;
          x3=bufx;    x4=bufx+1;     x5=bufx+2;     x6=bufx+3;
          x7=bufx*2;  x8=bufx*2+1;   x9=bufx*2+2;   x10=bufx*2+3;
                      x11=bufx*3+1;  x12=bufx*3+2;

   memset(bufptr,0,buffersize);           /* clear memory                 */
```

```
    for(i=0;i<icy;i++)                    /* y loop                          */
    {
      for(j=0;j<icx;j++)                  /* x loop                          */
      {
        ofst1=i*ix+j;                     /* offset into picture             */
        if(imgptr[ofst1]==255)
        {
           ofst2=i*bufx+j;                /* offset into dilation buffer     */
           bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
           bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255;
           bufptr[ofst2+x5]=255; bufptr[ofst2+x6]=255;
           bufptr[ofst2+x7]=255; bufptr[ofst2+x8]=255;
           bufptr[ofst2+x9]=255; bufptr[ofst2+x10]=255;
           bufptr[ofst2+x11]=255; bufptr[ofst2+x12]=255; /* dilate point */
        }
      }
    }
    h=0;
    for(k=0;k<buffersize;k++)
    {
       if(bufptr[k]==255) h++;            /* integrate dilated area          */
    }
    data[5]=(float)log10((double)h/12.0);
    data[4]=(float)log10((double)(icx+icy)/(sqrt(12.0)*2.0));
    sprintf(msg,"%d pixels at 12 pixel dilation",h);
    DRAW_MESSAGE(2,msg);

/*                    Stage 5, 21 pixel dilation                            */

                  x1=1;          x2=2;          x3=3;
   x4=bufx;       x5=bufx+1;     x6=bufx+2;     x7=bufx+3;     x8=bufx+4;
   x9=bufx*2;     x10=bufx*2+1;  x11=bufx*2+2;  x12=bufx*2+3;  x13=bufx*2+4;
   x14=bufx*3;    x15=bufx*3+1;  x16=bufx*3+2;  x17=bufx*3+3;  x18=bufx*3+4;
                  x19=bufx*4+1;  x20=bufx*4+2;  x21=bufx*3+3;

   memset(bufptr,0,buffersize);           /* clear memory                    */
   for(i=0;i<icy;i++)                     /* y loop                          */
   {
     for(j=0;j<icx;j++)                   /* x loop                          */
     {
       ofst1=i*ix+j;                      /* offset into picture             */
       if(imgptr[ofst1]==255)
       {
          ofst2=i*bufx+j;                 /* offset into dilation buffer     */
          bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
          bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255;
          bufptr[ofst2+x5]=255; bufptr[ofst2+x6]=255;
          bufptr[ofst2+x7]=255; bufptr[ofst2+x8]=255;
          bufptr[ofst2+x9]=255; bufptr[ofst2+x10]=255;
          bufptr[ofst2+x11]=255; bufptr[ofst2+x12]=255;
          bufptr[ofst2+x13]=255; bufptr[ofst2+x14]=255;
          bufptr[ofst2+x15]=255; bufptr[ofst2+x16]=255;
          bufptr[ofst2+x17]=255; bufptr[ofst2+x18]=255;
          bufptr[ofst2+x19]=255; bufptr[ofst2+x20]=255;
```

```
            bufptr[ofst2+x21]=255;                          /* dilate point */
          }
        }
      }
      h=0;
      for(k=0;k<buffersize;k++)
      {
         if(bufptr[k]==255) h++;            /* integrate dilated area       */
      }
      data[7]=(float)log10((double)h/21.0);
      data[6]=(float)log10((double)(icx+icy)/(sqrt(21.0)*2.0));
      sprintf(msg,"%d pixels at 21 pixel dilation",h);
      DRAW_MESSAGE(2,msg);

/*                      Stage 6, 32 pixel dilation                          */

              x1=1;          x2=2;          x3=3;          x4=4;
x5=bufx;    x6=bufx+1;    x7=bufx+2;    x8=bufx+3;    x9=bufx+4;    x10=bufx+5;
x11=bufx*2; x12=bufx*2+1; x13=bufx*2+2; x14=bufx*2+3; x15=bufx*2+4;
x16=bufx*2+5;
x17=bufx*3; x18=bufx*3+1; x19=bufx*3+2; x20=bufx*3+3; x21=bufx*3+4;
x22=bufx*3+5;
x23=bufx*4; x24=bufx*4+1; x25=bufx*4+2; x26=bufx*4+3; x27=bufx*4+4;
x28=bufx*4+5;
              x29=bufx*5+1; x30=bufx*5+2; x31=bufx*5+3; x32=bufx*5+4;

     memset(bufptr,0,buffersize);          /* clear memory                 */
     for(i=0;i<icy;i++)                    /* y loop                       */
     {
       for(j=0;j<icx;j++)                  /* x loop                       */
       {
         ofst1=i*ix+j;                     /* offset into picture          */
         if(imgptr[ofst1]==255)
         {
            ofst2=i*bufx+j;                /* offset into dilation buffer  */
            bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
            bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255;
            bufptr[ofst2+x5]=255; bufptr[ofst2+x6]=255;
            bufptr[ofst2+x7]=255; bufptr[ofst2+x8]=255;
            bufptr[ofst2+x9]=255; bufptr[ofst2+x10]=255;
            bufptr[ofst2+x11]=255; bufptr[ofst2+x12]=255;
            bufptr[ofst2+x13]=255; bufptr[ofst2+x14]=255;
            bufptr[ofst2+x15]=255; bufptr[ofst2+x16]=255;
            bufptr[ofst2+x17]=255; bufptr[ofst2+x18]=255;
            bufptr[ofst2+x19]=255; bufptr[ofst2+x20]=255;
            bufptr[ofst2+x21]=255; bufptr[ofst2+x22]=255;
            bufptr[ofst2+x23]=255; bufptr[ofst2+x24]=255;
            bufptr[ofst2+x25]=255; bufptr[ofst2+x26]=255;
            bufptr[ofst2+x27]=255; bufptr[ofst2+x28]=255;
            bufptr[ofst2+x29]=255; bufptr[ofst2+x30]=255;
            bufptr[ofst2+x31]=255; bufptr[ofst2+x32]=255; /* dilate point */
         }
       }
     }
```

```
    h=0;
    for(k=0;k<buffersize;k++)
    {
       if(bufptr[k]==255) h++;           /* integrate dilated area       */
    }
    data[9]=(float)log10((double)h/32.0);
    data[8]=(float)log10((double)(icx+icy)/(sqrt(32.0)*2.0));
    sprintf(msg,"%d pixels at 32 pixel dilation",h);
    DRAW_MESSAGE(2,msg);

/*                    Stage 7, 57 pixel dilation                          */

                                    x1=3;        x2=4;        x3=5;
                          x4=bufx+2;    x5=bufx+3;    x6=bufx+4;    x7=bufx+5;
x8=bufx+6;
             x9=bufx*2+1;  x10=bufx*2+2; x11=bufx*2+3; x12=bufx*2+4;
x13=bufx*2+5; x14=bufx*2+6; x15=bufx*2+7;
x16=bufx*3; x17=bufx*3+1; x18=bufx*3+2; x19=bufx*3+3; x20=bufx*3+4;
x21=bufx*3+5; x22=bufx*3+6; x23=bufx*3+7; x24=bufx*3+8;
x25=bufx*4; x26=bufx*4+1; x27=bufx*4+2; x28=bufx*4+3; x29=bufx*4+4;
x30=bufx*4+5; x31=bufx*4+6; x32=bufx*4+7; x33=bufx*4+8;
x34=bufx*5; x35=bufx*5+1; x36=bufx*5+2; x37=bufx*5+3; x38=bufx*5+4;
x39=bufx*5+5; x40=bufx*5+6; x41=bufx*5+7; x42=bufx*5+8;
             x43=bufx*6+1; x44=bufx*6+2; x45=bufx*6+3; x46=bufx*6+4;
x47=bufx*6+5; x48=bufx*6+6; x49=bufx*6+7;
                          x50=bufx*7+2; x51=bufx*7+3; x52=bufx*7+4;
x53=bufx*7+5; x54=bufx*7+6;
                                     x55=bufx*8+3; x56=bufx*8+4; x57=bufx*8+5;

   memset(bufptr,0,buffersize);          /* clear memory                 */
   for(i=0;i<icy;i++)                    /* y loop                       */
   {
     for(j=0;j<icx;j++)                  /* x loop                       */
     {
       ofst1=i*ix+j;                     /* offset into picture          */
       if(imgptr[ofst1]==255)
       {
          ofst2=i*bufx+j;                /* offset into dilation buffer  */
          bufptr[ofst2+x1]=255; bufptr[ofst2+x2]=255;
          bufptr[ofst2+x3]=255; bufptr[ofst2+x4]=255;
          bufptr[ofst2+x5]=255; bufptr[ofst2+x6]=255;
          bufptr[ofst2+x7]=255; bufptr[ofst2+x8]=255;
          bufptr[ofst2+x9]=255; bufptr[ofst2+x10]=255;
          bufptr[ofst2+x11]=255; bufptr[ofst2+x12]=255;
          bufptr[ofst2+x13]=255; bufptr[ofst2+x14]=255;
          bufptr[ofst2+x15]=255; bufptr[ofst2+x16]=255;
          bufptr[ofst2+x17]=255; bufptr[ofst2+x18]=255;
          bufptr[ofst2+x19]=255; bufptr[ofst2+x20]=255;
          bufptr[ofst2+x21]=255; bufptr[ofst2+x22]=255;
          bufptr[ofst2+x23]=255; bufptr[ofst2+x24]=255;
          bufptr[ofst2+x25]=255; bufptr[ofst2+x26]=255;
          bufptr[ofst2+x27]=255; bufptr[ofst2+x28]=255;
          bufptr[ofst2+x29]=255; bufptr[ofst2+x30]=255;
          bufptr[ofst2+x31]=255; bufptr[ofst2+x32]=255;
```

```
            bufptr[ofst2+x33]=255; bufptr[ofst2+x34]=255;
            bufptr[ofst2+x35]=255; bufptr[ofst2+x36]=255;
            bufptr[ofst2+x37]=255; bufptr[ofst2+x38]=255;
            bufptr[ofst2+x39]=255; bufptr[ofst2+x40]=255;
            bufptr[ofst2+x41]=255; bufptr[ofst2+x42]=255;
            bufptr[ofst2+x43]=255; bufptr[ofst2+x44]=255;
            bufptr[ofst2+x45]=255; bufptr[ofst2+x46]=255;
            bufptr[ofst2+x47]=255; bufptr[ofst2+x48]=255;
            bufptr[ofst2+x49]=255; bufptr[ofst2+x50]=255;
            bufptr[ofst2+x51]=255; bufptr[ofst2+x52]=255;
            bufptr[ofst2+x53]=255; bufptr[ofst2+x54]=255;
            bufptr[ofst2+x55]=255; bufptr[ofst2+x56]=255;
            bufptr[ofst2+x57]=255;                          /* dilate point */
         }
       }
    }
    h=0;
    for(k=0;k<buffersize;k++)
    {
       if(bufptr[k]==255) h++;          /* integrate dilated area       */
    }
    data[11]=(float)log10((double)h/57.0);
    data[10]=(float)log10((double)(icx+icy)/(sqrt(57.0)*2.0));
    sprintf(msg,"%d pixels at 57 pixel dilation",h);
    DRAW_MESSAGE(2,msg);


/*                      calculate dimension from numbers                */
/*                           by linear regression                       */

   n=6.0;
   *dimension=LINREG(n,data);

/*      Plot results to right half of screen for display                */

   xoffset=513;
   _setviewport(xoffset,1,xoffset+509,510);     /* set viewport        */
   _clearscreen(_GVIEWPORT);                    /* clear screen        */
   tmp2=((float)((int)(data[11]*2.0)))/2.0;     /* set min scale       */
   tmp3=((float)((int)(data[0]*2.0))+1.0)/2.0;
   tmp4=((float)((int)(data[1]*2.0))+1.0)/2.0;  /* set max scale       */
   tmp1=tmp2-tmp4+tmp3;                         /* ensure 1:1 scale    */
   strcpy(st1,"DILATION 2 DIMENSION PLOT");
   strcpy(st2,"Log number of squares vs Log scale");

   SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
   DATASET(12,data,CYAN,&PLT);
   TITLES(st1,st2,&PLT);
   DRAWBORDER(&PLT);
   DRAWDATA(&PLT);

   SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
   DATASET(12,data,BRIGHTWHITE,&PLT);
   DRAWDATA(&PLT);
```

```
   free(bufptr);                          /* free buffer used              */
   return;
}

/*-------------------------------------------------------------------------*/
/*                          LENGINE routine                                */
/*      This routine takes the values in the given LMAP structure          */
/*      and recursively builds the image in the memory allocated           */
/*      and pointed to in the given PICTURE structure.  Each line          */
/*      segment is assumed small enough that only the new end point        */
/*      is required to be added to the image.                              */
/*-------------------------------------------------------------------------*/

void LENGINE(RULES,PIC,iteration,command)
struct LMAP *RULES;                 /* pointer to the LMAP structure       */
struct PICTURE *PIC;                /* pointer to the PICTURE structure    */
int iteration;                      /* depth of recursion currently at     */
int command;                        /* command at this depth to follow     */
{
   long i,c,offset;

   if(iteration>0)                  /* not to bottom if > 0                */
   {
      switch(command)
      {
         case 'r' :
         case 'f' :                                 /* forward one l       */
            i=0;
            while((c=RULES->rule0[i])!='\0')
            {
               RULES->size[RULES->szptr++]=RULES->l; /* save l             */
               RULES->l=RULES->l*RULES->mult1;
               LENGINE(RULES,PIC,(iteration-1),c);   /* recursive call     */
               RULES->l=RULES->size[--RULES->szptr]; /* recall l           */
               i++;
            }
            return;
            break;

         case 'l' :
            i=0;
            while((c=RULES->rule1[i])!='\0')
            {
               RULES->size[RULES->szptr++]=RULES->l; /* save l             */
               RULES->l=RULES->l*RULES->mult1;
               LENGINE(RULES,PIC,(iteration-1),c);   /* recursive call     */
               RULES->l=RULES->size[--RULES->szptr]; /* recall l           */
               i++;
            }
            return;
            break;

         case '+' :                                 /* increase by angle0  */
```

```
            RULES->tangle+=1;                          /* increase angle       */
            break;

         case '-' :                                    /* decrease by angle0   */
            RULES->tangle-=1;                          /* decrease angle       */
            break;

         case 'b' :                                    /* branch               */
            RULES->branch[RULES->brptr++]=RULES->x;     /* save x              */
            RULES->branch[RULES->brptr++]=RULES->bx;    /* save x              */
            RULES->branch[RULES->brptr++]=RULES->y;     /* save y              */
            RULES->branch[RULES->brptr++]=RULES->by;    /* save y              */
            RULES->angl[RULES->angptr++]=RULES->tangle; /* save tangle         */
            break;

         case 'c' :                                    /* continue from branch */
            RULES->tangle=RULES->angl[--RULES->angptr]; /* restore tangle      */
            RULES->by=RULES->branch[--RULES->brptr];    /* restore y           */
            RULES->y=RULES->branch[--RULES->brptr];     /* restore y           */
            RULES->bx=RULES->branch[--RULES->brptr];    /* restore x           */
            RULES->x=RULES->branch[--RULES->brptr];     /* restore x           */
            break;

         case 'd' :                                    /* increase by angle 1  */
            RULES->tangle+=RULES->angle1;              /* increase angle       */
            break;

         case 'e' :                                    /* decrease by angle 1  */
            RULES->tangle-=RULES->angle1;              /* decrease angle       */
            break;

         case 's' :                                    /* short l segment      */
            i=0;
            while((c=RULES->rule0[i])!='\0')
            {
               RULES->size[RULES->szptr++]=RULES->l; /* save l               */
               RULES->l=RULES->l*RULES->mult1*RULES->mult2;
               LENGINE(RULES,PIC,(iteration-1),c);   /* recursive call       */
               RULES->l=RULES->size[--RULES->szptr]; /* recall l             */
               i++;
            }
            return;
            break;
      }
   }
   else
   {
      switch(command)
      {
         case 'r' :
         case 'l' :
         case 'f' :
            RULES->x = RULES->x + RULES->l * RULES->cosary[RULES->tangle];
            if(RULES->x > 0.01)
```

```
            {
               RULES->x-=0.01L;
               RULES->bx+=0.01L;
            }
            if(RULES->x < -0.01)
            {
               RULES->x+=0.01L;
               RULES->bx-=0.01L;
            }
            RULES->y = RULES->y + RULES->l * RULES->sinary[RULES->tangle];
            if(RULES->y > 0.01)
            {
               RULES->y-=0.01L;
               RULES->by+=0.01L;
            }
            if(RULES->y < -0.01)
            {
               RULES->y+=0.01L;
               RULES->by-=0.01L;
            }
            offset=PIC->imagewidth * (long)(RULES->by+RULES->y+0.5L)
                                   + (long)(RULES->bx+RULES->x+0.5L);
            PIC->pic_buf[offset]=255;            /* set pixel white      */
            break;

         case 's' :
           RULES->x = RULES->x + RULES->l * RULES->mult2 * RULES->cosary[RULES-
>tangle];
            if(RULES->x > 0.01)
            {
               RULES->x-=0.01L;
               RULES->bx+=0.01L;
            }
            if(RULES->x < -0.01)
            {
               RULES->x+=0.01L;
               RULES->bx-=0.01L;
            }
           RULES->y = RULES->y + RULES->l * RULES->mult2 * RULES->sinary[RULES-
>tangle];
            if(RULES->y > 0.01)
            {
               RULES->y-=0.01L;
               RULES->by+=0.01L;
            }
            if(RULES->y < -0.01)
            {
               RULES->y+=0.01L;
               RULES->by-=0.01L;
            }
            offset=PIC->imagewidth * (long)(RULES->by+RULES->y+0.5L)
                                   + (long)(RULES->bx+RULES->x+0.5L);
            PIC->pic_buf[offset]=255;            /* set pixel white      */
            break;
```

```
            case '+' :
               RULES->tangle+=1;                    /* increase angle       */
               break;

            case '-' :
               RULES->tangle-=1;                    /* decrease angle       */
               break;

            case 'b' :
               RULES->branch[RULES->brptr++]=RULES->x;      /* save x       */
               RULES->branch[RULES->brptr++]=RULES->bx;     /* save x       */
               RULES->branch[RULES->brptr++]=RULES->y;      /* save y       */
               RULES->branch[RULES->brptr++]=RULES->by;     /* save y       */
               RULES->angl[RULES->angptr++]=RULES->tangle; /* save tangle   */
               break;

            case 'c' :
               RULES->tangle=RULES->angl[--RULES->angptr]; /* restore tangle */
               RULES->by=RULES->branch[--RULES->brptr];     /* restore y    */
               RULES->y=RULES->branch[--RULES->brptr];      /* restore y    */
               RULES->bx=RULES->branch[--RULES->brptr];     /* restore x    */
               RULES->x=RULES->branch[--RULES->brptr];      /* restore x    */
               break;

            case 'd' :
               RULES->tangle+=RULES->angle1;        /* increase angle       */
               break;

            case 'e' :
               RULES->tangle-=RULES->angle1;        /* decrease angle       */
               break;
         }
      }
      return;
}

/*-------------------------------------------------------------------------*/
/*                              LMKOCH3                                    */
/*      This routine creates a 1024x1024 picture of a tree with            */
/*      dimension=1.104.  The curve is placed into memory and the          */
/*      necessary parameters set in the global PIC structure.              */
/*      If a picture already exists in memory, the memory is first         */
/*      freed before allocating the new memory array.                      */
/*-------------------------------------------------------------------------*/

int LMKOCH3(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                      Erase previous pic if required                     */
```

```
   if (PIC.valid_pic==1) {                /* ==1 if previous pic created  */
      free(PIC.pic_buf);                 /* free previous image          */
   }
/*                      Allocate new picture arrary                       */

   i=1024*1024;                           /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
      return(0);                          /* return with error flag       */
   }
   picptr=PIC.pic_buf;                    /* save picture location        */
   memset(picptr,0,i);                    /* clear memory                 */

/*              Allocate memory for stacks and angle arrays               */

   if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   TREE.szptr=0;
   TREE.brptr=0;

/*                      Ask for number of iterations                      */

   DRAW_MESSAGE(2,"Enter iterations (12) :");
   GET_MESSAGE(2,buffer);
   n=atoi(buffer);
   if(n<=0) n=12;

/*                      Write strings to LMAP structure                   */

   strcpy(TREE.rule0,"f+s--s+f");         /* LMKOCH3 rules                */
   pi=3.141592653589793L;
   TREE.angle0= -pi/4.0L;                 /* 45 degrees                   */

/*              initialize angle arrays                                   */

   for(i=0;i<100;i++)
   {
      TREE.cosary[i]=cos((double)(i-50)*TREE.angle0);
      TREE.sinary[i]=sin((double)(i-50)*TREE.angle0);
   }
   TREE.tangle=50;                        /* initial angle = 0            */

   TREE.l=1000.0L;                        /* size of dl at start          */
   TREE.mult1=1.0L/3.0L;                  /* multiplier                   */
```

```
    TREE.mult2=sqrt(0.50L);              /* multiplier                    */
    TREE.bx=10.0L;
    TREE.x=0.0L;
    TREE.by=512.0L;                      /* start of image                */
    TREE.y=0.0L;

/*                      Call LENGINE to create curve                     */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');           /* create image                  */

/*                      Calculate error in generated image               */

    error=sqrt((1010.0-TREE.bx-TREE.x)*(1010.0-TREE.bx-TREE.x)+
               (512.0-TREE.by-TREE.y)*(512.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;           /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    PIC.valid_pic=1;                     /* now valid picture             */
    free(TREE.size);
    free(TREE.branch);                   /* free stacks                   */
    free(TREE.cosary);                   /* free angle array              */
    free(TREE.sinary);                   /* free angle array              */
    return(1);                           /* signal good results           */
}

/*---------------------------------------------------------------------*/
/*                              LKOCH                                  */
/*      This routine creates a 1024x1024 picture of the standard       */
/*      Koch curve.  The curve is placed into memory and the           */
/*      necessary parameters set in the global PIC structure.          */
/*      If a picture already exists in memory, the memory is first     */
/*      freed before allocating the new memory array.                  */
/*---------------------------------------------------------------------*/

int LKOCH(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP KOCH;

/*                      Erase previous pic if required                   */

   if (PIC.valid_pic==1) {               /* ==1 if previous pic created  */
      free(PIC.pic_buf);                /* free previous image          */
```

```
    }
/*                      Allocate new picture array                          */

    i=1024*1024;                          /* size of new image             */
    if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
       return(0);                         /* return with error flag        */
    }
    picptr=PIC.pic_buf;                   /* save picture location         */
    memset(picptr,0,i);                   /* clear memory                  */

/*                      Allocate memory for stacks                          */

    if((KOCH.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag        */
    }
    if((KOCH.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag        */
    }
    if((KOCH.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag        */
    }
    if((KOCH.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag        */
    }
    KOCH.szptr=0;
    KOCH.brptr=0;

/*                      Ask for number of iterations                        */

    DRAW_MESSAGE(2,"Enter iterations (12) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=12;

/*                      Write strings to LMAP structure                     */

    strcpy(KOCH.rule0,"f+f--f+f");        /* KOCH rules                    */
    pi=3.141592653589793L;
    KOCH.angle0= -pi/3.0L;                /* 60 degrees                    */

/*              initialize angle arrays                                     */

    for(i=0;i<100;i++)
    {
       KOCH.cosary[i]=cos((double)(i-50)*KOCH.angle0);
       KOCH.sinary[i]=sin((double)(i-50)*KOCH.angle0);
    }
    KOCH.tangle=50;                       /* initial angle = 0             */

    KOCH.l=1000.0L;                       /* size of l at start            */
    KOCH.mult1=1.0L/3.0L;                 /* multiplier                    */
    KOCH.bx=10.0L;
    KOCH.x=0.0L;
    KOCH.by=512.0L;                       /* start of image                */
```

```
    KOCH.y=0.0L;

/*                      Call LENGINE to create curve                    */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&KOCH,&PIC,n,'f');          /* create image                */

/*                      Calculate error in generated image              */

    error=sqrt((1010.0-KOCH.bx-KOCH.x)*(1010.0-KOCH.bx-KOCH.x)+
               (512.0-KOCH.by-KOCH.y)*(512.0-KOCH.by-KOCH.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;          /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    PIC.valid_pic=1;                    /* now valid picture            */
    free(KOCH.branch);
    free(KOCH.size);                    /* free stacks                  */
    free(KOCH.cosary);                  /* free angle array             */
    free(KOCH.sinary);                  /* free angle array             */
    return(1);                          /* signal good results          */
}

/*--------------------------------------------------------------------*/
/*                              LMKOCH                                */
/*      This routine creates a 1024x1024 Koch type curve with         */
/*      dimension=1.34.  The curve is placed into memory and the      */
/*      necessary parameters set in the global PIC structure.         */
/*      If a picture already exists in memory, the memory is first    */
/*      freed before allocating the new memory array.                 */
/*--------------------------------------------------------------------*/

int LMKOCH(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                      Erase previous pic if required                  */

   if (PIC.valid_pic==1) {              /* ==1 if previous pic created  */
      free(PIC.pic_buf);               /* free previous image          */
   }
/*                      Allocate new picture arrary                     */

   i=1024*1024;                         /* size of new image           */
```

```
    if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
       return(0);                               /* return with error flag       */
    }
    picptr=PIC.pic_buf;                         /* save picture location        */
    memset(picptr,0,i);                         /* clear memory                 */

/*                        Allocate memory for stacks                            */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                               /* return with error flag       */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                               /* return with error flag       */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                               /* return with error flag       */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                               /* return with error flag       */
    }
    TREE.szptr=0;
    TREE.brptr=0;

/*                        Ask for number of iterations                          */

    DRAW_MESSAGE(2,"Enter iterations (10) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=10;

/*                        Write strings to LMAP structure                       */

    strcpy(TREE.rule0,"f-s+f+s-f");             /* MKOCH rules                  */
    pi=3.141592653589793L;
    TREE.angle0= pi/2.0L;                       /* 90 degrees                   */

/*                initialize angle arrays                                       */

    for(i=0;i<100;i+=4)
    {
       TREE.cosary[i]=-1.0L;      TREE.sinary[i]=0.0L;
       TREE.cosary[i+1]=0.0L;   TREE.sinary[i+1]=-1.0L;
       TREE.cosary[i+2]=+1.0L;  TREE.sinary[i+2]=0.0L;
       TREE.cosary[i+3]=0.0L;   TREE.sinary[i+3]=1.0L;
    }
    TREE.tangle=50;                             /* initial angle = 0            */

    TREE.l=1000.0L;                             /* size of dl at start          */
    TREE.mult1=1.0L/3.0L;                       /* multiplier                   */
    TREE.mult2=0.750L;                          /* multiplier                   */
    TREE.bx=10.0L;
    TREE.x = 0.0L;
    TREE.by=512.0L;                             /* start of image               */
    TREE.y = 0.0L;
```

```
/*                      Call LENGINE to create curve                    */

   PIC.imagewidth=1024;
   PIC.imagelength=1024;
   LENGINE(&TREE,&PIC,n,'f');          /* create image                  */

/*                      Calculate error in generated image              */

   error=sqrt((1010.0-TREE.bx-TREE.x)*(1010.0-TREE.bx-TREE.x)+
              (512.0-TREE.by-TREE.y)*(512.0-TREE.by-TREE.y));
   n=0;
   for(i=0;i<1024*1024;i++)
   {
      if(picptr[i]==255) n++;          /* count number of pixels used  */
   }
   sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
   DRAW_MESSAGE(2,buffer);
   i=getch();

   free(TREE.size);                    .
   free(TREE.branch);                 /* free stacks                   */
   free(TREE.cosary);                 /* free angle array              */
   free(TREE.sinary);                 /* free angle array              */
   PIC.valid_pic=1;                   /* now valid picture             */
   return(1);                         /* signal good results           */
}

/*-------------------------------------------------------------------------*/
/*                              LKOCH15                                    */
/*      This routine creates a 1024x1024 picture of a 3/2 dimension        */
/*      Koch type curve.  The curve is placed into memory and the          */
/*      necessary parameters set in the global PIC structure.              */
/*      If a picture already exists in memory, the memory is first         */
/*      freed before allocating the new memory array.                      */
/*-------------------------------------------------------------------------*/

int LKOCH15(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP KOCH;

/*                      Erase previous pic if required                  */

   if (PIC.valid_pic==1) {             /* ==1 if previous pic created  */
      free(PIC.pic_buf);              /* free previous image          */
   }
/*                      Allocate new picture array                      */

   i=1024*1024;                        /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
```

```
      return(0);                          /* return with error flag      */
   }
   picptr=PIC.pic_buf;                    /* save picture location       */
   memset(picptr,0,i);                    /* clear memory                */

/*                      Allocate memory for stacks                        */

   if((KOCH.branch=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag      */
   }
   if((KOCH.size=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag      */
   }
   if((KOCH.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag      */
   }
   if((KOCH.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag      */
   }
   KOCH.szptr=0;
   KOCH.brptr=0;

/*                      Ask for number of iterations                      */

   DRAW_MESSAGE(2,"Enter iterations (8) :");
   GET_MESSAGE(2,buffer);
   n=atoi(buffer);
   if(n<=0) n=8;

/*                      Write strings to LMAP structure                   */

   strcpy(KOCH.rule0,"f+f-f-ff+f+f-f"); /* KOCH15 rules                   */
   pi=3.141592653589793L;
   KOCH.angle0= -pi/2.0L;                 /* 90 degrees                  */

/*             initialize angle arrays                                    */

   for(i=0;i<100;i+=4)
   {
      KOCH.cosary[i]=-1.0L;    KOCH.sinary[i]=0.0L;
      KOCH.cosary[i+1]=0.0L;   KOCH.sinary[i+1]=1.0L;
      KOCH.cosary[i+2]=1.0L;   KOCH.sinary[i+2]=0.0L;
      KOCH.cosary[i+3]=0.0L;   KOCH.sinary[i+3]=-1.0L;
   }
   KOCH.tangle=50;                        /* initial angle = 0           */

   KOCH.l=1000.0L;                        /* size of l at start          */
   KOCH.mult1=1.0L/4.0L;                  /* multiplier                  */
   KOCH.bx=10.0L;
   KOCH.x=0.0L;
   KOCH.by=512.0L;                        /* start of image              */
   KOCH.y=0.0L;

/*                      Call LENGINE to create curve                      */
```

```
    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&KOCH,&PIC,n,'f');          /* create image                  */

/*                      Calculate error in generated image                */

    error=sqrt((1010.0-KOCH.bx-KOCH.x)*(1010.0-KOCH.bx-KOCH.x)+
               (512.0-KOCH.by-KOCH.y)*(512.0-KOCH.by-KOCH.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;          /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    PIC.valid_pic=1;                    /* now valid picture             */
    free(KOCH.branch);
    free(KOCH.size);                    /* free stacks                   */
    free(KOCH.cosary);                  /* free angle array              */
    free(KOCH.sinary);                  /* free angle array              */
    return(1);                          /* signal good results           */
}

/*---------------------------------------------------------------------*/
/*                              LTREE1                                 */
/*      This routine creates a 1024x1024 picture of a tree with        */
/*      dimension=1.465.  The curve is placed into memory and the      */
/*      necessary parameters set in the global PIC structure.          */
/*      If a picture already exists in memory, the memory is first     */
/*      freed before allocating the new memory array.                  */
/*---------------------------------------------------------------------*/

int LTREE1(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                      Erase previous pic if required                    */

   if (PIC.valid_pic==1) {              /* ==1 if previous pic created  */
      free(PIC.pic_buf);               /* free previous image          */
   }
/*                      Allocate new picture arrary                       */

   i=1024*1024;                         /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
      return(0);                        /* return with error flag       */
   }
```

```
    picptr=PIC.pic_buf;                    /* save picture location        */
    memset(picptr,0,i);                    /* clear memory                 */

 /*                        Allocate memory for stacks                       */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.angl=(int *)malloc(200*sizeof(int)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    TREE.szptr=0;
    TREE.brptr=0;
    TREE.angptr=0;

 /*                        Ask for number of iterations                     */

    DRAW_MESSAGE(2,"Enter iterations (10) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=10;

 /*                        Write strings to LMAP structure                  */

    strcpy(TREE.rule0,"fb+fcfb-fcf");      /* TREE1 rules                  */
    pi=3.141592653589793L;
    TREE.angle0= pi/8.0L;                  /* 22.5 degrees                 */

 /*               initialize angle arrays                                   */

    for(i=0;i<100;i++)
    {
       TREE.cosary[i]=cos((double)(i-50)*TREE.angle0);
       TREE.sinary[i]=sin((double)(i-50)*TREE.angle0);
    }
    TREE.tangle=46;                        /* initial angle = 90           */

    TREE.l=1000.0L;                        /* size of dl at start          */
    TREE.mult1=1.0L/3.0L;                  /* multiplier                   */
    TREE.bx=512.0L;
    TREE.x = 0.0L;
    TREE.by=1010.0L;                       /* start of image               */
    TREE.y = 0.0L;

 /*                        Call LENGINE to create curve                     */
```

```
    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');          /* create image                  */

/*                      Calculate error in generated image                */

    error=sqrt((512.0-TREE.bx-TREE.x)*(512.0-TREE.bx-TREE.x)+
               (10.0-TREE.by-TREE.y)*(10.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;          /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    free(TREE.size);
    free(TREE.angl);
    free(TREE.branch);                  /* free stacks                   */
    free(TREE.cosary);                  /* free angle array              */
    free(TREE.sinary);                  /* free angle array              */
    PIC.valid_pic=1;                    /* now valid picture             */
    return(1);                          /* signal good results           */
}

