

SKELETAL DOSIMETRY: A HYPERBOLOID REPRESENTATION OF THE
BONE-MARROW INTERFACE TO REDUCE VOXEL EFFECTS IN 3D IMAGES OF
TRABECULAR BONE

By

DIDIER A. RAJON

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2002

ACKNOWLEDGMENTS

Several people contributed greatly to this work. First, I would like to express my sincere thanks to Dr. Wesley E. Bolch for his support, encouragement, and guidance throughout my graduate work at the University of Florida. A mere thank you cannot express my gratitude for his tolerance and patience over the last two years. I appreciate the opportunity that he has provided me to pursue my interest in health physics. I also thank Dr. David Hintenlang, Dr. Samim Anghaie, Dr. Stephen Blackband, Dr. Anthony Ladd, and Dr. Lionel Bouchet for their suggestions and for being members of my committee.

I would also like to thank all of the students who contribute to the Bone Imaging Dosimetry project, for providing the bone-sample images that I used in this research. I also thank Dr. Derek Jokisch and Dr. Phillip Patton for their contributions and support.

I also thank all of the faculty and graduate students in the department of Nuclear and Radiological Engineering for their help and support during my five years of graduate research. They all contributed to my knowledge and made the work enjoyable. I also thank the staff of the department of Nuclear and Radiological Engineering for their devotion and comprehension in resolving various problems. I also thank the department of Nuclear and Radiological Engineering for its financial support over these years.

Finally I would like to thank some people very close to me. These include my parents and my sister for having supported my decisions and accepted a long separation

during these five years, and also all the friends I met since I arrived in Gainesville. They all contributed to the success of my experience by being present when I needed them.

TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGMENTS	ii
TABLE OF CONTENTS	iv
LIST OF FIGURES	viii
LIST OF TABLES	xii
ABSTRACT	xiv
 CHAPTER	
1 INTRODUCTION	1
2 BACKGROUND	7
Bone Structure and Physiology	7
Bone Cell Types	8
Variation of the Trabecular Bone Microstructure	9
Internal Dosimetry	11
Past Skeletal Dosimetry Works and Models	13
Use of Nuclear Magnetic Resonance	16
Bone Sample Imaging and Image Processing	18
3 VOXEL-SIZE EFFECTS IN 3D NMR MICROSCOPY PERFORMED FOR TRABECULAR BONE DOSIMETRY	25
Introduction	25
Previous Studies in Trabecular Bone Dosimetry	25
Use of NMR Microscopy	26
Voxel-Size Effects in Skeletal Dosimetry Calculations	27
Material and Methods	28
Construction of the Mathematical Model of Trabecular Bone	28
Construction of the Segmented Images of the Model	33
Electron-Transport Simulations	34
S-Value Calculations	35

Results and Discussion	36
Mathematical Sample	36
Segmented Images	37
Absorbed Fractions and their Absolute Errors	39
Absorbed Doses and their Relative Errors	44
Conclusion	46
4 SURFACE-AREA OVERESTIMATION WITHIN 3D DIGITAL IMAGES AND ITS CONSEQUENCES FOR SKELETAL DOSIMETRY	65
Introduction	65
Bone-Marrow Dosimetry	66
Voxelization Effects in Bone-Marrow Dosimetry	67
Material and Methods	69
Construction of the 3D Segmented Images of the Single-Sphere Model	69
Volume Fraction Occupied by the Spheres within the Segmented Images ..	71
Theoretical Surface Area of a Voxelized Sphere	71
Measurement of the Surface Area of a Voxelized Sphere	74
Consequences of an Error in Surface Area on the Absorbed Fraction	74
Electron-Transport Simulations	76
Statistical Analysis	78
Results and Discussion	80
Volume Fraction Occupied by Spheres within the Segmented Images	80
Surface Area of the 3D Segmented Images	80
Absorbed Fractions within the 3D Segmented Images	81
Conclusion	83
5 INTERACTION WITH 3D ISOTROPIC AND HOMOGENEOUS RADIATION FIELDS: A MONTE-CARLO SIMULATION ALGORITHM .	99
Introduction	99
Three-Dimensional Homogeneous and Isotropic Fields	100
Monte-Carlo Methods	101
Random Number Generators	101
Material and Methods	102
General Description of the Algorithm	102
Choosing a Random Point P on the Surface of the Sphere	104
Choosing a Random Point P' on the Surface of the Disk	105
Local Coordinate System of the Plane	105
Point P' in the Local Coordinate System	107
Coordinates of P' in the Initial Coordinate System	108
Direction of the Particle	108
Implementation of the Algorithm	109
Application to Chord-Length Distributions	111
Results and Discussion	112

General Algorithms	112
Chord-Length Distribution Examples	113
Conclusion	114
6 VOXEL EFFECTS WITHIN DIGITAL IMAGES OF TRABECULAR BONE AND THEIR CONSEQUENCES ON CHORD-LENGTH DISTRIBUTION MEASUREMENTS	121
Introduction	121
Background	122
Chord-Length Distributions within Trabecular Bone Samples	122
Voxel Effects on Chord-Length Distribution Measurements	124
Minimum Acceptable Chord (MAC) Selection Criteria	125
Mathematical Model of Trabecular Bone	126
Material and Methods	128
Chord-Length Distributions through the Mathematical Model	128
Chord-Length Distributions through Single-Sphere Models	133
Results and Discussion	134
Mathematical Model of Trabecular Bone	134
Single-Sphere Models	137
Conclusion	139
7 MARCHING-CUBE ALGORITHM: REVIEW AND TRILINEAR INTERPOLATION ADAPTATION FOR IMAGE-BASED DOSIMETRIC MODELS	156
Introduction	156
Marching-Cube Algorithm	159
Original MC Algorithm	160
Ambiguity Problem	163
Hole Problem	165
Image Size and Processing Time	166
Marching Tetrahedrons	167
Extended MC Algorithm	168
Facial Deciders	169
Optimizing the Image Size	171
Material and Methods	172
Mathematical Bone Sample	173
Direct and Reverse Extended MC Algorithm	175
Trilinear-Interpolation Isosurface	177
Intersection of a Straight Line with a Hyperboloid	181
Results and Discussion	184
Surface-Area Measurement	184
Chord-Length Distributions	185
Conclusion	188

8	HYPERBOLOID REPRESENTATION OF THE BONE-MARROW INTERFACE WITHIN 3D NMR IMAGES OF TRABECULAR BONE: APPLICATIONS TO SKELETAL DOSIMETRY	213
	Introduction	213
	Hyperboloid Marching-Cube Algorithm	214
	Material and Methods	216
	Revised Mathematical Model of Human Trabecular Bone	216
	AF Calculations within the Mathematical Bone Model	219
	S-Value Calculations within the Mathematical Bone Model	221
	Application to NMR Microscopy Images of Human Trabecular Bone	221
	Results and Discussion	223
	Volume Fraction and Surface Area	223
	Absorbed Fractions of Energy	224
	Radionuclide S Values	226
	Applications to NMR Microscopy Images of Human Trabecular Bone	230
	Conclusion	231
9	CONCLUSIONS AND FUTURE WORK	249
	Conclusions	249
	Future work	251
APPENDIX		
A	LOOK-UP TABLES FOR MARCHING-CUBE ALGORITHM	255
B	GENERAL TOOLS (C++ PROGRAMS)	276
C	LOOK-UP TABLE TOOLS (C++ PROGRAMS)	291
D	MARCHING-CUBE IMAGE TOOLS (C++ PROGRAMS)	311
E	MC TRIANGLE GENERATION (C++ PROGRAMS)	338
F	MC CHORD-LENGTH DISTRIBUTION (C++ PROGRAMS)	356
G	HMC CHORD-LENGTH DISTRIBUTION (C++ PROGRAMS)	369
H	HMC VOLUME FRACTION (C++ PROGRAMS)	380
I	HMC TRANSPORT CODE (EGSNRC USER CODE)	383
	REFERENCES	411
	BIOGRAPHICAL SKETCH	426

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
2-1 Femur head showing the different constituents of the bone structure	20
2-2 Microstructure of compact bone and trabecular bone	21
2-3 Microstructure of trabecular bone showing the types of bone cells and their location	22
2-4 Differing paths of two electrons emitted within anisotropic regions of trabecular bone	23
2-5 Trabecular bone 3D image obtained from a 4.7 Tesla NMR system	24
3-1 Electron micrograph of the trabecular latticework within a lumbar vertebra	49
3-2 Voxel-size effects on the dose calculation for a single electron track traveling within a bone trabecula	50
3-3 Edge effect when trying to position circles within a squared field	51
3-4 Transverse slice, 1.6 cm x 1.6 cm, through the ROI of the mathematical sample of trabecular bone	52
3-5 Chord-length distributions: Spiers cervical vertebra compared to the mathematical model	53
3-6 Segmentation of the mathematical sample	54
3-7 Geometrical parameters as a function of image resolution for the mathematical sample	55
3-8 Absorbed fraction differences between the segmented images and the reference values	56
3-9 Relative error in the absorbed fraction for the segmented images and using results from the mathematical bone sample as reference values	57

3-10	Relative error in the absorbed dose for monoenergetic electron sources in the marrow cavities	58
3-11	Relative error in the S value calculated for five radionuclides of interest in skeletal dosimetry	59
4-1	Cubical sample of trabecular bone reconstructed from a 3D NMR image obtained at 4.7 tesla	86
4-2	Perimeter of a circle as represented by a digital image	87
4-3	Analytical derivation of the surface area of a voxelized sphere	88
4-4	Consequence of the surface-area error on the absorbed-fraction calculation	89
4-5	Expected evolution of the cross-absorbed fraction overestimation as a function of the voxel size	90
4-6	Volume fraction occupied by the sphere within the segmented image as a function of the voxel size	91
4-7	Surface area of the segmented spheres, as a function of the voxel size	92
4-8	Relative error on absorbed fractions inside the sphere	93
4-9	Relative error on absorbed fractions outside the sphere	94
5-1	Technique used to simulate a homogeneous and isotropic infinite 3D field of radiation when it interacts with a fixed object	115
5-2	Geometric construction used to derive the trajectory of the particles	116
5-3	C program that implements Algorithm 1	117
5-4	C program that implements Algorithm 2	118
5-5	Chord-length distributions measured through a sphere of unit diameter	119
5-6	Chord-length distributions measured through a cube of unit width	120
6-1	Cubical sample of trabecular bone reconstructed from a 3D NMR image obtained at 4.7 tesla	144
6-2	Chord-length distributions measured by Spiers and colleagues within a thin slice of a human cervical vertebra	145

6-3	Bone trabecula chord-length distribution measured through a NMR image of a human sample of trabecular bone	146
6-4	Segmentation of the mathematical model of trabecular bone	147
6-5	Comparison between the theoretical and the measured distribution through the mathematical model	148
6-6	Three-dimensional chord-length distributions through segmented images of the mathematical model	149
6-7	Distributions measured through the mathematical model using different techniques	150
6-8	Distributions through a single sphere of radius 500 μm using different techniques	151
7-1	MC algorithm principle for a simple 2D image	191
7-2	MC algorithm configurations and patterns in 2D	192
7-3	MC patterns proposed by Lorensen for 3D images	193
7-4	Ambiguity problem in 2D	194
7-5	Hole problem as a consequence of the ambiguity problem	195
7-6	Hole problem for a 22 x 22 x 22 image of a trabecular bone sample	196
7-7	Marching-Tetrahedron method	197
7-8	MC patterns proposed to solve the hole problem using the direct design for ambiguous faces	198
7-9	MC patterns proposed to solve the hole problem using the reverse design for ambiguous faces	199
7-10	The four triangulations that solve the ambiguity problem for pattern 10	200
7-11	Transverse slice through the mathematical sample of trabecular bone	201
7-12	Synopsis of the database created by the MC algorithm	202
7-13	Localization of the eight vertices and twelve edges of a marching cube	203

7-14	Trilinear-interpolated isosurface examples within unit cubes	204
7-15	Trilinear interpolated isosurfaces within unit cubes	205
7-16	Trilinear interpolated isosurfaces within unit cubes of pattern 10	206
7-17	Trilinear interpolated isosurfaces within several adjacent cubes, showing how the surface connects	207
7-18	Surface area as a function of image resolution for the mathematical sample	208
7-19	Chord-length distributions measured through the mathematical bone sample for four different voxel sizes	209
7-20	Corrected chord-length distributions measured with the trilinear technique	210
8-1	Cubical sample of a human lumbar vertebra	234
8-2	3D representation of the triangulated bone-marrow interface obtained from the MC algorithm	235
8-3	Transverse slice, $1.2 \times 1.2 \text{ cm}^2$, through the revised mathematical sample of trabecular bone	236
8-4	Transverse slice, $0.896 \times 2.12 \text{ cm}^2$, through a NMR microscopy image of human trabecular bone	237
8-5	Geometrical parameters as a function of image resolution for the revised mathematical sample	238
8-6	Relative error of the absorbed fractions calculated for simulated images of the mathematical sample as a function of the voxel size	239
8-7	Relative error of the absorbed fractions calculated for simulated images of the mathematical sample as a function of the voxel size	240
8-8	Relative error of the S values calculated for four radionuclides of interest in skeletal dosimetry as a function of the voxel size	241
8-9	Relative error of the S values calculated for four radionuclides of interest for skeletal dosimetry as a function of the voxel size	242
8-10	Absorbed fraction: voxel code compared to HMC code	243
8-11	Relative error of the AF calculated with the voxel-code	244

LIST OF TABLES

<u>Table</u>	<u>page</u>
3-1 Characteristics of the 19 segmented images for a voxel-size range from 1000 μm to 16 μm	60
3-2 Tissue compositions used for bone and marrow in the mathematical bone sample	61
3-3 Radiation characteristics of the radionuclides used for S-value calculations	62
3-4 Absorbed fractions calculated within the mathematical bone sample for the marrow region, the bone region, and the region beyond the dosimetric region of interest	63
3-5 S values calculated within the mathematical bone sample for the five chosen radionuclides	64
4-1 Characteristics of the eight single-sphere models covering a typical range of bone-marrow cavity sizes.	95
4-2 Voxel sizes for the different 3D images of the single-sphere models	96
4-3 List of electron energies used in the study	97
4-4 Electron ranges in bone marrow and cortical bone	98
6-1 Characteristics of the 4 segmented images and of the mathematical trabecular bone model used in the chord-length distribution study	152
6-2 Characteristics of the 4 segmented images of the single-sphere marrow cavity model used in the chord-length distribution study	153
6-3 Mean chord lengths for the 4 segmented images and the mathematical bone model using the different techniques to record the chord-length distributions	154

6-4	Mean chord lengths for the 4 segmented images of the single-sphere marrow cavity model using the different techniques to record the chord-length distributions	155
7-1	Characteristics of the 17 segmented images of the mathematical bone sample	211
7-2	Gray-levels used for the isosurface equation of the cubes presented in Figures 7-14, 7-15, 7-16, and 7-17	212
8-1	Characteristics of the 19 segmented images of the mathematical bone sample	245
8-2	Energies used for AF calculations	246
8-3	Radiation characteristics of the radionuclides used for the S-value calculations	247
8-4	S values calculated within the real bone sample for both the voxel and the HMC representations of the bone-marrow interface	248

Abstract of Dissertation Presented to the Graduate School
Of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SKELETAL DOSIMETRY: A HYPERBOLOID REPRESENTATION OF THE
BONE-MARROW INTERFACE TO REDUCE VOXEL EFFECTS IN 3D IMAGES OF
TRABECULAR BONE

By

Didier A. Rajon

December 2002

Chairman: Dr. Wesley E. Bolch
Major Department: Nuclear and Radiological Engineering

Radiation damage to the hematopoietic bone marrow is clearly defined as the limiting factor to the development of internal emitter therapies. Current dosimetry models rely on chord-length distributions measured through the complex microstructure of the trabecular bone regions of the skeleton in which most of the active marrow is located. Recently, Nuclear Magnetic Resonance (NMR) has been used to obtain high-resolution three-dimensional (3D) images of small trabecular bone samples. These images have been coupled with computer programs to estimate dosimetric parameters such as chord-length distributions, and energy depositions by monoenergetic electrons. This new technique is based on the assumption that each voxel of the image is assigned either to bone tissue or to marrow tissue after application of a threshold value. Previous studies showed that this assumption had important consequences on the outcome of the computer calculations. Both the chord-length distribution measurements and the energy deposition calculations

are subject to voxel effects that are responsible for large discrepancies when applied to mathematical models of trabecular bone.

The work presented in this dissertation proposes first a quantitative study of the voxel effects. Consensus is that the voxelized representation of surfaces should not be used as direct input to dosimetry computer programs. Instead we need a new technique to transform the interfaces into smooth surfaces. The Marching Cube (MC) algorithm was used and adapted to do this transformation. The initial image was used to generate a continuous gray-level field throughout the image. The interface between bone and marrow was then simulated by the iso-gray-level surface that corresponds to a predetermined threshold value. Calculations were then performed using this new representation. Excellent results were obtained for both the chord-length distribution and the energy deposition measurements. Voxel effects were reduced to an acceptable level and the discrepancies found when using the voxelized representation of the interface were reduced to a few percent. We conclude that this new model should be used every time one performs dosimetry estimates using NMR images of trabecular bone samples.

CHAPTER 1 INTRODUCTION

Dosimetry assessment within the trabecular bone regions of the skeleton has been a challenging investigation over the past four decades. The importance of these studies is that bones serve as the “housing” for hematopoietic marrow, the tissue responsible for producing a variety of different blood cells. Therefore, irradiation of bone regions has the potential to cause severe damage to the bone marrow and, as a consequence, can be life threatening for the individual exposed to the irradiation.

Several situations may result in internal irradiation of trabecular bone regions.

These include

- Occupational exposures to bone-seeking radionuclides ([Brodsky 1996](#))
- Therapy procedures using injected radiopharmaceuticals that transit through the skeletal system ([Rubin and Scarantino 1978](#); [Sgouros 1993](#); [Siegel et al. 1990](#))
- Therapeutic procedures for palliation of bone pain associated with bone cancers ([Samaratunga et al. 1995](#)).

In most of these situations, the radionuclide is a beta or an alpha emitter and the amount of energy deposited in healthy marrow can be extremely important. Therefore, the active bone marrow is firmly established as the dose-limiting organ in radionuclide therapies ([Rubin and Scarantino 1978](#); [Sgouros 1993](#); [Siegel et al. 1990](#); [Zanzonico and Sgouros 1997](#)).

Occupational exposures of bone-seeking radionuclides will be a challenge to internal dosimetry clean-up and waste management programs within the U. S. Department

of Energy (DOE). Currently, numerous sites need to be safely decontaminated or decommissioned. Potential inhalation or ingestion of bone-seeking radionuclides (by clean-up workers or by off-site populations) is a concern in most of these sites (DOE 1995). Six such radionuclides are ^{90}Y , ^{91}Y , ^{95}Zr , ^{95}Nb , ^{90}Sr and ^{226}Ra . Over the past 50 years both ^{90}Sr and ^{226}Ra have been studied extensively in bone dosimetry (Hindmarsh et al. 1958; Spiers 1966a; Spiers 1967; Vaughan 1960; Vaughan 1973).

Radiopharmaceuticals have been used increasingly as therapy agents in recent decades. Therefore a more accurate trabecular bone dosimetry is needed to limit risk to patients. The risk is present in bone therapy using bone-seeking radiopharmaceuticals, and also in any therapy in which the radionuclide transits through the blood system. The bone marrow is continuously irrigated by blood vessels and is thus exposed to the radiation. More accurate trabecular bone dosimetric models will allow one to calculate the dose to both the bone and the bone marrow with more precision. Improved skeletal dosimetry will allow physicians to better understand the biological effects of specific therapy procedures, which in turn will help improve nuclear-medicine techniques by maximizing the administration of therapeutic doses of radiopharmaceuticals.

Radiopharmaceuticals are also used for palliation of bone pain. This treatment is accomplished with bone-seeking beta-emitting radionuclides. Phosphorus-32, ^{89}Sr , ^{131}I , and ^{186}Re are four such radionuclides considered for this treatment (Samaratunga et al. 1995). Radiopharmaceutical treatments of bone pain cause marrow to receive a significant amount of the deposited energy.

More accurate trabecular bone dosimetry has the potential to improve our understanding of the dangers associated with the scenarios previously mentioned.

Therapeutic applications of radiation and radioactive materials would benefit from better dosimetry of these regions, and health risks associated with bone-seeking radiopharmaceuticals could be calculated more accurately.

Within the trabecular bone regions, the bone trabeculae have a thickness on the order of 300 μm and the marrow cavities have widths on the order of 1 mm. The smallest geometrical features can be on the order of 100 μm for both the bone and the cavities. Because the features of such bone microstructure are so small, energetic electrons will be able to traverse several different marrow cavities while continuously depositing energy. Therefore, a trabecular bone dosimetry can be done by either a study of the path length of the particles through the different regions, or by coupling a trabecular bone model with a transport code that can perform a calculation in such a microscopic structure. Instead of a model that could be difficult to build, bone samples can be imaged and the digital image coupled with the transport code. Nuclear magnetic resonance (NMR) and quantitative computed tomography (QCT) are two nondestructive techniques investigated during the last decade to provide bone sample images. The last studies use very high resolution in-vitro NMR imaging ([Chung et al. 1996](#); [Chung et al. 1995a](#); [Hwang et al. 1997](#); [Link et al. 1998b](#); [Majumdar et al. 1996](#); [Wessels et al. 1997](#)) as well as QCT imaging ([Ciarelli et al. 1991](#); [Cody et al. 1989](#); [Cody et al. 1991](#); [Cody et al. 1996](#); [Engelke et al. 1993](#); [Goulet et al. 1994](#); [Kinney et al. 1995](#); [Kuhn et al. 1990](#); [Link et al. 1997](#); [Link et al. 1998a](#); [Link et al. 1998b](#); [McCubbrey et al. 1995](#); [Muller et al. 1994](#); [Muller and Ruegsegger 1996](#); [Ruegsegger et al. 1996](#)). A resolution of 50 μm can be achieved with both techniques.