/*-------------------------------------------------------------------------*/
/*                              LTREE2                                     */
/*      This routine creates a 1024x1024 picture of a tree with            */
/*      dimension=1.465.  The curve is placed into memory and the          */
/*      necessary parameters set in the global PIC structure.              */
/*      If a picture already exists in memory, the memory is first         */
/*      freed before allocating the new memory array.                      */
/*-------------------------------------------------------------------------*/

int LTREE2(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                      Erase previous pic if required                    */

   if (PIC.valid_pic==1) {              /* ==1 if previous pic created  */
      free(PIC.pic_buf);               /* free previous image          */
   }
/*                      Allocate new picture arrary                       */

   i=1024*1024;                         /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
      return(0);                        /* return with error flag       */
```

```
    }
    picptr=PIC.pic_buf;                    /* save picture location        */
    memset(picptr,0,i);                    /* clear memory                 */

 /*                      Allocate memory for stacks                         */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.angl=(int *)malloc(200*sizeof(int)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                          /* return with error flag       */
    }
    TREE.szptr=0;
    TREE.brptr=0;
    TREE.angptr=0;

/*                      Ask for number of iterations                       */

    DRAW_MESSAGE(2,"Enter iterations (10) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=10;

/*                      Write strings to LMAP structure                    */

    strcpy(TREE.rule0,"fb-fcfb+fcf");      /* TREE2 rules                  */
    pi=3.141592653589793L;
    TREE.angle0= pi/2.0L;                  /* 90 degrees                   */

/*              initialize angle arrays                                    */

    for(i=0;i<100;i+=4)
    {
       TREE.cosary[i]=-1.0L;      TREE.sinary[i]=0.0L;
       TREE.cosary[i+1]=0.0L;   TREE.sinary[i+1]=-1.0L;
       TREE.cosary[i+2]=+1.0L;  TREE.sinary[i+2]=0.0L;
       TREE.cosary[i+3]=0.0L;   TREE.sinary[i+3]=1.0L;
    }
    TREE.tangle=50;                        /* initial angle = 0            */

    TREE.l=1000.0L;                        /* size of dl at start          */
    TREE.mult1=1.0L/3.0L;                  /* multiplier                   */

    TREE.bx=10.0L;
    TREE.x = 0.0L;
```

```
    TREE.by=512.0L;                          /* start of image              */
    TREE.y = 0.0L;

 /*                        Call LENGINE to create curve                      */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');               /* create image                */

 /*                        Calculate error in generated image                */

    error=sqrt((1010.0-TREE.bx-TREE.x)*(1010.0-TREE.bx-TREE.x)+
               (512.0-TREE.by-TREE.y)*(512.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;               /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    free(TREE.size);
    free(TREE.angl);
    free(TREE.branch);                       /* free stacks                 */
    free(TREE.cosary);                       /* free angle array            */
    free(TREE.sinary);                       /* free angle array            */
    PIC.valid_pic=1;                         /* now valid picture           */
    return(1);                               /* signal good results         */
}

/*-------------------------------------------------------------------------*/
/*                                LTREE3                                   */
/*      This routine creates a 1024x1024 picture of a tree with            */
/*      dimension=1.5282.  The curve is placed into memory and the         */
/*      necessary parameters set in the global PIC structure.              */
/*      If a picture already exists in memory, the memory is first         */
/*      freed before allocating the new memory array.                      */
/*-------------------------------------------------------------------------*/

int LTREE3(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                        Erase previous pic if required                    */

   if (PIC.valid_pic==1) {                   /* ==1 if previous pic created */
      free(PIC.pic_buf);                     /* free previous image         */
   }
/*                        Allocate new picture arrary                       */
```

```
    i=1024*1024;                          /* size of new image            */
    if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
       return(0);                         /* return with error flag       */
    }
    picptr=PIC.pic_buf;                   /* save picture location        */
    memset(picptr,0,i);                   /* clear memory                 */

 /*                     Allocate memory for stacks                         */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag       */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag       */
    }
    if((TREE.angl=(int *)malloc(200*sizeof(int)))==NULL) {
       return(0);                         /* return with error flag       */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag       */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                         /* return with error flag       */
    }
    TREE.szptr=0;
    TREE.brptr=0;
    TREE.angptr=0;

 /*                     Ask for number of iterations                       */

    DRAW_MESSAGE(2,"Enter iterations (7) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=7;

 /*                     Write strings to LMAP structure                    */

    strcpy(TREE.rule0,"fb-fscfb+sb+scscfb-fcf"); /* TREE2 rules           */
    pi=3.141592653589793L;
    TREE.angle0= pi/4.0L;                 /* 45 degrees                   */

 /*             initialize angle arrays                                    */

    for(i=0;i<100;i++)
    {
       TREE.cosary[i]=cos((double)(i-50)*TREE.angle0);
       TREE.sinary[i]=sin((double)(i-50)*TREE.angle0);
    }
    TREE.tangle=48;                       /* initial angle = -90          */

    TREE.l=1000.0L;                       /* size of dl at start          */
    TREE.mult1=1.0L/4.0L;                 /* multiplier                   */
    TREE.mult2=0.700L;                    /* multiplier                   */
```

```
    TREE.bx=512.0L;
    TREE.x = 0.0L;
    TREE.by=1010.0L;                        /* start of image              */
    TREE.y = 0.0L;

 /*                        Call LENGINE to create curve                     */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');              /* create image                */

 /*                        Calculate error in generated image               */

    error=sqrt((512.0-TREE.bx-TREE.x)*(512.0-TREE.bx-TREE.x)+
               (10.0-TREE.by-TREE.y)*(10.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;              /* count number of pixels used */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
 tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    free(TREE.size);
    free(TREE.angl);
    free(TREE.branch);                      /* free stacks                 */
    free(TREE.cosary);                      /* free angle array            */
    free(TREE.sinary);                      /* free angle array            */
    PIC.valid_pic=1;                        /* now valid picture           */
    return(1);                              /* signal good results         */
 }

 /*-------------------------------------------------------------------------*/
 /*                                LTREE4                                   */
 /*      This routine creates a 1024x1024 picture of a tree with            */
 /*      dimension=1.5736.  The curve is placed into memory and the         */
 /*      necessary parameters set in the global PIC structure.              */
 /*      If a picture already exists in memory, the memory is first         */
 /*      freed before allocating the new memory array.                      */
 /*-------------------------------------------------------------------------*/

 int LTREE4(void)
 {
    long i,n;
    double error,pi;
    char *picptr,buffer[60];
    struct LMAP TREE;

 /*                        Erase previous pic if required                   */

    if (PIC.valid_pic==1) {                 /* ==1 if previous pic created */
```

```
      free(PIC.pic_buf);                  /* free previous image          */
   }
/*                        Allocate new picture arrary                      */

   i=1024*1024;                           /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
      return(0);                          /* return with error flag       */
   }
   picptr=PIC.pic_buf;                    /* save picture location        */
   memset(picptr,0,i);                    /* clear memory                 */

/*                        Allocate memory for stacks                       */

   if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.angl=(int *)malloc(200*sizeof(int)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                          /* return with error flag       */
   }
   TREE.szptr=0;
   TREE.brptr=0;
   TREE.angptr=0;

/*                        Ask for number of iterations                     */

   DRAW_MESSAGE(2,"Enter iterations (6) :");
   GET_MESSAGE(2,buffer);
   n=atoi(buffer);
   if(n<=0) n=6;

/*                        Write strings to LMAP structure                  */

   strcpy(TREE.rule0,"fb+sfcb-sfcfb+sscb-sscfb+scb-scf"); /* TREE4 rules */
   pi=3.141592653589793L;
   TREE.angle0= pi*55.0L/180.0L;          /* 55 degrees                   */

/*              initialize angle arrays                                    */

   for(i=0;i<100;i++)
   {
      TREE.cosary[i]=cos((double)(i-50)*TREE.angle0-pi/2.0L);
      TREE.sinary[i]=sin((double)(i-50)*TREE.angle0-pi/2.0L);
   }
   TREE.tangle=50;                        /* initial angle = -90          */
```

```
    TREE.l=1000.0L;                    /* size of dl at start          */
    TREE.mult1=1.0L/4.0L;              /* multiplier                   */
    TREE.mult2=0.520L;                 /* multiplier                   */

    TREE.bx=512.0L;
    TREE.x = 0.0L;
    TREE.by=1010.0L;                   /* start of image               */
    TREE.y = 0.0L;

/*                      Call LENGINE to create curve                   */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');         /* create image                 */

/*                      Calculate error in generated image             */

    error=sqrt((512.0-TREE.bx-TREE.x)*(512.0-TREE.bx-TREE.x)+
               (10.0-TREE.by-TREE.y)*(10.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;         /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    free(TREE.size);
    free(TREE.angl);
    free(TREE.branch);                 /* free stacks                  */
    free(TREE.cosary);                 /* free angle array             */
    free(TREE.sinary);                 /* free angle array             */
    PIC.valid_pic=1;                   /* now valid picture            */
    return(1);                         /* signal good results          */
}

/*--------------------------------------------------------------------*/
/*                              LSGASKET                              */
/*      This routine creates a 1024x1024 picture of the Sierpinski    */
/*      gasket.  The curve is placed into memory and the              */
/*      necessary parameters set in the global PIC structure.         */
/*      If a picture already exists in memory, the memory is first    */
/*      freed before allocating the new memory array.                 */
/*--------------------------------------------------------------------*/

int LSGASKET(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP KOCH;
```

```
/*                      Erase previous pic if required                    */

   if (PIC.valid_pic==1) {              /* ==1 if previous pic created  */
      free(PIC.pic_buf);               /* free previous image          */
   }
/*                      Allocate new picture array                        */

   i=1024*1024;                         /* size of new image            */
   if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
      return(0);                        /* return with error flag       */
   }
   picptr=PIC.pic_buf;                  /* save picture location        */
   memset(picptr,0,i);                  /* clear memory                 */

/*                      Allocate memory for stacks                        */

   if((KOCH.branch=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                        /* return with error flag       */
   }
   if((KOCH.size=(double *)malloc(200*sizeof(double)))==NULL) {
      return(0);                        /* return with error flag       */
   }
   if((KOCH.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                        /* return with error flag       */
   }
   if((KOCH.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
      return(0);                        /* return with error flag       */
   }
   KOCH.szptr=0;
   KOCH.brptr=0;

/*                      Ask for number of iterations                      */

   DRAW_MESSAGE(2,"Enter iterations (15) :");
   GET_MESSAGE(2,buffer);
   n=atoi(buffer);
   if(n<=0) n=15;

/*                      Write strings to LMAP structure                   */

   strcpy(KOCH.rule0,"-l+r+l-");        /* SGASKET rules                */
   strcpy(KOCH.rule1,"+r-l-r+");        /* SGASKET rules                */
   pi=3.141592653589793L;
   KOCH.angle0= -pi/3.0L;               /* 60 degrees                   */

/*              initialize angle arrays                                   */

   for(i=0;i<100;i++)
   {
      KOCH.cosary[i]=cos((double)(i-50)*KOCH.angle0);
      KOCH.sinary[i]=sin((double)(i-50)*KOCH.angle0);
   }
   KOCH.tangle=50;                      /* initial angle = 0            */
```

```
    KOCH.l=1000.0L;                     /* size of l at start          */
    KOCH.mult1=1.0L/2.0L;               /* multiplier                  */
    KOCH.bx=10.0L;
    KOCH.x=0.0L;
    KOCH.by=950.0L;                     /* start of image              */
    KOCH.y=0.0L;

 /*                    Call LENGINE to create curve                    */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&KOCH,&PIC,n,'l');          /* create image                */

 /*                    Calculate error in generated image              */

    error=sqrt((1010.0-KOCH.bx-KOCH.x)*(1010.0-KOCH.bx-KOCH.x)+
               (950.0-KOCH.by-KOCH.y)*(950.0-KOCH.by-KOCH.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;          /* count number of pixels used */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    PIC.valid_pic=1;                    /* now valid picture           */
    free(KOCH.branch);
    free(KOCH.size);                    /* free stacks                 */
    free(KOCH.cosary);                  /* free angle array            */
    free(KOCH.sinary);                  /* free angle array            */
    return(1);                          /* signal good results         */
}

/*-------------------------------------------------------------------------*/
/*                              LMKOCH2                                    */
/*      This routine creates a 1024x1024 Koch type curve with              */
/*      dimension ?=1.4649.  The curve is placed into memory and the       */
/*      necessary parameters set in the global PIC structure.              */
/*      If a picture already exists in memory, the memory is first         */
/*      freed before allocating the new memory array.                      */
/*      The curve created does not have a surface dimension equal to       */
/*      its Hausdorff dimension since the curve intersects at points.      */
/*-------------------------------------------------------------------------*/

int LMKOCH2(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                    Erase previous pic if required                    */
```

```
    if (PIC.valid_pic==1) {                 /* ==1 if previous pic created  */
       free(PIC.pic_buf);                  /* free previous image          */
    }
 /*                        Allocate new picture arrary                      */

    i=1024*1024;                            /* size of new image           */
    if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
       return(0);                           /* return with error flag       */
    }
    picptr=PIC.pic_buf;                     /* save picture location       */
    memset(picptr,0,i);                     /* clear memory                */

 /*                        Allocate memory for stacks                       */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                           /* return with error flag       */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                           /* return with error flag       */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                           /* return with error flag       */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                           /* return with error flag       */
    }
    TREE.szptr=0;
    TREE.brptr=0;

 /*                        Ask for number of iterations                     */

    DRAW_MESSAGE(2,"Enter iterations (9) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=9;

 /*                        Write strings to LMAP structure                  */

    strcpy(TREE.rule0,"f-f+f+f-f");         /* MKOCH rules                 */
    pi=3.141592653589793L;
    TREE.angle0= pi/2.0L;                   /* 90 degrees                  */

 /*              initialize angle arrays                                    */

    for(i=0;i<100;i++)
    {
       TREE.cosary[i]=cos((double)(i-50)*TREE.angle0);
       TREE.sinary[i]=sin((double)(i-50)*TREE.angle0);
    }
    TREE.tangle=50;                         /* initial angle = 0           */

    TREE.l=1000.0L;                         /* size of dl at start         */
    TREE.mult1=1.0L/3.0L;                   /* multiplier                  */
```

```
    TREE.mult2=0.750L;                  /* multiplier                   */
    TREE.bx=10.0L;
    TREE.x = 0.0L;
    TREE.by=512.0L;                     /* start of image               */
    TREE.y = 0.0L;

 /*                      Call LENGINE to create curve                   */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');          /* create image                 */

 /*                      Calculate error in generated image             */

    error=sqrt((1010.0-TREE.bx-TREE.x)*(1010.0-TREE.bx-TREE.x)+
               (512.0-TREE.by-TREE.y)*(512.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;          /* count number of pixels used  */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    free(TREE.size);
    free(TREE.branch);                  /* free stacks                  */
    free(TREE.cosary);                  /* free angle array             */
    free(TREE.sinary);                  /* free angle array             */
    PIC.valid_pic=1;                    /* now valid picture            */
    return(1);                          /* signal good results          */
}

/*--------------------------------------------------------------------------*/
/*                              LMKOCH3B    with angle                    */
/*      This routine creates a 1024x1024 picture of a tree with           */
/*      dimension=1.104.  The curve is placed into memory and the         */
/*      necessary parameters set in the global PIC structure.             */
/*      If a picture already exists in memory, the memory is first        */
/*      freed before allocating the new memory array.                     */
/*--------------------------------------------------------------------------*/

int LMKOCH3B(void)
{
   long i,n;
   double error,pi;
   char *picptr,buffer[60];
   struct LMAP TREE;

/*                      Erase previous pic if required                  */

   if (PIC.valid_pic==1) {              /* ==1 if previous pic created  */
      free(PIC.pic_buf);               /* free previous image          */
```

```
    }
/*                        Allocate new picture arrary                        */

    i=1024*1024;                              /* size of new image          */
    if((PIC.pic_buf=(char *)malloc(i))==NULL) {  /* memory allocation error */
       return(0);                             /* return with error flag     */
    }
    picptr=PIC.pic_buf;                       /* save picture location      */
    memset(picptr,0,i);                       /* clear memory               */

/*              Allocate memory for stacks and angle arrays                  */

    if((TREE.branch=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                             /* return with error flag     */
    }
    if((TREE.size=(double *)malloc(200*sizeof(double)))==NULL) {
       return(0);                             /* return with error flag     */
    }
    if((TREE.cosary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                             /* return with error flag     */
    }
    if((TREE.sinary=(double *)malloc(100*sizeof(double)))==NULL) {
       return(0);                             /* return with error flag     */
    }
    TREE.szptr=0;
    TREE.brptr=0;

/*                        Ask for number of iterations                       */

    DRAW_MESSAGE(2,"Enter iterations (12) :");
    GET_MESSAGE(2,buffer);
    n=atoi(buffer);
    if(n<=0) n=12;

/*                        Write strings to LMAP structure                    */

    strcpy(TREE.rule0,"f+s--s+f");            /* LMKOCH3 rules              */
    pi=3.141592653589793L;
    TREE.angle0= -pi/4.0L;                    /* 45 degrees                 */

/*              initialize angle arrays                                      */

    for(i=0;i<100;i++)
    {
       TREE.cosary[i]=cos((double)(i-50)*TREE.angle0+pi/8.0);
       TREE.sinary[i]=sin((double)(i-50)*TREE.angle0+pi/8.0);
    }
    TREE.tangle=50;                           /* initial angle = 0          */

    TREE.l=1000.0L;                           /* size of dl at start        */
    TREE.mult1=1.0L/3.0L;                     /* multiplier                 */
    TREE.mult2=sqrt(0.50L);                   /* multiplier                 */
    TREE.bx=10.0L;
    TREE.x=0.0L;
```

```
    TREE.by=512.0L;                          /* start of image              */
    TREE.y=0.0L;

 /*                        Call LENGINE to create curve                      */

    PIC.imagewidth=1024;
    PIC.imagelength=1024;
    LENGINE(&TREE,&PIC,n,'f');               /* create image                */

 /*                        Calculate error in generated image                */

    error=sqrt((1010.0-TREE.bx-TREE.x)*(1010.0-TREE.bx-TREE.x)+
               (512.0-TREE.by-TREE.y)*(512.0-TREE.by-TREE.y));
    n=0;
    for(i=0;i<1024*1024;i++)
    {
       if(picptr[i]==255) n++;               /* count number of pixels used */
    }
    sprintf(buffer,"Residual error= %5.3f pixels in %d pixels (key to con-
tinue)",error,n);
    DRAW_MESSAGE(2,buffer);
    i=getch();

    PIC.valid_pic=1;                         /* now valid picture           */
    free(TREE.size);
    free(TREE.branch);                       /* free stacks                 */
    free(TREE.cosary);                       /* free angle array            */
    free(TREE.sinary);                       /* free angle array            */
    return(1);                               /* signal good results         */
}


/*-------------------------------------------------------------------------*/
/*                                MASSDIMEN                                */
/*      The Mass Correlation Dimension routine takes the picture and       */
/*      calculates the Mass Correlation Dimension.  A cursor is            */
/*      displayed for the user to select the manually select the           */
/*      center of the image.  From this point the euclidean distance       */
/*      is used to segment the image.  The results of the segmentation     */
/*      is used to calculate the dimension.                                */
/*-------------------------------------------------------------------------*/

extern int right_button;          /* global mouse variables                */
extern int left_button;
extern int mouse_event;
extern int mouse_code;
extern int mouse_cx;
extern int mouse_dx;
extern int installed;
extern struct SREGS sregs;
extern union REGS inregs, outregs;
extern int (far *function_ptr)();
extern char msg1[60];
```

```
void MASSDIMEN(dimension,PIC)
float *dimension;
struct PICTURE *PIC;
{
   long imwd,imln,i,j,x0,y0,xs,ys,delta;
   long offset,doffset,radius,maxr;
   int maxx,maxy,d,oldx,oldy,oldcx,olddx;
   short dx,dy;
   unsigned char cursor[55],under[55];
   char *imgptr;
   char st1[60],st2[60];
   float data[200],tmp1,tmp2,tmp3,tmp4,pi;
   struct SCREEN PLT;

   imgptr=PIC->pic_buf;                  /* get picture location          */
   imwd=PIC->imagewidth;
   imln=PIC->imagelength;                /* get picture size              */
   delta=4096/min(imwd,imln);            /* fine tune step                */

/*      draw memory into left hand side                                    */

   if (PIC->valid_pic==1)
   {
      PIC_DRAW(0,PIC);
      DRAW_MESSAGE(0,msg1);
   }
   else
   {
      DRAW_MESSAGE(2,"No picture in memory to detect.  Press key to continue");
      d=getch();                         /* wait for key press    */
      return;
   }

/*      set cursor limits and initial position                             */
   if(installed==1)
   {
      inregs.w.ax = 0x0007;
      inregs.w.cx = 0;
      inregs.w.dx = 4092;
      int386( 0x33, &inregs, &outregs );        /* set x range  */
      inregs.w.ax = 0x0008;
      inregs.w.cx = 0;
      inregs.w.dx = 4092;
      int386( 0x33, &inregs, &outregs );        /* set y range  */
      inregs.w.ax = 0x0004;
      inregs.w.cx = 2048;
      inregs.w.dx = 2048;
      int386( 0x33, &inregs, &outregs );        /* set position */
      inregs.w.ax = 0x000F;
      inregs.w.cx = 8;
      inregs.w.dx = 8;
      int386( 0x33, &inregs, &outregs );        /* set sensitivity */
```

```
    }

/*      initialize cursor and position                                    */

    _setviewport(1,1,510,510);
    mouse_cx=2048;
    mouse_dx=2048;
    xs=510*mouse_cx/4092+1;
    ys=510*mouse_dx/4092+1;
    _getimage(xs-3,ys-3,xs+3,ys+3,under); /* get image under cursor      */

/*              cursor positioning routine                                */

    oldx=0;
    oldy=0;                         /* ensure first condition            */
    oldcx=0;
    olddx=0;
    mouse_event=0;
    left_button=0;
    right_button=0;
    inregs.w.ax = 0x000B;
    int386( 0x33, &inregs, &outregs );    /* reset delta mouse position   */
    while(1)                              /* cursor positioning loop      */
    {
       if(installed)                      /* if mouse try to use          */
       {
          inregs.w.ax = 0x000B;
          int386( 0x33, &inregs, &outregs ); /* read delta mouse position */
          dx=outregs.w.cx;
          dy=outregs.w.dx;
          mouse_cx+=dx;
          mouse_dx+=dy;
          mouse_cx=max(0,min(4092,mouse_cx));
          mouse_dx=max(0,min(4092,mouse_dx));     /* clamp movement       */
       }

       if(kbhit()!=0)                     /* see if keyboard used         */
       {
          if((d=getch())==0)
          {
             d=getch();                   /* get arrow keys               */
             switch(d)
             {
                case 72 :                 /* up arrow                     */
                   mouse_dx-=delta;
                   break;
                case 75 :                 /* left arrow                   */
                   mouse_cx-=delta;
                   break;
                case 77 :                 /* right arrow                  */
                   mouse_cx+=delta;
                   break;
                case 80 :                 /* down arrow                   */
                   mouse_dx+=delta;
```

```
                    break;
                }
                mouse_cx=max(0,min(4092,mouse_cx));
                mouse_dx=max(0,min(4092,mouse_dx)); /* clamp movement       */
            }
            else
            {
                if(d==27) return;
                if(d==13) break;
            }
        }
        xs=510*mouse_cx/4092+1;
        ys=510*mouse_dx/4092+1;
        if(oldx!=xs || oldy!=ys)                    /* check if cursor moved */
        {
            _putimage(oldx-3,oldy-3,under,_GPSET); /* erase old image      */
            oldx=xs;
            oldy=ys;                                /* save old location    */
            xs=510*mouse_cx/4092+1;
            ys=510*mouse_dx/4092+1;
            _getimage(xs-3,ys-3,xs+3,ys+3,under); /* get image under cursor  */
            memcpy(cursor,under,55);              /* copy image to cursor     */
            cursor[9]=150; cursor[16]=150; cursor[23]=150;
            cursor[27]=150; cursor[28]=150; cursor[29]=150;
            cursor[30]=150; cursor[31]=150; cursor[32]=150;
            cursor[33]=150; cursor[37]=150; cursor[44]=150;
            cursor[51]=150;                       /* put in green cursor      */
            _putimage(xs-3,ys-3,cursor,_GPSET);
        }
        if(oldcx!=mouse_cx || olddx!=mouse_dx) /* check if position changed */
        {
            x0=mouse_cx/delta;
            y0=mouse_dx/delta;
            sprintf(st1,"(%i,%i)",x0,y0);
            DRAW_MESSAGE(0,st1);
            _setviewport(1,1,510,510);
            oldcx=mouse_cx;
            olddx=mouse_dx;
        }
        if(left_button==1 || d==13)        /* check if selection was made  */
        {
            left_button=0;
            _putimage(x0-3,y0-3,under,_GPSET);
            break;
        }
    }

/*              Main counting loop                                          */

    maxx=x0;
    maxx=max(maxx,abs(x0-imwd));
    maxy=y0;
    maxy=max(maxy,abs(y0-imln));          /* find maximum distance         */
    maxr=(long)sqrt((double)(maxx*maxx+maxy*maxy)); /* maximum radius     */
```

```
    for(i=0;i<100;i++)                          /* reset data counters          */
    {
       data[i+i]=(float)(((i+1)*maxr)/(99)); /* set radius steps             */
       data[i+i+1]=0.0;                         /* reset data counters          */
    }

    offset=0;                                   /* set to starting point        */
    for(i=0;i<imln;i++)                         /* y loop                       */
    {
       for(j=0;j<imwd;j++)                      /* x loop                       */
       {
          if(imgptr[offset]==255)               /* if white then count          */
          {
            radius=min(maxr,(long)sqrt((double)((x0-j)*(x0-j)+(y0-i)*(y0-i))));
             doffset=(((radius*99)/maxr)*2+1);
             data[doffset]+=1.0;
          }
          offset++;                             /* move to next pixel           */
       }
    }

/*      convert data into form required by plotting routine                    */

    pi=3.141592653589793;
    for(i=0;i<198;i+=2)
    {
       data[i+3]=data[i+1]+data[i+3];     /* integrate mass counts        */
    }
    tmp4=0.0;
    for(i=0;i<200;i+=2)
    {
       data[i+1]=data[i+1]/(pi*data[i]*data[i]); /* calc prob. of mass   */
       data[i+1]=(float)log((double)data[i+1]);  /* take log of prob.    */
       data[i+1]=data[i+1]/(double)log(data[i]); /* convert to ratio     */
       tmp4=max(tmp4,data[i+1]);                 /* save max for plot    */
    }

/*                      calculate dimension from numbers                      */
/*                           by linear regression                             */

/* n=100.0;                    */
/* *dimension=LINREG(n,data); */

/*      Plot results to right half of screen for display                       */

   _setviewport(513,1,1022,510);                /* set viewport         */
   _clearscreen(_GVIEWPORT);                    /* clear screen         */
   tmp2=-3.0;     /* set min scale        */
   tmp3=data[198];
   tmp1=0.0;                            /*      */
   strcpy(st1,"MASS DIMENSION PLOT");
   strcpy(st2,"Number of pixels vs radius");
   SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
```

```
    DATASET(200,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(200,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);
    return;
}
```

C3 FILE: EDGES.C

```
/*                  Include files and prototypes                          */

#include <graph.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <stddef.h>
#include <dos.h>
#include <string.h>
#include <math.h>
#include "picture.h"

/*                  function prototypes                                   */

int LAPLACE5PT(void);
int EXP9PT(void);
int EXP13PT(void);
int EXP25PT(void);
int GAUSS9PT(void);
int GAUSS13PT(void);
int GAUSS25PT(void);

extern struct PICTURE PIC;
extern char *imgptr;
extern int imflag;
extern char msg1[60];

/*-------------------------------------------------------------------------*/
/*                            LAPLACE5PT                                   */
/*      This routine runs the picture though a Laplacian Edge              */
/*      Detector to remove all but the edges that are present in the       */
/*      picture.  The standard 5 element mask with the following           */
/*      weights are used:                                                  */
/*                                                                         */
/*                                -1000                                    */
/*                              -1000  4000 -1000                          */
/*                                -1000                                    */
/*                                                                         */
/*-------------------------------------------------------------------------*/

int LAPLACE5PT(void)
{
```

```
    char *picptr, *picptr_tmp;
    long i,j,k,a,b,c,d,x,y,tmp;

    picptr=PIC.pic_buf;                    /* get picture location          */
    x=PIC.imagewidth;                      /* get size of picture           */
    y=PIC.imagelength;                     /* get size of picture           */

    k=x*y;                                 /* size of new image             */
    if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
    {
          return(0);                           /* return with error flag       */
    }

/*                        set array to zero                                   */

    memset(picptr_tmp,0,k);

/*                        calculate pixel references                          */
/*      Each pixel reference will use the offsets calculated here.            */
/*      The pixel base is first set and all offsets from the base             */
/*      use one of the following numbers depending upon the position          */
/*      of the pixel required. ie:                                            */
/*                                                                            */
/*                                        a                                   */
/*                                   b    base    c                           */
/*                                        d                                   */
/*                                                                            */
    a=-x;
    b=-1;
    c=1;
    d=x;

/*                        calculate interior of picture                       */

    for(j=1;j<(y-1);j++)
    {
          for(i=1;i<(x-1);i++)
          {
          k=j*x+i;
          tmp=((long)picptr[k]*4000-(picptr[k+a]+picptr[k+b]+picptr[k+c]+
                    picptr[k+d])*1000)/400;
          picptr_tmp[k]=max(0,min(255,tmp));
          }
    }

    i=x*y;
    memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
    free(picptr_tmp);                      /* free working array           */

}

/*-------------------------------------------------------------------------*/
/*                              EXP9PT                                     */
/*      This routine runs the picture though a Exponential Edge            */
```

```
/*      Detector to remove all but the edges that are present in the    */
/*      picture.  The standard 9 element mask is used.                  */
/*                                                                      */
/*----------------------------------------------------------------------*/

int EXP9PT(void)
{
   char *picptr, *picptr_tmp;
   long i,j,k,a,b,c,d,e,f,g,h,x,y,tmp;

   picptr=PIC.pic_buf;                  /* get picture location        */
   x=PIC.imagewidth;                    /* get size of picture         */
   y=PIC.imagelength;                   /* get size of picture         */

   k=x*y;                               /* size of new image           */
   if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
   {
      return(0);                        /* return with error flag      */
   }

/*                      set array to zero                               */

   memset(picptr_tmp,0,k);

/*                      calculate pixel references                      */
/*      Each pixel reference will use the offsets calculated here.      */
/*      The pixel base is first set and all offsets from the base       */
/*      use one of the following numbers depending upon the position    */
/*      of the pixel required. ie:                                      */
/*                                      a      b      c                 */
/*                                      d    base     e                 */
/*                                      f      g      h                 */
/*                                                                      */
   a=-x-1;
   b=-x;
   c=-x+1;
   d=-1;
   e=1;
   f=x-1;
   g=x;
   h=x+1;

/*                      calculate interior of picture                   */

   for(j=1;j<(y-1);j++)
   {
      for(i=1;i<(x-1);i++)
      {
         k=j*x+i;
         tmp=((long)picptr[k]*5748
              -(picptr[k+a]+picptr[k+c]+picptr[k+f]+picptr[k+h])*1000
              -(picptr[k+b]+picptr[k+d]+picptr[k+e]+picptr[k+g])*437)/575;
         picptr_tmp[k]=max(0,min(255,tmp));
```

```
      }
    }

    i=x*y;
    memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
    free(picptr_tmp);                      /* free working array           */

}

/*-------------------------------------------------------------------------*/
/*                              EXP13PT                                    */
/*      This routine runs the picture though a 13 Point Exponential        */
/*      Edge Detector to remove all but the edges that are present in      */
/*      the picture.  A 13 element mask is used for best results.          */
/*                                                                         */
/*-------------------------------------------------------------------------*/

int EXP13PT(void)
{
   char *picptr, *picptr_tmp;
   long i,j,k,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,x,y;
   long tmp1,tmp2,tmp3,tmp4;

   picptr=PIC.pic_buf;                     /* get picture location        */
   x=PIC.imagewidth;                       /* get size of picture         */
   y=PIC.imagelength;                      /* get size of picture         */

   k=x*y;                                  /* size of new image           */
   if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
   {
        return(0);                            /* return with error flag      */
   }

/*                      set array to zero                                  */

  memset(picptr_tmp,0,k);

/*                      calculate pixel references                         */
/*      Each pixel reference will use the offsets calculated here.         */
/*      The pixel base is first set and all offsets from the base          */
/*      use one of the following numbers depending upon the position       */
/*      of the pixel required. ie:                                         */
/*                                                                         */
/*                                        p0                               */
/*                                   p1   p2   p3                          */
/*                              p4   p5   base p6   p7                     */
/*                                   p8   p9   p10                         */
/*                                        p11                              */
/*                                                                         */

                        p0=-2*x;
              p1=-x-1;  p2=-x;    p3=-x+1;
        p4=-2; p5=-1;             p6=1;      p7=2;
              p8=x-1;   p9=x;     p10=x+1;
                        p11=2*x;
```

```
 /*                        calculate interior of picture                    */

    for(j=2;j<(y-2);j++)
    {
          for(i=2;i<(x-2);i++)
          {
          k=j*x+i;
          tmp1=picptr[k+p2]+picptr[k+p5]+picptr[k+p6]+picptr[k+p9];
          tmp2=picptr[k+p1]+picptr[k+p3]+picptr[k+p8]+picptr[k+p10];
          tmp3=picptr[k+p0]+picptr[k+p4]+picptr[k+p7]+picptr[k+p11];
          tmp4=((long)picptr[k]*6288-tmp1*1000-tmp2*437-tmp3*135)/629;
          picptr_tmp[k]=max(0,min(255,tmp4));
          }
    }

    i=x*y;
    memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
    free(picptr_tmp);                      /* free working array           */