In 1995, an investigation of the feasibility of the use of NMR images to transport electrons through three-dimensional (3D) digitized representation of the trabecular bone

microstructure was initiated at the University of Florida ([Jokisch et al. 2001a](#); [Jokisch et al. 1998](#); [Jokisch et al. 2001b](#); [Patton et al. 2002a](#); [Patton et al. 2002b](#)) and satisfactory results were achieved. Chord-length distributions through both the bone trabeculae and the marrow cavities were also acquired using these 3D images. These distributions are of great interest for bone-marrow dosimetry since they can be compared with electron ranges to deduce energy deposition through the bone and the marrow regions. Furthermore, all current models of bone-marrow dosimetry are based on these distributions.

A few years ago, a study was initiated in an effort to assess the minimum voxel size required for NMR imaging in order to obtain reliable results for energy deposition calculation using Monte-Carlo codes to transport the particles through the images ([Rajon 1999](#)). Our aim was to reduce imaging time by choosing the largest image resolution possible without altering results, so that in vivo imaging could become a reality. This study reached two conclusions. First, for high-energy electrons, there is no need to obtain high-resolution images. With a voxel size as large as 300 μm the computer simulations are already in good agreement with the expected results. Second, for low-energy electrons the calculation could lead to overestimation of the cross-dose (marrow cavities irradiating the bone trabeculae or vice-versa) by up to 40 % for 50 keV electrons. This second conclusion was unexpected and stimulated further investigation of what caused the overestimation of the cross-dose and how the problem could be solved.

The work presented in this dissertation proposes to answer these two questions. [Chapter 2](#) gives the important background required to understand the whole development that follows. [Chapter 3](#) summarizes the previous study that led to the two conclusions sited above. It describes the development of a mathematical bone sample and how it was

used to benchmark the accuracy of the energy deposition calculation within 3D images. It shows that the digitization of a 3D image into voxels produces an overestimation of the surface area of the boundary between bone and marrow if measured along this voxelized representation of the boundary. As a consequence, the energy deposition by short-range electrons initiated from one side of the boundary and irradiating the other side is also overestimated. [Chapter 4](#) proposes to explain the origin of the surface-area overestimation in studying single sphere models. It also discusses how this overestimation of the surface area will lead to an overestimation of the cross-energy deposition calculated within the voxelized representation. Then, computer simulations of electron transport are conducted within these models to verify the prediction. The experimental results are shown to be in perfect agreement with the predicted results. Therefore, it is concluded that a better representation of the interface between bone and marrow is required. [Chapter 5](#) develops a new Monte-Carlo technique to simulate an isotropic and homogeneous radiation field that surrounds any object. This technique is then used in [Chapter 6](#) to measure chord-length distributions within representations of trabecular bone samples. In [Chapter 6](#), the mathematical models are used to assess the accuracy of the chord-length distribution measurements through the voxelized representation of the boundary. Older studies showed that these measurements were affected by voxel effects and some techniques were developed to reduce these effects ([Jokisch et al. 2001b](#)). A re-evaluation of these techniques is proposed and the overall results showed that their consequences depend on the voxel size. The conclusion is, again, that a better representation of the interface between bone and marrow is required.

[Chapter 7](#) proposes a new technique to model the bone-marrow interface by a smooth surface. The technique is an adaptation of the Marching-Cube (MC) algorithm and produces an isosurface within the gray level field of the image. This isosurface representation is shown to preserve the surface area of the interface and is expected to solve the problem of the energy-deposition calculation. It also gives excellent results when applied to chord-length distribution measurements and definitely solves the voxel effect problems. Finally, [Chapter 8](#) uses the same adaptation of the MC algorithm to couple the bone sample geometry with the Monte-Carlo transport code and to calculate the energy deposition by electrons through this representation of the bone-marrow interface. The technique is shown to eliminate the systematic overestimation found with the voxelized representation.

CHAPTER 2 BACKGROUND

Bone Structure and Physiology

Bone has two main types of histological structure: cortical bone (also referred to as hard compact bone) and trabecular bone (also referred to as spongy bone or cancellous bone). [Figure 2-1](#) shows the difference between compact bone and trabecular bone. Cortical bone comprises 80% of the skeletal mass ([Berne et al. 1993](#)). One of the few structures that penetrate the compactness of the cortical regions is the haversian canal system. [Figure 2-2](#) is a more detailed illustration of the compact bone structure. The Haversian canals are pathways used by the circulatory system to supply the living bone cells with nutrients. The dosimetry of cortical bone has been studied ([Akabani 1993](#); [Beddoe 1976a](#); [Beddoe 1977](#)). Recently Bouchet and Bolch ([1999](#)) developed a 3D transport model of cortical bone at the University of Florida.

Trabecular bone consists of a complex network of bone spicules (also called trabeculae) that surround cavities of marrow. [Figure 2-3](#) shows the complexity of the trabecular bone microstructure. The marrow cavities contain both active marrow (also called red marrow) and inactive marrow (also called yellow marrow). The active marrow is responsible for the production of blood cells, whereas inactive marrow is just marrow in which the hematopoietic cells have been replaced with fat cells. To allow the blood vessels to irrigate the bone marrow, the cavities are connected to one another thus also forming a complex network. As a consequence trabeculae and marrow cavities are two

separate lattices that interlace to each other ([Figure 2-3](#)). Trabecular bone exists in the inner regions of the vertebra, ribs, skull, pelvis and the end of the long bones. The presence of trabecular bone in the central part of long bones primarily occurs in newborns and very young children. But the trabecular regions gradually decay with age and rapidly disappear from these regions.

Where the bone surface interfaces either a Haversian canal (in cortical bone) or a marrow cavity (in trabecular bone), there is a thin layer of osteogenic cells called the endosteum. A similar layer exists on the exterior of the cortical bone called the periosteum. Because of the presence of red marrow the trabecular bone regions are of great importance for dosimetry studies. They constitute the area of interest in this research.

Bone Cell Types

The cellular components of cortical and trabecular bone are the same. Three types of bone cells control the production and destruction of osteoid matrix. Osteoblasts are responsible for ossification (the formation of bone). They are responsible for the formation of a collagen-based osteoid matrix and the deposition of calcium in the form of calcium phosphate into this matrix. Osteoblasts exist on the endosteal and periosteal surfaces, and are the most radiosensitive of the three bone cells ([Vaughan 1960](#)). The stem cells that produce the osteoblasts also exist on the endosteal and periosteal surfaces. These pre-osteoblasts are very radiosensitive.

Osteoclasts counteract the work of the osteoblasts by destroying or resorbing bone. Osteoclasts exist on the endosteal bone surfaces and are capable of removing

calcium phosphate and destroying the osteoid matrix. They do this by producing hydrolytic enzymes that digest minerals and bone matrix (Vaughan 1975).

Osteocytes, the third type of bone cells, are simply osteoblasts that bury themselves within the osteoid matrix over time. Obviously, osteocytes are not on the endosteal surface. However, osteocytes are believed to aid in the osteogenic processes by staying in contact with osteoblasts through channels called canaliculi. Osteocytes may become active osteoblasts again if they are uncovered by bone resorption later in life. For these reasons, the degree to which the bone volume is dosimetrically important is not fully understood. Figure 2-3 also shows the location of these different bone cells.

Variation of the Trabecular Bone Microstructure

The creation and resorption rates result in a process called bone remodeling that represents approximately a 15% bone mass turnover every year for an adult human (Berne et al. 1993). Bone mass peaks in humans between 20 and 30 years of age. Remodeling reaches equilibrium around 35 to 40 years of age, and then decreases for the remainder of life (Berne et al. 1993). Because women have smaller overall mass than men, this natural loss of bone, especially when coupled with hormonal losses due to menopause, can create bone structural problems such as osteoporosis.

Trabecular structure varies with age (Atkinson 1965; Atkinson 1967; Snyder et al. 1974), gender (Atkinson 1967; Mosekilde 1989), skeletal site (Eckerman 1985), and skeletal orientation (Atkinson 1967; Atkinson and Woodhead 1973; Hahn et al. 1992; Mosekilde 1989). Because of these variations, trabecular microstructure data is more properly characterized by distributions as opposed to mean values. The ossification and resorption rates mentioned above also vary with skeletal site and orientation. Obviously,

any change in the size of trabeculae corresponds to a change in marrow cavity size. Additionally, the percentage of active marrow that fills the cavities changes with age (Custer and Ahlfeldt 1932; Ellis 1961; Mechanik 1926). For these reasons, any study on the microstructure of trabecular regions of bone must pay attention to age and gender-related changes in that microstructure.

Trabecular bone structure can also be highly anisotropic in its structure (Cowin 1989; Turner 1992; Williams and Lewis 1982). Several studies found a difference between horizontal and vertical trabecular structure in bones such as vertebrae (Atkinson 1967; Atkinson and Woodhead 1973; Hahn et al. 1992; Mosekilde 1989). These studies suggest that the ossification and resorption processes in bone respond to compression and stress. In the case of a vertebra, the horizontal struts are not as structurally important as the vertical segments and they thin considerably faster than the vertical segments with subject age. At least one study did not find a horizontal resorption preference (Snyder et al. 1993).

Trabecular and cavity sizes also vary with skeletal location. The fractional mass of active marrow also varies with skeletal location. Some skeletal sites, such as the adult vertebrae, are more important to marrow dosimetry than others because they contain a larger portion of active marrow. Since yellow marrow does not contain appreciable populations of hematopoietic stem cells, skeletal regions containing yellow marrow are only of dosimetric importance in endosteal tissues.

The geometry and composition of the trabecular regions of the skeleton create several dosimetry problems. Since bone-marrow cavities are located within the trabecular bone structure, the dimensions of the two interlacing regions must be accurately known in

order to calculate the absorbed dose to these sites. The anisotropic structure of these regions further complicates any dosimetry studies because it is difficult to apply any sort of uniform modeling technique to such a complex geometry. Furthermore, the small sizes of trabecular and cavity regions, relative to the ranges of typical beta particles emitted from bone-seeking radionuclides, imply that an electron may traverse several cavities while continuously depositing kinetic energy. Two electrons with the same energy and starting point may take completely different paths, traversing differing amounts of marrow and bone as shown in [Figure 2-4](#).

Internal Dosimetry

Energy absorption by body tissue while exposed to internal ionizing radiation has been studied intensively and numerous assessment techniques have been proposed. In 1968, a new technique was introduced by the Society of Nuclear Medicine's Medical Internal Radiation Dose (MIRD) Committee ([Loevinger and Berman 1968a](#); [Loevinger and Berman 1968b](#)). This technique was improved over time and is now known as MIRD Primer ([Loevinger et al. 1991](#)). The purpose of this section is to briefly describe this technique.

Each organ or tissue of the body can be seen as a source of radiation if it has been contaminated by a radionuclide or as a target organ when it absorbs energy from one or several source organs. Of course, a source organ always irradiates itself and is also seen as a target organ. As a consequence of a contamination by a radionuclide, the dose D received by a target organ is the sum of the doses received from the different source organs and can be defined by

$$D_k = \sum_h D_{(k \leftarrow h)} . \quad (2-1)$$

In [Equation \(2-1\)](#) k represent the target organ and h the different source organs. The

MIRD Primer clearly divides the assessment of $D_{(k \leftarrow h)}$ into two separate problems:

- The cumulated activity \tilde{A}_h represents the total number of disintegrations of the radionuclide that occur during the contamination time.
- The S value $S_{(k \leftarrow h)}$ represents the average dose received by the target organ k per disintegration within the source h .

Using these definitions, [Equation \(2-1\)](#) can be rewritten as

$$D_k = \sum_h \tilde{A}_h S_{(k \leftarrow h)} . \quad (2-2)$$

The cumulated activity is calculated by integrating the activity within the source organ over time. Calculation of the activity is usually complex and must consider

- Build up of activity in the organ (may occur through various biological path ways)
- Physical decay of the radionuclide
- Several biological decays (such as transition to another organ or natural elimination of the radionuclide from the body).

This calculation is simplified by introducing a residence time ([Loevinger et al. 1991](#)) that relates the cumulated activity to the activity introduced to the body. The purpose of this dissertation is not the assessment of cumulated activity. Therefore, I do not discuss determination of residence time here.

The S value represents the average dose received by a target organ when a single disintegration occurs in a source organ. Depending on the radionuclide, several particles can be emitted per disintegration, or a particle can be associated to a probability of being

emitted: its yield. To account for these different situations, the S value is decomposed into

$$S_{(k \leftarrow h)} = \frac{\sum_i n_i E_i f_{i,(k \leftarrow h)}}{m_k}. \quad (2-3)$$

In Equation (2-3), i represents the different particles that can be emitted by the radionuclide. Each particle is defined by its type (photon, electron, alpha, etc) and its energy. The parameter n_i is the yield of this specific particle, E_i is its initial energy, and $f_{i,(k \leftarrow h)}$ is the average fraction of initial energy that is absorbed by the target organ. The parameter m_k is the mass of the target organ.

The yield n_i and the initial energy E_i are characteristics of the radionuclide. They are provided by the radionuclide spectrum. As an example, for a beta particle, i refers to a fraction of the beta spectrum that corresponds to an energy from E_i to $E_i + dE_i$. In this case, n_i represents the probability for the initial energy to be between E_i and $E_i + dE_i$. The absorbed fraction (AF) $f_{i,(k \leftarrow h)}$ depends on the geometry of the two organs, the tissue composition of the two organs, as well as the tissue composition and the geometry of the organs that lie between the source and the target. The AF can be determined by analytical methods such as Point Kernel, or by using Monte-Carlo transport codes. The purpose of this dissertation is to provide a more accurate calculation of the AF of energy within bone marrow.

Past Skeletal Dosimetry Works and Models

F. W. Spiers is responsible for most of the early work on skeletal dosimetry. While at the University of Leeds, Spiers (1949; 1951) began with relatively simple studies

on bone and soft tissue interface, and the unique dosimetry associated with that region. Spiers (1963) later looked at influences of the percentage of active marrow on trabecular dosimetry. He used active marrow distribution data derived by Ellis (1961) from the work of Mechanik (1926) and Custer and Ahlfeldt (1932).

Spiers (1966b; 1967) was the first to recognize that the anisotropic structure of trabecular bone required a unique method for characterizing the geometry in order to perform accurate skeletal internal dosimetry of beta-emitters. This characterization was originally performed using microscopes and visual inspection.

At this point, Spiers' group began working on a method to describe the geometry in terms of frequency distributions of linear path lengths through the trabecular and marrow regions. These distributions are called chord-length distributions. A variety of methods for obtaining these distributions exist. They depend on the origin and direction of the rays relative to the object. As pointed out by Eckerman et al. (1985), failure to account for the distinct nature of these distributions can result in misunderstanding some aspects of the radiation transport processes. Three fundamental methods of randomly obtaining these frequency distributions are relevant in trabecular dosimetry.

- Mean-free-path randomness (or μ -randomness). A chord of a convex body is defined by a point in space and a direction. The point and the direction are chosen randomly from independent uniform distributions. This kind of randomness results, for example, if the convex body is exposed to a uniform, isotropic field of straight lines.
- Interior radiator randomness (or I-randomness). A chord is defined by a point in the interior of the convex body and a direction. The point and the direction are chosen randomly from independent uniform distributions. This kind of randomness results, for example, if the convex body contains a uniform distribution of point sources, each of which emits radiation isotropically.
- Surface radiator randomness (or S-randomness). A chord is defined by a point on the surface of the convex body and a direction. The point and the direction are chosen

randomly from independent uniform distributions. This kind of randomness results, for example, if the surface of a convex body contains a uniform distribution of point sources, each of which emits radiation isotropically (Kellerer 1971).

When an electron starts from the bone-marrow interface (surface-seeking radionuclide) its first path length is better characterized by an S-randomness. When it starts from the volume of a bone trabecula or a marrow cavity, its first path length is better characterized by an I-randomness. After this electron has left the first medium, its next path lengths are better characterized by a μ -randomness. As seen previously (Figure 2-4), a single electron is likely to cross several bone and marrow regions before it loses its entire energy. Therefore, the overall electron transport is better characterized by a μ -randomness. As a consequence, μ -randomness is the method most investigated and thus used for our study.

Eventually, the Leeds group was able to automate the process of obtaining chord distributions (Beddoe 1976b; Beddoe et al. 1976; Darley 1972; Spiers 1969). They physically sectioned trabecular bone regions into thin slices and took contact radiographs of the slices. In their techniques, the radiograph is mounted on a turntable below a light microscope. Above the turntable is a photo-multiplier tube. As the turntable rotates, the radiograph is also moved radially, creating scan lines of minimal arc. The duration of a light pulse seen by the photo-multiplier corresponds to a marrow chord length. The opposite is true of trabecular chords. Darley (1972) developed the original apparatus. Later, Beddoe (1976a) used the system and improved on the radiography and preparation of the bone sections.

The distance between consecutive scan lines in the Leeds optical scanner is approximately 8 μm . The Leeds scanner had a dead time of 39 μs after the registration of

a pulse length. This corresponds to a lost path length of 100 μm as the turntable continued to rotate. They justified this loss by concluding that the lost path length occurred almost entirely over the features which were not measured (Beddoe 1976a). The bone and marrow chord lengths are measured in separate scans. Thus, if marrow chords were being measured and the next bone chord was less than 100 μm , the next marrow chord would be registered as being smaller than actual size. The opposite would be true for the bone chord scan. The effective resolution of their scanning system including film noise was reported as 11.5 μm (Beddoe 1976a).

To obtain omnidirectional distributions, Spiers and his colleagues had to make some symmetry assumptions. In the case of human vertebrae, the Beddoe (1976a) study found symmetry in one direction. The conclusion was that a scanning measurement in any set of parallel planes cut parallel to the symmetry axis was sufficient to generate an omnidirectional distribution.

Use of Nuclear Magnetic Resonance

Interest in bone micromorphology extends well beyond the radiation dosimetry community. Many studies focused on trabecular microstructure in an attempt to measure various structural parameters that might prove to be statistically reliable predictors of bone fracture risk in diseases such as osteoporosis. These studies have traditionally been performed using optical microscopy (Odgaard et al. 1990; Parfitt et al. 1983) or scanning electron microscopy (Boyde et al. 1986; Whitehouse and Dyson 1974; Whitehouse et al. 1971). These techniques require substantial sample preparation and are inherently destructive to the specimen. Furthermore, physical sectioning does not permit viewing the

sample in multiple planes. Considering the anisotropic nature of trabecular bone, multi-plane viewing is a highly desirable feature.

Over the past decade, several groups have investigated the use of two nondestructive techniques for high-resolution imaging of trabecular bone: QCT and NMR microscopy. QCT has been used to measure regional bone mineral density (Bone et al. 1994; Cody et al. 1989; Cody et al. 1991; Flynn and Cody 1993; Grampp et al. 1996; Kleerekoper et al. 1994a; Kleerekoper et al. 1994b; Link et al. 1997; Link et al. 1998a; Link et al. 1998b; Majumdar et al. 1997) and bone structural parameters (Chevalier et al. 1992; Durand and Ruegsegger 1992; Kuhn et al. 1990; Link et al. 1997; Link et al. 1998a; Link et al. 1998b; Majumdar et al. 1997; Muller and Ruegsegger 1996) at various skeletal sites *in vivo*. Several studies have used QCT for *in vitro* analyses of trabecular bone (Ciarelli et al. 1991; Cody et al. 1989; Cody et al. 1991; Cody et al. 1996; Engelke et al. 1993; Goulet et al. 1994; Kinney et al. 1995; Kuhn et al. 1990; Link et al. 1997; Link et al. 1998a; Link et al. 1998b; McCubbrey et al. 1995; Muller et al. 1994; Muller and Ruegsegger 1996; Ruegsegger et al. 1996).

NMR microscopy represents an alternative to QCT for analyzing trabecular bone microstructure. This imaging technique is ideally suited for this purpose because bone marrow is composed primarily of water and lipids, thus providing an abundant source of proton MR signal. Conversely, bone does not contain hydrogen in the abundance or chemical form needed to produce a sufficient signal. The bone-marrow interface thus ensures a high intrinsic contrast in the resulting image. Several investigators have used NMR microscopy to quantify trabecular microarchitecture both *in vivo* (Chung et al. 1994; Foo et al. 1992; Gordon et al. 1997; Grampp et al. 1996; Guglielmi et al. 1996; Jara

et al. 1993; Majumdar et al. 1997; Ouyang et al. 1997; Sebag and Moore 1990; Song et al. 1997; Wong et al. 1991) and *in vitro* (Chung et al. 1996; Chung et al. 1995a; Hwang et al. 1997; Majumdar et al. 1996; Wessels et al. 1997). Isotropic resolutions for these *in vitro* studies have been reported in the range of 50 to 150 μm . NMR microscopy of human subjects *in vivo* have yielded images with voxels ranging from 78 to 200 μm in plane and with slice thickness of 500 to 700 μm using clinical MR units at a field strength of 1.5 T (Jara et al. 1993; Majumdar and Genant 1995). Improved resolutions for *in vivo* NMR imaging would be possible using newer 3 Tesla clinical machines.