 }

 /*-------------------------------------------------------------------------*/
 /*                              EXP25PT                                    */
 /*      This routine runs the picture though a 25 Point Exponential        */
 /*      Edge Detector to remove all but the edges that are present in      */
 /*      the picture.  A 25 element mask is used for results.               */
 /*                                                                         */
 /*-------------------------------------------------------------------------*/

 int EXP25PT(void)
 {
    char *picptr, *picptr_tmp;
    long i,j,k,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11;
    long p12,p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p23,x,y;
    long tmp1,tmp2,tmp3,tmp4,tmp5,tmp6;

    picptr=PIC.pic_buf;                    /* get picture location         */
    x=PIC.imagewidth;                      /* get size of picture          */
    y=PIC.imagelength;                     /* get size of picture          */

    k=x*y;                                 /* size of new image            */
    if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
    {
          return(0);                           /* return with error flag       */
    }

/*                        set array to zero                                */

   memset(picptr_tmp,0,k);

/*                        calculate pixel references                       */
/*      Each pixel reference will use the offsets calculated here.         */
```

```
/*      The pixel base is first set and all offsets from the base       */
/*      use one of the following numbers depending upon the position    */
/*      of the pixel required. ie:                                      */
/*                                                                      */
/*                                        p0                            */
/*                                  p1    p2    p3                      */
/*                              p4  p5    p6    p7   p8                 */
/*                          p9  p10 p11   base  p12  p13  p14           */
/*                              p15 p16   p17   p18  p19                */
/*                                  p20   p21   p22                     */
/*                                        p23                           */
/*                                                                      */

                        p0=-3*x;
                p1=-2*x-1;  p2=-2*x;    p3=-2*x+1;
        p4=-x-2;  p5=-x-1;    p6=-x;      p7=-x+1;    p8=-x+2;
p9=-3;  p10=-2;   p11=-1;                 p12=1;      p13=2;     p14=2;
        p15=x-2;  p16=x-1;    p17=x;      p18=x+1;    p19=x+2;
                p20=2*x-1;  p21=2*x;    p22=2*x+1;
                            p23=3*x;

/*                      calculate interior of picture                   */

   for(j=3;j<(y-3);j++)
   {
         for(i=3;i<(x-3);i++)
         {
         k=j*x+i;
         tmp1=picptr[k+p6]+picptr[k+p11]+picptr[k+p12]+picptr[k+p17];
         tmp2=picptr[k+p5]+picptr[k+p7]+picptr[k+p16]+picptr[k+p18];
         tmp3=picptr[k+p2]+picptr[k+p10]+picptr[k+p13]+picptr[k+p21];
         tmp4=picptr[k+p1]+picptr[k+p3]+picptr[k+p4]+picptr[k+p8]+
                 picptr[k+p15]+picptr[k+p19]+picptr[k+p20]+picptr[k+p22];
         tmp5=picptr[k+p0]+picptr[k+p9]+picptr[k+p14]+picptr[k+p23];
         tmp6=((long)picptr[k]*7032-tmp1*1000-tmp2*437-tmp3*135-tmp4*84
                   -tmp5*18)/703;
         picptr_tmp[k]=max(0,min(255,tmp6));
         }
   }

   i=x*y;
   memmove(picptr,picptr_tmp,i);        /* copy result back into buffer */
   free(picptr_tmp);                    /* free working array           */

}

/*----------------------------------------------------------------------*/
/*                              GAUSS9PT                                */
/*      This routine runs the picture though a Gaussian Edge            */
/*      Detector to remove all but the edges that are present in the    */
/*      picture.  The standard 9 element mask is used.                  */
/*                                                                      */
/*----------------------------------------------------------------------*/

int GAUSS9PT(void)
{
```

```
    char *picptr, *picptr_tmp;
    long i,j,k,a,b,c,d,e,f,g,h,x,y,tmp;

    picptr=PIC.pic_buf;                    /* get picture location          */
    x=PIC.imagewidth;                      /* get size of picture           */
    y=PIC.imagelength;                     /* get size of picture           */

    k=x*y;                                 /* size of new image             */
    if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
    {
       return(0);                          /* return with error flag        */
    }

/*                        set array to zero                                  */

   memset(picptr_tmp,0,k);

/*                        calculate pixel references                         */
/*      Each pixel reference will use the offsets calculated here.           */
/*      The pixel base is first set and all offsets from the base            */
/*      use one of the following numbers depending upon the position         */
/*      of the pixel required. ie:                                           */
/*                                                                           */
/*                                      a      b     c                       */
/*                                      d    base    e                       */
/*                                      f      g     h                       */
/*                                                                           */
   a=-x-1;
   b=-x;
   c=-x+1;
   d=-1;
   e=1;
   f=x-1;
   g=x;
   h=x+1;

/*                        calculate interior of picture                      */

   for(j=1;j<(y-1);j++)
   {
      for(i=1;i<(x-1);i++)
      {
         k=j*x+i;
         tmp=((long)picptr[k]*6428
              -(picptr[k+a]+picptr[k+c]+picptr[k+f]+picptr[k+h])*1000
              -(picptr[k+b]+picptr[k+d]+picptr[k+e]+picptr[k+g])*607)/643;
         picptr_tmp[k]=max(0,min(255,tmp));
      }
   }

   i=x*y;
   memmove(picptr,picptr_tmp,i);           /* copy result back into buffer */
   free(picptr_tmp);                       /* free working array           */
```

```
}

/*-------------------------------------------------------------------------*/
/*                              GAUSS13PT                                  */
/*      This routine runs the picture though a 13 Point Gaussian Edge      */
/*      Detector to remove all but the edges that are present in the       */
/*      picture.  A 13 element mask is used for best results.              */
/*                                                                         */
/*-------------------------------------------------------------------------*/

int GAUSS13PT(void)
{
   char *picptr, *picptr_tmp;
   long i,j,k,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,x,y;
   long tmp1,tmp2,tmp3,tmp4;

   picptr=PIC.pic_buf;                  /* get picture location          */
   x=PIC.imagewidth;                    /* get size of picture           */
   y=PIC.imagelength;                   /* get size of picture           */

   k=x*y;                               /* size of new image             */
   if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
   {
        return(0);                          /* return with error flag       */
   }

/*                      set array to zero                                  */

   memset(picptr_tmp,0,k);

/*                      calculate pixel references                         */
/*      Each pixel reference will use the offsets calculated here.         */
/*      The pixel base is first set and all offsets from the base          */
/*      use one of the following numbers depending upon the position       */
/*      of the pixel required. ie:                                         */
/*                                          p0                             */
/*                                    p1    p2    p3                       */
/*                                p4  p5   base   p6   p7                  */
/*                                    p8    p9    p10                      */
/*                                          p11                            */
/*                                                                         */

                          p0=-2*x;
               p1=-x-1;  p2=-x;    p3=-x+1;
          p4=-2;  p5=-1;              p6=1;      p7=2;
               p8=x-1;   p9=x;      p10=x+1;
                          p11=2*x;

/*                      calculate interior of picture                      */

   for(j=2;j<(y-2);j++)
   {
        for(i=2;i<(x-2);i++)
        {
```

```
            k=j*x+i;
            tmp1=picptr[k+p2]+picptr[k+p5]+picptr[k+p6]+picptr[k+p9];
            tmp2=picptr[k+p1]+picptr[k+p3]+picptr[k+p8]+picptr[k+p10];
            tmp3=picptr[k+p0]+picptr[k+p4]+picptr[k+p7]+picptr[k+p11];
            tmp4=((long)picptr[k]*7320-tmp1*1000-tmp2*607-tmp3*223)/732;
            picptr_tmp[k]=max(0,min(255,tmp4));
            }
      }

      i=x*y;
      memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
      free(picptr_tmp);                      /* free working array           */

   }

   /*-------------------------------------------------------------------------*/
   /*                              GAUSS25PT                                  */
   /*      This routine runs the picture though a 25 Point Gaussian Edge      */
   /*      Detector to remove all but the edges that are present in the       */
   /*      picture.  A 25 element mask is used for results.                   */
   /*                                                                         */
   /*-------------------------------------------------------------------------*/

   int GAUSS25PT(void)
   {
      char *picptr, *picptr_tmp;
      long i,j,k,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11;
      long p12,p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p23,x,y;
      long tmp1,tmp2,tmp3,tmp4,tmp5,tmp6;

      picptr=PIC.pic_buf;                    /* get picture location         */
      x=PIC.imagewidth;                      /* get size of picture          */
      y=PIC.imagelength;                     /* get size of picture          */

      k=x*y;                                 /* size of new image            */
      if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
   error      */
      {
            return(0);                       /* return with error flag       */
      }

   /*                       set array to zero                                 */

      memset(picptr_tmp,0,k);

   /*                       calculate pixel references                        */
   /*      Each pixel reference will use the offsets calculated here.         */
   /*      The pixel base is first set and all offsets from the base          */
   /*      use one of the following numbers depending upon the position       */
   /*      of the pixel required. ie:                                         */
   /*                                                                         */
   /*                                        p0                               */
   /*                                  p1    p2    p3                         */
   /*                              p4  p5    p6    p7   p8                    */
   /*                          p9  p10 p11   base  p12  p13  p14              */
```

```
/*                                        p15  p16    p17   p18  p19          */
/*                                             p20    p21   p22               */
/*                                                    p23                     */
/*                                                                            */
                              p0=-3*x;
                  p1=-2*x-1;  p2=-2*x;    p3=-2*x+1;
        p4=-x-2;  p5=-x-1;    p6=-x;      p7=-x+1;    p8=-x+2;
p9=-3;  p10=-2;   p11=-1;                 p12=1;      p13=2;     p14=2;
        p15=x-2;  p16=x-1;    p17=x;      p18=x+1;    p19=x+2;
                  p20=2*x-1;  p21=2*x;    p22=2*x+1;
                              p23=3*x;

/*                      calculate interior of picture                         */

   for(j=3;j<(y-3);j++)
   {
         for(i=3;i<(x-3);i++)
         {
         k=j*x+i;
         tmp1=picptr[k+p6]+picptr[k+p11]+picptr[k+p12]+picptr[k+p17];
         tmp2=picptr[k+p5]+picptr[k+p7]+picptr[k+p16]+picptr[k+p18];
         tmp3=picptr[k+p2]+picptr[k+p10]+picptr[k+p13]+picptr[k+p21];
         tmp4=picptr[k+p1]+picptr[k+p3]+picptr[k+p4]+picptr[k+p8]+
                picptr[k+p15]+picptr[k+p19]+picptr[k+p20]+picptr[k+p22];
         tmp5=picptr[k+p0]+picptr[k+p9]+picptr[k+p14]+picptr[k+p23];
         tmp6=((long)picptr[k]*8480-tmp1*1000-tmp2*607-tmp3*223-tmp4*136
                                    -tmp5*18)/848;
         picptr_tmp[k]=max(0,min(255,tmp6));
         }
   }

   i=x*y;
   memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
   free(picptr_tmp);                      /* free working array           */

}
```

C4 FILE: FULD.C

```
/*                      Dimension calculating routines                        */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <malloc.h>
#include <string.h>
#include <dos.h>
#include <i86.h>
#include <conio.h>
#include "picture.h"
#include "screen.h"
#include "lmap.h"
```

```
extern struct PICTURE PIC;

void FULDILATION(float *dimension);
void MASSCORR(float *dimension,struct PICTURE *PIC);

extern void DATASET(int n,float *data,int color,struct SCREEN *PLT);
extern void TITLES(char *st1,char *st2,struct SCREEN *PLT);
extern void DRAWBORDER(struct SCREEN *PLT);
extern void DRAWDATA(struct SCREEN *PLT);
extern void DRAW_MESSAGE(int location,char *msg);
extern void PIC_DRAW(int side,struct PICTURE *DISP);
extern float LINREG(int n,float *data);
extern void SCREENSET(int xst,int yst,int xsp,int ysp,float xmn,float ymn,
               float xmx,float ymx,int typ,struct SCREEN *SCRN);
extern int  GET_MESSAGE(int side, char *file);
void lubksb(double *a,long n,long *indx,double *b);
void ludcmp(double *a,long n,long *indx,double *d);
void free_vector(double *v,int nl,int nh);
void nrerror(char *error_text);
double *vector(int nl,int nh);


/*-------------------------------------------------------------------------*/
/*                              FULDILATION                                */
/*      This routine takes the image in memory, creates a working          */
/*      buffer, and dilates each point in the image into the buffer.       */
/*      The buffer is then counted for set pixels.  The size of the        */
/*      dilation and the area the dilated picture covers gives the         */
/*      data required for calculating the dimension of the image.          */
/*         The results are then fitted to a log/log polynomial using       */
/*      least squares on the data.  Dilation here is with squares.         */
/*      This routine dilates each pixel over as large a range as           */
/*      possible and curve fits a user selected polynomial to the data     */
/*      in order to characterize the dimension of a multifractal as        */
/*      close as possible.  Finally, the maximum slope of the              */
/*      is found and given as the multifractal dimension of the image.     */
/*-------------------------------------------------------------------------*/

void FULDILATION(float *dimension)
{
   long h,i,j,k,ix,iy,isz,ofst1,ofst2,ofst3,buffersize;
   long ii,jj,xoffset,bufx,bufy,fit,size,t1,t2;
   long dilationsize[16],indx[5];
   float data[32],tp1,tp2,tp3,tp4;
   double tmp1,tmp2,tmp3,n,inv[5][5],ldim,cdim,pdim;
   double x1,y1,x2,y2,a1,a2,a3,a4;
   double At[5][17],A[17][5],y[17],x[5],AtA[5][5],b[5],dval;
   double *d;
   char c,msg[60],st1[60],st2[60];
   char *imgptr,*bufptr;
   struct SCREEN PLT;

   imgptr=PIC.pic_buf;                  /* locate picture              */
   ix=PIC.imagewidth;                   /* start at full resolution    */
```

```
    iy=PIC.imagelength;                    /* set y variable                */
    dilationsize[0]=1;   dilationsize[1]=2;    dilationsize[2]=3;
    dilationsize[3]=4;   dilationsize[4]=6;    dilationsize[5]=8;
    dilationsize[6]=12;  dilationsize[7]=16;   dilationsize[8]=24;
    dilationsize[9]=32;  dilationsize[10]=48;  dilationsize[11]=64;
    dilationsize[12]=96; dilationsize[13]=128; dilationsize[14]=192;
    dilationsize[15]=256;

/*      allocate memory for dilation buffer                               */

    bufx=ix+255;
    bufy=iy+255;
    buffersize=bufx*bufy;                  /* size of dilation buffer       */
   if((bufptr=(char *)malloc(buffersize))==NULL) {  /* memory allocation error
*/
      DRAW_MESSAGE(2,"memory allocation error");
      return;                              /* return with error flag        */
   }

/*                Stage 1, count all white pixels in picture               */

   h=0;                                    /* pixel counter                 */
   for(i=0;i<iy;i++)                       /* y loop                        */
   {
     for(j=0;j<ix;j++)                     /* x loop                        */
     {
       ofst1=i*ix+j;                       /* offset into picture           */
       if(imgptr[ofst1]==255) h++;         /* count all white pixels        */
     }
   }
   data[1]=-(float)log10((double)(h-1));          /* with end correction   */
   data[0]=0.0;
   sprintf(msg,"%d pixels at %dx%d dilation",(h-1),1,1);
   DRAW_MESSAGE(2,msg);

/*       Stage 2, dilate with 2,3,4,6 boxes with white pixels              */

   for(i=1;i<5;i++) {
     memset(bufptr,0,buffersize);          /* clear memory                  */
     isz=dilationsize[i];                  /* size of dilation              */
     h=0;                                  /* pixel counter                 */
     for(j=0;j<iy;j++)
     {
       for(k=0;k<ix;k++)
       {
         ofst1=j*ix+k;                     /* offset start of box           */
         c=imgptr[ofst1];                  /* get first value in box        */
         if(c==255)
         {
            ofst2=j*bufx+k;                /* offset into dilation buffer   */
            for(ii=0;ii<isz;ii++)          /* y dilation loop               */
            {
               ofst3=ii*bufx+ofst2;
               for(jj=0;jj<isz;jj++)       /* x dilation loop               */
```

```
                {
                   bufptr[ofst3+jj]=255; /* dilate point                */
                }
             }
          }
        }
      }
      h=0;
      for(k=0;k<buffersize;k++)
      {
         if(bufptr[k]==255) h++;          /* integrate dilated area       */
      }
     data[(i)*2+1]=-(float)log10((double)(h)/(isz*isz)-1.0); /* with end cor */
      data[(i)*2]=(float)log10((double)(isz));
      sprintf(msg,"%d pixels at %dx%d dilation",h,isz,isz);
      DRAW_MESSAGE(2,msg);
   }

/*       Stage 3, dilate with rest of boxes with white pixels           */

   for(i=5;i<16;i++) {
     memset(bufptr,0,buffersize);        /* clear memory                 */
     isz=dilationsize[i];                /* size of dilation             */
     h=0;                                /* pixel counter                */
     for(j=0;j<iy;j++)                   /* y image loop                 */
     {
       for(k=0;k<ix;k++)                 /* x image loop                 */
       {
         ofst1=j*ix+k;                   /* offset start of box          */
         c=imgptr[ofst1];                /* get first value in box       */
         if(c==255)
         {
            ofst2=j*bufx+k;              /* offset into dilation buffer  */
            for(ii=0;ii<isz;ii++)        /* y dilation loop              */
            {
               ofst3=ii*bufx+ofst2;
               for(jj=0;jj<isz;jj++)     /* x dilation loop              */
               {
                  bufptr[ofst3+jj]=255; /* dilate point                 */
               }
            }
            t1=min((k+isz-1),(ix-1));    /* find dilation dist.. or edge */
            for(t2=t1;t2>k;t2--)
            {
               if(imgptr[j*ix+t2]==255)
               {
                  k=t2-1;
                  break;
               }
            }
         }
       }
     }
    h=0;
```

```
      for(k=0;k<buffersize;k++)
      {
         if(bufptr[k]==255) h++;          /* integrate dilated area       */
      }

     data[(i)*2+1]=-(float)log10((double)(h)/(isz*isz)-1.0); /* with end cor */
      data[(i)*2]=(float)log10((double)(isz));
      sprintf(msg,"%d pixels at %dx%d dilation",h,isz,isz);
      DRAW_MESSAGE(2,msg);
   }
/*                      calculate dimension from numbers                    */
/*                           by linear regression                           */

   n=16.0;
   ldim=LINREG(n,data);

/*      Plot results to right half of screen for display                    */

   xoffset=513;
   _setviewport(xoffset,1,xoffset+509,510);     /* set viewport           */
   _clearscreen(_GVIEWPORT);                    /* clear screen           */
   tp1=0.0;                                     /* always zero!           */
   tp2=(float)((int)(data[1]*2.0-0.501))/2.0;   /* set min scale          */
   tp4=(float)((int)(data[31]*2.0+0.5))/2.0;    /* set max scale          */
   tp3=tp4-tp2;                                 /* set 1:1 scale          */
   strcpy(st1,"FULL DILATION DIMENSION PLOT");
   strcpy(st2,"Log area vs Log dilation");

   SCREENSET(525,15,480,480,tp1,tp2,tp3,tp4,1,&PLT);
   DATASET(32,data,CYAN,&PLT);
   TITLES(st1,st2,&PLT);
   DRAWBORDER(&PLT);
   DRAWDATA(&PLT);

   SCREENSET(525,15,480,480,tp1,tp2,tp3,tp4,0,&PLT);
   DATASET(32,data,BRIGHTWHITE,&PLT);
   DRAWDATA(&PLT);

/*      calculate dimension using a least squares cubic fit                 */
/*         double A[5][17],At[17][5],AtA[5][5],y[17],x[5];                  */
/*         data[odd]=y-value                                                */
/*         data[even]=x-value                                               */
/*                                                                          */
/*      put data into arrays                                                */

   for(i=1;i<=16;i++)           /* put data into arrays                    */
   {
      A[i][1]=data[i*2-1]*data[i*2-1]*data[i*2-1];
      A[i][2]=data[i*2-1]*data[i*2-1];
      A[i][3]=data[i*2-1];
      A[i][4]=1.0;
      At[1][i]=A[i][1];
      At[2][i]=A[i][2];
      At[3][i]=A[i][3];
```

```
        At[4][i]=A[i][4];
        y[i]=data[(i-1)*2];
    }

/*      multiply At[][] by A[][]                                          */

    for(i=1;i<=4;i++)
    {
       for(j=1;j<=4;j++)
       {
          AtA[i][j]=0.0;
          for(k=1;k<=16;k++)
          {
             AtA[i][j]=AtA[i][j]+At[i][k]*A[k][j];
          }
       }
    }

/*      calculate b=At*y                                                  */

    for(i=1;i<=4;i++)
    {
       b[i]=0.0;
       for(j=1;j<=16;j++)
       {
          b[i]=b[i]+At[i][j]*y[j];
       }
    }

/*      calculate the inverse of AtA[][]                                  */

    d=&dval;
    size=4;
    ludcmp(AtA,size,indx,d);
    for(j=1;j<=size;j++)
    {
       for(i=1;i<=size;i++) x[i]=0.0;
       x[j]=1.0;
       lubksb(AtA,size,indx,x);
       for(i=1;i<=size;i++) inv[i][j]=x[i];
    }

/*      multiply inv(AtA[][])*b to get cubic coefficients                 */

    for(j=1;j<=4;j++)
    {
       dval=0.0;
       for(i=1;i<=4;i++)
       {
          dval=dval+b[i]*inv[j][i];
       }
       x[j]=dval;
    }
```

```
/*      calculate the dimension using max slope of cubic spline          */

    tmp1=(-2.0*x[2])/(6.0*x[1]);          /* calculate location of min    */
    tmp2=(double)data[1];
    tmp3=(double)data[31];
    y1=max(tmp2,min(tmp3,tmp1));          /* constrain result to data     */
    tmp2=3.0*x[1]*y1*y1 + 2.0*x[2]*y1 + x[3]; /* calc inverse slope */
    cdim=1.0/tmp2;                        /* calculate slope              */
    x1=x[1]*y1*y1*y1 + x[2]*y1*y1 + x[3]*y1 + x[4];
    a1=x[1]; a2=x[2]; a3=x[3]; a4=x[4];  /* save variables               */

/*      calculate dimension using a least squares parabolic fit          */
/*         double At[5][17],A[17][5],y[17],x[5],AtA[5][5],b[5];           */
/*         data[odd]=y-value                                             */
/*         data[even]=x-value                                            */
/*                                                                       */
/*      put data into arrays                                             */

   for(i=1;i<=16;i++)           /* put data into arrays                  */
   {
      A[i][1]=data[(i-1)*2]*data[(i-1)*2];
      A[i][2]=data[(i-1)*2];
      A[i][3]=1.0;
      At[1][i]=A[i][1];
      At[2][i]=A[i][2];
      At[3][i]=A[i][3];
      y[i]=data[i*2-1];
   }

/*      multiply At[][] by A[][]                                         */

   for(i=1;i<=3;i++)
   {
      for(j=1;j<=3;j++)
      {
         AtA[i][j]=0.0;
         for(k=1;k<=16;k++)
         {
            AtA[i][j]=AtA[i][j]+At[i][k]*A[k][j];
         }
      }
   }

/*      calculate b=At*y                                                 */

   for(i=1;i<=3;i++)
   {
      b[i]=0.0;
      for(j=1;j<=16;j++)
      {
         b[i]=b[i]+At[i][j]*y[j];
      }
   }
```

```
/*      calculate the inverse of AtA[][]                                    */

    d=&dval;
    size=3;
    ludcmp(AtA,size,indx,d);
    for(j=1;j<=size;j++)
    {
       for(i=1;i<=size;i++) x[i]=0.0;
       x[j]=1.0;
       lubksb(AtA,size,indx,x);
       for(i=1;i<=size;i++) inv[i][j]=x[i];
    }

/*      multiply inv(AtA[][])*b to get parabolic coefficients               */

    for(j=1;j<=3;j++)
    {
       dval=0.0;
       for(i=1;i<=3;i++)
       {
          dval=dval+b[i]*inv[j][i];
       }
       x[j]=dval;
    }

/*      calculate the dimension using max slope of parabolic spline        */

    tmp2=(double)data[0];
    x2=tmp2;                              /* constrain result to data      */
    pdim=2.0*x[1]*x2 + x[2];              /* calc slope                    */
    y2=x[1]*x2*x2 + x[2]*x2 + x[3];

/*      output results to user                                              */

    _setviewport(1,542,1022,739);         /* set viewport                  */
    _clearscreen(_GVIEWPORT);             /* clear viewport                */

    sprintf(msg,"The Linear Dimension is: %6.4f ",ldim);
    _grtext(5,25,msg);

    sprintf(msg,"The Parabolic Dimension is: %6.4f at ( %6.4f , %6.4f )."
                "  The parabolic polynomial is: ",pdim,x2,y2);
    _grtext(5,60,msg);
    sprintf(msg,"y = %8e x^2 + %8e x + %8e",x[1],x[2],x[3]);
    _grtext(5,85,msg);

    sprintf(msg,"The Cubic Dimension is: %6.4f at ( %6.4f , %6.4f )."
                "  The cubic polynomial is: ",cdim,x1,y1);
    _grtext(5,135,msg);
    sprintf(msg,"x = %8e y^3 + %8e y^2 + %8e y + %8e",a1,a2,a3,a4);
    _grtext(5,160,msg);

/*      ask user what type of data fit to use                               */
```

```
    fit=0;
    while(fit==0)
    {
       DRAW_MESSAGE(2,"(1)st, (2)nd, or (3)rd order fit:");
       GET_MESSAGE(2,msg);
       i=atoi(msg);
       if(i>=1 && i<=3) fit=i;
    }

    if(fit==1) *dimension=ldim;
    else if(fit==2) *dimension=pdim;
    else *dimension=cdim;
    return;
}


/*=========================================================================*/
/*              Matrix.c is compiled from "NUMERICAL RECIPES in C"         */
/*=========================================================================*/

#define TINY 1.0e-40;

void ludcmp(a,n,indx,d)
long n,*indx;
double a[5][5];
double *d;
{
        int i,imax,j,k,sz;
        double big,dum,sum,temp;
        double *vv,*vector();
        void nrerror(),free_vector();

        sz=n+1;
        vv=vector(1,n);
        *d=1.0;
        for (i=1;i<=n;i++) {
                big=0.0;
                for (j=1;j<=n;j++)
                        if ((temp=fabs(a[i][j])) > big) big=temp;
                if (big == 0.0) nrerror("Singular matrix in routine LUDCMP");
                vv[i]=1.0/big;
        }
        for (j=1;j<=n;j++) {
                for (i=1;i<j;i++) {
                        sum=a[i][j];
                        for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                }
                big=0.0;
                for (i=j;i<=n;i++) {
                        sum=a[i][j];
                        for (k=1;k<j;k++)
                                sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
```

```
                        if ( (dum=vv[i]*fabs(sum)) >= big) {
                                big=dum;
                                imax=i;
                        }
                }
                if (j != imax) {
                        for (k=1;k<=n;k++) {
                                dum=a[imax][k];
                                a[imax][k]=a[j][k];
                                a[j][k]=dum;
                        }
                        *d = -(*d);
                        vv[imax]=vv[j];
                }
                indx[j]=imax;
                if (a[j][j] == 0.0) a[j][j]=TINY;
                if (j != n) {
                        dum=1.0/(a[j][j]);
                        for (i=j+1;i<=n;i++) a[i][j] *= dum;
                }
        }
        free_vector(vv,1,n);
}

#undef TINY

void lubksb(a,n,indx,b)
double a[5][5];
double b[];
long n,*indx;
{
        int i,ii=0,ip,j,sz;
        double sum;

        sz=n+1;
        for (i=1;i<=n;i++) {
                ip=indx[i];
                sum=b[ip];
                b[ip]=b[i];
                if (ii)
                        for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
                else if (sum) ii=i;
                b[i]=sum;
        }
        for (i=n;i>=1;i--) {
                sum=b[i];
                for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
                b[i]=sum/a[i][i];
        }
}


void free_vector(double *v,int nl,int nh)
{
```

```
    nh=0;
    free((v+nl));
}

void nrerror(char *error_text)
{
    printf("Numerical Recipes run-time error...\n");
    printf("%s\n",error_text);
    printf("exiting to system.\n");
    exit(1);
}


double *vector(int nl,int nh)
{
    double *v;
    v=(double *) malloc((unsigned) (nh-nl+1)*sizeof(double));
    if(!v) nrerror("\n Memory allocation error in vector()");
    return (v-nl);
}

/*-------------------------------------------------------------------------*/
/*                                MASSCORR                                 */
/*      The Mass Correlation Dimension routine takes the picture and       */
/*      calculates the Mass Correlation Dimension.  The euclidean          */
/*      distance is used to calculate the mass correlation.  The           */
/*      results of the correlation are then used to calculate the          */
/*      dimension.                                                         */
/*-------------------------------------------------------------------------*/

extern char msg1[60];

void MASSCORR(float *dimension,struct PICTURE *PIC)
{
    char *imgptr;
    char st1[60],st2[60],msg[80];
    long imwd,imln,i,j,k,n,val,test,valmax,dx,dy;
    long offset1,radius,rmax,indx[5],fit,size;
    long *pntx,*pnty;
    float data[32],tmp1,tmp2,tmp3,tmp4;
    double slope,tp1,tp2,tp3;
    double *cnt,*cal;
    double At[5][17],A[17][5],y[17],x[5],AtA[5][5],b[5],dval;
    double ldim,cdim,x1,y1,a1,a2,a3,a4,inv[5][5];
    double *d;
    struct SCREEN PLT;

/*      get picture information                                            */

    imgptr=PIC->pic_buf;                /* get picture location           */
    imwd=PIC->imagewidth;
    imln=PIC->imagelength;              /* get picture size               */

/*      calculate counting array size and allocate and clear               */
```

```
    i=(long)sqrt( (double) ( (imwd*imwd)+(imln*imln) ) );
    rmax=max(256,i);
    cnt=(double *)malloc(rmax*sizeof(double));
    if((cal=(double *)malloc(rmax*sizeof(double)))==NULL)
    {                                   /* memory allocation error      */
       DRAW_MESSAGE(2,"memory allocation error");
       return;                          /* return with error flag       */
    }
    for(i=0;i<rmax;i++)
    {
       cnt[i]=0.0;                      /* clear count array            */
       cal[i]=0.0;                      /* clear probability array      */
    }

/*      calculate probability array for discrete lattice                */

    DRAW_MESSAGE(2,"Calculating probability array");
    cal[0]=1.0;
    for(i=1;i<imln;i++)
    {
       for(j=0;j<imwd;j++)
       {
          radius=(double)((long)sqrt((double)(i*i+j*j)));
          cal[radius]+=4;
       }
    }
    for(i=1;i<rmax;i++)                          /* integrate result    */
    {
       cal[i]=cal[i]+cal[i-1];
    }

/*      count all white pixels for progress display                     */

    valmax=0;
    for(i=0;i<imln;i++)
    {
       offset1=i*imwd;
       for(j=0;j<imwd;j++)
       {
          if(imgptr[offset1+j]==255) valmax++;
       }
    }
    cnt[0]=(double)valmax;

/*      allocate position arrays                                        */

    if((pntx=(long *)malloc(valmax*sizeof(long)))==NULL)
    {                                   /* memory allocation error      */
       DRAW_MESSAGE(2,"memory allocation error");
       return;                          /* return with error flag       */
    }
    if((pnty=(long *)malloc(valmax*sizeof(long)))==NULL)
    {                                   /* memory allocation error      */
```

```
        DRAW_MESSAGE(2,"memory allocation error");
        return;                                  /* return with error flag      */
     }

  /*      put all pixels into position arrays                                    */

     DRAW_MESSAGE(2,"Creating pixel location arrays");
     k=0;                                        /* position counter            */
     for(i=0;i<imln;i++)
     {
        offset1=imwd*i;                          /* calculate offset to here    */
        for(j=0;j<imwd;j++)
        {
           if(imgptr[offset1+j]==255)            /* part of fractal ?           */
           {
              pntx[k]=j;
              pnty[k]=i;                         /* put position into arrays    */
              k++;
           }
        }
     }

  /*      perform count of pixel radii                                           */

     val=0;
     test=25;
     for(i=0;i<valmax-1;i++)
     {
        for(j=i+1;j<valmax;j++)
        {
           dx=pntx[i]-pntx[j];
           dy=pnty[i]-pnty[j];
           radius=(long)sqrt((double)(dx*dx+dy*dy));
           cnt[radius]+=2;                       /* count radius                */
        }
        val++;
        if(val>=test)
        {
           test=test+25;
           sprintf(st1,"%d of %d pixels processed",val,valmax);
           DRAW_MESSAGE(2,st1);
        }
     }

  /*      normalize data to number of pixels                                     */

     for(i=1;i<rmax;i++)
     {
        cnt[i]=cnt[i]/cnt[0];
     }
     cnt[0]=1.0;

  /*      perform integration of normalized pixel radii                          */
```

```
    for(i=1;i<rmax;i++)
    {
       cnt[i]=cnt[i]+cnt[i-1];
    }

/*      perform selection of data into plot array                          */
/*      this uses the same data range as for FULL DILATION !!              */

    data[0]=(float)log(1.0);      data[1]=(float)log(cnt[0]/cal[0]);
    data[2]=(float)log(2.0);      data[3]=(float)log(cnt[1]/cal[1]);
    data[4]=(float)log(3.0);      data[5]=(float)log(cnt[2]/cal[2]);
    data[6]=(float)log(4.0);      data[7]=(float)log(cnt[3]/cal[3]);
    data[8]=(float)log(6.0);      data[9]=(float)log(cnt[5]/cal[5]);
    data[10]=(float)log(8.0);     data[11]=(float)log(cnt[7]/cal[7]);
    data[12]=(float)log(12.0);    data[13]=(float)log(cnt[11]/cal[11]);
    data[14]=(float)log(16.0);    data[15]=(float)log(cnt[15]/cal[15]);
    data[16]=(float)log(24.0);    data[17]=(float)log(cnt[23]/cal[23]);
    data[18]=(float)log(32.0);    data[19]=(float)log(cnt[31]/cal[31]);
    data[20]=(float)log(48.0);    data[21]=(float)log(cnt[47]/cal[47]);
    data[22]=(float)log(64.0);    data[23]=(float)log(cnt[63]/cal[63]);
    data[24]=(float)log(96.0);    data[25]=(float)log(cnt[95]/cal[95]);
    data[26]=(float)log(128.0);   data[27]=(float)log(cnt[127]/cal[127]);
    data[28]=(float)log(192.0);   data[29]=(float)log(cnt[191]/cal[191]);
    data[30]=(float)log(256.0);   data[31]=(float)log(cnt[255]/cal[255]);

/*      calculate gyration radius                                          */

/*      Plot results to right half of screen for display                   */

    _setviewport(513,1,1022,510);              /* set viewport           */
    _clearscreen(_GVIEWPORT);                  /* clear screen           */
    tmp1=(0.0);
    tmp3=(float)(((long)(data[30]*2.0)+1.0)/2);
    tmp4=(1.0);
    tmp2= 1.0-(tmp3);
    strcpy(st1,"CORRELATION PLOT");
    strcpy(st2,"Ln P(radius) vs Ln radius");
    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,1,&PLT);
    DATASET(32,data,CYAN,&PLT);
    TITLES(st1,st2,&PLT);
    DRAWBORDER(&PLT);
    DRAWDATA(&PLT);

    SCREENSET(525,15,480,480,tmp1,tmp2,tmp3,tmp4,0,&PLT);
    DATASET(32,data,BRIGHTWHITE,&PLT);
    DRAWDATA(&PLT);

/*                calculate linear dimension from numbers                 */
/*                          by linear regression                          */

    n=10;
    slope=LINREG(n,&data[6]);
    ldim=2.0+slope;              /* calculate dimension from slope        */
```

```
/*      calculate dimension using a least squares cubic fit              */
/*         double At[5][17],A[17][5],y[17],x[5],AtA[5][5],b[5];           */
/*         data[odd]=y-value                                             */
/*         data[even]=x-value                                            */
/*                                                                       */
/*      put data into arrays                                             */

   for(i=1;i<=16;i++)          /* put data into arrays                   */
   {
      A[i][1]=(double)(data[(i-1)*2]*data[(i-1)*2]*data[(i-1)*2]);
      A[i][2]=(double)(data[(i-1)*2]*data[(i-1)*2]);
      A[i][3]=(double)(data[(i-1)*2]);
      A[i][4]=1.0;
      At[1][i]=A[i][1];
      At[2][i]=A[i][2];
      At[3][i]=A[i][3];
      At[4][i]=A[i][4];
      y[i]=(double)data[(i*2)-1];
   }

/*      multiply At[][] by A[][]                                         */

   for(i=1;i<=4;i++)
   {
      for(j=1;j<=4;j++)
      {
         AtA[i][j]=0.0;
         for(k=1;k<=16;k++)
         {
            AtA[i][j]=AtA[i][j]+At[i][k]*A[k][j];
         }
      }
   }

/*      calculate b=At*y                                                 */

   for(i=1;i<=4;i++)
   {
      b[i]=0.0;
      for(j=1;j<=16;j++)
      {
         b[i]=b[i]+At[i][j]*y[j];
      }
   }

/*      calculate the inverse of AtA[][]                                 */

   d=&dval;
   size=4;
   ludcmp(AtA,size,indx,d);
   for(j=1;j<=size;j++)
   {
      for(i=1;i<=size;i++) x[i]=0.0;
      x[j]=1.0;
```

```
        lubksb(AtA,size,indx,x);
        for(i=1;i<=size;i++) inv[i][j]=x[i];
    }

/*      multiply inv(AtA[][])*b to get cubic coefficients                */

    for(j=1;j<=4;j++)
    {
       dval=0.0;
       for(i=1;i<=4;i++)
       {
          dval=dval+b[i]*inv[j][i];
       }
       x[j]=dval;
    }

/*      calculate the dimension using min slope of cubic spline          */

    tp1=(-2.0*x[2])/(6.0*x[1]);          /* calculate location of min    */
    tp2=(double)data[0];
    tp3=(double)data[30];
    x1=max(tp2,min(tp3,tp1));            /* constrain result to data     */
    tp2=3.0*x[1]*x1*x1 + 2.0*x[2]*x1 + x[3];  /* calc slope              */
    cdim=2.0+tp2;
    y1=x[1]*x1*x1*x1 + x[2]*x1*x1 + x[3]*x1 + x[4];
    a1=x[1]; a2=x[2]; a3=x[3]; a4=x[4];  /* save variables               */

/*      output results to user                                           */

    _setviewport(1,542,1022,739);        /* set viewport                 */
    _clearscreen(_GVIEWPORT);            /* clear viewport               */

    sprintf(msg,"The Linear Dimension (3-96 pixel radius) is: %6.4f ",ldim);
    _grtext(5,25,msg);

    sprintf(msg,"The Cubic Dimension (all 16 points) is: %6.4f at ( %6.4f ,
%6.4f )."
                " Polynomial is:",cdim,x1,y1);
    _grtext(5,135,msg);
    sprintf(msg,"y = %8e x^3 + %8e x^2 + %8e x + %8e",a1,a2,a3,a4);
    _grtext(5,160,msg);

/*      ask user what type of data fit to use                            */

    fit=0;
    while(fit==0)
    {
       DRAW_MESSAGE(2,"(1)st, or (3)rd order fit:");
       GET_MESSAGE(2,msg);
       i=atoi(msg);
       if(i==1 || i==3) fit=i;
    }

    if(fit==1) *dimension=ldim;
```

```
    else *dimension=cdim;

    free(cnt);
    free(cal);
    free(pntx);
    free(pnty);
    return;
}

/*=========================================================================*/
/*                                GEN3D                                    */
/*     This routine generates a file with a 3-Dimensional set of           */
/*     points created from the fractal image in memory. This data           */
/*     is to be used as calibration sources for external programs.          */
/*=========================================================================*/

void GEN3D(void);

void GEN3D(void)
{
   FILE *out;                    /* declare out as the output file      */
   char *imgptr;
   char output[80];
   long imwd,imln,miny;
   long i,j,k,d,num,offset;
   long *x,*y;


/*                        get picture data                                 */

   imgptr=PIC.pic_buf;                  /* locate picture                 */
   imwd=PIC.imagewidth;                 /* get width                      */
   imln=PIC.imagelength;                /* get length                     */

/*                        get file name from user                          */

   DRAW_MESSAGE(2,"Enter file name for output :");
   d=GET_MESSAGE(2,output);
   if(d==27) return;

/*                        open file for writing                            */

   if ((out = fopen(output, "wt")) == NULL)      /* open file as text    */
   {
         return;
   }

/*                        count number of white pixels                     */

   num=0;
   for(i=0;i<imln;i++)
   {
      offset=i*imwd;
      for(j=0;j<imwd;j++)
```

```
        {
           if(imgptr[offset+j]==255) num++;
        }
     }

  /*                        allocate arrays for data points                  */

     x=(long *)malloc(num*sizeof(long));
     if((y=(long *)malloc(num*sizeof(long)))==NULL)
     {                                        /* memory allocation error     */
        DRAW_MESSAGE(2,"memory allocation error");
        return;                               /* return with error flag      */
     }

  /*                        put all points into the arrays                   */

     k=0;
     miny=0;
     for(i=0;i<imln;i++)
     {
        offset=i*imwd;
        for(j=0;j<imwd;j++)
        {
           if(imgptr[offset+j]==255)
           {
              x[k]=j;
              y[k]=i;                         /* put coordinates into arrays */
              miny=max(i,miny);               /* find lowest part of curve   */
              k++;
           }
        }
     }

  /*                        write 3D construction to file                    */

     fprintf(out,"%d %d\n",imwd,imln);

     for(i=0;i<num;i++)                       /* x constructor loop          */
     {
        for(j=0;j<num;j++)                    /* y constructor loop          */
        {
           fprintf(out,"%d %d %d\n",
                   x[i],x[j],miny-y[j]+miny-y[i]);
        }
     }

  /*                        free arrays and close file                       */

     fclose(out);
     free(x);
     free(y);
     return;
  }
```

C5 FILE: PIC_DRAW.C

```
/*                    Include files and prototypes                          */

#include <graph.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <stddef.h>
#include <dos.h>
#include <string.h>
#include <math.h>
#include <conio.h>
#include "picture.h"
#include "screen.h"

extern struct PICTURE PIC;
extern char *imgptr;

/*                Function Prototypes                                       */

int HIGH_PASS(void);
void PIC_DRAW(long position,struct PICTURE *PICT);
float LINREG(int n,float *array);
void DRAW_BORDER();
void DRAW_MESSAGE(int msgpos,char *msg);
short DRAW_MENU(int menu);
int GET_MESSAGE(int position,char *buffer);
void PLOT_HISTOGRAM(void);
void NEGATE_IMAGE(void);
void AUTO_SCALE(void);
int LOW_PASS(void);
extern void DATASET(int n, float *array, int m, struct SCREEN *SCRN);
extern void TITLES(char *str1, char *str2,struct SCREEN *SCRN);
extern void DRAWBORDER(struct SCREEN *SCRN);
extern void DRAWDATA(struct SCREEN *SCRN);
extern void SCREENSET(int xst,int yst,int xsp,int ysp,float xmn,float ymn,
              float xmx,float ymx,int typ,struct SCREEN *SCRN);


/*-------------------------------------------------------------------------*/
/*                                PIC_DRAW.                                */
/*      This routine takes the pointer given and draws the picture         */
/*      onto the screen.  The screen is assumed to be initialized          */
/*      by previous routines.  The plot is also reduced to 64 levels!      */
/*      The data values are reduced from 0-255 to 192-255 for display      */
/*      using the last 64 values in the palette.                           */
/*         The routine will draw the picture at 510x510 resolution in      */
/*      the half window specified in the call.  Note that this assumes     */
/*      the borders drawn by the routine DRAW_BORDER is in place.          */
/*                                                                         */
/*      Calling convention is:                                             */
/*         PIC_DRAW(pos,picture);              where                       */
/*              pos     = 0 for left side, 1 for right side               */
```

```
/*              picture = pointer to PICTURE structure                  */
/*--------------------------------------------------------------------*/
/*                        Start of routine                            */

void PIC_DRAW(long pos,struct PICTURE *PICT) {

   char c;
   long i,j;
   long k,xoffset,yoffset,a,b,tmp;
   long poffset,boffset,m,ypos,xpos;
   char *n;
   char *pic_addr;                          /* address of data          */
   _setviewport(0,0,1023,767);              /* set viewport to max      */

   if (pos==0) xoffset=1;                   /* which side is picture?   */
   else xoffset=513;
   yoffset=1;
   _setviewport(xoffset,yoffset,xoffset+509,yoffset+509); /* set viewport */

   m=PICT->imagewidth;
   n=PICT->pic_buf;
   k=0;
   imgptr[0]=254;
   imgptr[1]=1;
   imgptr[2]=254;
   imgptr[3]=1;
   imgptr[4]=8;
   imgptr[5]=0;                             /* init array by hand       */
   a=PICT->imagewidth;
   b=PICT->imagelength;

/*              Picture is smaller than display buffer                  */

   if(a<510 || b<510) {

      for (i=0;i<510;i++) {                 /* start of y loop          */
         ypos=(i*b)/510;                    /* calc y position          */
         tmp=ypos*a;
         for (j=0;j<510;j++) {              /* start of x loop          */
            xpos=(j*a)/510;                 /* calc x position          */
            poffset=tmp+xpos;               /* calc offset into pic array */
            pic_addr=n+poffset;             /* calculate address of pixel */
            imgptr[k+6]=((*pic_addr)>>2|192); /* transfer pixel         */
            k++;
         }
      }

   }

/*                  Picture is larger than buffer                       */

   else {

      memset(imgptr+6,0,260100);            /* clear buffer memory      */
```

```
      for (i=0;i<b;i++) {                    /* start of y loop          */
         tmp=i*a;
         ypos=(i*510)/b;                     /* calc y position          */
         for (j=0;j<a;j++) {                 /* start of x loop          */
            xpos=(j*510)/a;                  /* calc x position          */
            boffset=ypos*510+xpos+6;         /* buffer offset            */
            poffset=tmp+j;                   /* picture offset           */
            pic_addr=n+poffset;              /* calculate address of pixel */
            c=((*pic_addr)>>2|192);          /* calc new pixel           */
            imgptr[boffset]=max(imgptr[boffset],c); /* transfer pixel   */
         }
      }

   }
   _putimage(0,0,imgptr,_GPSET);
}


/*-------------------------------------------------------------------------*/
/*                              DRAW_BORDER                                */
/*      This routine initializes the SVGA monitor to 1024x768 pixels       */
/*      and draws the default border.  It also sets the greyscale          */
/*      values in the palette.   No arguments are required in calling.     */
/*-------------------------------------------------------------------------*/

#include <graph.h>

void DRAW_BORDER() {

/*                              SVGAINIT                                   */
/*      This routine initializes the screen to 1024x768x256 SVGA           */
/*      No checks are done to ensure that this can be done                 */


   short int mode;
   char parray[4];                       /* length of long int           */
   long *ptr=(long*)parray;
   short int i;
   long color;

   mode=_XRES256COLOR;
   _setvideomode(mode);

/*                              GREY_PALETTE                               */
/*      This routine calculates and then sets the palette to allow         */
/*      greyscale images to be displayed as such.                          */
/*      The standard SVGA card allows 256 colors selected from a           */
/*      palette of 256k colors.  Each primary color has a range of         */
/*      0 to 63.  This means that only 64 grey scale values can be         */
/*      created.  This program calculates the new palette and sets         */
/*      the SVGA card accordingly.  The last 64 values in the palette      */
/*      are reset to the available grey scales. "192" is black and "255"   */
/*      is white.  Palette values from 128 to 191 are all set to the       */
/*      color green so that drawing a selection box on the screen is       */
```

```
/*      easier.  The other palette values from 0 to 127 are not         */
/*      changed by this program and are free for use by other sub-      */
/*      routines.                                                       */

   parray[3]=0;
   for (i=0;i<64;i++) {
      parray[2]=i;                        /* blue component             */
      parray[1]=i;                        /* green component            */
      parray[0]=i;                        /* red component              */
      _remappalette(i+192,*ptr);          /* remap last 64 colors to grey */
   }

   parray[2]=0;                           /* blue component             */
   parray[1]=63;                          /* green component            */
   parray[0]=0;                           /* red component              */
   for (i=0;i<64;i++) {
      _remappalette(i+128,*ptr);          /* remap 128 to 191 to green  */
   }
   color=_LIGHTRED;
   _remappalette(LIGHTRED,color);
   color=_GREEN;
   _remappalette(GREEN,color);
   color=_YELLOW;
   _remappalette(YELLOW,color);
   color=_CYAN;
   _remappalette(CYAN,color);
   color=_LIGHTGREEN;
   _remappalette(LIGHTGREEN,color);
   color=_BRIGHTWHITE;
   _remappalette(BRIGHTWHITE,color);

/*                              DRAW BORDERS                            */
/*      This routine draws lines on the screen to seperate input and    */
/*      output picture windows.                                         */

   _setcolor(9);                          /* set color to light blue    */
   _moveto(0,0);
   _lineto(1023,0);
   _lineto(1023,767);
   _lineto(0,767);
   _lineto(0,0);
   _moveto(0,511);
   _lineto(1023,511);
   _moveto(512,0);
   _lineto(512,540);
   _moveto(511,0);
   _lineto(511,540);
   _moveto(0,541);
   _lineto(1023,541);
   _moveto(0,740);
   _lineto(1023,740);

   return;
}
```

```
/*-------------------------------------------------------------------------*/
/*                                DRAW_MESSAGE                             */
/*      This routine puts a text string onto the screen into one of        */
/*      several boxes defined in the pictures.  These boxes are to         */
/*      inform the user what happening within the system.                  */
/*                                                                         */
/*      The format of the call is:                                         */
/*                                                                         */
/*              DRAW_MESSAGE(msgpos,message);                              */
/*      Where                  msgpos=0 for left picture message           */
/*                                     1 for right picture message         */
/*                                     2 for bottom status message         */
/*                             message=null terminated message string      */
/*-------------------------------------------------------------------------*/

void DRAW_MESSAGE(int msgpos,char *msg)
{
   _setcolor(7);
   _setcharsize(26,13);

   switch (msgpos) {

      case 0 :                                /* left picture caption      */
         _setviewport(1,513,509,540);         /* set viewport              */
         _clearscreen(_GVIEWPORT);            /* clear viewport            */
         _grtext(2,2,msg);                    /* output message            */
         break;
      case 1 :                                /* right picture caption     */
         _setviewport(513,513,1022,540);      /* set viewport              */
         _clearscreen(_GVIEWPORT);            /* clear viewport            */
         _grtext(2,2,msg);
         break;
      case 2 :                                /* bottom status line        */
         _setviewport(1,741,1022,766);        /* set viewport              */
         _clearscreen(_GVIEWPORT);            /* clear viewport            */
         _grtext(2,2,msg);
         break;
   }

   return;
}

/*-------------------------------------------------------------------------*/
/*                                DRAW_MENU                                */
/*      This routine takes care of drawing the selected menu onto the      */
/*      screen and waits for the user to press a selection key.  The       */
/*      routine then returns to the calling routine with the key code      */
/*      that was pressed.                                                  */
/*                                                                         */
/*      The format of the call is:                                         */
/*                                                                         */
/*                      select=DRAW_MENU(menu);                            */
/*              Where   select = item selection number                     */
```

```
/*                          menu    = menu to be outputted to screen          */
/*--------------------------------------------------------------------------*/

short DRAW_MENU(int menu)
{
   static oldmenu=99;
   int i,j,k;
   char s[2],t[25];

   _setcharsize(26,13);
   strcpy(t,"Enter Selection: ");
   i=5;
   j=31;
   _setcolor(7);
   _setviewport(1,542,1022,739);  /* set viewport                  */
   if (oldmenu!=menu)
   {
      _clearscreen(_GVIEWPORT);   /* clear viewport                */
   }
   oldmenu=menu;
   switch (menu)
   {

      case 0 :
         _grtext(i,i,         "1) File input/output selection.");
         _grtext(i,i+j,       "2) Draw memory into left frame.");
         _grtext(i,i+2*j,     "3) Draw memory into right frame.");
         _grtext(i,i+3*j,     "4) L - Systems fractal selection.");
         _grtext(i,i+4*j,     "5) Dimension calculations.");
         _grtext(i+512,i,     "6) Picture Enhancement selection.");
         _grtext(i+512,i+j,   "7) Edge Detection selection.");
         _grtext(i+512,i+2*j,"8) SAVE current image into memory.");
         _grtext(i+512,i+3*j,"9) RECALL saved image from memory.");
         _grtext(i+512,i+4*j,"0) Simulation selection.");
         _grtext(i+512,i+5*j,"ESC) Quit Program.");

         DRAW_MESSAGE(2,t);
         while(kbhit()) k=getch();              /* drain keyboard buf   */
         while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
         if(k==27) return(999);                 /* esc = end of program */
         s[0]=k;
         s[1]='\0';
         strcat(t,s);
         DRAW_MESSAGE(2,t);
         k=k-48;                                /* convert to number    */
         return (k);
         break;

      case 1 :
         _grtext(i,i,         "1) 1024x1024 Koch curve 1, D=1.1042");
         _grtext(i,i+j,       "2) 1024x1024 The Koch curve, D=1.2619");
         _grtext(i,i+2*j,     "3) 1024x1024 Koch curve 3, D=1.3403");
         _grtext(i,i+3*j,     "4) 1024x1024 Koch curve 4, D=1.5000");
         _grtext(i,i+4*j,     "5) 1024x1024 Tree 1 curve, D=1.4650");
```

```
    _grtext(i+512,i,     "6) 1024x1024 Tree 2 curve, D=1.4650");
    _grtext(i+512,i+j,   "7) 1024x1024 Tree 3 curve, D=1.5282");
    _grtext(i+512,i+2*j,"8) 1024x1024 Tree 4 curve, D=1.5736");
    _grtext(i+512,i+3*j,"9) 1024x1024 Sier. Arrowhead, D=1.5850");
    _grtext(i+512,i+4*j,"0) 1024x1024 Koch curve, D?=1.4650");
    _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

    DRAW_MESSAGE(2,t);
    while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
    if(k==27) return(998);                  /* esc = end of submenu */
    s[0]=k;
    s[1]='\0';
    strcat(t,s);
    DRAW_MESSAGE(2,t);
    k=k-38;                  /* convert to number, range 10-19 */
    return (k);
    break;

 case 2 :
    _grtext(i,i,         "1) Negate picture image.");
    _grtext(i,i+j,       "2) Display grayscale histogram.");
    _grtext(i,i+2*j,     "3) Auto grayscale shift and scale.");
    _grtext(i,i+3*j,     "4) 9 Pt Low pass filter.");
    _grtext(i,i+4*j,     "5) 5 Pt High pass filter.");
    _grtext(i+512,i,     "6) Zero level cutoff.");
    _grtext(i+512,i+j,   "7) Max level cutoff.");
    _grtext(i+512,i+2*j,"8) 9 Pt Median filter.");
    _grtext(i+512,i+3*j,"9) Subimage selection.");
    _grtext(i+512,i+4*j,"0) Draw memory into right frame.");
    _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

    DRAW_MESSAGE(2,t);
    while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
    if(k==27) return(998);                  /* esc = end of submenu */
    s[0]=k;
    s[1]='\0';
    strcat(t,s);
    DRAW_MESSAGE(2,t);
    k=k-28;                  /* convert to number, range 20-29 */
    return (k);
    break;

case 3 :
    _grtext(i,i,         "1) 5 Pt Laplacian Edge Detection.");
    _grtext(i,i+j,       "2) 9 Pt Exponential Edge Detection.");
    _grtext(i,i+2*j,     "3) 13 Pt Exponential Edge Detection.");
    _grtext(i,i+3*j,     "4) 25 Pt Exponential Edge Detection.");
    _grtext(i+512,i+j,   "5) 9 Pt Gaussian Edge Detection.");
    _grtext(i+512,i+2*j,"6) 13 Pt Gaussian Edge Detection.");
    _grtext(i+512,i+3*j,"7) 25 Pt Gaussian Edge Detection.");
    _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

    DRAW_MESSAGE(2,t);
    while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
```

```
        if(k==27) return(998);                  /* esc = end of submenu */
        s[0]=k;
        s[1]='\0';
        strcat(t,s);
        DRAW_MESSAGE(2,t);
        k=k-18;                  /* convert to number, range 30-39 */
        return (k);
        break;

      case 4 :
        _grtext(i,i,          "1) Box Dimension calculation.");
        _grtext(i,i+j,        "2) Dilation Dimension calc.");
        _grtext(i,i+2*j,      "3) Better Box Dimension calc.");
        _grtext(i,i+3*j,      "4) Full Dilation calculation.");
        _grtext(i,i+4*j,      "5) Correlation Dimension.");
        _grtext(i+512,i,      "6) 2,3,4 Image Reduction.");
        _grtext(i+512,i+1*j,"7) Generate 3D file from image.");
        _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

        DRAW_MESSAGE(2,t);
        while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
        if(k==27) return(998);                  /* esc = end of submenu */
        s[0]=k;
        s[1]='\0';
        strcat(t,s);
        DRAW_MESSAGE(2,t);
        k=k-8;                   /* convert to number, range 40-49 */
        return (k);
        break;

      case 5 :
        _grtext(i,i,          "Use arrow keys to position box.");
        _grtext(i,i+j,        "Use + or - to change box size, F or S for
speed.");
        _grtext(i,i+2*j,      "Press x or X  and y or Y to change X and Y
size.");
        _grtext(i,i+3*j,      "Press m or M to magnify selection.");
        _grtext(i,i+4*j,      "Press u or U to select subimage for use.");
        _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

        DRAW_MESSAGE(2,t);
        return(0);
        break;

      case 6 :
        _grtext(i,i,          "1) Read TIF file.");
        _grtext(i,i+j,        "2) Save image in memory to file.");
        _grtext(i,i+2*j,      "3) Save current screen to file.");
        _grtext(i,i+3*j,      "4) Set a label into file (Max 39 chars).");
        _grtext(i,i+4*j,      "5) Read label (then key to continue).");
        _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

        DRAW_MESSAGE(2,t);
        while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
```

```
         if(k==27) return(998);                    /* esc = end of submenu */
         s[0]=k;
         s[1]='\0';
         strcat(t,s);
         DRAW_MESSAGE(2,t);
         k=k+12;                  /* convert to number, range 60-69       */
         return (k);
         break;

      case 7 :
         _grtext(i,i,          "1) Parallel Tree Simulation, G.S.");
         _grtext(i,i+j,        "2) Parallel Tree Simulation, J.");
         _grtext(i,i+2*j,      "3) Circular Tree Simulation, G.S.");
         _grtext(i,i+3*j,      "4) Circular Tree Sim 2, G.S.");
         _grtext(i,i+4*j,      "5) Animate image in memory.");
         _grtext(i+512,i,      "6) Calc. circular field for fractal.");
         _grtext(i+512,i+j,    "7) New Model Simulation.");
         _grtext(i+512,i+2*j,"8) Parallel Field convergence.");
         _grtext(i+512,i+3*j,"9) Circular Field convergence.");
         _grtext(i+512,i+4*j,"0) Set iterations for simulations.");
         _grtext(i+512,i+5*j,"ESC) Quit sub menu.");

         DRAW_MESSAGE(2,t);
         while(((k=getch())>'9' || k<'0') && k!=27) {}; /* wait for key */
         if(k==27) return(998);                    /* esc = end of submenu */
         s[0]=k;
         s[1]='\0';
         strcat(t,s);
         DRAW_MESSAGE(2,t);
         k=k+22;                  /* convert to number, range 70-79       */
         return (k);
         break;
   }
}

/*-------------------------------------------------------------------------*/
/*                              GET_MESSAGE                                */
/*      This routine gets an input string from the user and puts it        */
/*      into the buffer specified.  The input is also echoed for the       */
/*      user using the graphics text currently being used.                 */
/*                                                                         */
/*      The format of the call is                                          */
/*                                                                         */
/*            result=GET_MESSAGE(position,buffer);                         */
/*      where                                                              */
/*             result   = integer last key pressed                         */
/*             position = position of string echo                          */
/*             buffer   = output string array                              */
/*-------------------------------------------------------------------------*/

int GET_MESSAGE(int position,char *buffer)
{
   int i,c;
   i=0;
```

```
      buffer[0]='\0';
      switch(position) {
         case 2 :
            while((c=getch())!=13 && c!=27) {
               _setviewport(450,741,1022,766);   /* set viewport        */
               _clearscreen(_GVIEWPORT);      /* clear viewport         */
               if(c!=8 && c!=127 && c!=13 && c!=13 && c!=10) {
                  buffer[i]=c;                /* store character        */
                  buffer[i+1]='\0';
                  i++;                        /* count characters       */
                  strupr(buffer);
                  _grtext(2,2,buffer);
               }
               else if((c==8 || c==127 || i>40) && i>0) {
                  i--;
                  buffer[i]='\0';
                  _grtext(2,2,buffer);
               }
               else if (c==13 || c==10) break;
            }
            break;
      }
      return(c);
   }

   /*-------------------------------------------------------------------*/
   /*                          NEGATE_IMAGE                             */
   /*      This routine creates a negative image of the picture in      */
   /*      memory.  This allows neagative images to be used by the      */
   /*      same routines as other images.                               */
   /*-------------------------------------------------------------------*/

   void NEGATE_IMAGE(void)
   {
      char *picptr;
      long i,j;

      picptr=PIC.pic_buf;                 /* get picture location       */
      i=PIC.imagewidth*PIC.imagelength;   /* calculate size of picture  */
      for(j=0;j<i;j++)
      {
         picptr[j]=255-picptr[j];         /* negate each pixel          */
      }
   }


   /*-------------------------------------------------------------------*/
   /*                          PLOT_HISTOGRAM                           */
   /*      This routine creates a histogram of the current image in     */
   /*      memory.  The user can then see the intensity content of the  */
   /*      image and can select the appropriate routines to run on the  */
   /*      image.  The routine counts the number of pixels at each      */
   /*      intensity level and then plots the result into the right hand*/
   /*      part of the display screen.  No data is saved.               */
```

```
/*-------------------------------------------------------------------------*/

void PLOT_HISTOGRAM(void)
{
   long i,k,xoffset;
   float gray[512];
   double r,t;
   char *picptr;
   char st1[60],st2[60];
   struct SCREEN PLT;                    /* structure for screen        */


   for(i=0;i<512;i+=2)
   {
      gray[i]=(float)(i/2);              /* set x value                 */
      gray[i+1]=0;                       /* clear counter array         */
   }
   k=PIC.imagewidth*PIC.imagelength;
   picptr=PIC.pic_buf;
   for(i=0;i<k;i++)
   {
      gray[(int)(picptr[i])*2+1]+=1.0;   /* count pixels in each shade  */
   }
   t=0.0;
   for(i=1;i<512;i+=2)
   {
       t=max(gray[i],t);                 /* find maximum value in array */
   }
   r=(float)((int)(log10(t))+1);         /* convert to log value        */

   for(i=1;i<512;i+=2)
   {
      if(gray[i]>0.0)
      {
         gray[i]=(float)log10((double)gray[i]);
      }
   }
   xoffset=513;
   _setviewport(xoffset,1,xoffset+509,510);     /* set viewport         */
   _clearscreen(_GVIEWPORT);                    /* clear screen         */


   strcpy(st1,"INTENSITY HISTOGRAM");
   strcpy(st2,"Log number of pixels vs Intensity");
   SCREENSET(525,15,480,480,0.0,0.0,255.0,r,1,&PLT);
   DATASET(512,gray,YELLOW,&PLT);
   TITLES(st1,st2,&PLT);
   DRAWBORDER(&PLT);
   DRAWDATA(&PLT);

}


/*-------------------------------------------------------------------------*/
```

```
/*                                AUTO_SCALE                                */
/*      This routine creates a scales the image in memory to fit            */
/*      the available range of brightness values.  A histogram of the       */
/*      image is calculated and the upper and lower bounds of the           */
/*      image is stretched to cover the 255 gray scales available.          */
/*      This routine is mainly for enhancing the image that the user        */
/*      sees so that better decisions can be made for further steps.        */
/*--------------------------------------------------------------------------*/

void AUTO_SCALE(void)
{
   long i,k,gray[256],min,max,delta;
   char *picptr;

   for(i=0;i<256;i++)
   {
      gray[i]=0;                          /* clear counter arrary          */
   }
   k=PIC.imagewidth*PIC.imagelength;
   picptr=PIC.pic_buf;
   for(i=0;i<k;i++)
   {
      gray[(int)(picptr[i])]++;
   }
   min=0;
   while(gray[min]==0) min++;              /* find minimum gray value       */
   max=255;
   while(gray[max]==0) max--;              /* find maximum gray value       */
   delta=max-min;
   for(i=0;i<k;i++)
   {
     picptr[i]=(int)(picptr[i]-min)*255/delta; /* calculate new pixel value */
   }
}

/*--------------------------------------------------------------------------*/
/*                                 LOW_PASS                                 */
/*      This routine runs the picture though a low pass filter to           */
/*      remove some of the noise that may be present in the picture.        */
/*      The routine uses a 3x3 pixel mask and averages the pixel in         */
/*      the middle with the surrounding 8 pixels.                           */
/*      The pixel weight is :                                               */
/*                            1/9 1/9 1/9                                   */
/*                            1/9 1/9 1/9                                   */
/*                            1/9 1/9 1/9                                   */
/*--------------------------------------------------------------------------*/

int LOW_PASS(void)
{
   char *picptr, *picptr_tmp;
   long i,j,k,a,b,c,d,e,f,g,h,x,y;

   picptr=PIC.pic_buf;                    /* get picture location          */
   x=PIC.imagewidth;                      /* get size of picture           */
```

```
   y=PIC.imagelength;                    /* get size of picture          */

   k=x*y;                                /* size of new image            */
   if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
   {
      return(0);                         /* return with error flag       */
   }

/*                      calculate pixel references                        */
/*      Each pixel reference will use the offsets calculated here.        */
/*      The pixel base is first set and all offsets from the base         */
/*      use one of the following numbers depending upon the position      */
/*      of the pixel required. ie:                                        */
/*                                        a      b     c                  */
/*                                        d    base    e                  */
/*                                        f      g     h                  */
/*                                                                        */
   a=-x-1;
   b=-x;
   c=-x+1;
   d=-1;
   e=1;
   f=x-1;
   g=x;
   h=x+1;

/*                       calculate corners of image                       */

   k=0;
   picptr_tmp[k]=(int)(picptr[k]+picptr[k+e]+          /* first corner  */
                       picptr[k+g]+picptr[k+h])/4;
   k=x-1;
   picptr_tmp[k]=(int)(picptr[k+d]+picptr[k]+          /* second corner */
                       picptr[k+f]+picptr[k+g])/4;
   k=x*(y-1);
   picptr_tmp[k]=(int)(picptr[k+b]+picptr[k+c]+        /* third corner  */
                       picptr[k]+picptr[k+e])/4;
   k=x*y-1;
   picptr_tmp[k]=(int)(picptr[k+a]+picptr[k+b]+        /* fourth corner */
                         picptr[k+d]+picptr[k])/4;

/*                       calculate boundary of picture                    */

   i=x*(y-1);
   for(k=1;k<(x-1);k++)
   {
      picptr_tmp[k]=(int)(picptr[k+d]+picptr[k]+picptr[k+e]+    /* top  */
                          picptr[k+f]+picptr[k+g]+picptr[k+h])/6;
      j=i+k;
      picptr_tmp[j]=(int)(picptr[j+a]+picptr[j+b]+picptr[j+c]+  /* bot  */
                          picptr[j+d]+picptr[j]+picptr[j+e])/6;
   }
```

```
    for(k=1;k<(y-1);k++)
    {
       i=k*x;
       picptr_tmp[i]=(int)(picptr[i+b]+picptr[i+c]+picptr[i]+   /* left  */
                           picptr[i+e]+picptr[i+g]+picptr[i+h])/6;
       i=(k+1)*x-1;
       picptr_tmp[i]=(int)(picptr[i+a]+picptr[i+b]+picptr[i]+   /* right */
                           picptr[i+d]+picptr[i+f]+picptr[i+g])/6;
    }