NMR microscopy presents three distinct advantages over QCT for studying the microarchitecture of trabecular bone for improved radiation dosimetry modeling. First, NMR does not use ionizing radiation and thus applications to *in vivo* imaging are not dose limited. Second, NMR images may be acquired in any arbitrary plane without sample or subject repositioning. QCT would require sample positioning to acquire a specific orientation. Finally, NMR microscopy can be used to assess the fat and lipid mass fractions of the marrow spaces (Ballon et al. 1991; Ballon et al. 1996; Dixon 1984; Glover and Schneider 1991). NMR microscopy allows one to assess the trabecular and marrow chord-length distributions through the sample, and also to determine the fraction of marrow space occupied by active marrow (i.e., the regional marrow cellularity).

Bone Sample Imaging and Image Processing

Since Spiers' study, very little has been done in trabecular bone dosimetry. The chord-length distributions measured at Leeds during the 1970s became the reference and are still used today. While the research of Spiers and his students was superb in its level of detail and completeness, future improvements in skeletal dosimetry, particularly for

medical applications, require an expanded database of marrow and trabecular chord distributions. These databases should more fully encompass variations in trabecular microstructure with both subject age and gender. Newer techniques in medical imaging can now be applied to this particular task.

A few years ago, studies were initiated at the University of Florida ([Jokisch 1997](#); [Jokisch 1999](#); [Jokisch et al. 2001a](#); [Jokisch et al. 1998](#); [Jokisch et al. 2001b](#); [Patton 1998](#); [Patton 2000](#); [Patton et al. 2002a](#); [Patton et al. 2002b](#); [Rajon 1999](#)) to look at techniques that could be used to improve on, or expand, the microstructure database gathered at Leeds. This work resulted in choosing nuclear magnetic resonance (NMR) microscopy coupled with image-processing techniques to obtain chord distributions and Monte-Carlo transport codes to perform dose calculations. [Figure 2-5](#) shows an example of 3D image obtained with an NMR system. This raw image is a digital image in which each voxel is a gray level number between 0 and 255. If the image on [Figure 2-5](#) shows perfectly the location of the bone regions and the marrow regions, it's because your eyes constitute a powerful tool that can analyze the image. A computer program does not have this ability and the image needs to be processed in order to separate the bone and marrow voxels. This was done by [Jokisch et al. \(1998\)](#) using an image-processing technique that can be divided into three steps.

- Image extraction in which a region of interest is extracted from the entire image (already done in [Figure 2-5](#))
- Image thresholding and segmentation using a mathematical process to determine the gray level limit between bone and marrow
- Image filtering to eliminate abnormal high or low intensity voxels arising from signal noise.

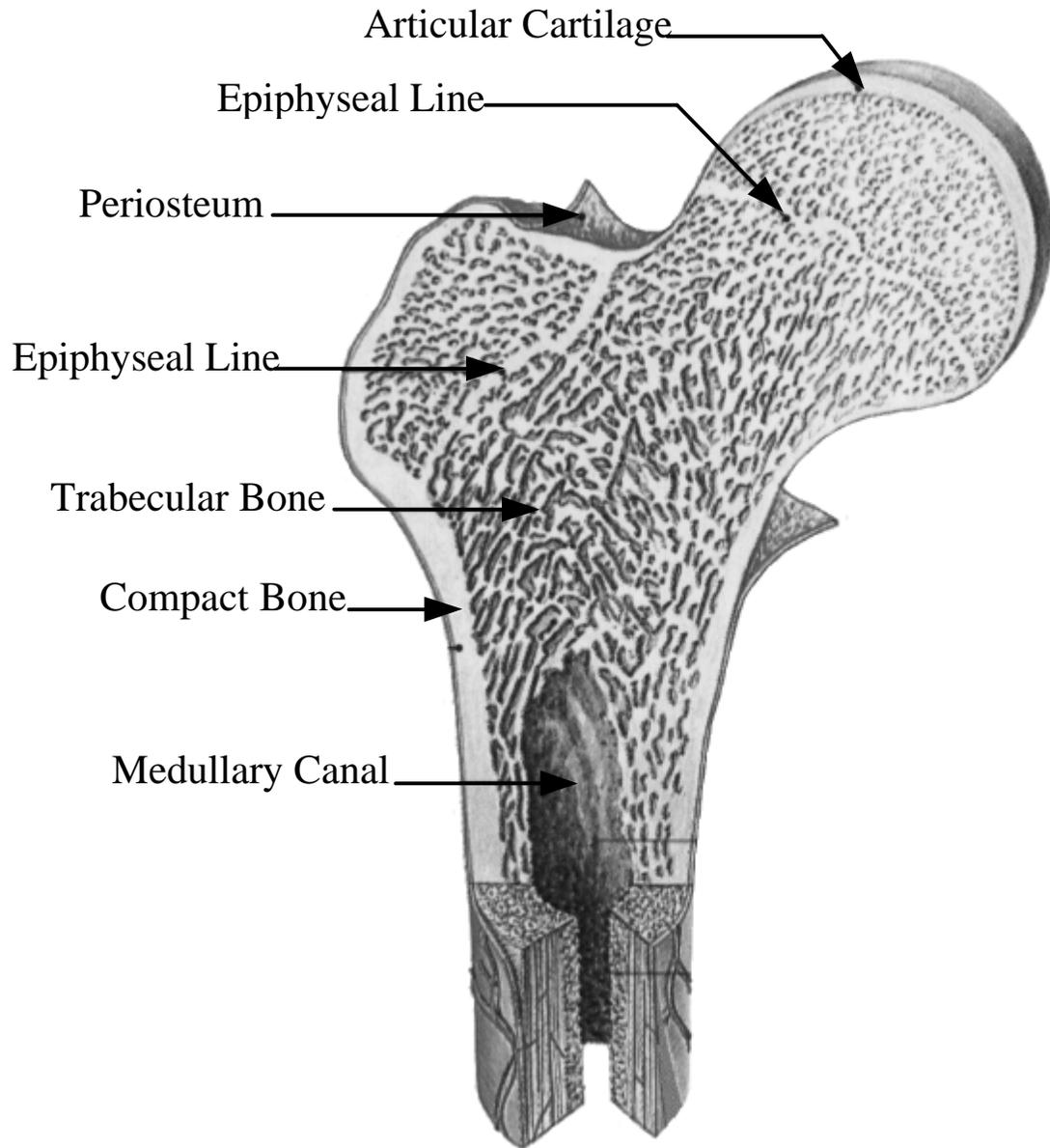


Figure 2-1. Femur head showing the different constituents of the bone structure. Adapted from a study by Takahashi (1994).

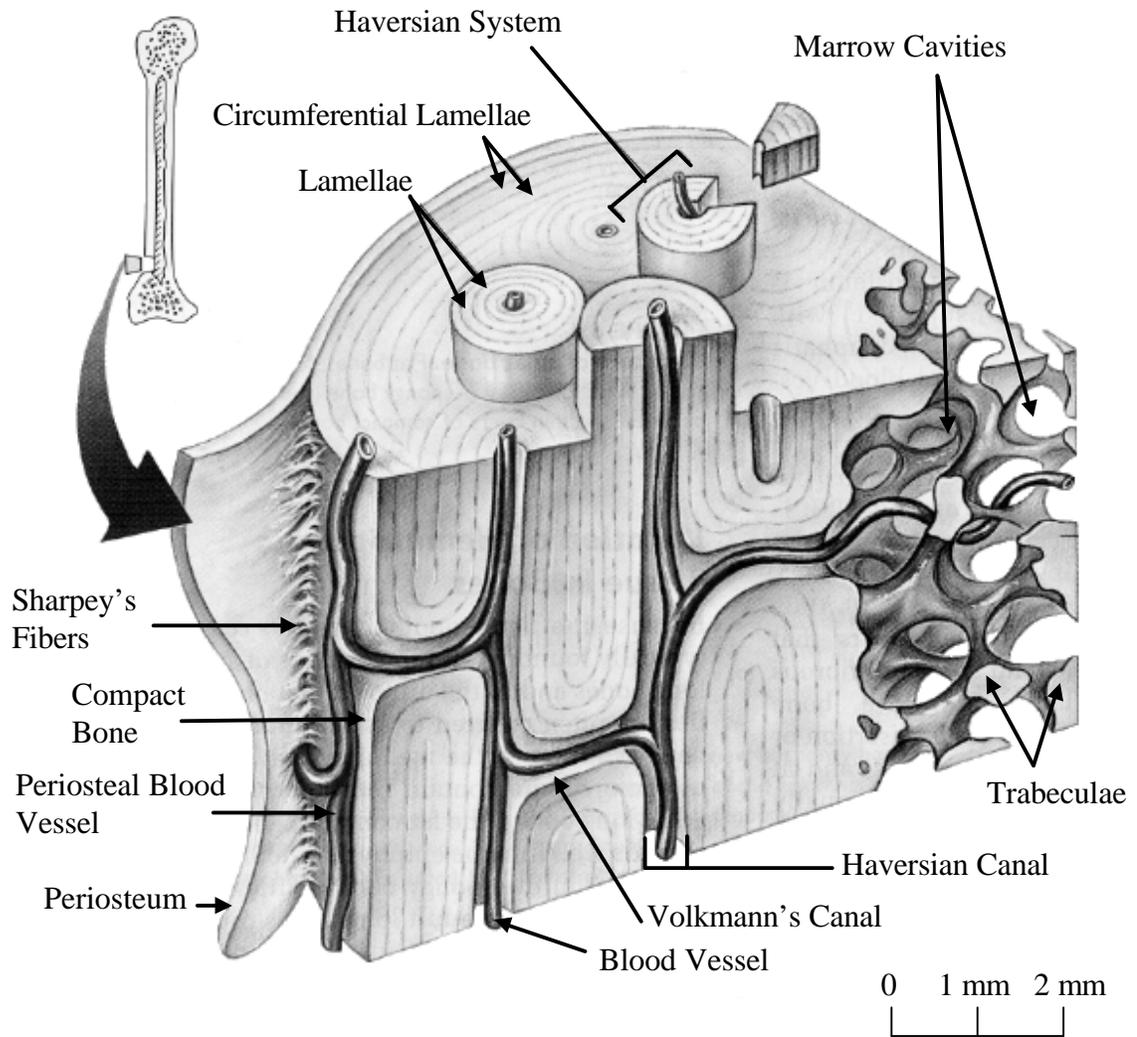


Figure 2-2. Microstructure of compact bone and trabecular bone. Adapted from a study by Marieb (1998).

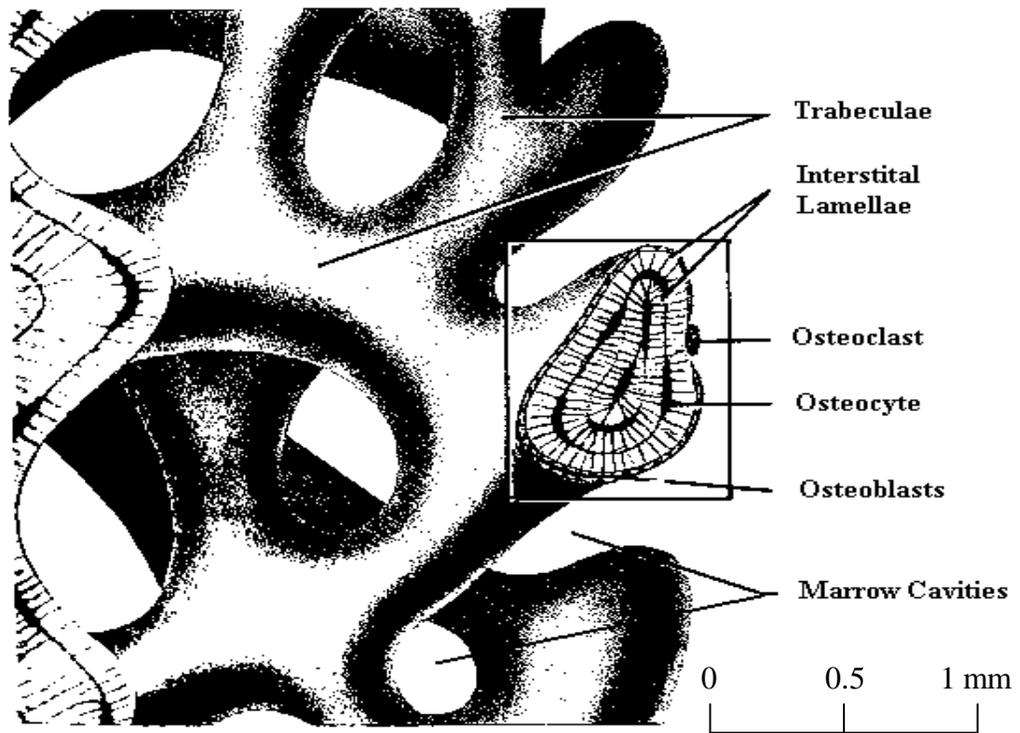


Figure 2-3. Microstructure of trabecular bone showing the types of bone cells and their location. Adapted from a study by Tortora (1992).

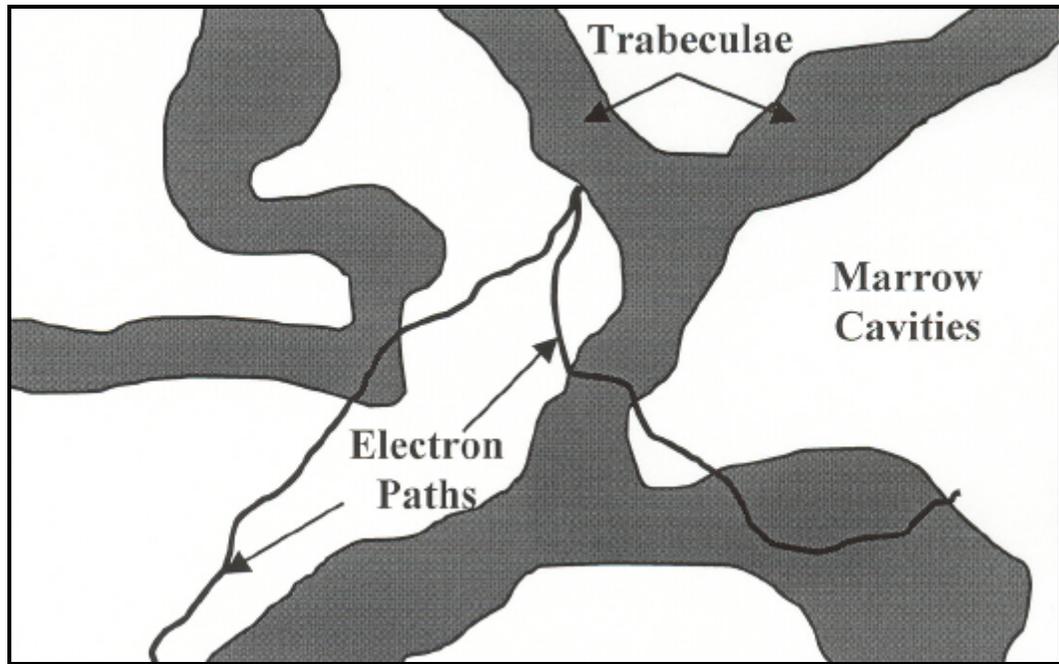


Figure 2-4. Differing paths of two electrons emitted within anisotropic regions of trabecular bone.

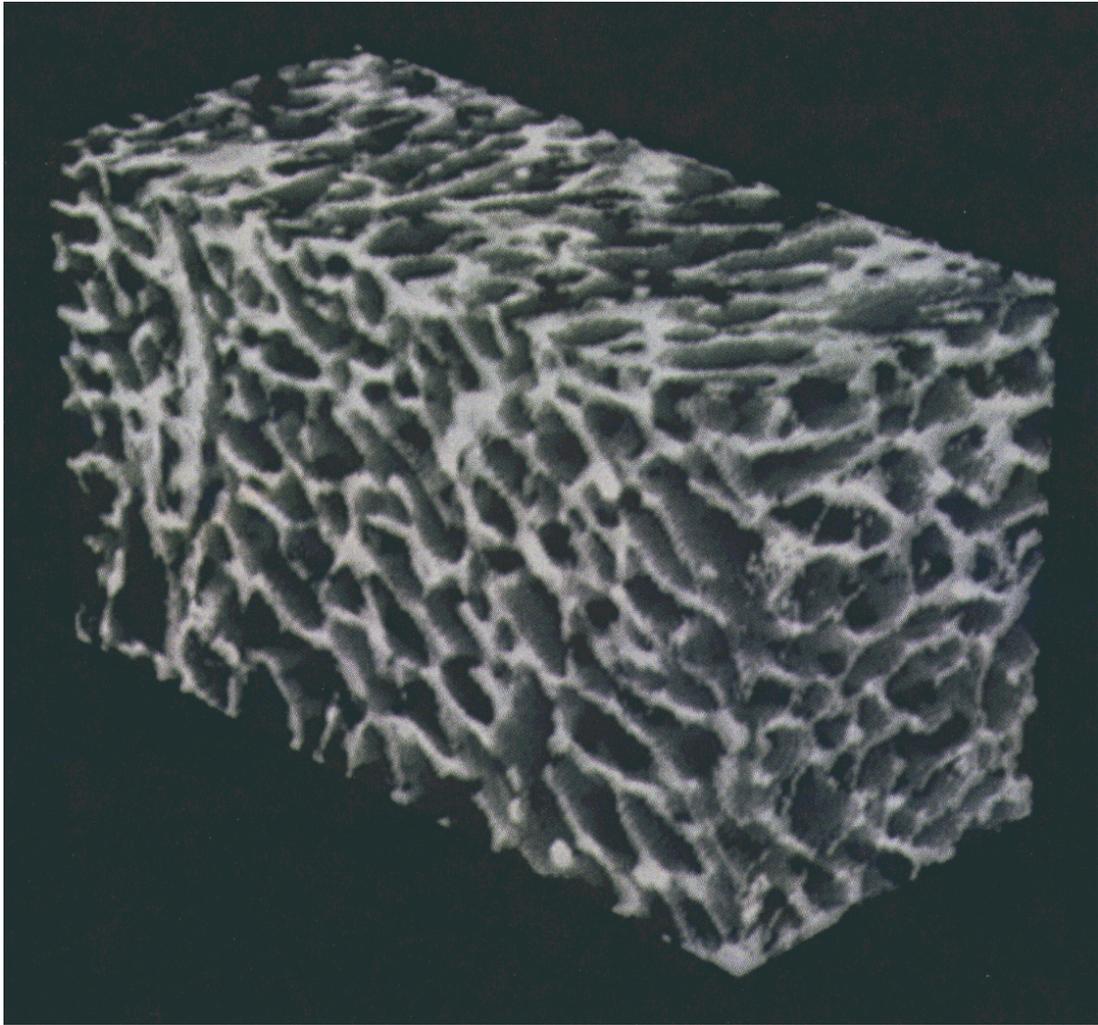


Figure 2-5. Trabecular bone 3D image obtained from a 4.7 Tesla NMR system. The image was taken over an 11 hour and 10 minute acquisition time (TR = 600 ms, TE = 9.1 ms, spectral width: 123,457 Hz, 2 averages, matrix: 512 x 256 x 256, field of view: 4.5 x 2.25 x 2.25 cm³). The sample comes from the right femur head of a 51-year-old man. The sample size is 1.81 x 1.06 x 0.80 cm³. The resolution is 88 x 88 x 88 μm³.

CHAPTER 3
VOXEL-SIZE EFFECTS IN 3D NMR MICROSCOPY PERFORMED FOR
TRABECULAR BONE DOSIMETRY¹

Introduction

Assessment of radiation absorbed dose within trabecular regions of the skeleton continues to be an important challenge in medical dosimetry. Skeletal dosimetry is important in that trabecular bone serves as the “housing” for hematopoietic marrow, the tissue responsible for the production of a variety of different blood cells. Several situations may result in internal irradiation of bone marrow. These include occupational exposures to bone-seeking radionuclides ([Brodsky 1996](#)), radionuclide therapy of tumors ([Rubin and Scarantino 1978](#); [Sgouros 1993](#); [Siegel et al. 1990](#)), and bone pain palliation treatments ([Samaratunga et al. 1995](#)).

Previous Studies in Trabecular Bone Dosimetry

Active bone marrow (also referred to as red marrow) is located within the trabecular regions of the skeleton ([Clayman 1995](#); [Marieb 1998](#)). The difficulty in assessing marrow dose is due to the complex structure of the trabecular regions. [Figure 3-1](#) shows how the trabecular lattice interlaces with the marrow cavities. In [Figure 3-1](#), the thickness of the trabeculae is ~300 μm and the sizes of the marrow cavities are on the order of 1000 μm . Furthermore, several studies have shown that the bone microstructure varies as a function

¹ This chapter was published by Medical Physics in November 2000: Rajon DA, Jokisch DW, Patton PW, Shah AP, and Bolch WE. 2000. Voxel size effects in three-dimensional nuclear magnetic resonance microscopy performed for trabecular bone dosimetry. *Med Phys* 27: 2624-2635.

of age (Atkinson 1965; Ouyang et al. 1997; Snyder et al. 1993) and between the sexes at a given age (Atkinson 1967; Mosekilde 1989).

F. W. Spiers and colleagues conducted the first comprehensive studies on bone dosimetry at the University of Leeds during the 1960's. Their work demonstrated that a trabecular bone sample could be characterized by its chord-length distributions acquired through both the bone trabeculae and the marrow cavities. Consequently, one of their main accomplishments was to measure these two chord-length distributions within thin physical sections of trabecular bone using an optical scanning system (Beddoe 1976b; Spiers 1966b; Spiers 1967). His results were the first attempt to characterize the trabecular microstructure, and they have since been used as the basis for almost all subsequent skeletal dosimetry models (Bouchet and Bolch 1999; Bouchet et al. 2000; Bouchet et al. 1999b; Eckerman 1985; Spiers et al. 1978a; Spiers et al. 1978b; Stabin 1996). Today, Spiers' research still serves as a reference data source in trabecular bone dosimetry and his distributions are used to compare the results of this present study.

Use of NMR Microscopy

During the 1990s, nuclear magnetic resonance (NMR) was shown to be an important tool for quantifying the trabecular microstructure. Several investigators have used NMR microscopy to obtain 3D images of in-vitro samples with image resolutions below 100 μm (Chung et al. 1996; Chung et al. 1995b; Hwang et al. 1997; Link et al. 1998b; Wessels et al. 1997). Recently, Jokisch et al. (1998) investigated a technique to calculate the chord-length distribution in these high-resolution images. Using a 600-MHz NMR spectrometer they obtained a 59 x 59 x 78 μm^3 -resolution image of a thoracic vertebra. After image filtration and image segmentation of the raw data, Jokisch et al.

showed that the chord-length distributions calculated from this image were consistent with the Spiers' distributions obtained in both the cervical and lumbar vertebrae for a similarly aged male subject. This study showed that NMR is an excellent tool for acquiring 3D images of trabecular bone. Coupling these images with a particle transport code thus allows one to assess the absorbed dose in bone marrow using particle-transport techniques within the full realistic 3D structure of the skeletal region.

At the University of Florida, we have acquired images of trabecular bone samples for subsequent dosimetry analysis at both high and moderate magnetic field strengths (14.1 T and 4.7 T) with acquisition times of several hours. One of the many parameters to be optimized in these images is the voxel resolution of the final 3D image as controlled by the image matrix size and physical size of the sample. A decreasing voxel size necessarily increases the required acquisition time to maintain a reasonable signal-to-noise ratio (SNR). Consequently, it is important to consider the dosimetric consequences of image voxel size in order to optimize our NMR image-acquisition techniques.

Voxel-Size Effects in Skeletal Dosimetry Calculations

Voxel-size effects are demonstrated in [Figure 3-2](#). This figure shows a schematic slice of a voxelized image, as in those obtained by Jokisch et al. (1998) via NMR microscopy, after image segmentation and filtration. The two curved lines represent the true boundaries between a bone trabecula and its two adjacent marrow cavities. The grid represents the pixels of the image. Each pixel may be assigned to marrow (the white pixels) if more than 50% of its volume is within marrow cavities. The pixel is assigned to bone (the dark pixels) if less than 50% of its volume is composed of marrow. Consider an electron traveling within a bone trabecula as indicated by the black arrow. In the real bone

sample, the electron will deposit its entire energy in bone. In the digital image composed of pixels, the particle travels across both marrow (white pixels) and bone (dark pixels). As a result, the electron traveling in the digital image deposits its energy in both marrow and bone overestimating the energy deposition within marrow. Obviously, this error will be reduced if the voxel size is reduced, but what is not known is the minimum voxel size needed to reduce the error to an acceptable value. In some cases, the overestimate of absorbed dose to the marrow (situation described above) may be compensated by an overestimate of dose to the bone for a particle traveling on the other side of the boundary. A few other effects can be expected from the voxelization of the sample. Among them, one can image the variation of the volume of each media when the voxel size is increased and also the variation of the surface area of the boundary between bone and marrow when a curved interface is approximated by a series of orthogonal rectangular surfaces. These two effects are discussed in detail when analyzing the results of this chapter.

Material and Methods

Construction of the Mathematical Model of Trabecular Bone

A mathematical bone model was created in this study to evaluate the effects of the voxel size on electron transport calculations. The model had to be constructed so that charged particles deposit their energy within the different media as they would if traveling within a real bone sample. From the studies of Spiers *et al.*, it is known that this condition can be reasonably met if both the trabecular and marrow cavity chord-length distributions within the mathematical sample match their corresponding distributions within a real skeletal sample. In this study, the Spiers chord distributions for the cervical vertebrae were thus selected for initial investigation.

Figure 3-1 shows that the boundaries between bone and marrow in the trabecular regions are smooth curved surfaces. Consequently, the mathematical sample had to be built with smooth surfaces and without sharp angles. Surface discontinuities are responsible for voxel effects and must be avoided within the mathematical sample. To satisfy these characteristics, a cubical bone volume was filled with spheres each representing a marrow cavity. The bone trabeculae are represented by the spaces between the spheres. The size of the sample cube had to be large enough so that it could represent a contiguous piece of trabecular bone, yet small enough so that the computer simulations of particle transport could treat the segmented images in a reasonable amount of time. The sample size that satisfies these two conditions was found to be $1.6 \times 1.6 \times 1.6 \text{ cm}^3$.

The sample had to be uniform in its microstructure over its entire volume and for this one needed to deal with “the edge effect”. To understand “the edge effect”, consider the case of a 2D image. Imagine a square in which one tries to fit circles (Figure 3-3). If the centers of the circles are randomly chosen inside the exterior square shown in Figure 3-3, at least one quarter of the area of each circle overlaps the exterior square (this fraction becomes one eighth the volume of a sphere for a 3D model). In Figure 3-3, this situation applies for the exterior square where one does not see spherical overlaps smaller than a quarter-circle. Much smaller fractions of marrow cavities, however, are found along the cut edges of a real trabecular bone sample. Next, consider the interior square shown in Figure 3-3. With the same distribution of spheres, and if the distance between the two squares is greater than the maximum radius of the circles, the distribution of spheres within the interior square is uniform over its entire volume and no “edge effects” are present. For the current study, the maximum radius of the spheres is less than 3 mm,

and thus the exterior cube of our mathematical sample must be 6 mm wider than the interior cube. As a result, the mathematical trabecular bone sample was built by positioning the centers of the marrow spheres randomly throughout a $2.2 \times 2.2 \times 2.2 \text{ cm}^3$ cubical region, but only the central $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ cube was selected as the region of interest (ROI) for dosimetry studies.

The next decision in building the mathematical sample was to choose the size of the spheres. Their radii had to follow a probability distribution that provides, for the whole sample, a marrow chord-length distribution similar to that of the Spiers cervical-vertebra distribution. To avoid surface discontinuities, the spheres needed to be located so that they do not overlap with one another. After several trials using various distributions of radii, it was found that an exponential distribution worked best:

$$P(r) = P_m e^{-P_m r}, \quad (3-1)$$

where r is the radius of the spheres and P_m is a parameter that allows changes to the average value of the distribution. The value of P_m that gives the best fit to the Spiers' marrow chord-length distribution for the cervical vertebrae was found to be 44 cm^{-1} . An advantage of this mathematical sample is that the number of spheres does not affect the marrow chord-length distribution. While keeping the same radius distribution, an increase in the number of spheres within the sample will increase the volume fraction of marrow but not its chord-length distribution.

Increasing the number of spheres inside the cube will also reduce the bone volume and will obviously reduce the mean trabecula chord length. Therefore, once the marrow radius distribution is defined, the only concern is to try to fit as many spheres as needed within the mathematical sample cube so that the mean trabecula chord-length distribution

is reduced to a value comparable to the Spiers' value. A C-program was written to build the sample by fitting the spheres inside the sample cube. The filling of the sample cube was a long process and the program was not able to fit more than 28,200 spheres in the sample cube, giving a bone chord-length distribution slightly skewed from that of Spiers.

The final mathematical sample thus contains 28,200 spherical marrow cavities. This large number is a major concern if one thinks about a transport code having to deal with such a large number of regions. With the sample cube occupying a volume of $2.2 \times 2.2 \times 2.2 \text{ cm}^3$, and as only the central $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ is considered the dosimetric region of interest, more than 60% of the volume of the original sample cube is contained outside the central ROI. Consequently, many spheres located along the edge will never overlap the region of interest. Therefore, a last step before the mathematical sample is complete is to remove all spheres located in the $2.2 \times 2.2 \times 2.2 \text{ cm}^3$ bone sample that do not in some way intersect the central ROI. The total number of remaining marrow spheres within the ROI is thus reduced to 11,605.

Using the radius distribution of Equation (3-1), it is possible to calculate both the total volume of marrow and the total surface area of the bone-marrow interface. The analytical expressions for both the marrow volume and the interface area are:

$$V = \frac{8pN}{P_m^3}, \quad \text{and} \quad (3-2)$$

$$S = \frac{8pN}{P_m^2}, \quad (3-3)$$

where N is the number of spheres. With $N = 11,605$ and $P_m = 44 \text{ cm}^{-1}$, the total marrow volume and interface surface area are 3.424 cm^3 and 150.7 cm^2 , respectively. The mean

marrow chord throughout the mathematical bone sample may be estimated through Cauchy's theorem ($4V/S$) as $909 \mu\text{m}$. Note that these values are inclusive of all marrow spheres whereas some spheres only partially overlap the ROI boundary. Consequently, the true values of the marrow volume fraction and the surface area of the bone-marrow interface within the dosimetric ROI are expected to be smaller than predicted by [Equations \(3-2\)](#) and [\(3-3\)](#).

A C-program was also created to calculate the chord-length distribution for both the marrow cavities and bone trabeculae within the mathematical sample. As the sample is made of spheres, chord lengths are isotropic, and the calculation can be performed using a uniform field of parallel rays crossing the entire sample along one of the three axes.

Another program calculates the volume fraction of marrow within the sample. As it is difficult to find an analytical expression for the volume fraction of a sphere that overlaps a cube, a Monte-Carlo calculation was performed. The accuracy of this method can be measured by the standard deviation of the result. When a point is selected randomly inside the cubical sample, it can be either in bone or in marrow. The probability of each event follows a binomial distribution and the standard deviation of a sample of N points is given by

$$\mathbf{s} = \sqrt{\frac{p(1-p)}{N}}. \quad (3-4)$$

In [Equation \(3-4\)](#), p is the probability for the point to be in marrow. This value of \mathbf{s} will be maximal for $p = 0.5$. With $N = 1,000,000$ the standard deviation is then less than 0.05%.

Construction of the Segmented Images of the Model

Once the mathematically defined bone model is created representing a cubical specimen of trabecular bone, a grid structure is placed over the sample representing the voxel dimensions of an NMR image acquisition. Segmentation of the grid structure yields a binary image similar to that obtained by Jokisch et al. Segmentation of the mathematical sample is achieved by assigning each voxel to either bone or marrow according to the percentage of marrow (volume inside the spheres) and bone (volume outside the spheres) it contains. A total of 19 samples were created with cubical voxel sizes ranging from 16 μm to 1000 μm (Table 3-1).

A simplified segmentation algorithm was adopted in which, for each voxel of the image, the volume fraction of marrow it contains is determined. If this fraction is greater than 50%, the voxel is assigned to marrow and if not, it is assigned to bone. The complexity comes from the calculation of the volume of intersection between a cubical voxel and a marrow sphere. Many different situations exist and few have analytical solutions. Consequently, a combination of analytical and Monte-Carlo methods were used for image segmentation.

Three different C-programs were written to perform the segmentation process. The first creates a file that contains one byte per voxel within setting the tissue medium of each voxel. The second program calculates the location of all voxels that are fully inside each marrow sphere of the sample and assigns these voxels to marrow. The third program sweeps through all voxels of the image previously undefined and for each it checks how many spheres overlap the voxel. If no overlapping regions are found, the voxel is assigned to bone. If at least one sphere overlaps the voxel, the program uses a Monte-Carlo

technique to calculate the percentage of voxel volume intersected by one or more spheres. If this cumulative percentage exceeds 50%, the voxel is assigned to marrow; otherwise, it is assigned to bone. The decision to segment at a volume fraction of 50% is somewhat arbitrary, but sufficient for the present study as it was more important to be consistent across all voxelized images. Other percentages or even a mass fraction (considering the different tissue densities) could have also been employed.

Once the segmentation is performed, another program is used to compress the data file so that each byte of the file can store eight voxels. Finally, two programs calculate, for each segmented image, the marrow volume fraction within the image and the surface area of the bone-marrow interface. The two series of results are in [Table 3-1](#).

Electron-Transport Simulations

The next step in the study is to couple the segmented voxelized images with a Monte-Carlo transport code to study the evolution of absorbed-fraction results while the voxel size is changed. The goal is to calculate the absorbed fractions (f) for a source of monoenergetic electrons born within the marrow cavities and for the bone trabeculae and marrow cavities as target regions. For both sets of absorbed fractions, simulations of electron transport are made within both the mathematical bone sample and within each of its simulated, segmented voxel images. Particle transport calculations were performed using the EGS4-PRESTA code.

Transport simulations were performed only within the central $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ ROI of the bone sample. Every particle leaving this volume was discarded. A homogeneous source of monoenergetic electrons was used with particle emissions beginning within those marrow spheres completely enclosed in the ROI. For both the

bone and the marrow as target regions, the absorbed fraction deposited in that region was calculated. A total of six electron energies were selected ranging 0.05 MeV to 2 MeV.

Two different EGS4 user codes were developed: one for the mathematical sample (“reference code”) and a second for the 19 segmented images (“test codes”). For both programs, the same calculation was performed. The only difference was the way the geometry was defined. For the “reference code” the geometry is defined by equations of spheres, whereas the “test code” is defined by equations of planes delineating the voxel slices of the particular image.

The parameters used by the EGS4 transport code were also kept constant between both programs. The media used for both bone and marrow were those defined by the ICRU (1992) Report 46 for an adult. The bone regions were composed of pure skeleton-cortical bone and the marrow regions were made of pure skeleton-red marrow. Descriptions of both tissues are given in Table 3-2. The densities are 1.92 g/cm³ for bone and 1.03 g/cm³ for marrow. In both bone and marrow regions, the energy cut off for both electron and photon transport was set to 10 keV. For the PRESTA extension, the ESTEPE parameter was set at 0.05. Each calculation (6 times the “reference code” and 114 times the “test code”) was performed for 10 runs of 100,000 initial electrons. The results were averaged over the 10 runs and a standard deviation was calculated to estimate the standard error.

S-Value Calculations

The absorbed fractions calculated with EGS4 were then used to calculate the S values (dose per unit cumulated activity) as defined by the Medical Internal Radiation Dose (MIRD) Committee for several radionuclides (Loevinger et al. 1991). Five nuclides

were chosen for their interest in trabecular bone dosimetry (^{131}I , ^{32}P , ^{33}P , ^{153}Sm , and $^{117\text{m}}\text{Sn}$) and because they cover a large range of beta particle energies. [Table 3-3](#) shows the radiological characteristics of the five radionuclides selected. S values were calculated using the decay scheme tables of Eckerman et al. (1993). In this study, only marrow sources of electrons were considered, even though two of these radionuclides (^{153}Sm and $^{117\text{m}}\text{Sn}$) are associated with radiochemicals that exclusively localize in or on the surface of the bone trabeculae. As discussed later, however, conclusions drawn regarding voxelization errors in the cross-dose to bone will hold equally true for the cross-dose to marrow when the source-target designations are reversed.

Results and Discussion

Mathematical Sample

The mathematical sample of trabecular bone was created as a cube (2.2 cm on edge) that contains 28,200 marrow spheres. Only 11,605 spheres remained in the sample after exclusion of those spheres fully outside the central $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ ROI. The average radius for all spheres is $227 \mu\text{m}$. The volume fraction of marrow within the ROI is 0.6979 ± 0.0005 . The total volume of marrow strictly found within the ROI is thus 2.859 cm^3 , a value less than that predicted by [Equation \(3-2\)](#) as discussed earlier. The true surface area of the bone-marrow interface found strictly within the ROI may be estimated as $(150.7 \text{ cm}^2) \times (2.859 \text{ cm}^3 / 3.424 \text{ cm}^3)$ or 125.8 cm^2 .

[Figure 3-4](#) shows a $1.6 \times 1.6 \text{ cm}^2$ cross section of the final sample within the ROI. The image is perpendicular to the X-axis, and is located at $X=0$ (the center of the cube). Black regions represent the lattice of trabeculae while white circles represent the marrow cavities.

In [Figure 3-5](#), the chord-length distributions for both the marrow cavities ([Figure 3-5-A](#)) and the bone trabeculae ([Figure 3-5-B](#)) are compared with those for the cervical vertebra in the data of Spiers and colleagues ([Whitwell 1973](#)). Within the mathematical sample, the mean chord lengths are 871 μm and 377 μm for the marrow cavities and the bone regions, respectively. The former approaches that given by [Equations \(3-2\) and \(3-3\)](#) and Cauchy's theorem (909 μm). The corresponding values from Spiers data are 909 μm and 280 μm , respectively. As noted earlier, the difference between the values of 871 and 909 μm are attributed to those spherical marrow cavities that only partially overlap the ROI boundary. [Figure 3-5-B](#) shows a large difference in shape between the chord-length distributions for the bone trabeculae. This difference is due to the difficulty in fitting a sufficient number of marrow spheres within the bone cube and is reflected in the difference between the mean chord lengths. Nevertheless, our goal was not to have an exact representation of trabecular bone, but to create a mathematical model representative of trabecular bone. The shapes of both distributions are similar, however. One may also interpret our mathematical sample as a specimen of trabecular bone with trabeculae slightly thicker than those measured by Spiers for his cervical-vertebra specimen.

Segmented Images

Segmented images were created for 19 different voxel sizes listed in [Table 3-1](#). The fifth column of [Table 3-1](#) gives the volume fraction of marrow in each segmented image. As the voxel size increases, the volume fraction increases reaching 100% at very poor resolution. As there is more than 50 percent marrow in the mathematical sample (69.79%), the larger the voxel size, the more likely each is to contain more than 50% marrow. Therefore, as the voxel size increases, there is a higher probability for each voxel to be

assigned to marrow, and the percentage of marrow voxels within the entire sample thus increases.

Figure 3-6 shows the result of segmentation for four different image resolutions taken for the same slice through the mathematical bone sample. The resolutions are 667 μm (24 voxels per dimension) for Figure 3-6-A, 286 μm (56 voxels per dimension) for Figure 3-6-B, 80 μm (200 voxels per dimension) for Figure 3-6-C, and 25 μm (640 voxels per dimension) for Figure 3-6-D. For a voxel size of 286 μm , the image appears blocky and visually inaccurate, yet the volume fraction of marrow approaches the true value (70.84 % compared to 69.79 %). For a voxel size smaller than 300 μm , the error on the volume fraction is only about one percent. Because of poor resolution, some voxels overestimate their true volume fraction of marrow and some overestimate their volume fraction of bone, with both errors canceling one another yielding an acceptable marrow fraction over the entire image. Figure 3-7-A shows how the marrow volume fraction converges toward the reference value (the horizontal dotted straight line) as the resolution is improved.

The last column of Table 3-1 shows the total surface area of the bone-marrow interface within the segmented sample. The results are compared with the reference value (the horizontal dotted straight line) in Figure 3-7-B. This area increases almost linearly from almost zero at 1000 μm to 191 cm^2 at very high resolution. This is partly because of the volume fraction of marrow. For a large voxel size, only a few voxels are assigned to bone. As a consequence, the surface area between bone and marrow is reduced (see Figure 3-6-A). Between Figures 3-6-B and 3-6-D the volume fraction of marrow is almost the same, but the surface area in Figure 3-6-D is still larger than that in Figure

3-6-B (Table 3-1). In Figure 3-6-B one can see that, because of the lower image resolution, marrow spheres are connected to each other through marrow voxels, thus reducing the interface surface area. This problem does not appear in Figure 3-6-D (or it appears very infrequently). This explains why the surface area continues to increase whereas the volume fraction remains constant. At high resolution, the area converges just above 191 cm^2 , which is ~50 % larger than the surface area of marrow spheres within the true mathematical sample ($\sim 125.8 \text{ cm}^2$). Voxelization of the images thus introduces an overestimation of the interface surface area that is not reduced with further reductions in voxel size.

Absorbed Fractions and their Absolute Errors

EGS4 transport simulations were made for monoenergetic electron sources located within the trabecular marrow space of the 19 segmented images as well as the reference mathematical trabecular bone sample. For each simulation, absorbed fractions of energy were estimated for targets in the marrow cavities, the bone trabeculae, and regions outside the dosimetric ROI. The latter was used to check the energy deposition balance within the system. In no case did the standard deviation on the absorbed fraction exceed 0.0015. Reference absorbed fractions for the mathematical sample are listed in Table 3-4. They are shown to decrease with increasing electron energy, as more electron kinetic energy is lost to the skeletal regions outside the dosimetric ROI.

The convergence of the absorbed fraction to marrow is shown in Figure 3-8-A as a function of voxel size and electron energy, while Figure 3-8-B shows the corresponding values for bone as the target tissue. The abscissa in Figures 3-8-A and 3-8-B is the voxel size ranging from high resolution ($16 \text{ }\mu\text{m}$) to low-resolution ($500 \text{ }\mu\text{m}$) images. The

ordinate is the difference between the reference absorbed fraction and the value found within each of the 19 segmented images:

$$\Delta_f = \mathbf{f}_{seg} - \mathbf{f}_{ref} . \quad (3-5)$$

A positive value corresponds to an overestimate of the absorbed fraction within the segmented image and a negative value corresponds to an underestimate.