 /*                        calculate interior of picture                    */

    for(j=1;j<y-1;j++)
    {
       for(i=1;i<x-1;i++)
       {
          k=j*x+i;
          picptr_tmp[k]=(int)(picptr[k+a]+picptr[k+b]+picptr[k+c]+
                              picptr[k+d]+picptr[k]+picptr[k+e]+
                              picptr[k+f]+picptr[k+g]+picptr[k+h])/9;
       }
    }

    i=x*y;
    memmove(picptr,picptr_tmp,i);          /* copy result back into buffer */
    free(picptr_tmp);                      /* free working array           */

 }

 /*-------------------------------------------------------------------------*/
 /*                                HIGH_PASS                                */
 /*      This routine runs the picture though a high pass filter to         */
 /*      enhance the higher spectral frequencies in the picture.            */
 /*-------------------------------------------------------------------------*/
 int HIGH_PASS(void)
 {
    char *picptr, *picptr_tmp;
    long i,j,k,a,b,c,d,e,f,g,h,x,y,tmp;

    picptr=PIC.pic_buf;                    /* get picture location         */
    x=PIC.imagewidth;                      /* get size of picture          */
    y=PIC.imagelength;                     /* get size of picture          */

    k=x*y;                                 /* size of new image            */
    if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
 error      */
    {
       return(0);                          /* return with error flag       */
    }

 /*                        calculate pixel references                       */
 /*      Each pixel reference will use the offsets calculated here.         */
 /*      The pixel base is first set and all offsets from the base          */
 /*      use one of the following numbers depending upon the position       */
```

```
/*      of the pixel required. ie:                                      */
/*                                                                      */
/*                                        a      b      c              */
/*                                        d    base     e              */
/*                                        f      g      h              */
/*                                                                      */
   a=-x-1;
   b=-x;
   c=-x+1;
   d=-1;
   e=1;
   f=x-1;
   g=x;
   h=x+1;

/*                        calculate corners of image                    */

   k=0;
   picptr_tmp[k]=(int)(picptr[k]*4-picptr[k+e]-          /* first corner  */
                       picptr[k+g]-picptr[k+h]);
   k=x-1;
   picptr_tmp[k]=(int)(picptr[k]*4-picptr[k+d]-          /* second corner */
                       picptr[k+f]-picptr[k+g]);
   k=x*(y-1);
   picptr_tmp[k]=(int)(picptr[k]*4-picptr[k+b]-          /* third corner  */
                       picptr[k+c]-picptr[k+e]);
   k=x*y-1;
   picptr_tmp[k]=(int)(picptr[k]*4-picptr[k+a]-          /* fourth corner */
                       picptr[k+b]-picptr[k+d]);

/*                        calculate boundary of picture                 */

   i=x*(y-1);
   for(k=1;k<(x-1);k++)
   {
      picptr_tmp[k]=(int)(picptr[k]*5-picptr[k+d]-picptr[k+e]-  /* top  */
                          picptr[k+f]-picptr[k+h]);
      j=i+k;
      picptr_tmp[j]=(int)(picptr[j]*5-picptr[j+a]-              /* bot  */
                          picptr[j+c]-picptr[j+d]-picptr[j+e]);
   }

   for(k=1;k<(y-1);k++)
   {
      i=k*x;
      picptr_tmp[i]=(int)(picptr[i]*5-picptr[i+b]-picptr[i+c]-  /* left */
                          picptr[i+g]-picptr[i+h]);
      i=(k+1)*x-1;
      picptr_tmp[i]=(int)(picptr[i]*5-picptr[i+a]-picptr[i+b]-  /* right */
                          picptr[i+f]-picptr[i+g]);
   }

/*                        calculate interior of picture                 */

   for(j=1;j<y-1;j++)
```

```
    {
       for(i=1;i<(x-1);i++)
       {
          k=j*x+i;
          tmp=((int)picptr[k]*5-picptr[k+b]-picptr[k+d]-picptr[k+e]-
                    picptr[k+g]);
          picptr_tmp[k]=max(0,min(255,tmp));
       }
    }
    i=x*y;
    memmove(picptr,picptr_tmp,i);        /* copy result back into buffer */
    free(picptr_tmp);                    /* free working array           */

}

/*                                  LINREG                                  */
/*                         calculate slope from array given                 */
/*                              by linear regression                        */
/*      Subroutine is called using the following parameters:                */
/*                                                                          */
/*              slope=LINREG(n,array);                                      */
/*      where                                                               */
/*              n       = number of data points (x,y)                       */
/*              array   = data array of [(x0,y0),(x1,y1)...]                */
/*                        points (float)                                    */
/*              slope   = slope of resultant curve                          */

float LINREG(n,data)
int n;
float *data;
{
   int i;
   float sx,sy,sxy,sx2,tmp1,tmp2,slope;

   sx=0.0;
   sy=0.0;
   sxy=0.0;
   sx2=0.0;
   for(i=0;i<(n*2);i+=2)
   {
      tmp1=data[i];
      tmp2=data[i+1];
      sxy=sxy+tmp1*tmp2;
      sx=sx+tmp1;
      sy=sy+tmp2;
      sx2=sx2+tmp1*tmp1;
   }
   slope=((float)n*sxy-sx*sy)/((float)n*sx2-sx*sx);
   return(slope);
}
```

C6 FILE: ROUTINES.C

```
/*                    Include files and prototypes                          */
```

```
#include <graph.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <stddef.h>
#include <dos.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include "picture.h"

/*                    function prototypes                                  */

int ZERO_CUT(void);
int MAX_CUT(void);
int SAVE(void);
int RECALL(void);
int MEDIAN(void);
void picsrt(short n,short *arr);
int GETSUBIMAGE(void);
void DRAW_BOX(int x,int y, int dx, int dy, int state);
void TIFF_WRITE(char *out_file,int mode);
extern int DRAW_MENU(int menu);
extern void DRAW_MESSAGE(int pos,char *msg);
extern void GET_MESSAGE(int pos, char *msg);
extern void PIC_DRAW(long position,struct PICTURE *PICT);

extern struct PICTURE PIC;
extern char *imgptr;
extern int imflag;
extern char msg1[60];


/*-------------------------------------------------------------------------*/
/*                              ZERO_CUT                                   */
/*      This routine takes the value entered by the user as a cuttoff      */
/*      value and sets all pixels in the picture below this value to       */
/*      zero.  The routine can be called many times to allow the user      */
/*      to see what effects the level selection is having.                 */
/*                                                                         */
/*-------------------------------------------------------------------------*/

int  ZERO_CUT(void)
{
   char buffer[50],*picptr;
   int i,j,zerocut,x,y;

   DRAW_MESSAGE(2,"Enter Zero Cuttoff Level (int) :");
   GET_MESSAGE(2,buffer);
   zerocut=atoi(buffer);

   picptr=PIC.pic_buf;                  /* get picture location          */
   x=PIC.imagewidth;                    /* get size of picture           */
```

```
    y=PIC.imagelength;                    /* get size of picture          */
    j=x*y;

    for(i=0;i<j;i++)
    {
     if(picptr[i]<zerocut) picptr[i]=0;
    }
    return(0);
 }

 /*-------------------------------------------------------------------------*/
 /*                              MAX_CUT                                    */
 /*      This routine takes the value entered by the user as a cuttoff      */
 /*      value and sets all pixels in the picture above this value to       */
 /*      255.   The routine can be called many times to allow the user      */
 /*      to see what effects the level selection is having.                 */
 /*                                                                         */
 /*-------------------------------------------------------------------------*/

 int  MAX_CUT(void)
 {
    char buffer[50],*picptr;
    int i,j,zerocut,x,y;

    DRAW_MESSAGE(2,"Enter Max. Cuttoff Level (int) :");
    GET_MESSAGE(2,buffer);
    zerocut=atoi(buffer);

    picptr=PIC.pic_buf;                   /* get picture location         */
    x=PIC.imagewidth;                     /* get size of picture          */
    y=PIC.imagelength;                    /* get size of picture          */
    j=x*y;

    for(i=0;i<j;i++)
    {
     if(picptr[i]>zerocut) picptr[i]=255;
    }
    return(0);
 }

 /*-------------------------------------------------------------------------*/
 /*                              SAVE                                       */
 /*      This routine takes the current picture in memory and saves it      */
 /*      into a backup array.  This array can then be recalled at any       */
 /*      time to retrieve this image.  For long sequences of calculation    */
 /*      this SAVE can be used to recall a previously calculated image      */
 /*      or to save disk reading time.                                      */
 /*-------------------------------------------------------------------------*/

 int SAVE(void)
 {
    char *picptr, *save_ptr;
    static long x,y;
    long i,k;
```

```
    if (PIC.save==1)
    {
     free(PIC.saveptr);                  /* free old save if required    */
    }

    picptr=PIC.pic_buf;                   /* get picture location         */
    x=PIC.imagewidth;                     /* get size of picture          */
    y=PIC.imagelength;                    /* get size of picture          */

    k=x*y;                                /* size of save image           */
    if((save_ptr=(char *)malloc(k))==NULL)  /* memory allocation error      */
    {
     return(0);                          /* return with error flag       */
    }

    PIC.savewidth=x;
    PIC.savelength=y;
    PIC.saveptr=save_ptr;
    PIC.save=1;
    i=x*y;
    memmove(save_ptr,picptr,i);          /* copy result into save buffer */
}

/*-------------------------------------------------------------------------*/
/*                              RECALL                                     */
/*      This routine takes the current picture saved and sets it           */
/*      into the main picture array.  The saved picture is not affected    */
/*      by the operation.  This allows the user to recall an image         */
/*      quickly to save disk reading time.                                 */
/*-------------------------------------------------------------------------*/

int RECALL(void)
{
   char *picptr, *save_ptr;
   static long x,y;
   long i,k;

   if (PIC.valid_pic==1)
   {
    free(PIC.pic_buf);                  /* free old picture             */
   }
   x=PIC.savewidth;                      /* get size of picture          */
   y=PIC.savelength;                     /* get size of picture          */
   save_ptr=PIC.saveptr;                 /* get picture location         */

   k=x*y;                                /* size of save image           */
   if((picptr=(char *)malloc(k))==NULL)  /* memory allocation error      */
   {
    return(0);                          /* return with error flag       */
   }

/*                      recall old picture                                 */
```

```
    PIC.imagewidth=x;
    PIC.imagelength=y;
    PIC.valid_pic=1;
    PIC.pic_buf=picptr;
    i=x*y;
    memmove(picptr,save_ptr,i);          /* copy result into buffer      */
}

/*--------------------------------------------------------------------------*/
/*                                  MEDIAN                                  */
/*      This routine runs the picture though a 9 Point Median filter        */
/*      to remove shot type noise in the picture.  The filter puts the      */
/*      elements of a 3x3 mask into an array, sorts the array and re-       */
/*      places the middle pixel by the value in the middle of the sort.     */
/*--------------------------------------------------------------------------*/

int MEDIAN(void)
{
   char *picptr, *picptr_tmp;
   long i,j,k,x,y;
   short a,b,c,d,e,f,g,h,sort[9];

   picptr=PIC.pic_buf;                   /* get picture location         */
   x=PIC.imagewidth;                     /* get size of picture          */
   y=PIC.imagelength;                    /* get size of picture          */

   k=x*y;                                /* size of new image            */
   if((picptr_tmp=(char *)malloc(k))==NULL)     /* memory allocation
error      */
   {
    return(0);                          /* return with error flag       */
   }

/*                       calculate pixel references                       */
/*      Each pixel reference will use the offsets calculated here.        */
/*      The pixel base is first set and all offsets from the base         */
/*      use one of the following numbers depending upon the position      */
/*      of the pixel required. ie:                                        */
/*                                                                        */
/*                                    a     b    c                       */
/*                                    d    base  e                       */
/*                                    f     g    h                       */
/*                                                                        */
         a=-x-1;  b=-x;    c=-x+1;
         d=-1;             e=1;
         f=x-1;   g=x;     h=x+1;


/*                       calculate interior of picture                    */

   for(j=1;j<(y-1);j++)
   {
    for(i=1;i<(x-1);i++)
    {
    k=j*x+i;
```

```
      sort[0]=picptr[k+a]; sort[1]=picptr[k+b]; sort[2]=picptr[k+c];
      sort[3]=picptr[k+d]; sort[4]=picptr[k]; sort[5]=picptr[k+e];
      sort[6]=picptr[k+f]; sort[7]=picptr[k+g]; sort[8]=picptr[k+h];
      picsrt(8,sort);
      picptr_tmp[k]=sort[4];
      }
     }

 /*                        set corners of image                          */

    k=x+1;
    picptr_tmp[0]=picptr_tmp[k];

    k=x-1;
    picptr_tmp[k]=picptr_tmp[k+x-1];

    k=x*y-1;
    picptr_tmp[k]=picptr_tmp[k-x-1];

    k=x*(y-1);
    picptr_tmp[k]=picptr_tmp[k-x+1];

 /*                        set boundary of picture                       */

    i=x*(y-1);
    for(k=1;k<(x-1);k++)
    {
     picptr_tmp[k]=picptr_tmp[k+x];
     j=i+k;
     picptr_tmp[j]=picptr_tmp[j-x];
    }

    for(k=2;k<(y-1);k++)
    {
     i=k*x-1;
     picptr_tmp[i]=picptr_tmp[i-1];
     i=i+1-x;
     picptr_tmp[i]=picptr_tmp[i+1];
    }


    i=x*y;
    memmove(picptr,picptr_tmp,i);        /* copy result back into buffer */
    free(picptr_tmp);                    /* free working array           */

}

/*                  SHELL SORT function for MEDIAN                       */

void picsrt(short n,short *arr)
{
   short i,j,a;

   for(j=2;j<=n;j++)              /* pick out each element in turn       */
```

```
    {
     a=arr[j];
     i=j-1;
     while(i>0 && arr[i]>a)     /* look for place to insert it          */
     {
     arr[i+1]=arr[i];
     i--;
     }
     arr[i+1]=a;                /* insert it                            */
    }
 }

 /*--------------------------------------------------------------------*/
 /*                            CUBEDIMENSION                           */
 /*      This routine takes the image in memory, calculates a Cubic    */
 /*      Dimension and returns the result.  The routine works          */
 /*      by counting the cubes in the picture that contain the picture.*/
 /*      The intensity of each pixel is interpreted as a height or Z   */
 /*      value.  Ajoining pixels are assumed connected by a surface    */
 /*      and all cubes containing this surface counted.  Note that a   */
 /*      a 1x1x1 cube requires 4 pixels, 2x2x2 requires 9 pixels etc.  */
 /*         The results are then fitted to a log/log line using linear */
 /*      regression on the data.                                       */
 /*--------------------------------------------------------------------*/

 void CUBEDIMENSION(float *dimension)
 {
    long h,i,j,k,m1,m2,ix,iy,isz,ofst1,ofst2;
    long icx,icy,cnt[8],size[8],cubes;
    double tmp1,tmp2;
    double n,sx,sy,sxy,sx2;
    unsigned char c,d;
    char msg[60];
    unsigned char *imgptr;

    imgptr=PIC.pic_buf;                 /* locate picture              */
    ix=PIC.imagewidth;                  /* start at full resolution    */
    iy=PIC.imagelength;                 /* set y variable              */

 /*      To eliminate edge effects, ensure same area used each iteration */

    icx=(ix/30)*30;                     /* ensure size divisible by 30 */
    icy=(iy/30)*30;                     /* ensure size divisible by 30 */

 /*                      1,2,3,4,5,6  size cubes                       */

    for(i=1;i<7;i++) {
    isz=i;                              /* size of cube                */
    cubes=0;                            /* cube counter                */
    for(j=0;j<icx;j+=i) {
      for(k=0;k<icy;k+=i) {
     ofst1=k*ix+j;                      /* offset start of box         */
     c=imgptr[ofst1];                   /* get first value in box      */
     d=imgptr[ofst1];
```

```
        for(m1=0;m1<=isz;m1++) {          /* get each row in box          */
        ofst2=ofst1+m1*ix;              /* get start of each row        */
        for(m2=0;m2<=isz;m2++) {        /* get each pixel in box        */
          c=max(c,imgptr[ofst2+m2]);    /* find greatest pixel          */
          d=min(d,imgptr[ofst2+m2]);    /* find minimum pixel           */
        }
        }
        cubes=cubes+((long)(c-d)+(isz-1))/isz+1;   /* count cubes     */
      }
      }
      cnt[i-1]=cubes;                   /* save result                  */
      size[i-1]=((icx-1)+(icy-1))/((isz)*2);   /* average picture size  */
      h=isz;                            /* cube size used               */
      sprintf(msg,"%d cubes at %dx%dx%d cube size counted",cubes,h,h,h);
      DRAW_MESSAGE(2,msg);
     }

 /*                      calculate dimension from numbers                */
 /*                           by linear regression                       */

    n=6.0;
    sx=0.0;
    sy=0.0;
    sxy=0.0;
    sx2=0.0;
    for(i=0;i<6;i++) {
     tmp1=log10((float) size[i]);
     tmp2=log10((float) cnt[i]);
     sxy=sxy+tmp1*tmp2;
     sx=sx+tmp1;
     sy=sy+tmp2;
     sx2=sx2+tmp1*tmp1;
    }
    *dimension=(n*sxy-sx*sy)/(n*sx2-sx*sx);
    return;
}

/*                                GETSUBIMAGE                             */
/*      This routine when called allows the operator to select a          */
/*      subimage of the memory.  The first step is to write the           */
/*      current memory to the left side of the screen.  Next a            */
/*      selection box and menu is created.  The operator can then         */
/*      select the part of the image required for more analysis.          */

int GETSUBIMAGE(void)
{
   struct PICTURE SUBIMAGE;
   int i,j,k,x,y,dx,dy,c,step;
   int imx,imy,imdx,imdy,imwd,imln;
   unsigned char *picptr,*subptr;
   char msg[100];

   DRAW_MENU(5);                    /* subimage capture menu            */
   imwd=PIC.imagewidth;
```

```
imx=imwd/4;
imln=PIC.imagelength;
imy=imln/4;
imdx=imwd/2;
imdy=imln/2;                    /* calculate initial box size          */
x=510*imx/imwd;
y=510*imy/imln;
dx=510*imdx/imwd;
dy=510*imdy/imln;               /* screen box size                     */
step=1;                         /* initial step size                   */
DRAW_BOX(x,y,dx,dy,0);          /* draw initial box                    */

while((c=getch())!=27)          /* subimage capture loop               */
{
 if(c==0) c=getch()+256; /* get control key                          */
 switch (c)
 {

 case 43 :                /* + key                                    */
   if((imdx+2)<=imwd && (imdy+2)<=imln && imx>0 && imy>0 &&
    (imx+imdx+1)<=imwd && (imy+imdy+1)<=imln)
   {
    imdx+=2;
    imdy+=2;           /* increase box size                        */
    imx-=1;
    imy-=1;
   }
   break;

 case 45 :                /* - key                                    */
   if((imdx-2)>0 && (imdy-2)>0)
   {
    imdx-=2;
    imdy-=2;           /* decrease box size                        */
    imx+=1;
    imy+=1;
   }
   break;

 case 52  :
 case 331 :               /* left arrow                               */
   if((imx-step)>=0)
   {
    imx-=step;         /* shift image left                         */
   }
   break;

 case 54  :
 case 333 :               /* right arrow                              */
   if((imx+step+imdx)<=imwd)
   {
    imx+=step;         /* shift image right                        */
   }
   break;
```

```
 case 56  :
 case 328 :              /* up arrow                          */
   if((imy-step)>=0)
   {
    imy-=step;         /* shift image up                    */
   }
   break;

 case 50  :
 case 336 :              /* down arrow                        */
   if((imy+step+imdy)<=imln)
   {
    imy+=step;         /* shift image down                  */
   }
   break;

 case 120 :              /* x key                             */
   if((imdx-step)>0)
   {
    imdx-=step;        /* decrease X size                   */
   }
   break;

 case 88  :              /* X key                             */
   if((imx+imdx+step)<=imwd)
   {
    imdx+=step;        /* increase X size                   */
   }
   break;

 case 121 :              /* y key                             */
   if((imdy-step)>0)
   {
    imdy-=step;        /* decrease Y size                   */
   }
   break;

 case 89  :              /* Y key                             */
   if((imy+imdy+step)<=imln)
   {
    imdy+=step;        /* increase Y size                   */
   }
   break;

 case 70  :
 case 102 :
   step=4;              /* shift to faster speed             */
   break;

 case 83  :
 case 115 :
   step=1;              /* shift to slow speed               */
   break;
```

```
     case 77  :
     case 109 :                              /* magnify image              */
        i=imdx*imdy;
        if(imflag==1) free(subptr);          /* free old picture           */
       if((subptr=(unsigned char *)malloc(i))==NULL) /* memory allocation error
*/
        {
         return(0);                        /* return with error          */
        }
        imflag=1;                            /* memory now allocated       */
        k=0;
        picptr=PIC.pic_buf;                 /* get input picture location   */
        for(i=imy;i<(imdy+imy);i++)  /* transfer selection to buffer */
        {
         for(j=imx;j<(imdx+imx);j++)
         {
          subptr[k++]=picptr[i*imwd+j]; /* move subimage to mem  */
         }
        }
        SUBIMAGE.imagewidth=imdx;
        SUBIMAGE.imagelength=imdy;
        SUBIMAGE.pic_buf=subptr;     /* save picture location         */
        sprintf(msg,"(%d,%d)   Image size = %dx%d",imx,imy,imdx,imdy);
        DRAW_MESSAGE(1,msg);
        PIC_DRAW(1,&SUBIMAGE);
        break;

    case 85  :
    case 117 :                           /* use subimage                */
       i=imdx*imdy;
       if(imflag==1) free(subptr);     /* free old picture           */
       if((subptr=(unsigned char *)malloc(i))==NULL)    /* memory allocation
error   */
       {
        return(0);                  /* return with error          */
       }
       imflag=1;
       k=0;
       picptr=PIC.pic_buf;          /* get input picture location   */
       for(i=imy;i<(imdy+imy);i++) /* transfer selection to buffer */
       {
        for(j=imx;j<(imdx+imx);j++)
        {
        subptr[k++]=picptr[i*imwd+j];
        }
       }
       SUBIMAGE.imagewidth=imdx;
       SUBIMAGE.imagelength=imdy;
       SUBIMAGE.pic_buf=subptr;     /* save picture location         */
       sprintf(msg1,"(%d,%d), Subimage size = %dx%d",imx,imy,imdx,imdy);
       picptr=PIC.pic_buf;          /* get input picture location   */
       free(picptr);                /* free old image               */
       PIC.imagewidth=SUBIMAGE.imagewidth;
```

```
        PIC.imagelength=SUBIMAGE.imagelength;
        PIC.pic_buf=SUBIMAGE.pic_buf;       /* use new image        */
        return(1);
      }
      x=510*imx/imwd;
      y=510*imy/imln;
      dx=510*imdx/imwd;
      dy=510*imdy/imln;          /* screen box size                 */
      DRAW_BOX(x,y,dx,dy,1);     /* draw box                        */
    }
    DRAW_BOX(x,y,dx,dy,2);     /* erase box                          */
    return(1);
}

/*                                DRAW_BOX                            */
/*      This routine takes the parameters passed and draws a green    */
/*      box on the screen.  The old box is first erased before the    */
/*      new one is drawn if the state flag is set.                    */

void DRAW_BOX(int x,int y, int dx, int dy, int state)
{
   static int oldx,oldy,olddx,olddy;
   int i,distx,disty;

   _setviewport(1,1,511,511);                  /* set viewport       */
   distx=oldx+olddx;
   disty=oldy+olddy;
   if(state==1 || state==2)                    /* erase old box      */
   {
    for(i=oldx;i<=distx;i+=3)
    {
    _setcolor(_getpixel(i,oldy)+64);
    _setpixel(i,oldy);                        /* top of box         */
    _setcolor(_getpixel(i,disty)+64);
    _setpixel(i,disty);                       /* bottom of box      */
    }
    for(i=oldy+1;i<disty;i+=3)
    {
    _setcolor(_getpixel(oldx,i)+64);
    _setpixel(oldx,i);                        /* left side of box   */
    _setcolor(_getpixel(distx,i)+64);
    _setpixel(distx,i);                       /* right side of box  */
    }
  }
  if(state==0 || state==1)
  {
   distx=x+dx;
   disty=y+dy;
   for(i=x;i<=distx;i+=3)                       /* draw new box      */
   {
   _setcolor(_getpixel(i,y)-64);
   _setpixel(i,y);                            /* top of box         */
   _setcolor(_getpixel(i,disty)-64);
   _setpixel(i,disty);                        /* bottom of box      */
```

```
      }
      for(i=y+1;i<disty;i+=3)
      {
      _setcolor(_getpixel(x,i)-64);
      _setpixel(x,i);                          /* left side of box     */
      _setcolor(_getpixel(distx,i)-64);
      _setpixel(distx,i);                      /* right side of box    */
      }
      oldx=x;
      oldy=y;
      olddx=dx;
      olddy=dy;                                  /* save old values      */
    }
 }

 /*                                TIFF_WRITE                                */
 /*      The purpose of this routine is to write the active image in         */
 /*      memory or the screen image to a specified file name.  The           */
 /*      format is reduced as much as possible with a fixed header size.     */
 /*      This will allow simpler programs to be written to further           */
 /*      process the results if required.                                    */

 void TIFF_WRITE(char *out_file,int mode)
 {
    long int i,j;
    int  c,d;
    unsigned char buf[220];       /* fixed header size for TIF file       */
    short int *ibuf;
    long int  *lbuf;
    FILE *output;

    ibuf=(short int*) &buf[10];
    lbuf=(long int*) &buf[10];

    if ((output = fopen(out_file, "wb")) == NULL) /* open file as binary */
    {
     DRAW_MESSAGE(2,"File error has occurred.");
     return;
    }

 /*              Create TIFF file header for writing                         */

    buf[0]='I';
    buf[1]='I';                              /* byte order identifier        */
    buf[2]= 42;                              /* 2 TIF file identifier        */
    buf[3]= 0;
    buf[4]= 8;
    buf[5]= 0;
    buf[6]= 0;
    buf[7]= 0;                               /* 6 first IFD offset           */
 /*                                IFD's                                     */
    buf[8]= 12;
    buf[9]= 0;                               /* 8 number of IFD's            */
 /*      IFD 1           */
```

```
    if(mode==0)
    {
     ibuf[0]= 256;                    /* 10 image width identifier    */
     ibuf[1]= 3;                      /* 12 short field type          */
     ibuf[2]= 1;
     ibuf[3]= 0;                      /* 14 one entry immediate       */
     ibuf[4]= PIC.imagewidth;
     ibuf[5]= 0;                      /* 18 image width entry         */
    }
    else
    {
     ibuf[0]= 256;                    /* 10 image width identifier    */
     ibuf[1]= 3;                      /* 12 short field type          */
     ibuf[2]= 1;
     ibuf[3]= 0;                      /* 14 one entry immediate       */
     ibuf[4]= (short int) 1024;       /* 18 image width entry         */
     ibuf[5]= 0;
    }
 /*      IFD 2           */
    if(mode==0)
    {
     ibuf[6]= 257;                    /* 22 image length identifier   */
     ibuf[7]= 3;                      /* 24 short field type          */
     ibuf[8]= 1;
     ibuf[9]= 0;                      /* 26 one entry immediate       */
     ibuf[10]= PIC.imagelength;
     ibuf[11]= 0;                     /* 30 image length entry        */
    }
    else
    {
     ibuf[6]= 257;                    /* 22 image length identifier   */
     ibuf[7]= 3;                      /* 24 short field type          */
     ibuf[8]= 1;
     ibuf[9]= 0;                      /* 26 one entry immediate       */
     ibuf[10]= (short int) 542;
     ibuf[11]= 0;                     /* 30 image length entry        */
    }
 /*      IFD 3           */
    ibuf[12]= 258;                    /* 34 number of grayscales      */
    ibuf[13]= 3;                      /* 36 short field type          */
    ibuf[14]= 1;
    ibuf[15]= 0;                      /* 38 one entry immediate       */
    ibuf[16]= 8;
    ibuf[17]= 0;                      /* 42 number of grayscales      */

 /*      IFD 4           */
    ibuf[18]= 259;                    /* 46 image compression identifier */
    ibuf[19]= 3;                      /* 48 long field type           */
    ibuf[20]= 1;
    ibuf[21]= 0;                      /* 50 one entry immediate       */
    ibuf[22]= 1;
    ibuf[23]= 0;                      /* 54 no compression used       */
 /*      IFD 5           */
    ibuf[24]= 262;                    /* 58 photometric interpretation */
```

```
    ibuf[25]= 3;                          /* 60 long field type          */
    ibuf[26]= 1;
    ibuf[27]= 0;                          /* 62 one entry immediate      */
    ibuf[28]= 1;
    ibuf[29]= 0;                          /* 66 photometric interpretation */
 /*      IFD 6          */
    ibuf[30]= 270;                        /* 70 strip offset             */
    ibuf[31]= 2;                          /* 72 ascii field type         */
    lbuf[16]= 40;                         /* 74 40 characters            */
    lbuf[17]= 174;                        /* 78 strip offset             */
 /*      IFD 7          */
    ibuf[36]= 273;                        /* 82 strip offset             */
    ibuf[37]= 4;                          /* 84 long field type          */
    lbuf[19]= 1;                          /* 86 one entry immediate      */
    lbuf[20]= 220;                        /* 90 strip offset             */
 /*      IFD 8          */
    if(mode==0)
    {
     ibuf[42]= 278;                       /* 94 Rows per strip           */
     ibuf[43]= 3;                         /* 96 short field type         */
     ibuf[44]= 1;
     ibuf[45]= 0;                         /* 98 one entry immediate      */
     ibuf[46]= PIC.imagelength;
     ibuf[47]= 0;                         /* 102 rows entry              */
    }
    else
    {
     ibuf[42]= 278;                       /* 94 Rows per strip           */
     ibuf[43]= 3;                         /* 96 short field type         */
     ibuf[44]= 1;
     ibuf[45]= 0;                         /* 98 one entry immediate      */
     ibuf[46]= (short int) 542;
     ibuf[47]= 0;                         /* 102 Rows per strip          */
    }
 /*      IFD 9          */
    if(mode==0)
    {
     ibuf[48]= 279;                       /* 106 Strip byte count        */
     ibuf[49]= 4;                         /* 108 long field type         */
     lbuf[25]= 1;                         /* 110 one entry immediate     */
     lbuf[26]= PIC.imagewidth*PIC.imagelength;
                  /* 114 Strip byte count        */
    }
    else
    {
     ibuf[48]= 279;                       /* 106 Strip byte count        */
     ibuf[49]= 4;                         /* 108 long field type         */
     lbuf[25]= 1;                         /* 110 one entry immediate     */
     lbuf[26]= (long) 1024*542;           /* 114 Strip byte count        */
    }
/*      IFD 10         */
    ibuf[54]= 282;                        /* 118 XResolution             */
    ibuf[55]= 5;                          /* 120 long field type         */
    lbuf[28]= 1;                          /* 122 one offset              */
```

```
    lbuf[29]= 158;                          /* 126 XResolution location     */
 /*      IFD 11          */
    ibuf[60]= 283;                          /* 130 YResolution              */
    ibuf[61]= 5;                            /* 132 long field type          */
    lbuf[31]= 1;                            /* 134 one offset               */
    lbuf[32]= 166;                          /* 138 YResolution location     */
 /*      IFD 12          */
    ibuf[66]= 296;                          /* 142 Resolution unit          */
    ibuf[67]= 3;                            /* 144 long field type          */
    ibuf[68]= 1;
    ibuf[69]= 0;                            /* 146 one entry immediate      */
    ibuf[70]= 2;
    ibuf[71]= 0;                            /* 150 Resolution unit          */
 /*      End of IFD flag                                                     */
    lbuf[36]= 0;                            /* 154                          */
 /*      Resolution arrays                                                   */
    lbuf[37]=100;                           /* 158                          */
    lbuf[38]=1;                             /* 162                          */
    lbuf[39]=100;                           /* 166                          */
    lbuf[40]=1;                             /* 170                          */
 /*      Save message string   */
    strncpy((char *)&lbuf[41],PIC.message,40);

    fwrite(buf,1,220,output);               /* write header to file         */
    if(mode==0)
    {
     i=PIC.imagewidth*PIC.imagelength;
     fwrite(PIC.pic_buf,1,i,output); /* write picture to file       */
    }
    else
    {
     _setviewport(0,0,1023,767);     /* set viewport                 */
     for(i=0;i<542;i++)
     {
        for(j=0;j<1024;j++)
        {
       c=_getpixel(j,i);        /* get screen pixel             */
       if(c>191)
       {                        /* convert screen to grayscale  */
          d=((int)((float)(c-192)*4.056));
       }
       else if (c==0) d=255;    /* convert black to white       */
       else d=0;                /* all else to black            */
       fputc(d,output);
        }
     }
    }
    fclose(output);                         /* close file after done        */
    return;
}
```

C7 FILE: SCREEN.C

```
/*                      Screen Plotting Routines                            */
```

```
/*      This purpose of these routines ss to allow the easy plotting    */
/*      of data on a screen at any resolution in 16 colors              */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <graph.h>
#include "screen.h"

/*                          Function Prototypes                         */

void DATASET(int n, float *array, int m,struct SCREEN *SCRN);
void TITLES(char *str1, char *str2,struct SCREEN *SCRN);
void DRAWBORDER(struct SCREEN *SCRN);
void DRAWDATA(struct SCREEN *SCRN);

/*                          DATASET sets the number of                  */
/*                          data points and pointer to the data         */

void DATASET(n,array,m,SCRN)
int n,m;
float *array;
struct SCREEN *SCRN;
{
   SCRN->data=array;
   SCRN->nn=n;
   SCRN->color=m;
}