Six general points can be made in viewing the data from [Figure 3-8](#). They are

- 1) Both diagrams are fairly symmetric. For all electron energies and voxel sizes, an overestimate of the marrow absorbed fraction leads to an underestimate of the bone absorbed fraction and vice versa.
- 2) The absorbed fraction for large voxel sizes is overestimated in marrow and underestimated in bone.
- 3) For high-energy electrons (>400 keV), the error decreases as the voxel size is reduced. [Figures 3-8-A](#) and [3-8-B](#) show that the absorbed fractions calculated within the segmented images converge to the reference value (the difference converges to zero) as the resolution is improved.
- 4) For electrons >400 keV, convergence of the results to the reference value is improved with increasing initial electron energy.
- 5) At low electron energies (<400 keV), the data of [Figure 3-8](#) show that for large voxel sizes (> 350 μm), smaller absorbed-fraction differences are seen at lower electron energies.
- 6) At low energy electrons and at high image resolution, the results do not converge to the reference value.

To further understand these trends, it is helpful to revisit the errors introduced via voxelization in both the marrow volume fraction (“volume-error effect”) and the surface area of the bone-marrow interface (“surface-error effect”) ([Figure 3-7](#)).

The influence of errors in the marrow volume fraction at large voxel sizes is easily understood. At low image resolution, the overestimation of the total marrow volume results in an increase in the absorbed fraction to marrow and a corresponding decrease in

the absorbed fraction to the bone trabeculae. The surface-error effect is more complex. As shown in [Figure 3-7-B](#), the surface area of the bone-marrow interface increases as the voxel size decreases. This error was shown to overestimate the true surface area at small voxel sizes and to underestimate the area at large voxel sizes. As the surface area is artificially increased in high-resolution images, particle escape at the interface surface is increased and the absorbed fraction to bone trabeculae is overestimated. Within low-resolution images, the surface area is underestimated and thus the absorbed fraction to bone is underestimated. For low energy electrons born within the marrow cavities, energy loss to bone is contributed by only those electrons emitted from a marrow layer immediately adjacent to the bone-marrow boundary. The thickness of this effective source layer is approximately equal to the electron range in marrow. If the electron range is much smaller than the voxel size, the effective source volume is roughly equal to the total surface area times the electron range in marrow. The right angles between the squared surfaces of the voxels do not introduce a significant contribution to this volume, and thus the variation of the effective source volume is roughly proportional to the corresponding variation in interface surface area. Consequently, an increase in surface area increases the effective source volume and the absorbed fraction to bone by the same proportion. On the other hand, if the electron range exceeds the voxel size, the effective source volume is more complex and becomes smaller than that predicted by the product of the surface area and the electron range. At higher electron energies, an increase in surface area does not necessarily increase the effective source volume by the same proportion, and thus overestimates in the absorbed fraction to bone are not as significant as they are at lower energies. A general conclusion is thus that overestimates in the absorbed fraction to bone

are only significant if the voxel size is large compared to the electron range. Note that identical but converse arguments would hold true if one were to consider instead a bone source of low-energy electrons irradiating the marrow in a high-resolution image. Let us now return to the six observations noted in [Figures 3-8-A](#) and [3-8-B](#).

Point 1): The total size of the bone sample and its interior ROI is the same whether it is taken as the pure mathematical sample or one of its 3D segmented images. The loss of energy due to particle escape, therefore, is constant and only depends on the initial electron energy.

Point 2) is a direct consequence of the errors in the marrow volume fraction. At large voxel sizes, the volume of bone is small compared to that within the mathematical sample and thus the bone absorbed fraction is underestimated. This error increases as the voxel size increases.

Point 3) is also a consequence of the errors in the marrow volume fraction. As the voxel size is reduced, the marrow volume approaches the reference volume and the absorbed fractions converge toward the reference values. For electrons exceeding 400 keV, their range exceeds the voxel dimensions (1300 μm for 400 keV electrons in water). Consequently, the surface-error effect only appears at large voxel sizes where it is completely masked by the volume-error effect that acts in the opposite direction. Absorbed fractions converge to reference values around 200 μm where the surface-error effect is too weak to have a noticeable consequence.

Point 4). For high-energy electrons, the deposition of energy is expected to be more uniform throughout the whole sample than for low-energy electrons. Consequently, the absorbed fractions to both marrow and bone would be those given by the ratio of masses

for the two media. One can therefore expect that the dependence of the results on the voxel size has its origin in the variation of the marrow volume fraction within the sample. Furthermore, convergence is also improved at high electron energies where the surface-error effect is smaller.

Point 5). For low-energy, short-ranged electrons below 400 keV, the absorbed fraction to marrow approaches unity. Consequently, the dependence of the absorbed fraction on the volume-error effect is weak for this energy range. A consequence of points 4) and 5) is that for large voxel sizes ($> 350 \mu\text{m}$) the absorbed-fraction error increases with electron energy below 400 keV, and decreases with electron energy above 400 keV. The calculation performed for 400 keV electrons seems to give the largest absorbed-fraction error in images of poor resolution.

Point 6). For small voxel sizes (below $400 \mu\text{m}$) the volume-error effect is no longer strong and only the surface-error effect is dominant. For low electron energies, the electron range becomes smaller than the voxel size (e.g., $43 \mu\text{m}$ in water for 50 keV electrons). The surface-error effect thus leads to an overestimate of the bone absorbed fraction. When the voxel size is very small and becomes smaller than the electron range, the absorbed fraction converges again toward the reference value. Absorbed-fraction curves in [Figure 3-8-B](#) for both 50 and 100 keV electrons show an overestimate of the bone absorbed fraction that increases to a voxel size approximately equal to their range ($43 \mu\text{m}$ and $143 \mu\text{m}$, respectively). At higher voxel resolutions, the absorbed-fraction errors decrease again as they approach their reference values. The surface-error effect no longer influences the calculation if the voxel size becomes significantly smaller than the electron range.

Absorbed Doses and their Relative Errors

The discussion above reflects the absolute error made on the absorbed fraction with changes in voxel resolution. An alternative evaluation of the voxel-size effects on dose calculations to skeletal tissues using NMR microscopy is to look at the relative error of the absorbed fractions. [Figure 3-9](#) shows the relative error (in percentage) for the absorbed fraction to both marrow ([Figure 3-9-A](#)) and bone ([Figure 3-9-B](#)) expressed as:

$$\Delta r_f = \frac{\mathbf{f}_{seg} - \mathbf{f}_{ref}}{\mathbf{f}_{ref}} \cdot 100\% . \quad (3-6)$$

For the marrow results, [Figures 3-8-A](#) and [3-9-A](#) are almost identical since the absorbed fraction is on the order of unity for an electron source located within the marrow cavities. For the bone trabecula results, [Figure 3-9-B](#) shows that the relative error is very important for low energy electrons. The absorbed fractions to the bone trabeculae are small because of the short ranges of the particles in the marrow source region. Here the relative error can reach 65 % for 50 keV electrons. This error is directly attributable to the ~50% overestimate in the surface area of the bone-marrow interface ([Figure 3-7-B](#)). By dividing each absorbed fraction by the mass of the corresponding target region (segmented image or reference sample), the corresponding relative error on the absorbed dose to either marrow ([Figure 3-10-A](#)) or the bone ([Figure 3-10-B](#)) from monoenergetic electrons sources in marrow may be calculated:

$$\Delta r_f = \left(\frac{\frac{\mathbf{f}_{seg}}{m_{seg}} - \frac{\mathbf{f}_{ref}}{m_{ref}}}{\frac{\mathbf{f}_{ref}}{m_{ref}}} \right) 100\% . \quad (3-7)$$

Figures 3-9 and 3-10 are similar for small voxel sizes, but differ at large voxel sizes. Since the mass of the target is proportional to its volume, the absorbed-fraction error at large voxel sizes is attenuated by the mass error when the corresponding dose is calculated. Consequently, the error on the dose is smaller than that for the absorbed fraction. For small voxel sizes where the volume fraction of marrow is almost constant and equal to its true value, the relative error in the absorbed dose is approximately similar to the relative error in the absorbed fraction.

The results of Figures 3-8 to 3-10 are shown for monoenergetic electron sources in marrow. To investigate potential errors in marrow and bone dose due to radionuclides localized in marrow, radionuclide S values were then determined in which absorbed fractions for monoenergetic electrons were weighting across their beta-particle energy spectra. These S values are given in Table 3-5 for ^{32}P , ^{33}P , ^{131}I , ^{153}Sm , and $^{117\text{m}}\text{Sn}$. Figure 3-11 shows the relative error on the S value as a function of voxel size. For ^{32}P , the mean electron energy is high and Figure 3-11 shows good convergence of the S value across all voxel sizes. There is less than 1% error for both the bone and the marrow dose below 300 μm . Iodine-131 and ^{153}Sm are intermediate-energy electron emitters. For these radionuclides, the surface-area effect begins to become important and the relative error for $S_{(\text{bone} \leftarrow \text{marrow})}$ cannot be reduced to less than 5% even at high voxel resolution. For the very low-energy emitters $^{117\text{m}}\text{Sn}$ and ^{33}P , the relative error on the S value remains as high as 25% for the cross-dose to bone, but is less than 4% for the self-dose to marrow. At voxel resolutions of $\sim 450 \mu\text{m}$, the result is exact but no conclusions can be drawn. This value of 450 μm is a consequence of the reduction of the surface area of the bone-marrow interface as the voxel size is increased. This surface-area reduction depends on the marrow volume

fraction within the mathematical sample and it would be incorrect to suggest that 450 μm is an optimal value for either $^{117\text{m}}\text{Sn}$ or ^{33}P . It is only an optimal value for the specific geometry used in this study. For a different trabecular bone sample with a different marrow volume fraction, the optimal value would most likely be different.

Conclusion

The voxel-size impact on dose calculations within a voxelized and segmented 3D NMR image of trabecular bone may be summarized according to three different aspects. First, the geometry of the segmented image is obviously different from that of the true bone sample. [Figure 3-2](#) shows that cubical voxels may lead to overestimates and underestimates of energy deposition within the media if a particle is traveling close to a boundary. These errors tend to cancel one another on average and have probably no consequence on the dose calculation. Second, the volume fraction of marrow is overestimated at large voxel sizes, but as shown in [Figure 3-7-A](#) this overestimate is small below 300 μm and there is no consequence to the dose calculation for image resolutions below 300 μm . Third, the surface area of the bone-marrow interface is overestimated at small voxel sizes (below 300 μm) and underestimated at large voxel sizes as shown in [Figure 3-7-B](#). For high-energy electron emitters, this effect is without consequence in that the electron range exceeds the voxel dimensions. The surface-error effect occurs over a range of voxel sizes for which reductions in bone volume fraction with improved resolution attenuate the effect of increases in estimated surface area. The surface-error effect is thus without consequence for high-energy electrons. For low-energy electrons emitted within the marrow cavities, the surface-error effect has little consequence on the self-dose to the marrow cavities (maximum error <5% at 200 keV in [Figure 3-9-A](#)).

Nevertheless, for low-energy electrons, the surface-error effect does lead to an overestimate of the cross-dose to bone trabeculae for small voxel sizes and an underestimate of the bone dose at large voxel sizes. The overestimate in bone cross-dose approaches 25% for radionuclides with a mean beta energy of ~100 keV and is maximal for 100 μm voxels. This error approaches 5% for radionuclides with a mean beta energy of ~200 keV and is maximal also around 100 μm . No conclusion can be drawn for the optimal voxel size for low-energy beta emitters in that the voxel-size range for overestimates and underestimates in cross-region dose depends on the variation of the surface-error effect with voxel size which further depends upon the specific geometry of the sample. At voxel resolutions below 100 μm , the error in cross-dose is shown to turn over and again converge to the reference value as the voxel size becomes smaller than the electron range. This can be seen in both [Figure 3-10](#) and [Figure 3-11](#), but the convergence occurs at voxel sizes on the order of a few micrometers, well beyond the current capabilities of NMR on samples of this nature within reasonable acquisition time.

The largest error encountered in the dosimetry of voxelized images was shown to be the influence of the surface-error effect on the cross-region dose for low-energy electrons. Fortunately, in these instances, the self-dose will dominate because of the short range of the particles. In this study, the marrow cavities were selected as the source region of electron emissions. Because of the symmetry of the voxelized image, a similar overestimate in marrow dose would be expected for low-energy electron sources in the bone trabeculae. Again, the self-dose to bone would dominate and the uncertainty in the average cross-dose to the marrow cavity may be unimportant. Nevertheless, the surface-error effect warrants further investigation considering that studies by Lord (1990)

indicate that the majority of marrow stem cells are located near the endosteal surface where the dose would be higher than the average marrow cavity dose.

In summary, the present study found that an NMR image of trabecular bone composed of voxels $<300\text{-}400\ \mu\text{m}$ provides accurate dosimetry for high-energy electron emitters and for both the self-dose to marrow and the cross-dose to the bone trabeculae. For radionuclides with a mean beta energy of 300 keV or higher, the error is reduced to a few percent at this resolution. This finding is significant in that this voxel-size range approaches that obtainable for trabecular bone imaging via clinical MRI units at 1.5 T under conditions of minimal patient motion. While there is no impact of the surface-error effect on the self-dose to marrow for low-energy beta emitters, overestimates in the interface surface area must be taken into account when considering the cross-dose to bone at low electron energies. Further investigations are warranted including the use of additional mathematically defined bone samples of differing trabecular microstructure (e.g., cranium, ribs, etc.). In these studies, systematic dosimetry errors introduced by surface area overestimates with low-energy electrons and within high-resolution images can be evaluated. Correction factors might then be developed and applied to dosimetry results.

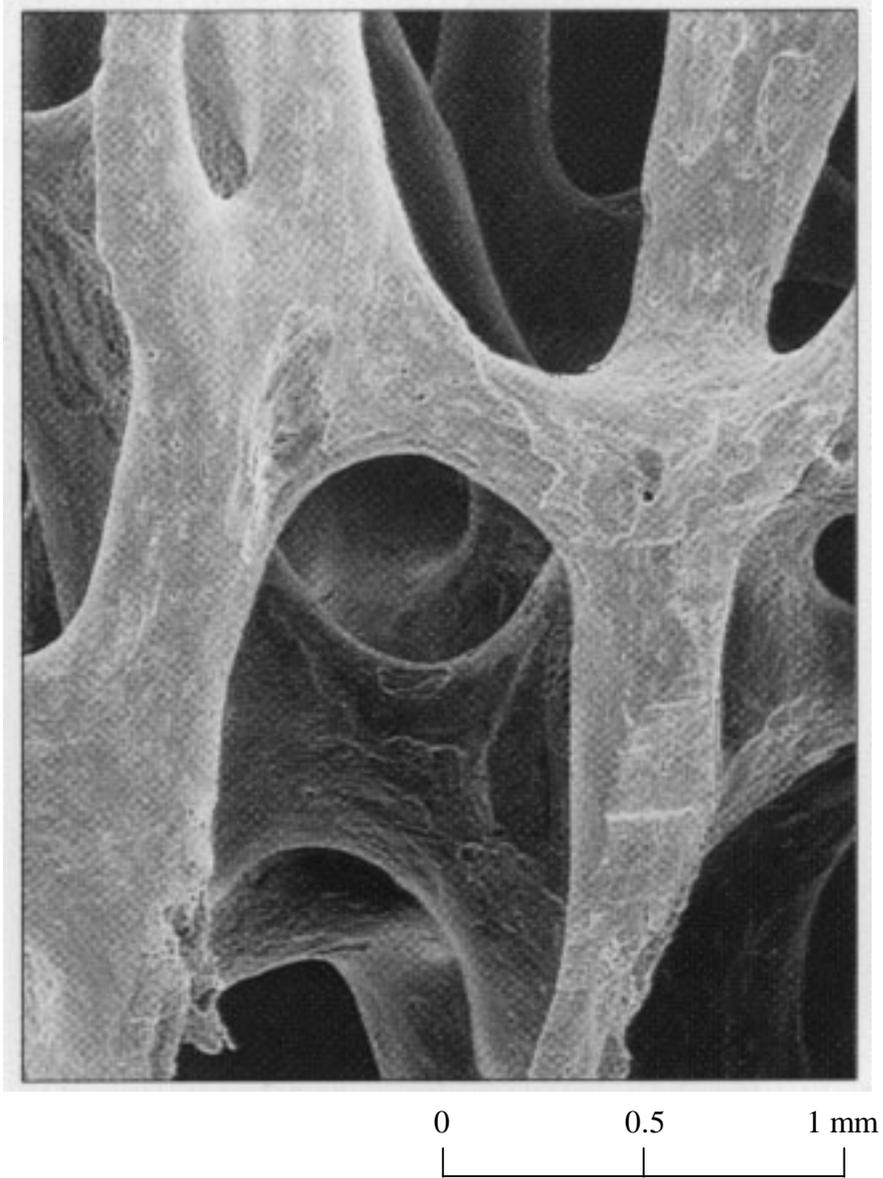


Figure 3-1. Electron micrograph of the trabecular latticework within a lumbar vertebra. Adapted from a study by Clayman (1995).

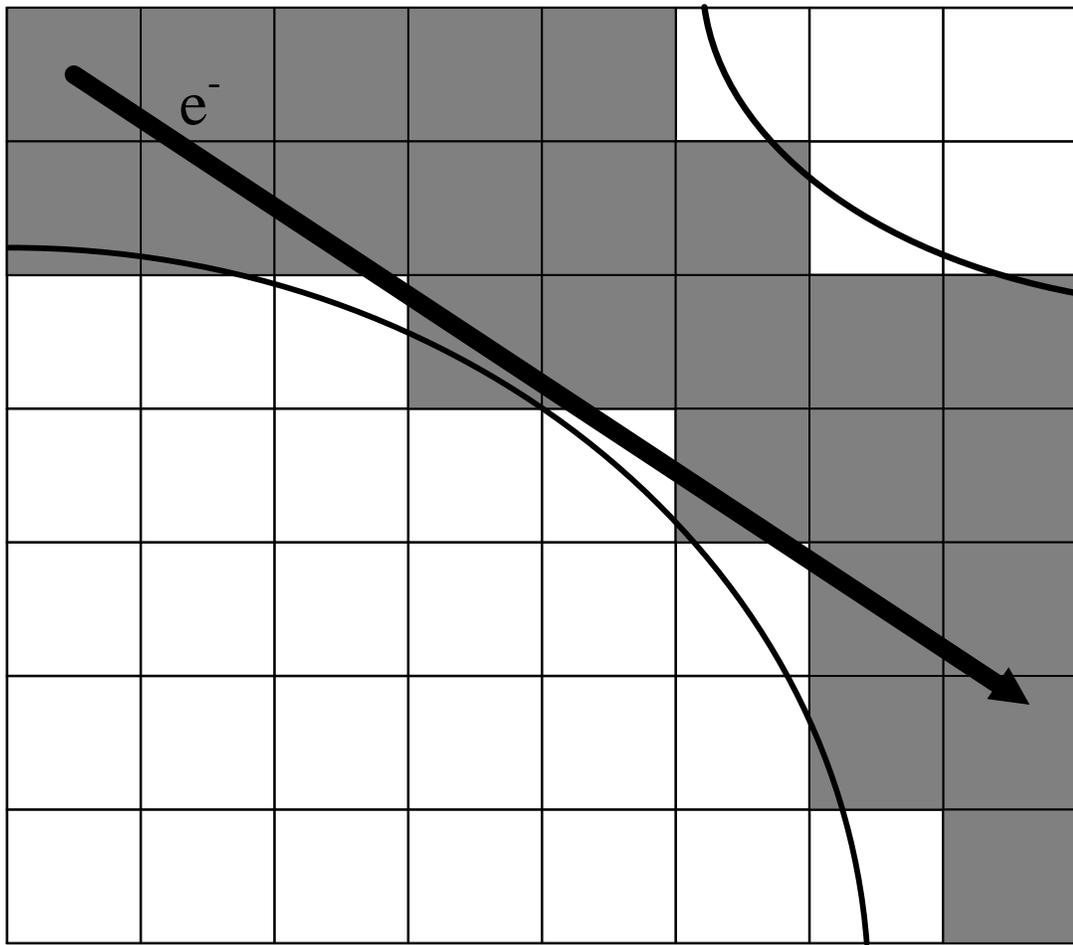


Figure 3-2. Voxel-size effects on the dose calculation for a single electron track traveling within a bone trabecula. The curved lines indicate the true surfaces of the bone-marrow interface. The gridlines indicate pixels within a corresponding digital image of the sample.