/*                          SCREENSET                                   */
/*      Sets the screen parameters for the display into the structure.  */
/*      This routine assumes that the display has already been          */
/*      initialized.                                                    */

void SCREENSET(int xst,int yst,int xsp,int ysp,float xmn,float ymn,
          float xmx,float ymx,int typ,struct SCREEN *SCRN)
{
   SCRN->xstart=xst;
   SCRN->ystart=yst;
   SCRN->xspan=xsp;
   SCRN->yspan=ysp;
   SCRN->xmin=xmn;
   SCRN->ymin=ymn;
   SCRN->xmax=xmx;
   SCRN->ymax=ymx;
   SCRN->type=typ;
}


/*                          TITLES sets the titles                      */
/*                          into the screen structure                   */

void TITLES(str1,str2,SCRN)
```

```
char *str1,*str2;
struct SCREEN *SCRN;
{
   strncpy(SCRN->title1,str1,39);
   strncpy(SCRN->title2,str2,39);
   SCRN->title1[39]='\0';
   SCRN->title2[39]='\0';
}

/*                          DRAWBORDER                                    */
/*      This routine draws the structure given onto the screen.           */
/*      It first installs the graph axes then draws the graph and         */
/*      finally puts on the screen titles.                                */

void DRAWBORDER(SCRN)
struct SCREEN *SCRN;
{
   int i,j,k,l;
   float dx,dy,tmp,xdif,ydif;
   char out[8];

   i=65;                              /* start graph after numbers     */
   j=SCRN->yspan-29;
   _settextalign(_LEFT,_TOP);
   _setcolor(GREEN);                  /* set the color for the lines   */
   _setlinestyle(SOLID_LINE);
   _setviewport(SCRN->xstart,SCRN->ystart,SCRN->xstart+SCRN->xspan,
            SCRN->ystart+SCRN->yspan);
   _clearscreen(_GVIEWPORT);
   _moveto(i,0);
   _lineto(i,j);
   _lineto(SCRN->xspan-11,j);
   j=SCRN->yspan-28;                  /* tics along x-axis             */
   k=SCRN->yspan-22;
   for (i=0;i<11;i++) {
      _setcolor(GREEN);               /* set the color for the lines   */
      _setlinestyle(SOLID_LINE);
      l=67+(SCRN->xspan-80)*i/10;
      _moveto(l,j);
      _lineto(l,k);
      _setcolor(LIGHTRED);            /* set the color for the lines   */
      _setlinestyle(DOTTED_LINE);
      _moveto(l,j);
      _lineto(l,0);
   }

   j=60;                              /* tics along y-axis             */
   k=64;
   for (i=0;i<11;i++) {
      _setcolor(GREEN);               /* set the color for the lines   */
      _setlinestyle(SOLID_LINE);
      l=(SCRN->yspan-30)*i/10;
      _moveto(j,l);
      _lineto(k,l);
```

```
        _setcolor(LIGHTRED);          /* set the color for the lines  */
        _setlinestyle(DOTTED_LINE);
        _moveto(k,1);
        _lineto(SCRN->xspan-11,1);
     }

     xdif=(SCRN->xmax-SCRN->xmin);
     ydif=(SCRN->ymax-SCRN->ymin);
     dx=xdif/5.0;
     dy=ydif/5.0;
     _setcolor(YELLOW);               /* set the color for the text   */
     _setcharsize(16,10);
     for (i=0;i<=5;i++) {                      /* draw in y-numbers    */
        k=(SCRN->yspan-29)*i/5+1;
        j=(SCRN->yspan-29)-k;
        tmp=SCRN->ymin+dy*i;
        sprintf(out,"%6.2f",tmp);
        _grtext(0,j,out);
     }
     for (i=0;i<=5;i++) {                      /* draw in x-numbers    */
        j=30+(SCRN->xspan-78)*i/5;
        tmp=SCRN->xmin+dx*i;
        sprintf(out,"%6.2f",tmp);
        _grtext(j,SCRN->yspan-16,out);
     }
}

/*                DRAWDATA puts the actual data onto the                */
/*        screen.                                                       */

void DRAWDATA(SCRN)
struct SCREEN *SCRN;
{
   int i,j,k,xpt,ypt,xold,yold;
   float xdif,ydif;
   float *data;

   data=SCRN->data;                  /* get address of data          */
   j=67;                             /* x-offset into graph          */
   k=SCRN->yspan-30;                 /* y-offset into graph          */
   xdif=(SCRN->xmax-SCRN->xmin);
   ydif=(SCRN->ymax-SCRN->ymin);
   _setviewport(j+SCRN->xstart,SCRN->ystart,SCRN->xspan+SCRN->xstart,
          SCRN->yspan+SCRN->ystart-30); /* bug here !!!! */
   switch (SCRN->type) {
      case 0:  for (i=0;i<SCRN->nn;i+=2) {
            ypt=k-((data[i+1]-SCRN->ymin)/ydif*(SCRN->yspan-30));
            xpt=((data[i]-SCRN->xmin)/xdif*(SCRN->xspan-80));
            _setcolor(SCRN->color);
            _setpixel(xpt-1,ypt-1);
            _setpixel(xpt,ypt-2);
            _setpixel(xpt,ypt-1);
            _setpixel(xpt+1,ypt-1);
            _setpixel(xpt-1,ypt);
```

```
                _setpixel(xpt-2,ypt);
                _setpixel(xpt,ypt);
                _setpixel(xpt+1,ypt);
                _setpixel(xpt+2,ypt);
                _setpixel(xpt-1,ypt+1);
                _setpixel(xpt,ypt+1);
                _setpixel(xpt,ypt+2);
                _setpixel(xpt+1,ypt+1);
             }
             break;
        case 1:  ypt=k-((data[1]-SCRN->ymin)/ydif*(SCRN->yspan-30));
             xpt=((data[0]-SCRN->xmin)/xdif*(SCRN->xspan-80));
             _setcolor(SCRN->color);
             _setlinestyle(SOLID_LINE);
             _moveto(xpt,ypt);            /* move to start        */
             for (i=2;i<SCRN->nn;i+=2) {
                xold=xpt;
                yold=ypt;
                ypt=k-((data[i+1]-SCRN->ymin)/ydif*(SCRN->yspan-30));
                xpt=((data[i]-SCRN->xmin)/xdif*(SCRN->xspan-80));
                _lineto(xpt,ypt);
             }
             break;
        case 2:  _setcolor(SCRN->color);
             _setlinestyle(SOLID_LINE);
             for (i=0;i<SCRN->nn;i+=2) {
                ypt=k-((data[i+1]-SCRN->ymin)/ydif*(SCRN->yspan-30));
                xpt=((data[i]-SCRN->xmin)/xdif*(SCRN->xspan-80));
                _moveto(xpt,k);
                _lineto(xpt,ypt);
             }
             break;
    }
    _setcharsize(20,12);
    _setcolor(LIGHTGREEN);
    _settextalign(_CENTER,_TOP);
    _grtext(SCRN->xspan/2-40,1,SCRN->title1);
    _setcharsize(20,11);
    _setcolor(CYAN);
    _grtext(SCRN->xspan/2-40,23,SCRN->title2);
    _settextalign(_LEFT,_TOP);
}
```

C8 FILE: SIM.C

```
/*                      Simulation of Laplacian Fractals                */

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <malloc.h>
#include <conio.h>
```

```
#include <graph.h>
#include <dos.h>
#include "picture.h"

#define SX 100                          /* maximum array size x direction */
#define SY 100                          /* maximum array size y direction */
#define maxv 1000000000                 /* maximum potential on electrode */
extern long localiter;                  /* must be even!!                 */
extern long globaliter;                 /* must be even!!                 */
long startnumber=-1;
long *idum=&startnumber;                /* number generator start         */

/*              Function Prototypes                                       */

int PARTREESIM1(struct PICTURE *PICT);
int PARTREESIM2(struct PICTURE *PICT);
int CIRTREESIM1(struct PICTURE *PICT);
int CIRTREESIM2(struct PICTURE *PICT);
int CIRTREESIM3(struct PICTURE *PICT);
int PARFIELD(struct PICTURE *PICT);
int CIRFIELD(struct PICTURE *PICT);
int CIRFIELDCAL(struct PICTURE *PICT);
long CALCFIELD1(unsigned long *L,long szx,long szy,long x,long y);
void CALCFIELD2(unsigned long *L1,unsigned long *L2,
                long szx,long szy,long x,long y);
int  CALCGRAD(unsigned long *L,long szx,long szy, long *x, long *y,
              char *picptr,long iter,long *color,double neta);
int CALCGRAD2(unsigned long *L,long szx, long szy, long *x, long *y,
              char *clrptr,long iter,long *colorcounter,double neta);
int  CONVERTPIC1(unsigned long *L,struct PICTURE *PICT,long szx,long szy);
int  CONVERTPIC2(struct PICTURE *PICT,long szx,long szy,char *clrptr);
int REDUCE(struct PICTURE *PICT);
double ran2(long *num);
void ANIMATE(void);
void sort2(long num, short *ra,long *rb);
void SETITER(void);
extern void DRAW_MESSAGE(int position, char *msg);
extern void GET_MESSAGE(int posn, char *buffer);
extern void PIC_DRAW(int side,struct PICTURE *DISP);

/*=========================================================================*/
/*            Tree Growth Simulator, Gauss-Siedel iteration                */
/*      This routine simulates tree growth with an x-y grid of sx by sy    */
/*      points.  This uses parallel electrodes on the top (maxv) and       */
/*      bottom of the array (0v) and uses the Laplace equation for         */
/*      field calculations.                                                */
/*=========================================================================*/

int PARTREESIM1(struct PICTURE *PICT)
{
   unsigned long *L;
   unsigned v;
   long i,j,offset,iter;
   long  x,y,szx,szy,maxdev;
```

```
    long color;
    char msg[60],buffer[60],c;
    char *picptr,*clrptr;
    double tmp,neta;
    int tst;

/*                      Get data from user                                  */

    DRAW_MESSAGE(2,"Enter X size of array (int) :");
    GET_MESSAGE(2,buffer);
    szx=atoi(buffer);
    DRAW_MESSAGE(2,"Enter Y size of array (int) :");
    GET_MESSAGE(2,buffer);
    szy=atoi(buffer);
    DRAW_MESSAGE(2,"Number of iterations (int) :");
    GET_MESSAGE(2,buffer);
    iter=atoi(buffer);
    DRAW_MESSAGE(2,"Exponent value (double, 1.0) :");
    GET_MESSAGE(2,buffer);
    neta=atof(buffer);
    if(neta==0) neta=1.0;

/*                      Erase previous pic if required                      */

   if (PICT->valid_pic==1) {            /* ==1 if previous pic created  */
      free(PICT->pic_buf);              /* free previous image          */
   }

/*              Allocate new picture and color arrays                       */

   i=szx*szy;                           /* size of new image            */
   if((PICT->pic_buf=(char *)malloc(i))==NULL)
   {                                    /* memory allocation error      */
      return(0);                        /* return with error flag       */
   }
   picptr=PICT->pic_buf;                /* save picture location        */
   memset(picptr,0,i);                  /* clear memory                 */
   if((clrptr=(char *)malloc(i))==NULL)
   {                                    /* memory allocation error      */
      return(0);                        /* return with error flag       */
   }
   memset(clrptr,0,i);                  /* clear memory                 */

/*              allocate and initialize field array                         */

   tmp=ran2(idum);
   *idum=1;
   i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/

/*              initialize laplacian field array to values                  */

   DRAW_MESSAGE(1,"Initializing field array");
```

```
    offset=0;
    for(i=0;i<szy;i++)                          /* y loop                    */
    {
       v=(unsigned long)((double)maxv*(double)(szy-i-1)/(double)szy);
       for(j=0;j<szx;j++)
       {
          L[offset++]=v;                        /* set potential             */
       }
    }
    offset=(szx*szy)-szx/2-szx;                 /* position of start value   */
    L[offset]=0;                                /* starting point            */
    clrptr[offset]=255;                         /* white starting point      */
    x=szx/2;
    y=szy-1;                                    /* starting point coordinates */

/*                        Main Iteration Loop                               */

    DRAW_MESSAGE(1,"Beginning iterations");
    DRAW_MESSAGE(2,"1) Performing G.S. iteration.  Press ESC to end.");
    color=0;                                    /* reset color counter       */
    for(i=0;i<(iter/100);i++)
    {
       for(j=0;j<100;j++)
       {
          maxdev=CALCFIELD1(L,szx,szy,x,y); /* laplacian field calculation */
          tst=CALCGRAD(L,szx,szy,&x,&y,clrptr,iter,&color,neta); /* find new
branch */
          if(kbhit()!=0 || tst==1)
          {
             if(tst==1)                         /* end of simulation         */
             {
                CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L);                        /* free laplacian field array */
                return(1);
             }
             else if((c=getch())==27)           /* operator intervention     */
             {
                CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L);                        /* free laplacian field array */
                return(1);
             }
          }
       }
       CONVERTPIC1(L,PICT,szx,szy);             /* convert array to grays    */
       PIC_DRAW(1,PICT);                        /* draw intermediate pictures */
       tmp=(double)maxdev/(double)maxv;
       sprintf(msg,"%d iterations. Field err= %6.1e",i*100+j,tmp);
       DRAW_MESSAGE(1,msg);
    }
    CONVERTPIC2(PICT,szx,szy,clrptr);           /* convert array to picture  */
    free(clrptr);
    free(L);                                    /* free laplacian field array */
```

```
   return(1);                                  /* return with success          */
}

/*=========================================================================*/
/*         Laplacian Field Calculation Routine, Gauss-Seidel               */
/*=========================================================================*/

long CALCFIELD1(unsigned long *L,long szx,long szy,long x,long y)
{
   long offset1,offset2,tmpoffset,gx,gy,maxdev;
   long i,j,k,x0,y0,x1,y1,a,b,c,d;

/*                       local field iteration loops                       */

   x0=max(1,x-25);
   x1=min(szx-1,x+25);
   y0=max(1,y-25);
   y1=min(szy-1,y+25);                    /* determine local field box     */
   tmpoffset=y0*szx;
   for(i=0;i<localiter;i++)               /* local field iteration loop    */
   {
      offset1=tmpoffset;
      for(j=y0;j<y1;j++)                  /* local field y loop            */
      {
         for(k=x0;k<x1;k++)               /* local field x loop            */
         {
            offset2=offset1+k;
            if(L[offset2]!=0 && L[offset2]!=maxv)
            {
               L[offset2]=(L[offset2-1]+L[offset2+1]+
                           L[offset2-szx]+L[offset2+szx])>>2;
            }
         }
         offset1+=szx;
      }
   }

/*                       Global Iteration Loop                             */

   gx=szx-1;
   gy=szy-1;
   for(i=0;i<globaliter-1;i++)            /* global field iteration loop  */
   {
      offset1=szx;
      for(j=1;j<gy;j++)                   /* global field y loop          */
      {
         L[offset1]=(L[offset1+1]+L[offset1+1]+
                     L[offset1-szx]+L[offset1+szx])>>2;
         for(k=1;k<gx;k++)                /* global field x loop          */
         {
            offset2=offset1+k;
            if(L[offset2]!=0 && L[offset2]!=maxv)
            {
               L[offset2]=(L[offset2-1]+L[offset2+1]+
```

```
                        L[offset2-szx]+L[offset2+szx])>>2;
                }
            }
            offset1=offset1+szx-1;
            L[offset1]=(L[offset2-1]+L[offset2-1]+
                       L[offset2-szx]+L[offset2+szx])>>2;
            offset1++;
        }
    }
    maxdev=0;                              /* reset max deviation          */
    offset1=szx;
    for(j=1;j<gy;j++)                      /* global field y loop          */
    {
          L[offset1]=(L[offset1+1]+L[offset1+1]+
                     L[offset1-szx]+L[offset1+szx])>>2;
          for(k=1;k<gx;k++)                /* global field x loop          */
          {
             offset2=offset1+k;
             if(L[offset2]!=0 && L[offset2]!=maxv)
             {
                a=L[offset2];             /* get old value                */
                L[offset2]=(L[offset2-1]+L[offset2+1]+
                           L[offset2-szx]+L[offset2+szx])>>2; /* calc new */
                b=abs(a-L[offset2]);
                maxdev=max(maxdev,b);     /* find max deviation           */
             }
          }
          offset1=offset1+szx-1;
          L[offset1]=(L[offset2-1]+L[offset2-1]+
                     L[offset2-szx]+L[offset2+szx])>>2;
          offset1++;
       }
    return(maxdev);
}

/*=========================================================================*/
/*                  Calculate gradient and probability                     */
/*=========================================================================*/

int CALCGRAD(unsigned long *L,long szx, long szy, long *x, long *y,
             char *clrptr,long iter,long *colorcounter,double neta)
{
   long i,j,offset1,offset2,gx,gy,maxg,color;
   double total,tst,a,b,c,d;

/*              Determine color of new branch                              */

   color=255-(*colorcounter)*150/iter;
   *colorcounter=*colorcounter+1;

/*              Sum up all gradient values around branches                 */

   gx=szx-1;
   gy=szy-1;
```

```
    offset1=szx;
    total=0.0;
    if(neta==1.0)
    {
       for(i=1;i<gy;i++)                    /* main y loop                    */
       {
          for(j=1;j<gx;j++)                 /* main x loop                    */
          {
             offset2=offset1+j;
             if(clrptr[offset2]!=0)
             {
                maxg=L[offset2-1]+L[offset2+1]+L[offset2-szx]+L[offset2+szx];
                if(maxg>maxv) return(1); /* end of iterations              */
                total=(double)maxg+total;
             }
          }
          offset1+=szx;
       }
    }
    else
    {
       for(i=1;i<gy;i++)                    /* main y loop                    */
       {
          for(j=1;j<gx;j++)                 /* main x loop                    */
          {
             offset2=offset1+j;
             if(clrptr[offset2]!=0)
             {
                a=pow((double)L[offset2-1],neta);
                b=pow((double)L[offset2+1],neta);
                c=pow((double)L[offset2-szx],neta);
                d=pow((double)L[offset2+szx],neta);
                total=a+b+c+d+total;
                maxg=L[offset2-1]+L[offset2+1]+L[offset2-szx]+L[offset2+szx];
                if(maxg>maxv) return(1); /* end of iterations              */
             }
          }
          offset1+=szx;
       }
    }

/*              Use probability to determine which site extends           */

    offset1=szx;
    tst=ran2(idum)*total;
    total=0.0;
    if(neta==1.0)
    {
       for(i=1;i<gy;i++)                    /* main y loop                    */
       {
          for(j=1;j<gx;j++)                 /* main x loop                    */
          {
             offset2=offset1+j;
             if(clrptr[offset2]!=0)
```

```
            {
               total=(double)L[offset2-1]+total;
               if(total>=tst)
               {
                  *x=j;
                  *y=i;
                  L[offset2-1]=0;
                  clrptr[offset2-1]=color;
                  return(0);
               }
               total=(double)L[offset2+1]+total;
               if(total>=tst)
               {
                  *x=j;
                  *y=i;
                  L[offset2+1]=0;
                  clrptr[offset2+1]=color;
                  return(0);
               }
               total=(double)L[offset2-szx]+total;
               if(total>=tst)
               {
                  *x=j;
                  *y=i;
                  L[offset2-szx]=0;
                  clrptr[offset2-szx]=color;
                  return(0);
               }
               total=(double)L[offset2+szx]+total;
               if(total>=tst)
               {
                  *x=j;
                  *y=i;
                  L[offset2+szx]=0;
                  clrptr[offset2+szx]=color;
                  return(0);
               }
            }
         }
         offset1+=szx;
      }
   }
   else
   {
      for(i=1;i<gy;i++)                    /* main y loop                    */
      {
         for(j=1;j<gx;j++)                 /* main x loop                    */
         {
            offset2=offset1+j;
            if(clrptr[offset2]!=0)
            {
               if(L[offset2-1]!=0) total=pow((double)L[offset2-1],neta)+total;
               if(total>=tst)
               {
```

```
                    *x=j;
                    *y=i;
                    L[offset2-1]=0;
                    clrptr[offset2-1]=color;
                    return(0);
                 }
                 if(L[offset2+1]!=0) total=pow((double)L[offset2+1],neta)+total;
                 if(total>=tst)
                 {
                    *x=j;
                    *y=i;
                    L[offset2+1]=0;
                    clrptr[offset2+1]=color;
                    return(0);
                 }
                 if(L[offset2-szx]!=0) total=pow((double)L[offset2-
 szx],neta)+total;
                 if(total>=tst)
                 {
                    *x=j;
                    *y=i;
                    L[offset2-szx]=0;
                    clrptr[offset2-szx]=color;
                    return(0);
                 }
                 if(L[offset2+szx]!=0) total=pow((dou-
 ble)L[offset2+szx],neta)+total;
                 if(total>=tst)
                 {
                    *x=j;
                    *y=i;
                    L[offset2+szx]=0;
                    clrptr[offset2+szx]=color;
                    return(0);
                 }
              }
           }
           offset1+=szx;
        }
     }
     return(0);
}

/*=========================================================================*/
/*          Convert Laplacian array to picture array                       */
/*=========================================================================*/

int CONVERTPIC1(unsigned long *L,struct PICTURE *PICT,long szx,long szy)
{
   long i,j,offset1;
   char *picptr;

   picptr=PICT->pic_buf;                  /* get address of picture       */
```

```
 /*                    Transfer picture from laplacian array                */

    offset1=0;
    for(i=0;i<(szy-1);i++)
    {
       for(j=0;j<szx;j++)
       {
          if(L[offset1]==0) picptr[offset1]=255;   /* white pixel values   */
          else picptr[offset1]=L[offset1]/(unsigned long)392157; /* gray pixel
 values */
          offset1++;
       }
    }
    for (j=0;j<szx;j++) picptr[offset1++]=0;      /* blacken bottom line  */
    PICT->imagewidth=szx;
    PICT->imagelength=szy;                        /* indicate size        */
    PICT->valid_pic=1;                            /* signal valid picture */
    return(1);
 }

 /*=========================================================================*/
 /*           Convert Color array to picture array                          */
 /*=========================================================================*/

 int CONVERTPIC2(struct PICTURE *PICT,long szx,long szy,char *clrptr)
 {
    long i,j,offset1;
    char *picptr;

    picptr=PICT->pic_buf;                 /* get address of picture       */

 /*                    Transfer picture from laplacian array                */

    offset1=0;
    for(i=0;i<(szy-1);i++)
    {
       for(j=0;j<szx;j++)
       {
          picptr[offset1]=clrptr[offset1];
          offset1++;
       }
    }
    PICT->imagewidth=szx;
    PICT->imagelength=szy;                        /* indicate size        */
    PICT->valid_pic=1;                            /* signal valid picture */
    return(1);
 }

 /*=========================================================================*/
 /*      Random number generator from "Numerical Recipies in C"             */
 /*=========================================================================*/

 #define M 714025
 #define IA 1366
```

```
#define IC 150889

double ran2(long *idum)
{
   static long iy,ir[98];
   static int iff=0;
   int j;

   if(*idum<0 || iff==0)
   {
      iff=1;
      if((*idum=(IC-(*idum))%M) < 0) *idum=-(*idum);
      for(j=1;j<97;j++)                    /* initialize table shuffle     */
      {
         *idum=(IA*(*idum)+IC)%M;
         ir[j]=(*idum);
      }
      *idum=(IA*(*idum)+IC)%M;
      iy=(*idum);
   }
   j=1+97.0*iy/M;
   if(j>97 || j<1) DRAW_MESSAGE(2,"RAND2: this cannot happen");
   iy=ir[j];
   *idum=(IA*(*idum)+IC)%M;
   ir[j]=(*idum);
   return ((double)iy/(double)M);
}

/*=========================================================================*/
/*            Tree Growth Simulator, Jacobi iteration                      */
/*      This routine simulates tree growth with an x-y grid of sx by sy    */
/*      points.  This uses parallel electrodes on the top (maxv) and       */
/*      bottom of the array (0v) and uses the Laplace equation for         */
/*      field calculations.                                                */
/*=========================================================================*/

int PARTREESIM2(struct PICTURE *PICT)
{
   unsigned long *L1,*L2;
   unsigned v;
   long i,j,offset,iter;
   long  x,y,szx,szy;
   long color;
   char msg[60],buffer[60],c;
   char *picptr,*clrptr;
   double tmp,neta;
   int tst;

/*                      Get data from user                                 */

   DRAW_MESSAGE(2,"Enter X size of array (int) :");
   GET_MESSAGE(2,buffer);
   szx=atoi(buffer);
   DRAW_MESSAGE(2,"Enter Y size of array (int) :");
```

```
    GET_MESSAGE(2,buffer);
    szy=atoi(buffer);
    DRAW_MESSAGE(2,"Number of iterations (int) :");
    GET_MESSAGE(2,buffer);
    iter=atoi(buffer);
    DRAW_MESSAGE(2,"Exponent value (double, 1.0) :");
    GET_MESSAGE(2,buffer);
    neta=atof(buffer);
    if(neta==0) neta=1.0;

/*                      Erase previous pic if required                  */

   if (PICT->valid_pic==1) {          /* ==1 if previous pic created  */
      free(PICT->pic_buf);            /* free previous image          */
   }

/*              Allocate new picture and color arras                    */

   i=szx*szy;                         /* size of new image            */
   if((PICT->pic_buf=(char *)malloc(i))==NULL)
   {                                  /* memory allocation error      */
      return(0);                      /* return with error flag       */
   }
   picptr=PICT->pic_buf;              /* save picture location        */
   memset(picptr,0,i);                /* clear memory                 */
   if((clrptr=(char *)malloc(i))==NULL)
   {                                  /* memory allocation error      */
      return(0);                      /* return with error flag       */
   }
   memset(clrptr,0,i);                /* clear memory                 */

/*              allocate and initialize field array                     */

   *idum=-1;
   tmp=ran2(idum);
   *idum=1;
   i=szx*szy*4;
  if((L1=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/
  if((L2=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/

/*              initialize laplacian field array to values              */

   DRAW_MESSAGE(1,"Initializing field array");
   color=0;
   offset=0;
   for(i=0;i<szy;i++)                 /* y loop                       */
   {
      v=(unsigned long)((double)maxv*(double)(szy-i-1)/(double)szy);
      for(j=0;j<szx;j++)
      {
         L1[offset]=v;                /* set potential                */
         L2[offset++]=v;
```

```
      }
    }
    offset=(szx*szy)-szx/2-szx;          /* position of start value      */
    L1[offset]=0;                        /* starting point               */
    clrptr[offset]=255;                  /* white starting point         */
    x=szx/2;
    y=szy-1;                             /* starting point coordinates   */

 /*                      Main Iteration Loop                             */

    DRAW_MESSAGE(1,"Beginning iterations");
    DRAW_MESSAGE(2,"2) Performing Jacobi iteration.  Press ESC to end.");
    for(i=0;i<(iter/100);i++)
    {
       for(j=0;j<100;j++)
       {
          CALCFIELD2(L1,L2,szx,szy,x,y); /* laplacian field calculation  */
          tst=CALCGRAD(L1,szx,szy,&x,&y,clrptr,iter,&color,neta); /* find new
 branch */
          if(kbhit()!=0 || tst==1)
          {
             if(tst==1)                  /* end of simulation            */
             {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L1);                /* free laplacian field array   */
                free(L2);                /* free laplacian field array   */
                return(1);
             }
             else if((c=getch())==27)    /* operator intervention        */
             {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L1);                /* free laplacian field array   */
                free(L2);                /* free laplacian field array   */
                return(1);
             }
          }
       }
       CONVERTPIC1(L1,PICT,szx,szy);     /* convert array to grays       */
       PIC_DRAW(1,PICT);                 /* draw intermediate pictures   */
       sprintf(msg,"Results after %d iter.",i*100+j);
       DRAW_MESSAGE(1,msg);
    }
    CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture     */
    free(L1);                         /* free laplacian field array   */
    free(L2);                         /* free laplacian field array   */
    free(clrptr);
    return(1);                        /* return with success          */
}

/*=========================================================================*/
/*       Laplacian Field Calculation Routine, Jacobi                       */
/*=========================================================================*/
```

```
void CALCFIELD2(unsigned long *L1,unsigned long *L2,
                long szx,long szy,long x,long y)
{
   long offset1,offset2,tmpoffset,gx,gy;
   long i,j,k,x0,y0,x1,y1;
   unsigned long *src,*dest,*tmp;

/*                        local field iteration loops                    */

   src=L1;
   dest=L2;                                  /* set source and dest arrays   */
   memcpy(dest,src,sizeof(long)*szy*szx); /* make arrays equal before start */

   x0=max(1,x-25);
   x1=min(szx-1,x+25);
   y0=max(1,y-25);
   y1=min(szy-1,y+25);                       /* determine local field box    */
   tmpoffset=y0*szx;
   for(i=0;i<localiter;i++)                  /* local field iteration loop   */
   {
      offset1=tmpoffset;
      for(j=y0;j<y1;j++)                     /* local field y loop           */
      {
         for(k=x0;k<x1;k++)                  /* local field x loop           */
         {
            offset2=offset1+k;
            if(src[offset2]!=0 && src[offset2]!=maxv)
            {
               dest[offset2]=(src[offset2-1]+src[offset2+1]+
                              src[offset2-szx]+src[offset2+szx])>>2;
            }
            else
            {
               dest[offset2]=src[offset2];
            }
         }
         offset1+=szx;
      }
      tmp=src;
      src=dest;
      dest=tmp;                              /* swap buffers around          */
   }

/*                      Global Iteration Loop                            */

   gx=szx-1;
   gy=szy-1;
   for(i=0;i<globaliter;i++)                 /* global field iteration loop  */
   {
      offset1=szx;
      for(j=1;j<gy;j++)                      /* global field y loop          */
      {
         for(k=1;k<gx;k++)                   /* global field x loop          */
```

```
            {
               offset2=offset1+k;
               if(src[offset2]!=0 && src[offset2]!=maxv)
               {
                  dest[offset2]=(src[offset2-1]+src[offset2+1]+
                              src[offset2-szx]+src[offset2+szx])>>2;
               }
               else
               {
                  dest[offset2]=src[offset2];
               }
            }
            dest[offset1]=(src[offset2+1]+src[offset2+1]+
                           src[offset2-szx]+src[offset2+szx])>>2;
            offset1=offset1+szx-1;
            dest[offset1]=(src[offset2-1]+src[offset2-1]+
                           src[offset2-szx]+src[offset2+szx])>>2;
            offset1++;
         }
         tmp=src;
         src=dest;
         dest=tmp;                              /* swap buffers around          */
      }
      return;
   }

   /*=========================================================================*/
   /*            Tree Growth Simulator, Gauss-Siedel iteration                */
   /*      This routine simulates tree growth with an x-y grid of sx by sy    */
   /*      points.  This uses circular electrodes with the outside set to     */
   /*      maxv and the center set to 0v and uses the Laplace equation for    */
   /*      field calculations.                                                */
   /*=========================================================================*/

   int CIRTREESIM1(struct PICTURE *PICT)
   {
      unsigned long *L;
      unsigned v;
      long i,j,offset,iter,color,maxdev;
      long  x,y,szx,szy,ctrx,ctry,radius;
      char msg[60],buffer[60],c;
      char *picptr,*clrptr;
      double tmp,ratio,neta;
      double rmax,rmaxv;
      int tst;

   /*                      Get data from user                                 */

      DRAW_MESSAGE(2,"Enter diameter of array (int) :");
      GET_MESSAGE(2,buffer);
      szx=atoi(buffer);
      szy=szx;
      DRAW_MESSAGE(2,"Number of iterations (int) :");
      GET_MESSAGE(2,buffer);
```

```
    iter=atoi(buffer);
    DRAW_MESSAGE(2,"Exponent value (double, 1.0) :");
    GET_MESSAGE(2,buffer);
    neta=atof(buffer);
    if(neta==0) neta=1.0;

 /*                        Erase previous pic if required                    */

    if (PICT->valid_pic==1) {              /* ==1 if previous pic created  */
       free(PICT->pic_buf);                /* free previous image          */
    }
 /*                Allocate new picture and color arrays                     */

    i=szx*szy;                             /* size of new image            */
    if((PICT->pic_buf=(char *)malloc(i))==NULL)
    {                                      /* memory allocation error      */
       return(0);                          /* return with error flag       */
    }
    picptr=PICT->pic_buf;                  /* save picture location        */
    memset(picptr,0,i);                    /* clear memory                 */
    if((clrptr=(char *)malloc(i))==NULL)
    {                                      /* memory allocation error      */
       return(0);                          /* return with error flag       */
    }
    memset(clrptr,0,i);                    /* clear memory                 */

 /*                allocate and initialize field array                       */

    *idum=-1;
    tmp=ran2(idum);
    *idum=1;
    i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
 */

 /*                initialize laplacian field array to values                */

    DRAW_MESSAGE(1,"Initializing field array");
    ctrx=szx/2;
    ctry=szy/2;                            /* calculate center of array    */
    rmax=(double)(ctrx-2);
    rmaxv=(double)maxv;
    ratio=log(rmax);
    offset=0;
    for(i=0;i<szy;i++)                     /* y loop                       */
    {
       for(j=0;j<szx;j++)                  /* x loop                       */
       {
          tmp=(double)((i-ctrx)*(i-ctrx))+(double)((j-ctry)*(j-ctry));
          radius=(long)(sqrt(tmp));        /* calculate radius from center */
          if(radius>=ctrx) v=maxv;
          else
          {
             v=(long)((1.0-(log(rmax/(double)(radius))/ratio))*(double)maxv);
```

```
            }
            L[offset++]=v;                    /* set potential              */
         }
      }
      offset=ctrx+ctry*szx;                   /* position of start value    */
      L[offset]=0;                            /* starting point             */
      clrptr[offset]=255;                     /* white starting point       */
      x=ctrx;
      y=ctry;                                 /* starting point coordinates */