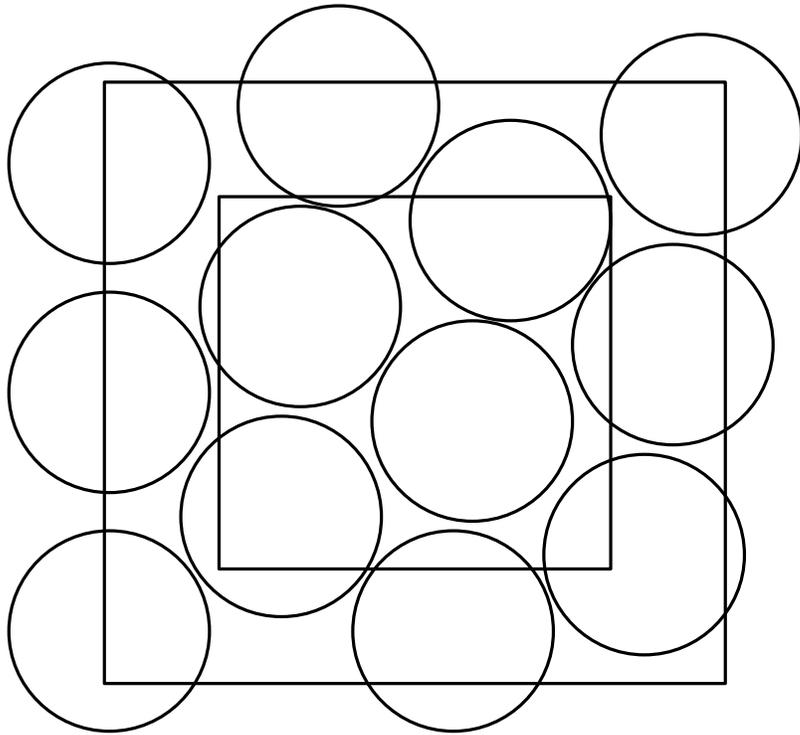


Figure 3-3. Edge effect when trying to position circles within a squared field.

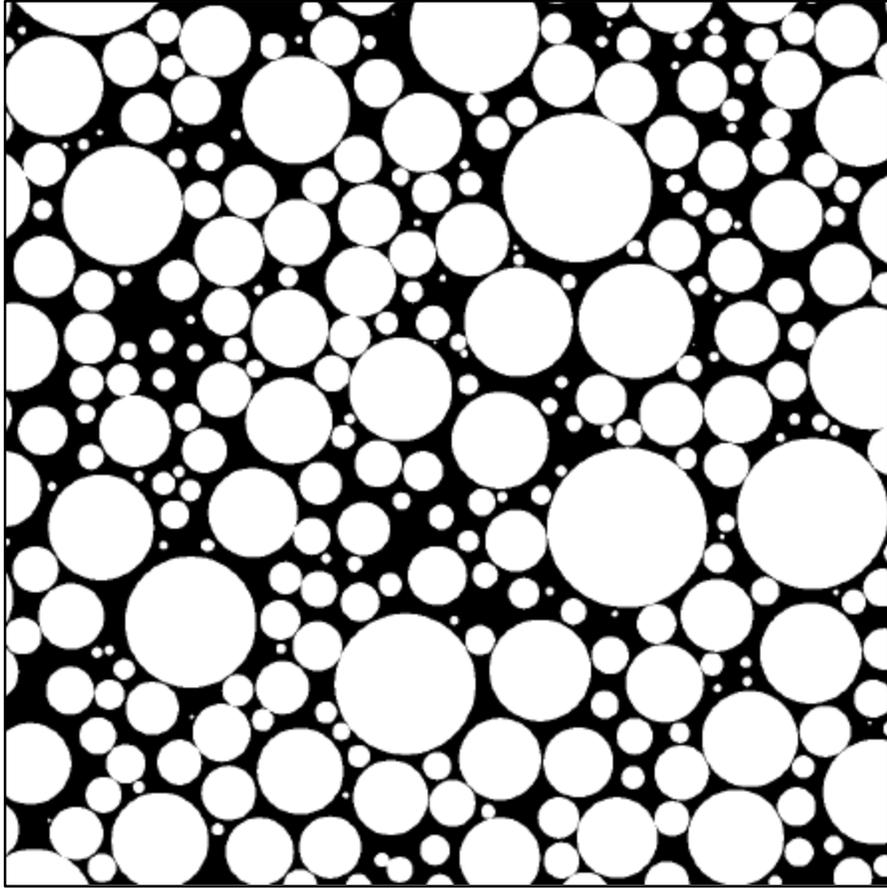


Figure 3-4. Transverse slice, 1.6 cm x 1.6 cm, through the ROI of the mathematical sample of trabecular bone.

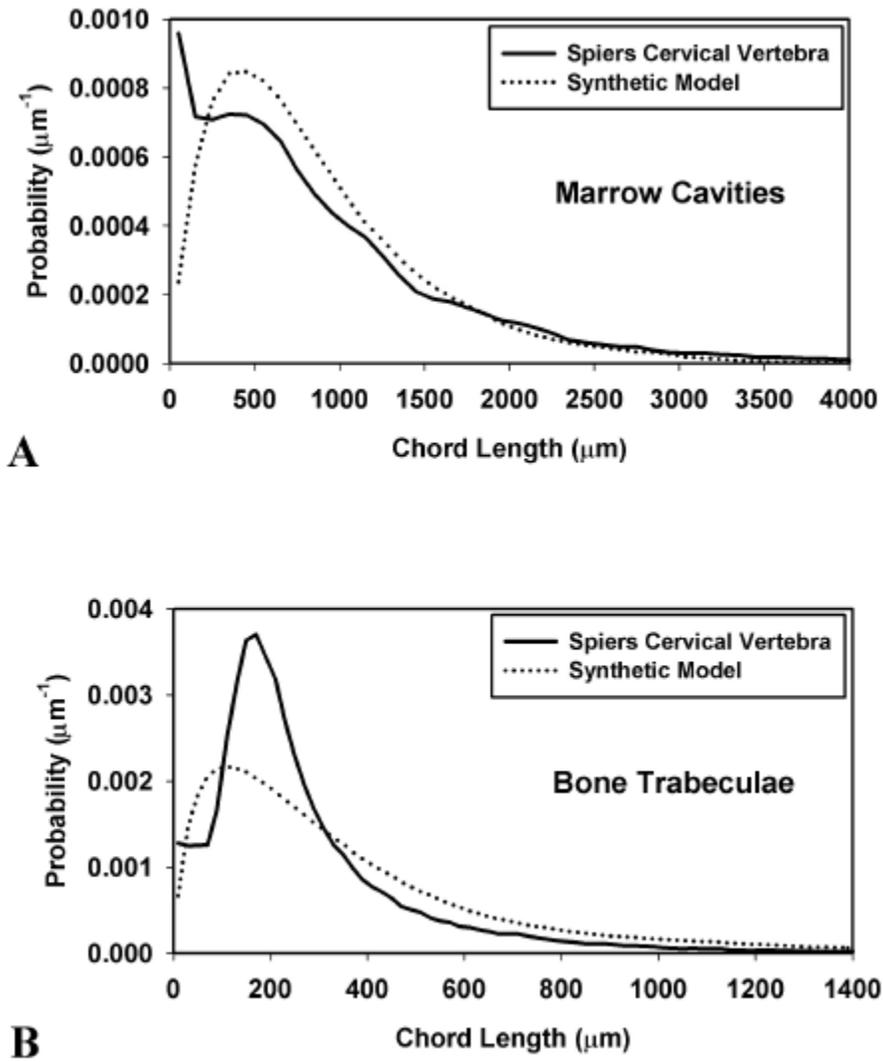


Figure 3-5. Chord-length distributions: Spiers cervical vertebra compared to the mathematical model. A) Marrow cavities. B) Bone trabeculae.

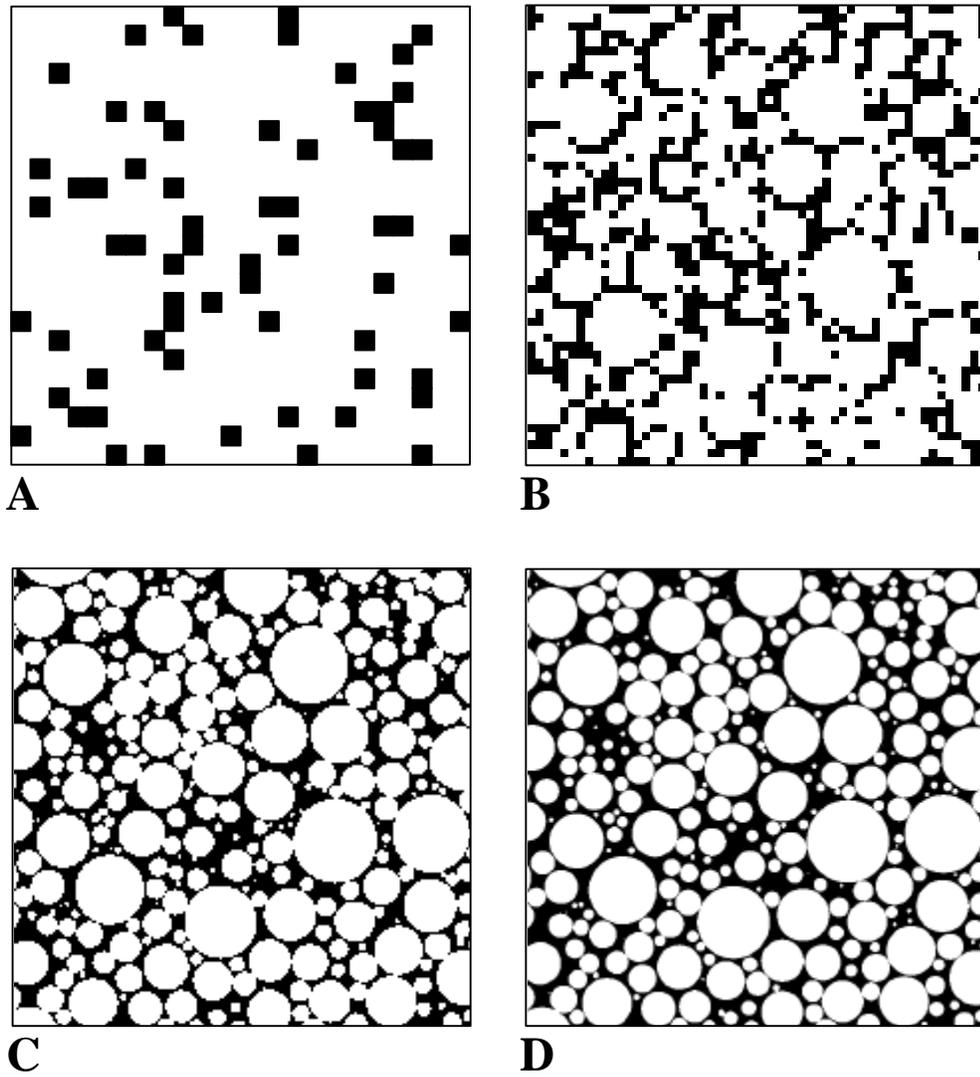


Figure 3-6. Segmentation of the mathematical sample. The four images have a different resolution, but represent the same slice through the mathematical sample. A) 667 μm . B) 286 μm . C) 80 μm . D) 25 μm .

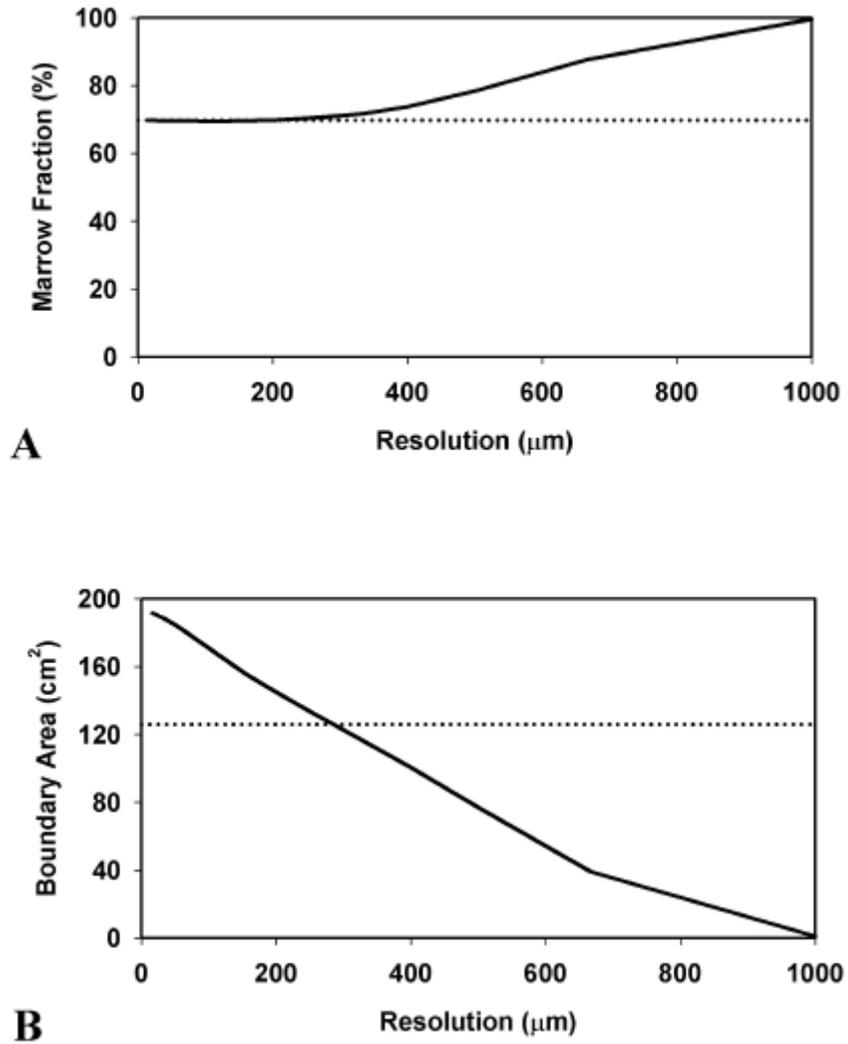


Figure 3-7. Geometrical parameters as a function of image resolution for the mathematical sample. A) Marrow volume fraction (reference value is 69.79%). B) Surface area of the bone-marrow interface (reference value is 125.8 cm^2).

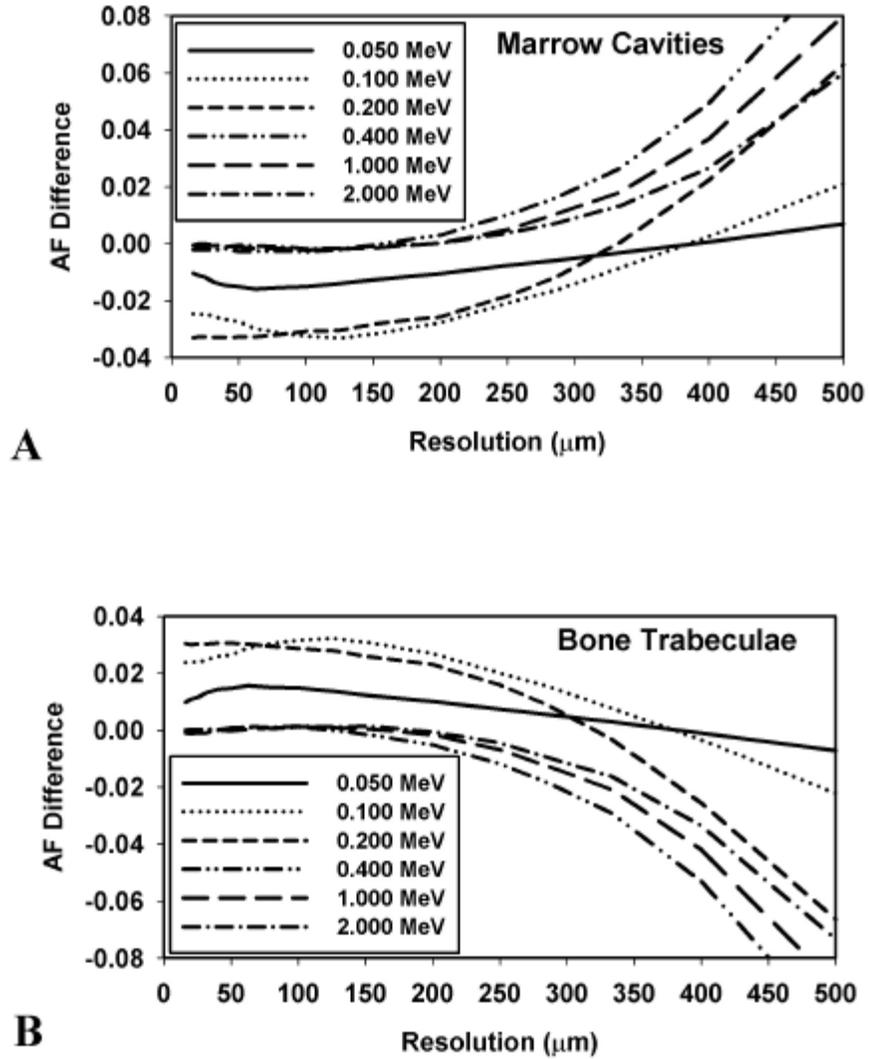


Figure 3-8. Absorbed-fraction differences between the segmented images and the reference values. The reference is obtained within the mathematical sample. The source of radiation is monoenergetic electrons emitted within the marrow cavities. A) Target regions considered are the marrow cavities. B) Target regions considered are the bone trabeculae.

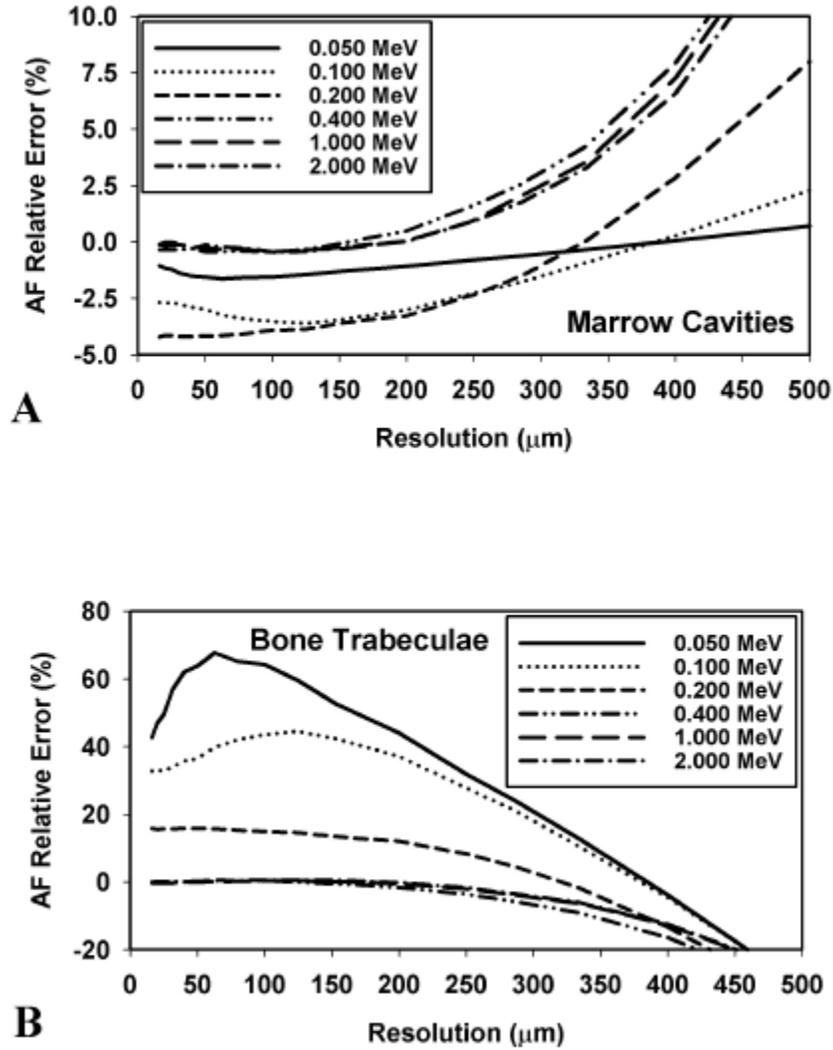


Figure 3-9. Relative error in the absorbed fraction for the segmented images and using results from the mathematical bone sample as reference values. The source of radiation is monoenergetic electrons emitted within the marrow cavities. A) Target regions considered are the marrow cavities. B) Target regions considered are the bone trabeculae.

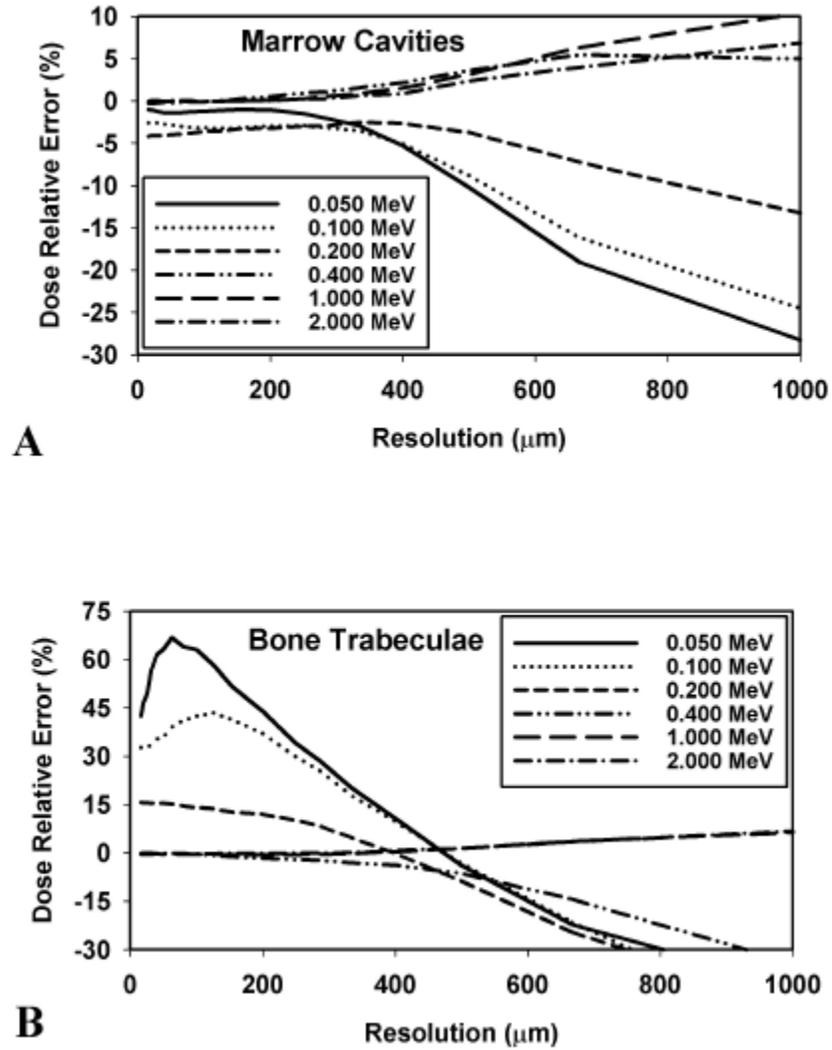


Figure 3-10. Relative error in the absorbed dose for monoenergetic electron sources in the marrow cavities. A) Target regions considered are the marrow cavities. B) Target regions considered are the bone trabeculae.

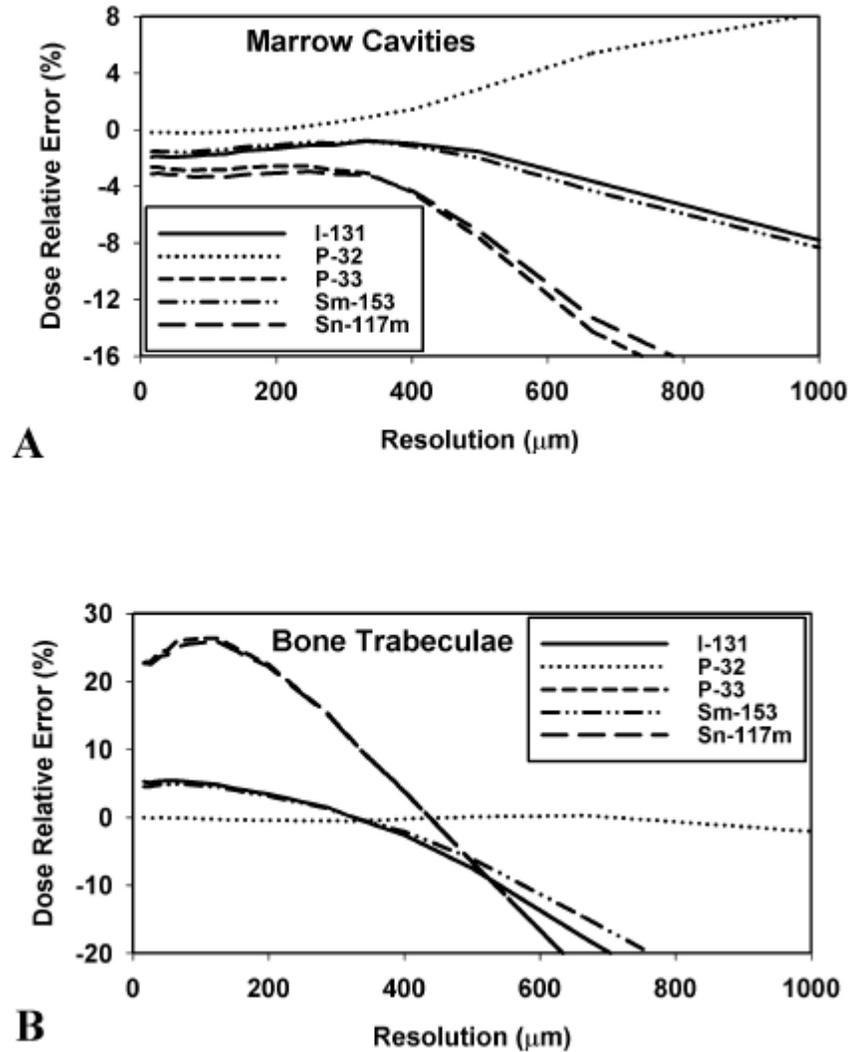


Figure 3-11. Relative error in the S value calculated for five radionuclides of interest in skeletal dosimetry. A) Target regions considered are the marrow cavities. B) Target regions considered are the bone trabeculae.

Table 3-1. Characteristics of the 19 segmented images for a voxel-size range from 1000 μm to 16 μm .

Voxel size (μm)	Voxels per dimension	Number of voxels (million)	File size (million of bytes)	Fraction of marrow voxels (%)	Surface Area of Interface (cm^2)
1000.00	16	0.004	0.000	99.56	1.04
666.67	24	0.014	0.002	87.60	38.99
500.00	32	0.033	0.004	78.31	77.01
400.00	40	0.064	0.008	73.75	100.36
333.33	48	0.111	0.014	71.64	115.24
285.70	56	0.176	0.022	70.84	125.82
250.00	64	0.262	0.033	70.28	133.37
200.00	80	0.512	0.064	69.78	144.84
153.85	104	1.124	0.141	69.60	156.10
125.00	128	2.097	0.262	69.56	163.90
100.00	160	4.096	0.512	69.56	170.84
80.00	200	8.000	1.000	69.60	176.51
62.50	256	16.777	2.097	69.63	181.30
50.00	320	32.768	4.096	69.67	184.56
40.00	400	64.000	8.000	69.70	186.91
31.25	512	134.217	16.777	69.72	188.83
25.00	640	262.144	32.768	69.74	190.00
20.00	800	512.000	64.000	69.75	190.84
16.00	1000	1,000.000	125.000	69.76	191.42

Table 3-2. Tissue compositions used for bone and marrow in the mathematical bone sample.

Element	Symbol	Atomic number	Mass fraction in bone (%)	Mass fraction in marrow (%)
Hydrogen	H	1	3.4	10.5
Carbon	C	6	15.5	41.4
Nitrogen	N	7	4.2	3.4
Oxygen	O	8	43.5	43.9
Sodium	Na	11	0.1	----
Magnesium	Mg	12	0.2	----
Phosphorus	P	15	10.3	0.1
Sulfur	S	16	0.3	0.2
Chlorine	Cl	17	----	0.2
Potassium	K	19	----	0.2
Calcium	Ca	20	22.5	----
Iron	Fe	26	----	0.1

Table 3-3. Radiation characteristics of the radionuclides used for S-value calculations.

	Mode of decay	Avg. Energy (MeV)	Max. Energy (MeV)	Half-life (days)
¹³¹ I	β^-	0.191	0.606	8.04
³² P	β^-	0.695	1.710	14.3
³³ P	β^-	0.077	0.249	25.3
¹⁵³ Sm	β^-	0.225	0.809	1.95
^{117m} S	I.T.	0.135	0.159	13.6

Table 3-4. Absorbed fractions calculated within the mathematical bone sample for the marrow region, the bone region, and the region beyond the dosimetric region of interest.

Electron energy (MeV)	$\phi_{(\text{marrow} \leftarrow \text{marrow})}$	$\phi_{(\text{bone} \leftarrow \text{marrow})}$	$\phi_{(\text{beyond ROI} \leftarrow \text{marrow})}$
0.05	0.9745	0.0232	0.0023
0.10	0.9200	0.0726	0.0074
0.20	0.7859	0.1926	0.0215
0.40	0.6222	0.3232	0.0546
1.00	0.5079	0.3292	0.1629
2.00	0.4034	0.2702	0.3264

Table 3-5. S values calculated within the mathematical bone sample for the five chosen radionuclides. The target masses used for the calculation correspond to the volume of tissue enclosed in the limit of the mathematical bone model.

Radionuclide	$S_{(\text{marrow} \leftarrow \text{marrow})}$ (mGy/MBq-s)	$S_{(\text{bone} \leftarrow \text{marrow})}$ (mGy/MBq-s)
^{131}I	7.83×10^{-3}	3.20×10^{-3}
^{32}P	2.08×10^{-2}	1.56×10^{-2}
^{33}P	3.72×10^{-3}	5.56×10^{-4}
^{153}Sm	1.07×10^{-2}	4.35×10^{-3}
$^{117\text{m}}\text{Sn}$	7.39×10^{-3}	1.22×10^{-3}

CHAPTER 4
SURFACE-AREA OVERESTIMATION WITHIN 3D DIGITAL IMAGES AND ITS
CONSEQUENCES FOR SKELETAL DOSIMETRY¹

Introduction

Trabecular bone is that portion of the adult human skeleton that houses the hematopoietic marrow, the tissue responsible for the production of various blood cells. The high mitotic activity of active bone marrow thus makes this organ one of high radiation sensitivity, and is thus of high interest in the dosimetry of radionuclide therapies where marrow toxicity is generally dose-limiting (Sgouros 1993; Sgouros et al. 2000). Internal irradiation of bone marrow may result from occupational exposures to bone-seeking radionuclides (Brodsky 1996), radionuclide therapy of tumors (Siegel et al. 1990), and bone pain palliation treatments (Samaratunga et al. 1995). In these different situations, the absorbed dose to the bone marrow results from both the physical aspects of the radiation transport and energy deposition, as well as the biological aspects that determine source location and the cumulative number of nuclear decays. In regard to the physical aspects of skeletal dosimetry, the microstructural complexity of trabecular bone (Clayman 1995; Marieb 1998) imposes challenges to the construction of dosimetry models of this skeletal region, particularly as needed to assess absorbed fractions of energy for electron sources.

¹ This chapter was published by Medical Physics in May 2002: Rajon DA, Patton PW, Shah AP, Watchman CJ, and Bolch WE. 2002. Surface area overestimation within three-dimensional digital images and its consequences for skeletal dosimetry. Med Phys 29: 682-693.

Bone-Marrow Dosimetry

Bone-marrow dosimetry has been investigated extensively during the past forty years. During the 1960s to early 1970s, Spiers and colleagues conducted the first comprehensive studies in skeletal dosimetry at the University of Leeds (Beddoe 1976b; Beddoe et al. 1976; Spiers 1966b; Spiers et al. 1978a; Spiers et al. 1978b; Whitwell 1973; Whitwell and Spiers 1976). After establishment of a reference-man skeletal model, other investigators have expanded these techniques to permit dose estimation specific to medical internal dosimetry (Bouchet and Bolch 1999; Bouchet et al. 2000; Eckerman 1985; Eckerman and Stabin 2000; Snyder et al. 1974; Snyder et al. 1975; Stabin 1996). Nevertheless, the complexity of the bone microstructure and its variations with patient age (Mosekilde 1986; Ouyang et al. 1997; Snyder et al. 1993) have made it difficult to define models for bone-marrow dosimetry that provide improved patient specificity over the single reference model for the adult male. During the 1990s, several groups have investigated the use of three-dimensional imaging techniques such as micro-computed tomography (microCT) and nuclear magnetic resonance (NMR) imaging to acquire 3D digital images of trabecular bone samples. These techniques have been used to measure regional bone mineral density and structural parameters (Chung et al. 1996; Chung et al. 1995b; Hwang et al. 1997; Link et al. 1998b; Wessels et al. 1997). Recently, NMR microscopy has been applied to the study of skeletal dosimetry (Jokisch et al. 2001a; Jokisch et al. 1998; Jokisch et al. 2001b). This technique uses high-field NMR spectrometers to acquire high-resolution (~60-90 μm) 3D images of trabecular bone. The images are then directly coupled to Monte-Carlo radiation transport codes to calculate the deposition of energy by monoenergetic electrons within bone marrow. The code used is

EGS4-PRESTA as it allows voxelized geometry containing millions of voxels. [Figure 4-1](#) shows a reconstruction of a 3D NMR image of a trabecular bone sample. On this image, the average thickness of the bone trabeculae is $\sim 300 \mu\text{m}$ and the average size of the marrow cavities is $\sim 1000 \mu\text{m}$.

Voxelization Effects in Bone-Marrow Dosimetry

A 3D image is a digital representation of a real object. In the case of trabecular bone, two media are present within the image: the bone trabeculae and the marrow cavities. In the digital image, each voxel is assigned to bone or to marrow by image thresholding and segmentation ([Jokisch et al. 1998](#)). Consequently, the interface between the two media, which is most likely a curved surface in the real bone sample, appears as a jagged surface in the 3D digital image because of the rectangular shape of the voxels. As a consequence, a 3D digital image is not a totally faithful representation of the real sample. This problem has already been studied for chord-length distributions within NMR images of trabecular bone ([Jokisch et al. 2001b](#)) and has been defined by what the author calls pixel (or voxel) effects A and B. Both pixel effects A and B are a consequence of the rectangular shape of the voxels (or pixels in 2D geometry). Pixel effect A creates some sharp spikes over the entire chord-length distribution, whereas pixel effect B overestimates the amount of short chord lengths within the geometry. Pixel effect B thus contributes to an underestimation of the mean of the chord-length distribution.

For calculations of energy deposition, the 3D digital image is then coupled to a Monte-Carlo radiation transport code. Consequently, voxel effects are also expected to result in dosimetry errors when radiation particles are transported close to the bone-marrow interface. To evaluate the magnitude of these voxelization effects, a

theoretical study was conducted previously using a mathematical model of trabecular bone as explained in [Chapter 3](#). The model was constructed of non-uniformly sized spheres representing marrow cavities, with the intervening spaces representing the bone trabeculae. The experiment involved

- Coupling the mathematical bone model with the radiation transport code EGS4
- Simulating 3D imaging of the bone model through its voxelization at various degrees of image resolution
- Coupling these voxelized images within EGS4
- Comparing the dosimetry results with and without image voxelization.

The study permitted the identification of three voxel effects.

First, a geometry effect occurs when a particle travels parallel to the bone-marrow interface and deposits its energy alternatively within bone and marrow as a direct result of the jagged representation of the interface. This effect overestimates or underestimates the absorbed fraction to marrow tissues depending upon which side of the true interface the particle is traveling (marrow or bone). These errors were shown to cancel when the particle transport is averaged across the entire image.

Second, a volume effect is seen to overestimate the volume fraction of marrow at large voxel sizes. Below 300 μm , however, the error in the marrow volume fraction becomes insignificant and without consequence for the dose calculation.

The third effect is a surface-area effect. Because of the voxelization of the image, the surface area of the bone-marrow interface is not respected. Measurements within the mathematical bone model show that the total surface area throughout the entire image increases continuously from near zero at poor resolutions to a convergence value at high resolution. Unfortunately, the convergence value is 50% higher than the true surface area

within the mathematical model of trabecular bone. This overestimation has important consequences when the electron range is small compared to the voxel size. In this situation, the cross-absorbed fraction (e.g., a marrow source irradiating bone, or a bone source irradiating marrow) can also be overestimated by up to 50%. According to the study (see [Chapter 3](#)), for some low-energy beta emitters used for skeletal dosimetry, this effect can lead to a 25% overestimation of the cross-region absorbed dose.

The purpose of the present chapter is to give a scientific explanation for the bone-marrow interface overestimation, and its consequence on absorbed-fraction calculations. The mathematical bone model developed for the previous study and detailed in [Chapter 3](#) was made of thousands of spheres uniformly distributed throughout a cube representing a sectioned piece of trabecular bone. To understand how it is possible that the surface area of the bone-marrow interface, when measured through a voxelized image, converges to a value different from its true value, a separate study is presented here with models representing isolated marrow-cavity spheres. In the following text, the previous model will be referred as the mathematical trabecular-bone model, whereas the models developed in the present study will be referred as single-sphere models.

Material and Methods

Construction of the 3D Segmented Images of the Single-Sphere Models

A set of eight models was created in which a single sphere was located at the center of a cube representing a small trabecular bone sample encompassing a single marrow cavity. The radius of the sphere and the size of the cube were selected according to the following criteria:

- The entire set should cover the range of typical marrow cavity sizes found in trabecular bone
- The series of cube sizes and sphere radii should follow approximately a geometric progression
- In each model, the sphere should be surrounded by a typical trabecular thickness
- The sphere radius should be chosen so that it does not represent an integer number of voxels.

The last condition ensures that the bone-marrow interface is rarely adjacent to a voxel side. Such an undesirable situation could introduce geometrical effects that could alter the results since it is never realized within a real bone sample where the marrow cavities are randomly located. [Table 4-1](#) contains the characteristics of the series of single-sphere models. The ten digits of the sphere radius guarantees the fourth condition defined above.

Once the single-sphere models are defined, a grid structure is placed over the cubes, representing the voxel dimensions of a 3D image acquisition. Different voxel sizes are used for each model to cover the largest possible range of image resolution. The minimum voxel size is set to 2.5 μm , which corresponds to the range of a 10 keV electron in bone marrow and of a 15 keV electron in cortical bone. The maximum voxel size is set so that each image contains at least one voxel of each media. [Table 4-2](#) lists the voxel sizes used for the study. A geometric progression is also used for the choice of the number of voxels per dimension.

The segmentation of the different images into bone and marrow voxels is based upon a technique similar to that developed for the mathematical bone model and explained in [Chapter 3](#). The principle is to calculate the volume fraction of each voxel inside the sphere. If this volume fraction exceeds 0.5, the voxel is assigned to marrow; otherwise, it is assigned to bone. When the volume fraction is not easy to calculate analytically, a

Monte-Carlo sampling technique is used. Once the segmentation is complete, a compression technique similar to that developed for the mathematical bone model is used. This technique permits images as large as two billion voxels to be handled by the EGS4 transport code.

Volume Fraction Occupied by the Spheres within the Segmented Images

The study detailed in [Chapter 3](#) showed that the volume fraction of marrow is not represented well for voxel sizes larger than 300 μm . To verify this result, and to include its consequences in the data interpretation of the present study, a computer program was created to calculate the marrow volume fraction in the voxelized single-sphere models.

Theoretical Surface Area of a Voxelized Sphere

As the resolution of a 3D image is improved, it would appear that the voxelized surface of the image would approach the exact surface of the real object. Nevertheless, the voxelized surface still follows rectangular shapes and the area measured within the 3D image does not converge to the real surface area, even though the voxel size is reduced to an infinitely small value. The problem is demonstrated in 2D as follows.

Consider a circle of radius R as shown in [Figure 4-2](#). If a 2D image of the circle is made with a digital imaging system, the result will be the grid structure shown in that same figure. For this example, let us assume that the imaging system assigns each pixel as dark or white according to the fraction of its perimeter that is outside or inside the circle. The white pixels will then represent the interior of the circle (marrow) and the dark pixels will represent the exterior (bone). Therefore, the perimeter of the circle, if measured within the image, is equal to the length of the cumulative interface between the white pixels and the dark pixels. Each pixel side that belongs to this perimeter can be moved as shown by

the arrows to form the large dashed square. Therefore, the perimeter of the voxelized circle is equal to the perimeter of the dashed square. The size of this square depends on the pixel size and can be slightly smaller or slightly larger than the diameter of the circle. When the pixel size is reduced to an infinitely small value, the size of the square converges to the diameter of the circle. The total perimeter of the image of the circle is therefore equal to 8 times the radius of the circle. This, if compared with the exact perimeter of the circle, corresponds to an overestimate equal to

$$\frac{8R - 2pR}{2pR} = 0.2732 \quad \Rightarrow \quad 27.32 \% . \quad (4-1)$$

This overestimate is the limit of convergence when the pixel size is reduced to an infinitely small value. For a larger pixel size, the overestimate will be slightly smaller or slightly larger than the convergence value. This variation about the convergence value results from the fact that, depending on the ratio between the circle radius and the pixel size, the circle will not exactly fit within the larger dashed square.

The 3D problem is more complex and can be solved analytically. A sphere of radius R can be considered as a set of slices along the z -axis. Each slice is a flat disk with radius r varying with the distance z . This is represented in [Figure 4-3](#) where a 2D projection of the sliced sphere is shown. The vertical rectangles represent the disks seen on edge. The thickness dz of the disks is assumed to be the voxel size of the 3D image. The surface area of the edge of each voxelized disk can be derived from the previous result for the circle by multiplying the perimeter of the disk (within the voxelized image) by its thickness (the voxel size). That is

$$ds = 8rdz . \quad (4-2)$$

When the voxel size is reduced to an infinitely small value, the total surface area can be calculated by the integration

$$S = 2 \int_0^R 8rdz . \quad (4-3)$$

However, this expression only accounts for the surface area of the edges of the voxelized disks. These edges are perpendicular to the x -axis or the y -axis but never perpendicular to the z -axis; therefore, and because of the symmetry of the problem, they account for only two thirds of the total surface area of the voxelized sphere. The total surface is then obtained by

$$S = 3 \int_0^R 8rdz . \quad (4-4)$$

Using a variable change:

$$r = R \sin \mathbf{q} , \quad \text{and} \quad z = R \cos \mathbf{q} \quad \Rightarrow \quad dz = -R \sin \mathbf{q} d\mathbf{q} . \quad (4-5)$$

The integration thus becomes:

$$S = 24R^2 \int_0^{\frac{\pi}{2}} (\sin \mathbf{q})^2 d\mathbf{q} . \quad (4-6)$$

The calculation gives the result:

$$S = 6\mathbf{p}R^2 . \quad (4-7)$$

Equation (4-7) shows that reducing the voxel size will make the surface area of a voxelized sphere converge to a value that is 50% higher than the real surface area of the sphere ($4\mathbf{p}R^2$). For larger voxel sizes the surface area will be slightly larger or slightly smaller than the convergence value according to the ratio of the voxel size and the sphere radius.