   /*                      Main Iteration Loop                               */

      DRAW_MESSAGE(1,"Beginning Iterations");
      DRAW_MESSAGE(2,"3) Performing G.S. iteration.  Press ESC to end.");
      color=0;
      for(i=0;i<(iter/100);i++)
      {
         for(j=0;j<100;j++)
         {
            maxdev=CALCFIELD1(L,szx,szy,x,y); /* laplacian field calculation */
            tst=CALCGRAD(L,szx,szy,&x,&y,clrptr,iter,&color,neta); /* find new
   branch */
            if(kbhit()!=0 || tst==1)
            {
               if(tst==1)                     /* end of simulation          */
               {
                  CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                  free(clrptr);
                  free(L);                    /* free laplacian field array */
                  return(1);
               }
               else if((c=getch())==27)       /* operator intervention      */
               {
                  CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                  free(clrptr);
                  free(L);                    /* free laplacian field array */
                  return(1);
               }
            }
         }
         CONVERTPIC1(L,PICT,szx,szy);         /* convert array to grays     */
         PIC_DRAW(1,PICT);                    /* draw intermediate pictures */
         tmp=(double)maxdev/(double)maxv;
         sprintf(msg,"%d iterations. Field err= %6.1e",i*100+j,tmp);
         DRAW_MESSAGE(1,msg);
      }
      CONVERTPIC2(PICT,szx,szy,clrptr);    /* convert array to picture     */
      free(clrptr);
      free(L);                             /* free laplacian field array   */
      return(1);                           /* return with success          */
   }

   /*=========================================================================*/
   /*            Tree Growth Simulator, Gauss-Siedel iteration                */
```

```
/*      This routine simulates tree growth with an x-y grid of sx by sy */
/*      points.  This uses circular electrodes with the outside set to  */
/*      maxv and the center set to 0v and uses the Laplace equation for */
/*      field calculations.  An inner radius of 10 % is used.           */
/*=======================================================================*/

int CIRTREESIM2(struct PICTURE *PICT)
{
   unsigned long *L;
   unsigned v;
   long i,j,offset,iter,color,rmin;
   long  x,y,szx,szy,ctrx,ctry,radius,maxdev;
   char msg[60],buffer[60],c;
   char *picptr,*clrptr;
   double tmp,ratio,neta;
   double rmax,rmaxv;
   int tst;

/*                        Get data from user                             */

   DRAW_MESSAGE(2,"Enter diameter of array (int) :");
   GET_MESSAGE(2,buffer);
   szx=atoi(buffer);
   szy=szx;
   DRAW_MESSAGE(2,"Number of iterations (int) :");
   GET_MESSAGE(2,buffer);
   iter=atoi(buffer);
   DRAW_MESSAGE(2,"Exponent value (double, 1.0) :");
   GET_MESSAGE(2,buffer);
   neta=atof(buffer);
   if(neta==0) neta=1.0;

/*                      Erase previous pic if required                   */

   if (PICT->valid_pic==1) {            /* ==1 if previous pic created  */
      free(PICT->pic_buf);              /* free previous image          */
   }
/*              Allocate new picture and color arrays                    */

   i=szx*szy;                           /* size of new image            */
   if((PICT->pic_buf=(char *)malloc(i))==NULL)
   {                                    /* memory allocation error      */
      return(0);                        /* return with error flag       */
   }
   picptr=PICT->pic_buf;                /* save picture location        */
   memset(picptr,0,i);                  /* clear memory                 */
   if((clrptr=(char *)malloc(i))==NULL)
   {                                    /* memory allocation error      */
      return(0);                        /* return with error flag       */
   }
   memset(clrptr,0,i);                  /* clear memory                 */

/*              allocate and initialize field array                      */
```

```
    *idum=-1;
    tmp=ran2(idum);
    *idum=1;
    i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/

 /*              initialize laplacian field array to values                */

    DRAW_MESSAGE(1,"Initializing field array");
    ctrx=szx/2;
    ctry=szy/2;                              /* calculate center of array    */
    rmax=(double)(ctrx-2);
    rmaxv=(double)maxv;
    rmin=(ctrx-2)/10;
    ratio=log(rmax/(double)rmin);
    offset=0;
    for(i=0;i<szy;i++)                       /* y loop                       */
    {
       for(j=0;j<szx;j++)                    /* x loop                       */
       {
          tmp=(double)((i-ctrx)*(i-ctrx))+(double)((j-ctry)*(j-ctry));
         radius=(long)sqrt(tmp);                 /* calculate radius from center */
          if(radius>=ctrx) v=maxv;
          else if(radius==rmin)
          {
             v=0;
             clrptr[offset]=255;             /* white starting points        */
          }
          else
          {
             v=(long)((1.0-(log(rmax/(double)radius)/ratio))*(double)maxv);
          }
          L[offset++]=v;                     /* set potential                */
       }
    }
    x=ctrx;
    y=ctry;                                  /* starting point coordinates   */

/*                        Main Iteration Loop                               */

    DRAW_MESSAGE(1,"Beginning Iterations");
    DRAW_MESSAGE(2,"4) Performing G.S. iteration.  Press ESC to end.");
    color=0;
    for(i=0;i<(iter/100);i++)
    {
       for(j=0;j<100;j++)
       {
          maxdev=CALCFIELD1(L,szx,szy,x,y); /* laplacian field calculation */
          tst=CALCGRAD(L,szx,szy,&x,&y,clrptr,iter,&color,neta); /* find new
branch */
          if(kbhit()!=0 || tst==1)
          {
             if(tst==1)                      /* end of simulation            */
```

```
            {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
               free(clrptr);
               free(L);                    /* free laplacian field array   */
               return(1);
            }
            else if((c=getch())==27)     /* operator intervention        */
            {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
               free(clrptr);
               free(L);                    /* free laplacian field array   */
               return(1);
            }
         }
      }
      CONVERTPIC1(L,PICT,szx,szy);        /* convert array to grays       */
      PIC_DRAW(1,PICT);                   /* draw intermediate pictures   */
      tmp=(double)maxdev/(double)maxv;
      sprintf(msg,"%d iterations. Field err= %6.1e",i*100+j,tmp);
      DRAW_MESSAGE(1,msg);
   }
   CONVERTPIC2(PICT,szx,szy,clrptr);   /* convert array to picture     */
   free(clrptr);
   free(L);                             /* free laplacian field array   */
   return(1);                           /* return with success          */
}

/*=========================================================================*/
/*                              ANIMATE                                    */
/*      This routine takes the image in memory and displays it on the      */
/*      left hand side of the screen.  After display, the imgptr buffer    */
/*      used is then converted into a pixel array with the brightest       */
/*      pixels first and the darkest last.  These pixels are then sent     */
/*      to the screen in this order to animate the picture.                */
/*=========================================================================*/

extern char *imgptr;
extern struct PICTURE PIC;
void ANIMATE(void)
{
   long i,j,k,num,d;
   char c;
   long offset,xoffset,yoffset,a,b,tmp;
   long poffset,boffset,m,ypos,xpos;
   char *n;
   char *pic_addr;                      /* address of data              */
   short *anbuf1,*anbuf2;               /* sort arrays                  */
   long  *anbuf3;                       /* mirror array                 */

/*          check if picture exists and then write to left side        */

   if (PIC.valid_pic==1)
   {
      _setviewport(0,0,1023,767);       /* set viewport to max          */
```

```
        m=PIC.imagewidth;
        n=PIC.pic_buf;
        k=0;
        imgptr[0]=254;
        imgptr[1]=1;
        imgptr[2]=254;
        imgptr[3]=1;
        imgptr[4]=8;
        imgptr[5]=0;                          /* init array by hand          */
        a=PIC.imagewidth;
        b=PIC.imagelength;

/*              Picture is smaller than display buffer                       */

        if(a<510 || b<510)
        {
           for (i=0;i<510;i++)
           {                                  /* start of y loop             */
              ypos=(i*b)/510;                 /* calc y position             */
              tmp=ypos*a;
              for (j=0;j<510;j++)
              {                               /* start of x loop             */
                 xpos=(j*a)/510;              /* calc x position             */
                 poffset=tmp+xpos;            /* calc offset into pic array  */
                 pic_addr=n+poffset;          /* calculate address of pixel  */
                 imgptr[k+6]=(*pic_addr);     /* transfer full pixel         */
                 k++;
              }
           }

        }

/*                  Picture is larger than buffer                            */

        else
        {
           memset(imgptr+6,0,260100);         /* clear buffer memory         */
           for (i=0;i<b;i++)
           {                                  /* start of y loop             */
              tmp=i*a;
              ypos=(i*510)/b;                 /* calc y position             */
              for (j=0;j<a;j++)
              {                               /* start of x loop             */
                 xpos=(j*510)/a;              /* calc x position             */
                 boffset=ypos*510+xpos+6;     /* buffer offset               */
                 poffset=tmp+j;               /* picture offset              */
                 pic_addr=n+poffset;          /* calculate address of pixel  */
                 c=(*pic_addr);               /* calc new pixel              */
                 imgptr[boffset]=max(imgptr[boffset],c); /* transfer pixel   */
              }
           }
        }
     }
     else
```

```
    {
       DRAW_MESSAGE(2,"No picture in memory to animate.  Press key to con-
tinue");
       d=getch();                                    /* wait for key press   */
       return;
    }

/*                count number of non black pixels                          */

    num=0;
    for(i=6;i<260106;i++)
    {
       if(imgptr[i]!=0)
       {
          num++;
       }
    }

/*       Allocate arrays for animation of image                             */

    if((anbuf1=(short *)malloc(num*2+6))==NULL) /* allocate intensity array */
    {                                          /* memory allocation error      */
       DRAW_MESSAGE(2,"Not enough memory to animate.  Press key to continue");
       d=getch();                                    /* wait for key press   */
       return;
    }
    if((anbuf2=(short *)malloc(num*4+12))==NULL)
    {                                          /* memory allocation error      */
       DRAW_MESSAGE(2,"Not enough memory to animate.  Press key to continue");
       d=getch();                                    /* wait for key press   */
       free(anbuf1);
       return;
    }
    anbuf3=(long *)anbuf2;                        /* point to same array  */

/*       Move all non black pixels into animation buffer                    */

    DRAW_MESSAGE(2,"Creating animation array.");
    offset=6;                                  /* offset into imgptr buffer    */
    k=1;                                       /* offset into animation array  */
    for(i=0;i<510;i++)                         /* y loop                       */
    {
       for(j=0;j<510;j++)                      /* x loop                       */
       {
          if(imgptr[offset]!=0)
          {
             anbuf1[k]=imgptr[offset];         /* save intensity               */
             anbuf2[k+k]=j;                    /* save x coordinate            */
             anbuf2[k+k+1]=i;                  /* save y coordinate            */
             k++;                              /* point to next location       */
          }
          offset++;
       }
    }
```

```
/*      Sort the animation buffer according to intensity of pixel       */

   DRAW_MESSAGE(2,"Sorting animation array.");
   sort2(num,anbuf1,anbuf3);              /* use heapsort to sort arrays  */

/*      Convert the color of all pixels to white                        */

   for(i=1;i<=num;i++)
   {
      anbuf1[i]=255;
   }

/*      Animate result by putting each pixel onto the screen            */

   DRAW_MESSAGE(2,"Performing animation.  Press ESC to end.");
   xoffset=513;
   yoffset=1;
   _setviewport(xoffset,yoffset,xoffset+509,yoffset+509); /* set viewport */
   while(1)
   {
      _clearscreen(_GVIEWPORT);          /* clear old picture            */
      delay(500);                        /* wait 1/2 second!             */
      j=0;                               /* offset counter into array    */
      for(i=1;i<=num;i++)
      {
         if(kbhit()!=0)
         {
            if((c=getch())==27)
            {
               _clearscreen(_GVIEWPORT); /* clear old picture            */
               free(anbuf2);
               free(anbuf1);
               return;                   /* end of animation             */
            }
         }
         _setcolor(anbuf1[j]);           /* set color of pixel           */
         _setpixel(anbuf2[j+j],anbuf2[j+j+1]);
         j++;
      }
   }
   return;
}

/*              Heapsort routine from "Numerical Recipes in C"          */
/*      Routine modified to sort largest to smallest with two arrays    */

void sort2(long n,short *ra,long *rb)
{
   long l,j,ir,i;
   short rra;
   long rrb;

   l=(n>>1)+1;
```

```
      ir=n;
      for(;;)
      {
         if(l>1)
         {
            rra=ra[--l];
            rrb=rb[l];
         }
         else
         {
            rra=ra[ir];
            rrb=rb[ir];
            ra[ir]=ra[1];
            rb[ir]=rb[1];
            if(--ir == 1)
            {
               ra[1]=rra;
               rb[1]=rrb;
               return;
            }
         }
         i=l;
         j=l<<1;
         while(j<=ir)
         {
            if(j<ir && ra[j]>ra[j+1]) ++j;
            if(rra>ra[j])
            {
               ra[i]=ra[j];
               rb[i]=rb[j];
               j+=(i=j);
            }
            else j=ir+1;
         }
         ra[i]=rra;
         rb[i]=rrb;
      }
   }

/*=========================================================================*/
/*                                CIRFIELDCAL                              */
/*      This routine takes the image in memory and calculates the          */
/*      Laplacian field distribution around the image.  The input           */
/*      image must be white on black for the process to work.  The          */
/*      image is first placed into an array whose size is specified        */
/*      by the user.  No checks are done on the image to ensure that       */
/*      the image fits into this array!                                    */
/*=========================================================================*/

int CIRFIELDCAL(struct PICTURE *PICT)
{
   unsigned long *L;
   unsigned v;
   long i,j,k,offset,iter,color,rmin,imszx,imszy,gx,gy;
```

```
    long x,y,szx,szy,ctrx,ctry,radius,startx,starty,r,rmax;
    long offset1,offset2;
    char msg[60],buffer[60],c;
    char *picptr,*out_pic;
    double tmp,ratio;
    int tst;

/*                      Check if pic is available                          */

    if (PICT->valid_pic!=1) {           /* ==1 if pic available           */
       return(0);                       /* return with error flag         */
    }

/*                      Get data from user                                 */

    picptr=PICT->pic_buf;               /* get address of image           */
    imszx=PICT->imagewidth;
    imszy=PICT->imagelength;            /* get size of image              */
    DRAW_MESSAGE(2,"Enter array size (int>image x,y):");
    GET_MESSAGE(2,buffer);
    szx=max(imszx,atoi(buffer));
    szy=szx;
    DRAW_MESSAGE(2,"Field iterations (int > 1000):");
    GET_MESSAGE(2,buffer);
    iter=max(10,atoi(buffer));

/*              Allocate field array and output array                      */

    i=szx*szy*sizeof(char);
    if((out_pic=(char *)malloc(i))==NULL)
    {                                   /* memory allocation error        */
       return(0);                       /* return with error flag         */
    }
    memset(out_pic,0,i);                /* set memory                     */
    i=szx*szy*sizeof(long);             /* size of field image            */
    if((L=(unsigned long *)malloc(i))==NULL)
    {                                   /* memory allocation error        */
       return(0);                       /* return with error flag         */
    }
    memset(L,100,i);                    /* set memory non zero            */

/*              place image into field array                               */

    startx=(szx-imszx)/2;
    starty=(szy-imszy)/2;               /* calc start of image in array   */
    for(i=0;i<imszy;i++)                /* y transfer loop                */
    {
       for(j=0;j<imszx;j++)             /* x transfer loop                */
       {
          k=i*imszx+j;                  /* position in picture array      */
          if(picptr[k]!=0)
          {
             k=(i+startx)*szx+j+starty; /* offset into result arrays      */
             L[k]=0;                    /* inner Dirichlet boundary       */
```

```
        }
      }
    }
    x=szx/2-1;
    y=szy/2-1;
    rmax=(long) (x-1);
    for(i=0;i<szy;i++)
    {
       for(j=0;j<szx;j++)
       {
          r=(long)(sqrt( (double)((i-x)*(i-x))
                        +(double)((j-y)*(j-y)) ) ); /* calc radius  */
          if(r>=rmax)
          {
             k=i*szx+j;                    /* calc offset into array       */
             L[k]=maxv;                    /* outer Dirichlet boundary     */
          }
       }
    }


/*              Calculate laplacian field array                            */

   DRAW_MESSAGE(1,"Calculating field array");
   gx=szx-1;
   gy=szy-1;
   for(i=0;i<iter;i++)
   {

/*                      Global Field Iteration Loop                        */

      offset1=szx;
      for(j=1;j<gy;j++)                  /* global field y loop          */
      {
         L[offset1]=(L[offset1+1]+L[offset1+1]+
                     L[offset1-szx]+L[offset1+szx])>>2;
         for(k=1;k<gx;k++)               /* global field x loop          */
         {
            offset2=offset1+k;
            if(L[offset2]!=0 && L[offset2]!=maxv)
            {
               L[offset2]=(L[offset2-1]+L[offset2+1]+
                           L[offset2-szx]+L[offset2+szx])>>2;
            }
         }
         offset1=offset1+szx-1;
         L[offset1]=(L[offset2-1]+L[offset2-1]+
                     L[offset2-szx]+L[offset2+szx])>>2;
         offset1++;
      }
   }
   free(PICT->pic_buf);                  /* free old picture             */
   PICT->pic_buf=out_pic;
   CONVERTPIC1(L,PICT,szx,szy);          /* convert array to picture     */
```

```
    PIC_DRAW(1,PICT);                          /* draw picture                */
    sprintf(msg,"Circular Field Calc. (G.S.), %4dx%4d",
            PICT->imagewidth,PICT->imagelength);
    DRAW_MESSAGE(1,msg);
    free(L);                                   /* free laplacian field array  */
    return(1);                                 /* return with success         */
}

/*=========================================================================*/
/*                              REDUCE                                     */
/*      This routine takes the image in memory and reduces it by the       */
/*      factor entered by the user.  Instead of decimating the image       */
/*      in the traditional sense, the image is reduced.  This is           */
/*      accomplished by testing if the image exists in the mesh size       */
/*      entered by the user.  This effectively tests the image at a        */
/*      scale in similar fashion to the test fractal generation.           */
/*=========================================================================*/

int REDUCE(struct PICTURE *PICT)
{
   long i,j,k,ii,jj,red,newx,newy,tst;
   long offset1,offset2,szx,szy;
   long poffset,boffset,m,ypos,xpos;
   char *picptr;                               /* address of picture          */
   char *outbuf;                               /* address of result           */
   char buffer[40];

/*      Ask the user for the reduction number                                */

   picptr=PICT->pic_buf;                       /* get address of image        */
   szx=PICT->imagewidth;
   szy=PICT->imagelength;                      /* get size of image           */
   i=0;
   while(i==0)
   {
      DRAW_MESSAGE(2,"Enter Reduction (2,3,4):");
      GET_MESSAGE(2,buffer);
      if((red=atoi(buffer))>1 && red<5) i=1;
   }

/*      Allocate array for reduction result                                  */

   newx=szx/red;
   newy=szy/red;
   i=newx*newy*sizeof(char);                   /* calculate size of result    */
   if((outbuf=(char *)malloc(i))==NULL)
   {                                           /* memory allocation error     */
      return(0);                               /* return with error flag      */
   }

/*      Reduce picture using reduction factor entered                        */

   for(i=0;i<newy;i++)                         /* picture y loop              */
   {
```

```
        for(j=0;j<newx;j++)                    /* picture x loop                */
        {
           tst=0;
           offset1=i*red*szx+j*red;            /* offset into original picture */
           for(ii=0;ii<red;ii++)               /* reduction y loop              */
           {
              offset2=offset1+ii*szx;          /* offset into box               */
              for(jj=0;jj<red;jj++)            /* reduction x loop              */
              {
                 tst=max(tst,picptr[offset2+jj]);
              }
           }
           k=i*newx+j;
           outbuf[k]=tst;
        }
     }
     free(picptr);                             /* free previous image           */
     PICT->pic_buf=outbuf;                     /* point to result               */
     PICT->imagewidth=newx;
     PICT->imagelength=newy;                   /* get size of image             */
     return(1);
  }

  /*=========================================================================*/
  /*            PARALLEL FIELD convergence test                              */
  /*      This routine initializes a field with MAXV potential at the        */
  /*      top and 0 potential at the bottom.  All the other values           */
  /*      in the field are then initialized to 0.  The field is then         */
  /*      calculated and results put to the screen to test field             */
  /*      convergence using Gauss Seidel iteration.                          */
  /*=========================================================================*/

  int PARFIELD(struct PICTURE *PICT)
  {
     unsigned long *L;
     unsigned v;
     long i,j,k,ii,jj,offset,offset1,offset2,iter;
     long  x,y,szx,szy,maxdev,gx,gy,a,b,oldmax,line;
     char msg[60],buffer[60],c;
     char *picptr;
     double tmp;

  /*                        Get data from user                               */

     DRAW_MESSAGE(2,"Enter X size of array (int) :");
     GET_MESSAGE(2,buffer);
     szx=atoi(buffer);
     DRAW_MESSAGE(2,"Enter Y size of array (int) :");
     GET_MESSAGE(2,buffer);
     szy=atoi(buffer);

  /*                        Erase previous pic if required                   */

     if (PICT->valid_pic==1) {              /* ==1 if previous pic created  */
```

```
      free(PICT->pic_buf);                    /* free previous image          */
   }

/*                Allocate new picture array                                   */

   i=szx*szy;                                 /* size of new image            */
   if((PICT->pic_buf=(char *)malloc(i))==NULL)
   {                                          /* memory allocation error      */
      return(0);                              /* return with error flag       */
   }
   picptr=PICT->pic_buf;                      /* save picture location        */
   memset(picptr,0,i);                        /* clear memory                 */

/*                allocate and initialize field array                          */

   i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/

/*                initialize laplacian field array to values                   */

   DRAW_MESSAGE(1,"Initializing field array");
   offset=0;
   for(i=0;i<szy;i++)
   {
      L[offset++]=maxv;                       /* top row to maxv              */
   }
   for(i=1;i<szy-1;i++)                       /* y loop                       */
   {
      for(j=0;j<szx;j++)                      /* x loop                       */
      {
         L[offset++]=0;                       /* set potential to 0           */
      }
   }
   for(i=0;i<szy;i++)                         /* bottom row to zero           */
   {
      L[offset++]=0;                          /* bottom row to zero           */
   }
   oldmax=100000000;

/*                        Main Iteration Loop                                  */

   DRAW_MESSAGE(1,"Beginning iterations");
   DRAW_MESSAGE(2,"8) Performing G.S. iteration.  Press ESC to end.");
   _setviewport(1,542,1022,739);              /* set viewport                 */
   _clearscreen(_GVIEWPORT);                  /* clear viewport               */
   ii=0;
   jj=0;
   line=0;
   while(ii!=-1)
   {
      ii++;
      jj++;
```

```
/*                      Global Field Iteration Loop                        */

      maxdev=0;
      gx=szx-1;
      gy=szy-1;
      offset1=szx;
      for(j=1;j<gy;j++)                 /* global field y loop          */
      {
         L[offset1]=(L[offset1+1]+L[offset1+1]+
                     L[offset1-szx]+L[offset1+szx])>>2;
         for(k=1;k<gx;k++)              /* global field x loop          */
         {
            offset2=offset1+k;
            a=L[offset2];               /* get old value                */
            L[offset2]=(L[offset2-1]+L[offset2+1]+
                        L[offset2-szx]+L[offset2+szx])>>2; /* calc new */
            b=abs(a-L[offset2]);
            maxdev=max(maxdev,b);       /* find max deviation           */
         }
         offset1=offset1+szx-1;
         L[offset1]=(L[offset2-1]+L[offset2-1]+
                     L[offset2-szx]+L[offset2+szx])>>2;
         offset1++;
      }

/*              End of global field iteration loop                        */
/*              Decide if it is time to output result                     */

      if(maxdev<oldmax)
      {
          _setviewport(1,542,1022,739); /* set viewport                   */
          tmp=(double)oldmax/(double)maxv;
          sprintf(msg,"%6d iterations. Field error now < %6.1e",ii,tmp);
          _grtext(5,2+line*24,msg);
          line++;
          oldmax=oldmax/10;             /* reduce sensitivity by 10       */
          if(oldmax<10)
          {
            DRAW_MESSAGE(2,"End of test. Press key to continue");
            c=getch();                  /* wait for key press             */
            free(L);                    /* free laplacian field array     */
            return(1);
          }
      }

      if(kbhit()!=0)
      {
         if((c=getch())==27)            /* operator intervention          */
         {
            free(L);                    /* free laplacian field array     */
            return(1);
         }
      }
```

```
      if(jj>299)
      {
         jj=0;
         CONVERTPIC1(L,PICT,szx,szy);       /* convert array to grays       */
         PIC_DRAW(1,PICT);                  /* draw intermediate pictures   */
         tmp=(double)maxdev/(double)maxv;
         sprintf(msg,"%d iterations. Field err= %6.1e",ii,tmp);
         DRAW_MESSAGE(1,msg);
      }
   }
   free(L);                                 /* free laplacian field array   */
   return(1);                               /* return with success          */
}

/*=========================================================================*/
/*                    CIRCULAR FIELD convergence test                      */
/*      This routine initializes a field with MAXV potential at the        */
/*      outer electrode and 0 potential at a point in the middle.  All     */
/*      the other values in the field are then initialized to 10.          */
/*      The field is then calculated and results put to the screen to      */
/*      test field convergence using Gauss Seidel iteration.               */
/*=========================================================================*/

int CIRFIELD(struct PICTURE *PICT)
{
   unsigned long *L;
   unsigned v;
   long i,j,k,ii,jj,offset,offset1,offset2,iter,rmax;
   long  x,y,szx,szy,maxdev,gx,gy,a,b,oldmax,line,r;
   char msg[60],buffer[60],c;
   char *picptr;
   double tmp;

/*                       Get data from user                                */

   DRAW_MESSAGE(2,"Enter diameter of array (int) :");
   GET_MESSAGE(2,buffer);
   szx=atoi(buffer);
   szy=szx;

/*                       Erase previous pic if required                    */

   if (PICT->valid_pic==1) {            /* ==1 if previous pic created  */
      free(PICT->pic_buf);              /* free previous image          */
   }

/*              Allocate new picture array                                 */

   i=szx*szy;                           /* size of new image            */
   if((PICT->pic_buf=(char *)malloc(i))==NULL)
   {                                    /* memory allocation error      */
      return(0);                        /* return with error flag       */
   }
   picptr=PICT->pic_buf;                /* save picture location        */
```

```
   memset(picptr,0,i);                          /* clear memory                */

/*              allocate and initialize field array                             */

   i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
*/

/*              initialize laplacian field array to values                      */

   DRAW_MESSAGE(1,"Initializing field array");
   x=szx/2;
   y=szy/2;                                     /* calculate center of field   */
   rmax=x-2;
   offset=0;
   for(i=0;i<szy;i++)                           /* y loop                      */
   {
      for(j=0;j<szx;j++)                        /* x loop                      */
      {
         r=(long)(sqrt((double)((x-j)*(x-j)+(y-i)*(y-i))));
         if(r>=rmax) L[offset++]=maxv;
         else L[offset++]=1000;                 /* set potential to 1000 <<maxv */
      }
   }
   offset=x+y*szx;
   L[offset++]=0;
   oldmax=100000000;

/*                      Main Iteration Loop                                     */

   DRAW_MESSAGE(1,"Beginning iterations");
   DRAW_MESSAGE(2,"8) Performing G.S. iteration.  Press ESC to end.");
   _setviewport(1,542,1022,739);                /* set viewport                */
   _clearscreen(_GVIEWPORT);                    /* clear viewport              */
   ii=0;
   jj=0;
   line=0;
   while(ii!=-1)
   {
      ii++;
      jj++;

/*                      Global Field Iteration Loop                             */

      maxdev=0;
      gx=szx-1;
      gy=szy-1;
      offset1=szx;
      for(j=1;j<gy;j++)                         /* global field y loop         */
      {
         L[offset1]=(L[offset1+1]+L[offset1+1]+
                     L[offset1-szx]+L[offset1+szx])>>2;
         for(k=1;k<gx;k++)                      /* global field x loop         */
         {
```

```
            offset2=offset1+k;
            if(L[offset2]!=0 && L[offset2]!=maxv)
            {
               a=L[offset2];          /* get old value                */
               L[offset2]=(L[offset2-1]+L[offset2+1]+
                          L[offset2-szx]+L[offset2+szx])>>2; /* calc new */
               b=abs(a-L[offset2]);
               maxdev=max(maxdev,b); /* find max deviation             */
            }
         }
         offset1=offset1+szx-1;
         L[offset1]=(L[offset2-1]+L[offset2-1]+
                     L[offset2-szx]+L[offset2+szx])>>2;
         offset1++;
      }

/*             End of global field iteration loop                      */
/*             Decide if it is time to output result                   */

      if(maxdev<oldmax)
      {
         _setviewport(1,542,1022,739); /* set viewport                 */
         tmp=(double)oldmax/(double)maxv;
         sprintf(msg,"%6d iterations. Field error now < %6.1e",ii,tmp);
         _grtext(5,2+line*24,msg);
         line++;
         oldmax=oldmax/10;             /* reduce sensitivity by 10     */
         if(oldmax<10)
         {
           DRAW_MESSAGE(2,"End of test. Press key to continue");
           c=getch();                  /* wait for key press           */
           free(L);                    /* free laplacian field array   */
           return(1);
         }
      }

      if(kbhit()!=0)
      {
         if((c=getch())==27)           /* operator intervention        */
         {
            free(L);                   /* free laplacian field array   */
            return(1);
         }
      }

      if(jj>299)
      {
         jj=0;
         CONVERTPIC1(L,PICT,szx,szy);      /* convert array to grays    */
         PIC_DRAW(1,PICT);                 /* draw intermediate pictures*/
         tmp=(double)maxdev/(double)maxv;
         sprintf(msg,"%d iterations. Field err= %6.1e",ii,tmp);
         DRAW_MESSAGE(1,msg);
      }
```

```
    }
    free(L);                                /* free laplacian field array   */
    return(1);                              /* return with success          */
}

/*=========================================================================*/
/*                        SETITER subroutine                               */
/*      This routine sets the number of iterations used in the field       */
/*      calculations for all of the simulations.  The values must be       */
/*      even so are made so.                                               */
/*=========================================================================*/

void SETITER(void)
{
   char buffer[80];
   long flag,tmp;

   flag=0;
   while(flag==0)
   {
      DRAW_MESSAGE(2,"Even # local field iter. (50): ");
      GET_MESSAGE(2,buffer);
      tmp=atoi(buffer);
      if(tmp==0) tmp=50;
      if(tmp>=2 && tmp<=1000)
      {
         localiter=(tmp/2)*2;           /* make even                    */
         flag=1;
      }
   }
   flag=0;
   while(flag==0)
   {
      DRAW_MESSAGE(2,"Even # of global field iter. (8): ");
      GET_MESSAGE(2,buffer);
      tmp=atoi(buffer);
      if(tmp==0) tmp=8;
      if(tmp>=2 && tmp<=1000)
      {
         globaliter=(tmp/2)*2;          /* make even                    */
         flag=1;
      }
   }
   sprintf(buffer,"%d local iterations. %d global iterations",localiter,glo-
baliter);
   DRAW_MESSAGE(2,buffer);
   delay(1500);
}

/*=========================================================================*/
/*                  Calculate gradient and probability 2                   */
/*=========================================================================*/

int CALCGRAD2(unsigned long *L,long szx, long szy, long *x, long *y,
```

```
              char *clrptr,long iter,long *colorcounter,double neta)
{
   long i,j,offset1,offset2,gx,gy,maxg,color,test;
   double total,tst,a,b,c,d;

   test=maxv/2;

/*              Determine color of new branch                          */

   color=255-(*colorcounter)*150/iter;
   *colorcounter=*colorcounter+1;

/*              Sum up all gradient values around branches             */

   gx=szx-1;
   gy=szy-1;
   offset1=szx;
   total=0.0;
   if(neta==1.0)
   {
      for(i=1;i<gy;i++)                  /* main y loop               */
      {
         for(j=1;j<gx;j++)               /* main x loop               */
         {
            offset2=offset1+j;
            if(clrptr[offset2]==0)
            {
               if(clrptr[offset2-1]!=0 || clrptr[offset2+1]!=0 ||
                  clrptr[offset2-szx]!=0 || clrptr[offset2+szx]!=0)
               {
                  maxg=L[offset2];
                  if(maxg>test) return(1); /* end of iterations       */
                  total=(double)maxg+total;
               }
            }
         }
         offset1+=szx;
      }
   }
   else
   {
      for(i=1;i<gy;i++)                  /* main y loop               */
      {
         for(j=1;j<gx;j++)               /* main x loop               */
         {
            offset2=offset1+j;
            if(clrptr[offset2]==0)
            {
               if(clrptr[offset2-1]!=0 || clrptr[offset2+1]!=0 ||
                  clrptr[offset2-szx]!=0 || clrptr[offset2+szx]!=0)
               {
                  maxg=L[offset2];
                  if(maxg>test) return(1); /* end of iterations       */
                  total=pow((double)maxg,neta)+total;
```

```
                }
            }
         }
         offset1+=szx;
      }
   }

/*              Use probability to determine which site extends          */

   offset1=szx;
   tst=ran2(idum)*total;
   total=0.0;
   if(neta==1.0)
   {
      for(i=1;i<gy;i++)                    /* main y loop                 */
      {
         for(j=1;j<gx;j++)                 /* main x loop                 */
         {
            offset2=offset1+j;
            if(clrptr[offset2]==0)
            {
               if(clrptr[offset2-1]!=0 || clrptr[offset2+1]!=0 ||
                  clrptr[offset2-szx]!=0 || clrptr[offset2+szx]!=0)
               {
                  maxg=L[offset2];
                  if(maxg>test) return(1); /* end of iterations          */
                  total=pow((double)maxg,neta)+total;
                  if(total>=tst)
                  {
                     *x=j;
                     *y=i;
                     L[offset2]=0;
                     clrptr[offset2]=color;
                     return(0);
                  }
               }
            }
         }
         offset1+=szx;
      }
   }
   else
   {
      for(i=1;i<gy;i++)                    /* main y loop                 */
      {
         for(j=1;j<gx;j++)                 /* main x loop                 */
         {
            offset2=offset1+j;
            if(clrptr[offset2]==0)
            {
               if(clrptr[offset2-1]!=0 || clrptr[offset2+1]!=0 ||
                  clrptr[offset2-szx]!=0 || clrptr[offset2+szx]!=0)
               {
                  maxg=L[offset2];
```

```
                    if(maxg>test) return(1); /* end of iterations          */
                    total=pow((double)maxg,neta)+total;
                    if(total>=tst)
                    {
                       *x=j;
                       *y=i;
                       L[offset2]=0;
                       clrptr[offset2]=color;
                       return(0);
                    }
                 }
              }
           }
           offset1+=szx;
        }
     }
     return(0);
  }

  /*=========================================================================*/
  /*            Tree Growth Simulator, Gauss-Siedel iteration 3              */
  /*      This routine simulates tree growth with an x-y grid of sx by sy    */
  /*      points.  This uses circular outer electrode with the outside       */
  /*      set to maxv and the center set to 0v and uses the Laplace          */
  /*      equation for field calculations.  Uses New Model gradient calc.    */
  /*=========================================================================*/

  int CIRTREESIM3(struct PICTURE *PICT)
  {
     unsigned long *L;
     unsigned v;
     long i,j,offset,iter,color,maxdev;
     long  x,y,szx,szy,ctrx,ctry,radius;
     char msg[60],buffer[60],c;
     char *picptr,*clrptr;
     double tmp,ratio,neta;
     double rmax,rmaxv;
     int tst;

  /*                        Get data from user                               */

     DRAW_MESSAGE(2,"Enter diameter of array (int) :");
     GET_MESSAGE(2,buffer);
     szx=atoi(buffer);
     szy=szx;
     DRAW_MESSAGE(2,"Number of iterations (int) :");
     GET_MESSAGE(2,buffer);
     iter=atoi(buffer);
     DRAW_MESSAGE(2,"Exponent value (double, 1.0) :");
     GET_MESSAGE(2,buffer);
     neta=atof(buffer);
     if(neta==0) neta=1.0;

  /*                        Erase previous pic if required                   */
```

```
    if (PICT->valid_pic==1) {              /* ==1 if previous pic created  */
       free(PICT->pic_buf);                /* free previous image          */
    }
 /*              Allocate new picture and color arrays                     */

    i=szx*szy;                             /* size of new image           */
    if((PICT->pic_buf=(char *)malloc(i))==NULL)
    {                                      /* memory allocation error     */
       return(0);                          /* return with error flag      */
    }
    picptr=PICT->pic_buf;                  /* save picture location       */
    memset(picptr,0,i);                    /* clear memory                */
    if((clrptr=(char *)malloc(i))==NULL)
    {                                      /* memory allocation error     */
       return(0);                          /* return with error flag      */
    }
    memset(clrptr,0,i);                    /* clear memory                */

 /*              allocate and initialize field array                       */

    *idum=-1;
    tmp=ran2(idum);
    *idum=1;
    i=szx*szy*4;
  if((L=(unsigned long *)malloc(i))==NULL) return(0);  /* return with error
 */

 /*              initialize laplacian field array to values                */

    DRAW_MESSAGE(1,"Initializing field array");
    ctrx=szx/2;
    ctry=szy/2;                            /* calculate center of array   */
    rmax=(double)(ctrx-2);
    rmaxv=(double)maxv;
    ratio=log(rmax);
    offset=0;
    for(i=0;i<szy;i++)                     /* y loop                      */
    {
       for(j=0;j<szx;j++)                  /* x loop                      */
       {
          tmp=(double)((i-ctrx)*(i-ctrx))+(double)((j-ctry)*(j-ctry));
          radius=(long)(sqrt(tmp));        /* calculate radius from center */
          if(radius>=ctrx) v=maxv;
          else
          {
             v=(long)((1.0-(log(rmax/(double)(radius))/ratio))*(double)maxv);
          }
          L[offset++]=v;                   /* set potential               */
       }
    }
    offset=ctrx+ctry*szx;                  /* position of start value     */
    L[offset]=0;                           /* starting point              */
    clrptr[offset]=255;                    /* white starting point        */
```

```
    x=ctrx;
    y=ctry;                                  /* starting point coordinates   */

 /*                        Main Iteration Loop                                */

    DRAW_MESSAGE(1,"Beginning Iterations");
    DRAW_MESSAGE(2,"7) Performing G.S. iteration.  Press ESC to end.");
    color=0;
    for(i=0;i<(iter/100);i++)
    {
       for(j=0;j<100;j++)
       {
          maxdev=CALCFIELD1(L,szx,szy,x,y); /* laplacian field calculation */
          tst=CALCGRAD2(L,szx,szy,&x,&y,clrptr,iter,&color,neta); /* find new
branch */
          if(kbhit()!=0 || tst==1)
          {
             if(tst==1)                      /* end of simulation            */
             {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L);                     /* free laplacian field array   */
                return(1);
             }
             else if((c=getch())==27)        /* operator intervention        */
             {
               CONVERTPIC2(PICT,szx,szy,clrptr); /* convert array to picture */
                free(clrptr);
                free(L);                     /* free laplacian field array   */
                return(1);
             }
          }
       }
       CONVERTPIC1(L,PICT,szx,szy);          /* convert array to grays       */
       PIC_DRAW(1,PICT);                     /* draw intermediate pictures   */
       tmp=(double)maxdev/(double)maxv;
       sprintf(msg,"%d iterations. Field err= %6.1e",i*100+j,tmp);
       DRAW_MESSAGE(1,msg);
    }
    CONVERTPIC2(PICT,szx,szy,clrptr);    /* convert array to picture     */
    free(clrptr);
    free(L);                              /* free laplacian field array   */
    return(1);                            /* return with success          */
}
```

C9 FILE: TIF_READ.C

```
/*                              TIFF_READ                                     */
/*      The purpose of this routine is to read a TIFF file into               */
/*      memory.  A call to TIFF_READ follows the format:                      */
/*                                                                            */
/*              TIFF_READ ( filestring ) ;                                    */
/*                                                                            */
/*              where PIC        = global picture structure                   */
```

```
/*                    filestring = string containing the file name      */
/*                                                                      */
/*      The file structure is as follows:                               */
/*   byte 0,1  = byte order in file                                     */
/*             = "II" (4949H) if 1234H is stored on disk as 3412        */
/*             = "MM" (4D4DH) if 1234H is stored on disk as 1234        */
/*               DOS systems store 01234567H as 67452301 so inversion   */
/*               may be needed for each short or long integer.          */
/*   byte 2,3  = 42 (002AH or 2A00H based on 0,1), TIFF file identifier */
/*   byte 4-7  = offset to first Image File Directory (IFD, from byte 0)*/
/*               the offset essentually points to the start of the first*/
/*               IFD entry whose structure is below (labelled A).       */
/*                                                                      */
/*   IFD format:                                                        */
/*   byte A             = number of Directory Entries (labelled B).     */
/*   byte A+2           = Dictionary entry 1.                           */
/*   byte A+14          = Dictionary entry 2.                           */
/*     ...              = ...                                           */
/*   byte A+2+(B-1)*12  = Dictionary entry B.                           */
/*   byte A+2+B*12      = Offset to next IFD (from 0) or 0000H if none  */
/*                                                                      */
/*   Dictionary format:                                                 */
/*   byte 0,1  = Tag               (short int)                          */
/*   byte 2,3  = Type              (short int)                          */
/*   byte 4-7  = Count             (long int)                           */
/*   byte 8,11 = Value or offset   (various types)                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */

/*==========machine data definitions and global variables===============*/
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <dos.h>
#include <malloc.h>
#include <string.h>
#include "picture.h"
#define BUFSIZE 4096            /* data buffer size                     */
#define DOS     1               /* reverse order storage                */
#define SPARC   0
#define MAC     0

extern struct PICTURE PIC;

long MACH;
long AT_EOF;                    /* at end of file flag                  */
long BEG_INBUF,END_INBUF,BEG_RQST,END_RQST;     /* buffer indexes       */


/*==========================Function Prototypes=========================*/

void TIFF_READ(char *filename);
void SWAP2(char *buffer);               /* swap short int               */
void SWAP4(char *buffer);               /* swap long int                */
```

```
void SWAPIFD(char *buffer,int reverse); /* swap IFD                          */
long GET_DATA(FILE *in,char *buf);      /* more data is required            */
extern void DRAW_MESSAGE(int position,char *message);

/*=========================================================================*/

void TIFF_READ(char *input)
{

/*============================data declarations============================*/

   FILE *in;                          /* declare in as the input file      */

/*                          General data storage                           */

   long i,j,k,l,revs,offset,entries,errflg,TAG;
   long stripnumber,offsetsize,bytecounts,bytecntsize,check,sum;
   char *inbuf;
   char *picptr;
   char *ctmp1;
   short int *sstpoffsarray=NULL;
   long *lstpoffsarray=NULL;
   short int *sbytecntarray=NULL;
   long *lbytecntarray=NULL;
   short *intptr,atmp,btmp;
   short int *itmp1;
   long *tmp1;
   long *longptr;
   unsigned long size;

/*===============file opening and reading first chunk===================*/

   MACH=DOS;
   PIC.file_error=0;                                  /* reset file error  */
   if ((in = fopen(input, "rb")) == NULL)             /* open file as binary */
   {
    PIC.file_error=1;
    return;
   }

/*                        allocate memory for arrays                       */

   inbuf=(char*) malloc(BUFSIZE);                     /* input data buffer */
   tmp1=(long*) malloc(1024*sizeof(long));            /* working buffer 1  */
   if(inbuf==NULL||tmp1==NULL) {
    DRAW_MESSAGE(2,"Not enough memory available to read TIF file.");
    PIC.file_error=2;
    goto CLOSE_FILE;
   }
   itmp1=(short int*)tmp1;                            /* short equiv.      */
   ctmp1=(char*)tmp1;                                 /* char equiv.       */

/*                          read chunk of file                             */
```

```
    BEG_INBUF=0;                                    /* at beginning of file */
    END_INBUF=0;
    BEG_RQST=0;
    END_RQST=2;
    AT_EOF=0;

    errflg=GET_DATA(in,inbuf);          /* read data from file          */
    if (errflg!=0) {
     PIC.file_error=3;
     goto CLOSE_FILE;
    }
    offset=BEG_INBUF;                   /* beginning of buffer          */

 /*==================determine storage byte order=======================*/

    if(inbuf[0]=='I' && inbuf[1]=='I') i=1; /* reverse order storage     */
    else if(inbuf[0]=='M' && inbuf[1]=='M') i=0; /* normal order storage */
    else {
     PIC.file_error=4;                 /* improper file format flag    */
     goto CLOSE_FILE;                  /* return with error noted      */
    }
    if (MACH==i) revs=0;
    else revs=1;
    offset=2;

 /*==================Read TIFF File Identifier==========================*/

    memcpy(ctmp1,&inbuf[offset],2);      /* get two bytes from buffer    */
    if (revs==1) SWAP2(ctmp1);           /* swap bytes if required       */
    atmp=itmp1[0];                       /* get short int from buffer    */
    if (atmp!=42) {
     PIC.file_error=5;                 /* improper file format flag    */
     goto CLOSE_FILE;                  /* return with error noted      */
    }
    offset+=2;

 /*==========================Read Offset to IFD============================*/

    memcpy(ctmp1,&inbuf[offset],4);      /* get long int from buffer     */
    if (revs==1) SWAP4(ctmp1);           /* swap bytes if required       */
    BEG_RQST=tmp1[0];                    /* read IFD offset from buffer  */


 /*=========================Read IFD data into memory====================*/

    END_RQST=BEG_RQST+2;                 /* looking for short int        */
    errflg=GET_DATA(in,inbuf);           /* more data from file if rqd   */
    if (errflg!=0) {                     /* seek error                   */
     PIC.file_error=6;                 /* improper file format flag    */
     goto CLOSE_FILE;                  /* return with error noted      */
    }
    offset=BEG_RQST-BEG_INBUF;           /* find offset into buffer      */
    memcpy(ctmp1,&inbuf[offset],2);      /* get short int from buffer    */
```

```
    if (revs==1) SWAP2(ctmp1);              /* swap bytes if necessary      */
    entries=itmp1[0];                       /* # of entries in IFD          */
    PIC.DICT_number=entries;                /* save number of valid entries */

    END_RQST=entries*12+BEG_RQST;           /* range of data rqd for dict.  */
    errflg=GET_DATA(in,inbuf);              /* more data from file if rqd   */
    if (errflg!=0) {                        /* seek error                   */
     PIC.file_error=7;                     /* improper file format flag    */
     goto CLOSE_FILE;                      /* return with error noted      */
    }

    offset+=2;                              /* position of data in buffer   */
    for (i=0;i<entries;i++)
    {
     j=offset+i*12;                        /* calculate offset into buffer */
     memcpy(ctmp1,&inbuf[j],12);           /* copy IFD to working area     */
     SWAPIFD(ctmp1,revs);                  /* convert IFD                  */
     memmove(&PIC.IFD[i],ctmp1,12);        /* copy IFD into structure      */
    }
    PIC.samplesperpixel=1;                  /* default value into structure */


/*===============Dictionary entries now must be decoded================*/
/*       and put into a standard format in the picture structure        */

    for (i=0;i<entries;i++)
    {
     intptr=(short int*)&PIC.IFD[i];
     atmp=*intptr;                         /* get tag from dictionary      */
     TAG=atmp;                             /* convert tag to long          */

switch (TAG) {

   case 262:  intptr=&PIC.IFD[i][4];            /* read bilevel data    */
       atmp=*intptr;                          /* get data             */
       PIC.photometric=atmp;                  /* put into structure   */
       break;

   case 259:  intptr=&PIC.IFD[i][4];            /* read compression     */
       atmp=*intptr;                          /* get data             */
       PIC.compression=atmp;                  /* put into structure   */
       break;

   case 257:  intptr=&PIC.IFD[i][4];            /* read imagelength     */
       atmp=*intptr;                          /* get data             */
       PIC.imagelength=atmp;                  /* put into structure   */
       break;

   case 256:  intptr=&PIC.IFD[i][4];            /* read imagewidth      */
       atmp=*intptr;                          /* get data             */
       PIC.imagewidth=atmp;                   /* put into structure   */
       break;

   case 296:  intptr=&PIC.IFD[i][4];            /* read resolution Unit */
```

```
      atmp=*intptr;                        /* get data              */
      PIC.resolutionunit=atmp;             /* put into structure    */
      break;

   case 282:  longptr=(long*)&PIC.IFD[i][4];    /* read resolution Unit */
      BEG_RQST=*longptr;                   /* get address of data  */
      if(revs==1) SWAP4((char *) &BEG_RQST); /* swap offset */
      END_RQST=BEG_RQST+8;                 /* looking for 2 longs  */
      errflg=GET_DATA(in,inbuf);           /* more data from file  */
      if (errflg!=0) {                     /* seek error           */
         PIC.file_error=8;                 /* improper format flag */
         goto CLOSE_FILE;                  /* return, error noted  */
      }
      offset=BEG_RQST-BEG_INBUF;           /* offset into buffer   */
      memcpy(ctmp1,&inbuf[offset],8); /* get longs from buffer */
      if (revs==1) SWAP4(ctmp1);           /* swap bytes if required */
      if (revs==1) SWAP4(&ctmp1[4]); /* swap bytes if required */
      PIC.xresolution[0]=tmp1[0];          /* save data            */
      PIC.xresolution[1]=tmp1[1];          /* save data            */
      break;

   case 283:  longptr=(long*)&PIC.IFD[i][4];    /* read resolution Unit */
      BEG_RQST=*longptr;                   /* get address of data  */
      if(revs==1) SWAP4((char *) &BEG_RQST); /* swap offset  */
      END_RQST=BEG_RQST+8;                 /* looking for 2 longs  */
      errflg=GET_DATA(in,inbuf);           /* more data from file  */
      if (errflg!=0) {                     /* seek error           */
         PIC.file_error=9;                 /* improper format flag */
         goto CLOSE_FILE;                  /* return, error noted  */
      }
      offset=BEG_RQST-BEG_INBUF;           /* offset into buffer   */
      memcpy(ctmp1,&inbuf[offset],8); /* get longs from buffer */
      if (revs==1) SWAP4(ctmp1);           /* swap bytes if required */
      if (revs==1) SWAP4(&ctmp1[4]); /* swap bytes if required */
      PIC.yresolution[0]=tmp1[0];          /* save data            */
      PIC.yresolution[1]=tmp1[1];          /* save data            */
      break;

   case 278:  intptr=&PIC.IFD[i][1];            /* address data format  */
      atmp=*intptr;                        /* read format          */
      if(atmp==3) {
      intptr=&PIC.IFD[i][4];               /* address rows per strip */
      btmp=*intptr;                        /* read rows per strip  */
      j=btmp;                              /* convert to long      */
      PIC.rowsperstrip=j;                  /* read rows per strip  */
      }
      else {
      longptr=(long*)&PIC.IFD[i][4]; /* address long format  */
      PIC.rowsperstrip=*longptr;           /* read rows per strip  */
      }
      break;

   case 273:  intptr=&PIC.IFD[i][1];            /* address data format  */
      atmp=*intptr;                        /* read format          */
```

```
 longptr=(long*)&PIC.IFD[i][2];     /* get number of strips */
 stripnumber=*longptr;              /* save number of strips */
 if(atmp==3) {                      /* short format         */
 intptr=&PIC.IFD[i][4];           /* address strip offsets */
 btmp=*intptr;                    /* strip offsets start  */
 offsetsize=2;                    /* offset size (short)  */
 k=2*stripnumber;                 /* size of offset array */
 BEG_RQST=btmp;                   /* start of offset array */
 END_RQST=btmp+k;                 /* end of offset array  */
 sstpoffsarray=malloc(k);         /* allocate short array */
 if(sstpoffsarray==NULL) {
    PIC.file_error=10;
    goto CLOSE_FILE;
 }
 sstpoffsarray[0]=btmp;           /* special case         */
 if(stripnumber>1) {
    errflg=GET_DATA(in,inbuf);  /* get data for array   */
    if (errflg!=0) {            /* seek error           */
     PIC.file_error=11;     /* improper format flag */
     goto CLOSE_FILE;       /* return, error noted  */
    }
    offset=BEG_RQST-BEG_INBUF;
    memcpy(sstpoffsarray,&inbuf[offset],k); /* save array */
    if(revs==1)                 /* swap array if reqd   */
    {
       for(j=0;j<stripnumber;j++)
       {
     SWAP2((char *) &sstpoffsarray[j]);
       }
    }
 }
 }
 else {                             /* long format          */
 longptr=(long*)&PIC.IFD[i][4]; /* address long format  */
 j=*longptr;                    /* strip offsets start  */
 offsetsize=4;                  /* offset size (long)   */
 k=4*stripnumber;               /* size of offset array */
 BEG_RQST=j;                    /* start of offset array */
 END_RQST=j+k;                  /* end of offset array  */
 lstpoffsarray=malloc(k);       /* allocate array       */
 if(lstpoffsarray==NULL) {
    PIC.file_error=10;
    goto CLOSE_FILE;
 }
 lstpoffsarray[0]=j;            /* special case         */
 if(stripnumber>1) {
    errflg=GET_DATA(in,inbuf);  /* get data for array   */
    if (errflg!=0) {            /* seek error           */
     PIC.file_error=11;       /* improper format flag */
     goto CLOSE_FILE;         /* return, error noted  */
    }
    offset=BEG_RQST-BEG_INBUF;
    memcpy(lstpoffsarray,&inbuf[offset],k); /* save array */
    if(revs==1)                 /* swap array if reqd   */
```

```
        {
           for(j=0;j<stripnumber;j++)
           {
         SWAP4((char *)&lstpoffsarray[j]);
           }
        }
     }
     }
     break;

case 279:  intptr=&PIC.IFD[i][1];            /* address data format  */
      atmp=*intptr;                        /* read format          */
      longptr=(long*)&PIC.IFD[i][2];       /* get number of strips */
      bytecounts=*longptr;                 /* save number of strips */
      if(atmp==3) {                        /* short format         */
    intptr=&PIC.IFD[i][4];          /* address strip offsets */
    btmp=*intptr;                   /* strip offsets start  */
    bytecntsize=2;                  /* offset size (short)  */
    k=2*bytecounts;                 /* size of offset array */
    BEG_RQST=btmp;                  /* start of offset array */
    END_RQST=btmp+k;                /* end of offset array  */
    sbytecntarray=malloc(k);        /* allocate short array */
    if(sbytecntarray==NULL) {
       PIC.file_error=10;
       goto CLOSE_FILE;
    }
    sbytecntarray[0]=btmp;          /* special case         */
    if(bytecounts>1) {
       errflg=GET_DATA(in,inbuf);  /* get data for array   */
       if (errflg!=0) {            /* seek error           */
        PIC.file_error=11;     /* improper format flag */
        goto CLOSE_FILE;       /* return, error noted  */
       }
       offset=BEG_RQST-BEG_INBUF;
       memcpy(sbytecntarray,&inbuf[offset],k); /* save array */
       if(revs==1)                  /* swap array if reqd   */
       {
          for(j=0;j<bytecounts;j++)
          {
        SWAP2((char *)&sbytecntarray[j]);
          }
       }
    }
      }
      else {                              /* long format          */
    longptr=(long*)&PIC.IFD[i][4]; /* address long format  */
    j=*longptr;                    /* strip offsets start  */
    bytecntsize=4;                 /* offset size (long)   */
    k=4*bytecounts;                /* size of offset array */
    BEG_RQST=j;                    /* start of offset array */
    END_RQST=j+k;                  /* end of offset array  */
    lbytecntarray=malloc(k);       /* allocate array       */
    if(lbytecntarray==NULL) {
       PIC.file_error=10;
```

```
            goto CLOSE_FILE;
         }
         lbytecntarray[0]=j;              /* special case          */
         if(bytecounts>1) {
            errflg=GET_DATA(in,inbuf);   /* get data for array    */
            if (errflg!=0) {             /* seek error            */
             PIC.file_error=11;         /* improper format flag */
             goto CLOSE_FILE;           /* return, error noted  */
            }
            offset=BEG_RQST-BEG_INBUF;
            memcpy(lbytecntarray,&inbuf[offset],k); /* save array */
            if(revs==1)                  /* swap array if reqd    */
            {
               for(j=0;j<bytecounts;j++)
               {
             SWAP4((char *)&lbytecntarray[j]);
               }
            }
         }
           }
           break;

    case 258:  intptr=&PIC.IFD[i][4];            /* read bits per sample */
        atmp=*intptr;                  /* get data              */
        PIC.bitspersample=atmp;        /* put into structure    */
        break;

    case 274:  intptr=&PIC.IFD[i][4];            /* read image orientation */
        atmp=*intptr;                  /* get data              */
        PIC.orientation=atmp;          /* put into structure    */
        break;

    case 277:  intptr=&PIC.IFD[i][4];            /* read samples per pixel */
        atmp=*intptr;                  /* get data              */
        PIC.samplesperpixel=atmp;      /* put into structure    */
        break;

    case 270:  longptr=(long*)&PIC.IFD[i][4];    /* read msg location     */
        BEG_RQST=*longptr;             /* get address of msg    */
        if(revs==1) SWAP4((char *) &BEG_RQST); /* swap offset   */
        END_RQST=BEG_RQST+40;          /* max 40 chars          */
        errflg=GET_DATA(in,inbuf);     /* more data from file   */
        if (errflg!=0) {               /* seek error            */
           PIC.file_error=17;          /* improper format flag */
           goto CLOSE_FILE;            /* return, error noted  */
        }
        offset=BEG_RQST-BEG_INBUF;     /* offset into buffer    */
        strncpy(PIC.message,&inbuf[offset],40); /* get message */
        PIC.message[39]='\0';          /* null end character    */
        break;

     }
    }
```

```
/*====================Picture sizing and reading=========================*/

/*              size and allocate memory for picture                     */

   picptr=PIC.pic_buf;
   size=(long)PIC.samplesperpixel*(long)PIC.imagelength*(long)PIC.imagewidth;
   if(PIC.valid_pic==1)                  /* check if old picture exists  */
   {
    free(picptr);                    /* free old picture memory      */
   }
   if((picptr=(char *)malloc(size))==NULL) {  /* memory allocation error
*/
    PIC.file_error=10;               /* improper file format flag    */
    goto CLOSE_FILE;                 /* return with error noted      */
   }
   PIC.pic_buf=picptr;

/*                       Read picture data                              */
/*                       Strip Loop                                     */

   sum=0;
   check=0;
   for (i=0;i<stripnumber;i++) {    /* one loop per strip           */

/*                       Get address of strip                           */

    if(offsetsize==2) {
    if(stripnumber==1) j=sstpoffsarray[0];
    else j=sstpoffsarray[i];
    }
    else if(offsetsize==4) {
    if(stripnumber==1) j=lstpoffsarray[0];
    else j=lstpoffsarray[i];
    }                                    /* j=address of strip           */

/*                       Get number of bytes in strip                   */

    if(bytecntsize==2) {
    if(bytecounts==1) k=sbytecntarray[0];
    else k=sbytecntarray[i];
    }
    else if(bytecntsize==4) {
    if(bytecounts==1) k=lbytecntarray[0];
    else k=lbytecntarray[i];
    }                                    /* k=byte count in strip        */

/*                       Picture Gathering Loop                         */

    if(PIC.compression==1 && PIC.photometric==1) { /* easiest case    */

    for (l=0;l<k;l+=PIC.imagewidth) { /* loop for each scan line   */
       BEG_RQST=j+l;                /* calculate start of scan line */
       END_RQST=j+l+PIC.imagewidth; /* calc end of scan line        */
       errflg=GET_DATA(in,inbuf);     /* data from file if rqd     */
```

```
        if (errflg!=0) {                /* seek error                  */
         PIC.file_error=15;           /* improper file format flag */
         goto CLOSE_FILE;             /* return with error noted   */
        }
        offset=BEG_RQST-BEG_INBUF;      /* calculate offset into buffer */
        atmp=PIC.imagewidth;
        memcpy(PIC.pic_buf+1+sum,&inbuf[offset],atmp); /* get scan      */
        check=check+atmp;               /* count pixels moved          */
     }
     sum=sum+k;
     }
     else {
        PIC.file_error=16;            /* picture type not decoded flag */
        goto CLOSE_FILE;              /* return with error noted     */
    }



    }
```

```
/*====================file closing and cleanup==========================*/

    CLOSE_FILE:
     fclose(in);
     free(inbuf);
     free(tmp1);
     free(sstpoffsarray);
     free(lstpoffsarray);
     free(sbytecntarray);
     free(lbytecntarray);
    return;

}

/*====================SWAP subroutines==================================*/
/*      These routines swap bytes in the buffer around so that          */
/*      the data matches how the machine will use it.                   */

void SWAP2(char *buffer)                /* swap short int              */
{
   char c;
   c=buffer[0]; buffer[0]=buffer[1],buffer[1]=c;        /* do byte swap */
}

void SWAP4(char *buffer)                /* swap long int               */
{
   char c;
   c=buffer[3]; buffer[3]=buffer[0];buffer[0]=c;        /* do byte swap */
   c=buffer[2]; buffer[2]=buffer[1];buffer[1]=c;        /* do byte swap */
```

```
}

void SWAPIFD(char *buffer,int revs)     /* swap and convert IFD          */
{
   int i;
   short int *ptrs;
   if (revs==1) SWAP2(buffer);          /* swap TAG                      */
   buffer+=2;
   if (revs==1) SWAP2(buffer);          /* swap Field Type               */
   ptrs=(short int*) buffer;            /* point to Field Type           */
   i=*ptrs;                             /* get Field Type                */
   buffer+=2;
   if (revs==1) SWAP4(buffer);          /* swap Count Field              */
   buffer+=4;
   if (i==4 && revs==1) SWAP4(buffer);  /* swap Long Value Offset        */
   else if(i==3 && revs==1) SWAP2(buffer); /* swap Short Value Offset    */
}


/*======================Read Data from File============================*/
/*  in = already opened file handler                                   */
/*  buf = character input buffer                                       */
/*                                                                     */
/*  BEG_INBUF,END_INBUF,BEG_RQST,END_RQST =  global buffer indexes     */
/*                                                                     */


long GET_DATA(FILE *in,char *inbuf)
{
   int i,j,k,err;

   j=BUFSIZE;
   err=0;
   if (END_RQST>=END_INBUF || BEG_RQST < BEG_INBUF) {   /* need more data  */
    err=fseek(in,BEG_RQST,SEEK_SET);  /* seek requested data             */
    if (err!=0) return(err);
    i=0;
    while (i<j && (k=fgetc(in))!=EOF) inbuf[i++]=k; /* read data         */
    END_INBUF=i+BEG_RQST;             /* end of buffer                  */
    BEG_INBUF=BEG_RQST;               /* beginning of buffer            */
   }
   return(err);
}
```

C10 FILE: PICTURE.H

```
/*                                PICTURE structure                    */
/*      This file contains the structure of a picture file in memory   */
/*      Space is allocated to accomodate the file size expected        */
/*                                                                     */

struct PICTURE
{
  long valid_pic;          /* code=1 if a valid pic exists in struct   */
```

```
   long file_error;          /* code !=0 if error occured in file          */
   long DICT_number;         /* # of valid entries in dictionary           */
   short int IFD[32][6];     /* 32 dictionary item area reserved           */
   long photometric;         /* 0=B/W,1=W/B or GREYSCALE, 3=COLOR          */
   long compression;         /* 1=none, 2=CCITT Group 3, 32773=Pack Bits   */
   long imagelength;         /* number of picture rows                     */
   long imagewidth;          /* number of picture columns                  */
   long resolutionunit;      /* 1=none,2=inch,3=centimeter   unit          */
   long xresolution[2];      /* [0]=# of x pixels per [1] units            */
   long yresolution[2];      /* [0]=# of y pixels per [1] units            */
   long rowsperstrip;        /* scan rows per compressed strip             */
   long bitspersample;       /* bits per component or pixel                */
   long orientation;         /* picture orientation                        */
   long samplesperpixel;     /* samples per pixel                          */
   char *pic_buf;            /* declare pointer to picture array           */
   long save;                /* code=1 if a picture has been saved         */
   long savewidth;           /* width of saved picture                     */
   long savelength;          /* length of saved picture                    */
   char *saveptr;            /* pointer to saved picture                   */
   char savestr[80];         /* string to save picture message             */
   char message[40];         /* string saved/recovered with file           */
}
;
```

C11 FILE: LMAP.H

```
/*                          L-Systems routines                             */
/*                                                                         */
/*      The following structure is used when creating an L-System          */
/*      image.  The necessary values in the structure are set and          */
/*      the LENGINE routine is then called.  This then sets the            */
/*      points in the previously declared picture structure.               */

struct LMAP
{
   char start[80];            /* starter string used for image            */
   char rule0[80];            /* rule string 0                            */
   char rule1[80];            /* rule string 1                            */
   char rule2[80];            /* rule string 2                            */
   char rule3[80];            /* rule string 3                            */
   char rule4[80];            /* rule string 4                            */
   double angle0;             /* default angle and angle 0                */
   double angle1;             /* angle 1 when called upon                 */
   double angle2;             /* angle 2 when called upon                 */
   double *cosary;            /* pointer to cosine array, 50=0.0          */
   double *sinary;            /* pointer to sine array, 50=0.0            */
   int tangle;                /* total image angle steps                  */
   double l;                  /* line segment length                      */
   double x;                  /* x position in image                      */
   double bx;                 /* big x position in image                  */
   double y;                  /* y position in image                      */
   double by;                 /* big y position in image                  */
   double mult1;              /* line segment multiplier 1                */
   double mult2;              /* line segment multiplier 2                */
```

```
    int brptr;                  /* branch pointer into branch stack     */
    double *branch;             /* branch stack                         */
    int szptr;                  /* size pointer into size stack         */
    double *size;               /* size stack                           */
    int angptr;                 /* angle pointer into angle stack       */
    int *angl;                  /* angle stack                          */
};
```

C12 FILE: SCREEN.H

```
/*                      Screen Plotting Structure                       */
/*      This purpose of this class is to allow the easy plotting of     */
/*      data on a screen of any resolution using the first 16 colors    */

struct SCREEN
{
   int xstart;              /* starting point on screen 1<x<MaxX    */
   int ystart;              /* starting point on screen 1<y<MaxY    */
   int xspan;               /* x window size                        */
   int yspan;               /* y window size                        */
   float xmin;              /* minimum x value plotted              */
   float ymin;              /* mimimum y value plotted              */
   float xmax;              /* maximum x value plotted              */
   float ymax;              /* maximum y value plotted              */
   int type;                /* 0=point,1=line,2=bar plot type       */
   int nn;                  /* number of data points in data array  */
   int color;               /* color of points in graph             */
   float *data;             /* pointer to data array to be plotted  */
   char title1[40];         /* graph title string #1                */
   char title2[40];         /* gragh title string #2                */
};

/*                      Some color constants                            */

#define SOLID_LINE 65535
#define DOTTED_LINE 257
#define LIGHTRED 17
#define GREEN 18
#define YELLOW 19
#define CYAN 20
#define LIGHTGREEN 21
#define BRIGHTWHITE 22
```

C13 FILE: MAKEFILE

```
# Makefile for building test.exe from its components

test.exe : test.obj pic_draw.obj tif_read.obj routines.obj edges.obj
screen.obj dimen.obj sim.obj fuld.obj
        wlink debug all file test,pic_draw,tif_read,rou-
tines,edges,screen,dimen,sim, fuld

test.obj : test.c picture.h
        wcc386p test /w3 /d1
```

```
pic_draw.obj : pic_draw.c picture.h
        wcc386p pic_draw /w3 /d1

tif_read.obj : tif_read.c picture.h
        wcc386p tif_read /w3 /d1

routines.obj : routines.c picture.h
        wcc386p routines /w3 /d1

edges.obj : edges.c picture.h
        wcc386p edges /w3 /d1

screen.obj : screen.c screen.h
        wcc386p screen /w3 /d1

dimen.obj : dimen.c
        wcc386p dimen /w3 /d1

sim.obj : sim.c
        wcc386p sim /ot

fuld.obj : fuld.c makefile
        wcc386p fuld /w3 /ot
```