Measurement of the Surface Area of a Voxelized Sphere

To check the theoretical result given by [Equation \(4-7\)](#), the surface area of the spheres was measured within each simulated image. A computer program was created to perform this task. Its principle is to sweep every voxel of the image and to check if the six adjacent voxels (or less if the voxel is on the edge of the cube) are composed of a different medium. Each time one adjacent voxel is found of a different medium, the surface area of the interface between the current and the adjacent voxel is added to the total surface area. By doing this, each surface is counted twice, and therefore the final result is reduced by a factor of two to obtain the true surface area of the voxelized sphere.

Consequences of an Error in Surface Area on the Absorbed Fraction

The overestimation of the surface area will affect the result of the Monte-Carlo transport code when the particles are emitted close to the boundary of the two media, as each electron will have a greater than normal chance of crossing that boundary. This surface-area effect will vary for different voxel sizes as demonstrated in [Figure 4-4](#).

[Figure 4-4-A](#) demonstrates the situation of a voxel size small compared to the electron range. The diagram is in 2D but it can be easily transposed to the 3D situation. The thick solid straight line represents the real boundary between the medium of the radiation source (lower right side of the line), and the medium of the irradiated target (upper left side). The thin solid jagged line represents the same boundary, but as seen within the corresponding digital image. This boundary follows the shape of the rectangular image pixels. The arrows represent electrons (assumed monoenergetic) crossing the boundary. In both situations (the real boundary and the jagged one), electrons crossing the boundary must leave the source at a distance shorter than their

range in the source medium. Therefore, in the real situation, the effective source area from which the electrons can reach the other medium extends from the boundary to the thick dashed straight line, the distance between the two lines being the electron range. In the case of the digital image, the effective source area is more complex, as the boundary is no longer straight. Each dashed circle represents the limit of the effective source area from which the electrons can reach each corner of the image boundary located on the source side of the real interface. The radius R is equal to the electron range. These circles extend slightly over the thick dashed straight line since their radius is equal to the distance between the two thick lines. The overall limit of the effective source area is the envelope of these circles. One can see in [Figure 4-4-A](#) that the source area enclosed between the thin jagged line and the envelope of the circles is nearly identical to the source area enclosed between the two thick lines. Therefore, if the voxel size is small compared to the electron range, the error in interface length introduces little error on the cross-region absorbed-fraction calculation since approximately the same number of electrons cross the boundary in the real case as within the digital image.

[Figure 4-4-B](#) represents exactly the same geometrical features as in [Figure 4-4-A](#), but for a voxel size that is larger than the electron range. Only the circles centered at the summits of the thin solid line have been represented, but one can easily understand that the envelope of the effective source area from which the electrons can reach the other medium extends from the solid thin jagged line to the dashed thin line (envelope of the circles). In this case, the two source areas (between the thick lines and between the thin lines) are different. Since the electron range is smaller than the voxel size, the shape of the interface affects the source area from which the electrons can reach the other side. This source area

converges to the length of the thin solid line multiplied by the electron range when the voxel size becomes very large compared to the electron range.

The 3D situation is similar and the effective source volume from which the electrons can reach the other side of the boundary is proportional to its surface area. Therefore, a 50 percent overestimation of the surface area overestimates the effective source volume for cross irradiation by 50 percent as well. Correspondingly, the cross-region absorbed fraction is also overestimated by as much as 50%.

The real boundary in the example problem of [Figure 4-4](#) is a straight line; this represents a geometry large in comparison to the electron range. In a smaller relative geometry, the shapes would be more complex but the principle would remain the same.

For the single-sphere models, let us assume that monoenergetic electrons start from the outside of the sphere (the bone trabeculae) and that the absorbed fraction within the sphere (the marrow cavity) is to be calculated. Taking into account the surface-area effect alone, and for a very large voxel size, the resulting absorbed fraction is expected to overestimate its exact value by 50% since the electron range is much smaller than the voxel size. By reducing the voxel size, the voxel dimensions will approach the electron range and thus the absorbed fraction would become closer to its exact value. If the voxel size is chosen very small, the results should converge. This scenario is shown in [Figure 4-5](#). The voxel size at which the convergence appears should be around, or at least function of, the electron range in the source region.

Electron-Transport Simulations

The next step in the study is to calculate the absorbed fractions of energy and to analyze their evolution with the voxel size. For this, particle transport calculations were

performed using the EGS4-PRESTA code. Two transport simulations were performed. One for a uniform monoenergetic electron source located within the sphere (e.g., marrow source) and a second for a uniform monoenergetic electron source located outside the sphere (e.g., bone source) but within the limits of the bone cube. For each simulation, six electron energies were used. As the surface-area effect only influences the calculation for low-energy electrons (see [Chapter 3](#)), the maximum energy was set to 320 keV. The minimum energy was set to 10 keV. For each electron energy and for both source regions, one set of EGS4 runs was performed for each single-sphere model. Each set represents one simulation within the exact single-sphere model and one simulation within each 3D image (each voxel size shown in [Table 4-2](#)). Therefore, two different EGS4 user codes were developed: a “reference code” for the exact sphere model, using the equation of the sphere delineating the marrow cavity boundary, and an “image code” for the different segmented images, using equations of planes separating the voxels. The “reference code” was executed for each model and for each electron energy. The “image code” was executed for each model, for each electron energy, and for each image resolution.

The parameters used by the EGS4 transport were the same for both codes. The characteristics of the bone and marrow media were identical to those used for the study of the mathematical trabecular bone model (see [Chapter 3](#)). For the PRESTA extension, the ESTEPE parameter was set to 0.05. For each execution, absorbed fractions of energy were estimated for targets inside the sphere, outside the sphere but inside the cube, and outside the cube (energy lost for this study). Only the cross absorbed-fraction results are shown and analyzed in this work.

The absorbed fractions calculated with the “image code” were compared to the absorbed fractions calculated with the “reference code”. The absorbed-fraction relative error Δr_f produced by the overestimation of the boundary surface area was then evaluated using the expression:

$$\Delta r_f = \left(\frac{f_{ima} - f_{ref}}{f_{ref}} \right) \times 100\% , \quad (4-8)$$

where f_{ima} is the absorbed fraction calculated using the “image code” and f_{ref} is the absorbed fraction calculated using the “reference code”.

Statistical Analysis

Combining all parameters described above, a total of 1,764 EGS4 simulations were performed. To minimize the calculation time, a statistical analysis was performed prior to the EGS4 simulations. Instead of one execution of N histories, 100 runs were performed. For each run, only one hundredth of N histories were used. This allows calculating a mean and a standard deviation within the sample of 100 runs. The standard deviation of the mean can also be estimated, as well as a 95 percent confidence interval assuming a normal distribution of the mean. As it is not known whether or not the distribution of the absorbed fraction follows a normal distribution, the later assumption is verified through application of the central limit theorem. This explains the large sample (100 runs in this case) used for this analysis. The number of histories was chosen so that the 95 percent confidence interval for the mean was reduced to about $\pm 2\%$ for each absorbed fraction calculated. As the accuracy of the result varies mostly with the electron energy and the source region, a different number of histories was set for each situation to reach the 2%

goal and to minimize the calculation time. [Table 4-3](#) shows the electron energies used and the number of histories per run for both types of sources.

Note that this 95 percent confidence interval is representative of the statistical fluctuation of the results because of the randomness of the Monte-Carlo method. It does not represent the accuracy of the technique used. It signifies that another experiment using the same Monte-Carlo transport code and the same experimental conditions would have a 95 percent chance to find a result within the confidence interval. The purpose of this work is not to assess the accuracy of the dosimetry technique, but to quantify the error due to the surface-area overestimate. To calculate the 95 percent confidence interval for the relative error, the error-propagation technique was used. According to the theory ([Knoll 2000](#); [Turner 1995](#)), the standard deviation of function $f(x,y)$ is given by the equation

$$\mathbf{s}_f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 \mathbf{s}_x^2 + \left(\frac{\partial f}{\partial y}\right)^2 \mathbf{s}_y^2}. \quad (4-9)$$

In our situation, using [Equation \(4-8\)](#) to calculate the partial derivatives, the expression becomes

$$\mathbf{s}_{\Delta r_f} = \frac{\sqrt{\mathbf{f}_{ref}^2 \mathbf{s}_{f_{ima}}^2 + \mathbf{f}_{ima}^2 \mathbf{s}_{f_{ref}}^2}}{\mathbf{f}_{ref}^2}. \quad (4-10)$$

In [Equation \(4-10\)](#), $\mathbf{s}_{f_{ref}}$ and $\mathbf{s}_{f_{ima}}$ are the standard deviations of the mean calculated with

$N = 100$ runs of respectively the “reference code” and the “image code” using the relation

$$\mathbf{s}_f = \sqrt{\frac{\sum_{i=1}^N (\mathbf{f}_i - \mathbf{f})^2}{N(N-1)}}. \quad (4-11)$$

The 95 percent confidence interval is therefore obtained by:

$$Err_{95\%} = \pm 1.96 \sqrt{\frac{\mathbf{f}_{ref}^2 \mathbf{s}_{f_{ma}}^2 + \mathbf{f}_{ima}^2 \mathbf{s}_{f_{ref}}^2}{\mathbf{f}_{ref}^2}}. \quad (4-12)$$

Results and Discussion

The results from the eight single-sphere models are quite consistent. Therefore, only 4 models are shown below. The models chosen are 1400, 0728, 0350, and 0069.

Volume Fraction Occupied by the Spheres within the Segmented Images

The volume fraction occupied by the spheres within the segmented images is compared with the exact value in [Figure 4-6](#) for each model. In this figure, the horizontal solid lines represent the exact volume fraction of the spheres, with values given in column 6 of [Table 4-1](#). For each model, the volume fraction converges toward its exact value as the voxel size is reduced. The convergence occurs around a voxel size that decreases as the radius of the marrow sphere decreases.

The study of the mathematical trabecular bone model ([Chapter 3](#)) showed that the error on the volume fraction at large voxel sizes, referred to as the volume-fraction effect, was not a concern for dosimetry calculations since it occurs only above 300 μm in a typical bone sample. Therefore, in the remainder of this study, we focus only on voxel sizes small enough so that the surface-area effect can be studied in isolation.

Surface Area of the 3D Segmented Images

The surface areas calculated within the segmented images are compared with their exact values in [Figure 4-7](#). In this figure, the horizontal solid lines represent the exact surface areas of the marrow spheres whose values are given in column 4 of [Table 4-1](#). The dashed lines are at 1.5 times the exact values. These results are in excellent

agreement with predicted values calculated in the materials and methods section of this chapter. Furthermore, the surface area is expected to oscillate around a convergence value. It can be larger or smaller than the convergence value according to the ratio between the voxel size and the sphere radius. These oscillations are particularly evident for the small spheres; for the larger ones, they are barely visible, but still present. At large voxel sizes, the oscillations disappear for all spheres since the surface-area effect is overwhelmed by the volume fraction effect. For very large voxel sizes, the surface area drops to zero since the volume fraction also decreases toward zero.

Absorbed Fractions within the 3D Segmented Images

The next step was to calculate the absorbed fraction of energy deposited by electrons within the different models. In this section, only the cross-absorbed fractions are considered since the self-absorbed fractions are reasonably accurate even at small voxel sizes (according to the results of [Chapter 3](#)). The results are presented in [Figure 4-8](#) for electron sources in bone irradiating marrow, and in [Figure 4-9](#) for electron sources in the marrow irradiating bone. In each figure, the four selected models are shown: [Figure 4-8-A](#) and [4-9-A](#) are for model 1400, [Figure 4-8-B](#) and [4-9-B](#) are for model 0728, [4-8-C](#) and [4-9-C](#) are for model 0350, [4-8-D](#) and [4-9-D](#) are for model 0069. For each graph, the curves represent the relative error in the cross-absorbed fraction given by [Equation \(4-8\)](#) as a function of the voxel size and electron energy. The error bars show the 95 percent confidence intervals calculated by [Equation \(4-12\)](#). These curves are to be compared with the expected results shown schematically in [Figure 4-5](#).

First, for voxel sizes larger than the resolution at which the volume fraction converges, the results cannot be interpreted by the surface-area error alone since it is

overwhelmed and largely influenced by the volume fraction. This is why, at these voxel sizes, the absorbed fraction can be overestimated or underestimated, depending upon the volume fraction of the segmented sphere. At this resolution, one can see that the shapes of the absorbed-fraction curves follow more or less the shapes of the volume fraction curves.

Second, for a voxel size small enough so as to minimize the volume-fraction effect, the surface-area effect is visible in each model as predicted in the material and methods section of this chapter. The 50% overestimate of the surface area leads to a 50% overestimate of the cross-absorbed fraction down to a voxel size that depends on the electron energy. Below this voxel size, the absorbed fraction converges to its exact value. This convergence is only visible at high energies; at very low-energy electron sources (e.g., 10 and 20 keV), the shape of the absorbed-fraction curve indicates that a similar effect would occur at a voxel size below the range used in the study. These results are in strong agreement with [Figure 4-5](#).

Third, the resolution at which the absorbed fraction starts to converge toward the exact value can be compared with the electron range as shown in [Table 4-4](#) (data from ICRU (1984) Publication 37). At electron energies from 20 keV to 80 keV, the electron range is equal to the voxel size that corresponds to a relative error of ~25% (i.e., the middle of the convergence slope). For the 10 keV curves, the voxel size is not extended to a value small enough to show the same consequence, but an extrapolation of these curves can be easily made. At higher electron energies (160 and 320 keV curves), the convergence slope falls above the volume-fraction convergence value. Once again, the

convergence slope can be deduced from the lower part of the curves, and the extrapolated shapes would show a convergence slope at a voxel size close to the electron range.

Finally, one can see that, at voxel sizes that are not affected by the volume fraction effect, the sphere radius does not change the shape of the curves. The surface-area effect depends only on the ratio between the electron range and the voxel size, as predicted in the material and methods section of this chapter.

Conclusion

The surface-area effect resulting from the segmentation of an object within its 3D digital image has been investigated using single-sphere models of trabecular bone-marrow cavities. It has been shown, both analytically and experimentally, that a consequence of the image segmentation is an overestimation of the interface surface area. This overestimate, identified as a surface-area effect, is a result of the rectangular shape of the voxels that constitute the image and cannot be reduced by improved resolution. In the case of single spheres, the overestimate is as high as 50%. This finding is consistent with the studies of presented in [Chapter 3](#) where a similar overestimate of bone-marrow interface area was found in a mathematical model of trabecular bone.

The dosimetry consequence of the surface-area effect is an overestimate of the absorbed fraction by electrons cross-irradiating a region adjacent to the source region (bone source irradiating marrow, for example). At a voxel size larger than the electron range, this overestimate is equal to the overestimate of the surface area (50% for spherical objects). As the voxel size is reduced, the error decreases and becomes insignificant when the voxel size falls below the electron range. For high-energy electrons, this effect is inconsequential since the electron range is significantly larger than the voxel size used to

image trabecular bone samples. For electron energies below 100 keV, however, the effect is significant to a voxel size on the order of a few micrometers. With typical resolutions used in NMR microscopy (from 60 to 100 μm), and for very low-energy beta emitters (e.g., $^{117\text{m}}\text{Sm}$ and ^{33}P), the dose calculation using these absorbed fractions leads to a 25% overestimate within the mathematical model of trabecular bone, as shown in [Chapter 3](#). This mathematical model was designed with dosimetry features as close as possible to those of a real trabecular bone sample, and thus the same consequence is expected within NMR images of real samples.

As shown in [Figure 4-5](#), the problem can be incrementally solved by improving the image resolution. [Figures 4-8](#) and [4-9](#) shows that for low-energy electrons (let us say 40 keV) the resolution needs to be pushed downward to $\sim 5 \mu\text{m}$ in order to achieve a close estimate of the cross-dose. This resolution is ~ 10 times better than what is currently performed in NMR microscopy of trabecular bone ([Patton et al. 2002a](#)) and ~ 6 times better than available with microCT systems ([Ruegsegger et al. 1996](#)). Increased resolution of the NMR microscopy images are possible via improvements in the signal-to-noise ratio of the NMR system acquisition via higher field strengths (e.g., 20 T) and/or longer imaging times. Compromises occur, however, in that one must contend with susceptibility artifacts at the bone-marrow interface at higher fields, and longer imaging sessions can become cost prohibitive. In addition, as the images become larger with decreasing voxel size, computer storage can potentially be an issue.

Nevertheless, other solutions to the problem may be sought. One such solution would be to perform an interpolation of the gray-level values of the original NMR image to calculate a polygonal isosurface that would represent the bone-marrow interface. In

this manner, image voxels at the interface would encompass one or more polygons separating the marrow tissues from the osseous bone tissue. The exact positions and relative angles between polygons would be a function of both the relative gray level of the interface voxels and the resolution of the image (i.e., voxel size). The resulting collection of polygons would thus permit a more smooth representation of the true bone-marrow interface surface and would thus better represent the true surface area. Such a polygonal isosurface has already been used with trabecular bone microstructure (Muller et al. 1994) and its construction is based on the Marching-Cube algorithm developed during the 1980's by Lorensen and Cline (1987). In this algorithm, the resulting isosurface is only made of triangles that insure that each piece of the surface can be modeled with equations of planes. Through the application of this technique, previous problems identified in either the assessment of bone and marrow chord-length distributions (Jokisch et al. 2001b), as well as absorbed fractions for cross-irradiation by low-energy electrons (see Chapter 3) would be significantly reduced from those found using voxelized geometries. The storage capacity of the computer would still remain a major concern, since the number of triangles would be on the same order of the number of voxels, but the image resolution no longer needs to be tremendously improved, and may even be reduced if the voxel effects are shown to be significantly reduced.

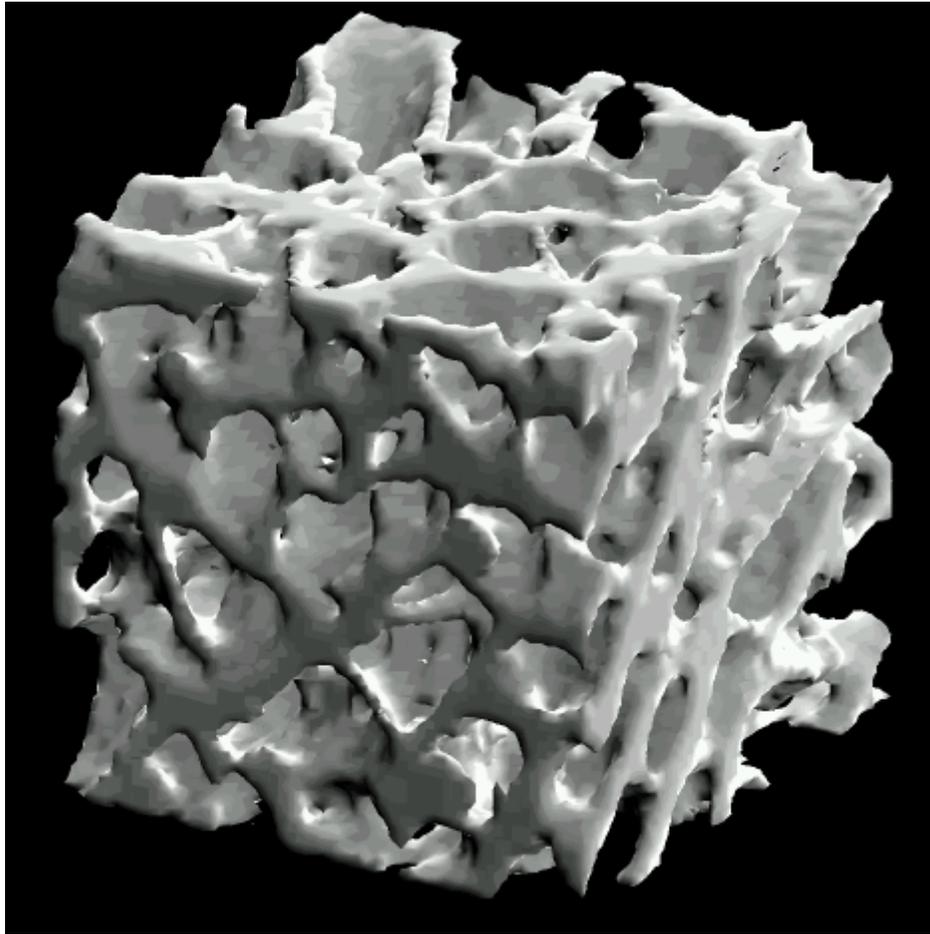


Figure 4-1. Cubical sample of trabecular bone reconstructed from a 3D NMR image obtained at 4.7 tesla. The image was taken over an 11 hour and 10 minute acquisition time (TR = 600 ms, TE = 9.1 ms, spectral width: 123,457 Hz, 2 averages, matrix: 512 x 256 x 256, field of view: 4.5 x 2.25 x 2.25 cm³). The sample was sectioned from the right femoral head of a 51-year male. The sample size is 5.6 x 5.6 x 5.6 mm³. The image resolution is 88 x 88 x 88 μm³.

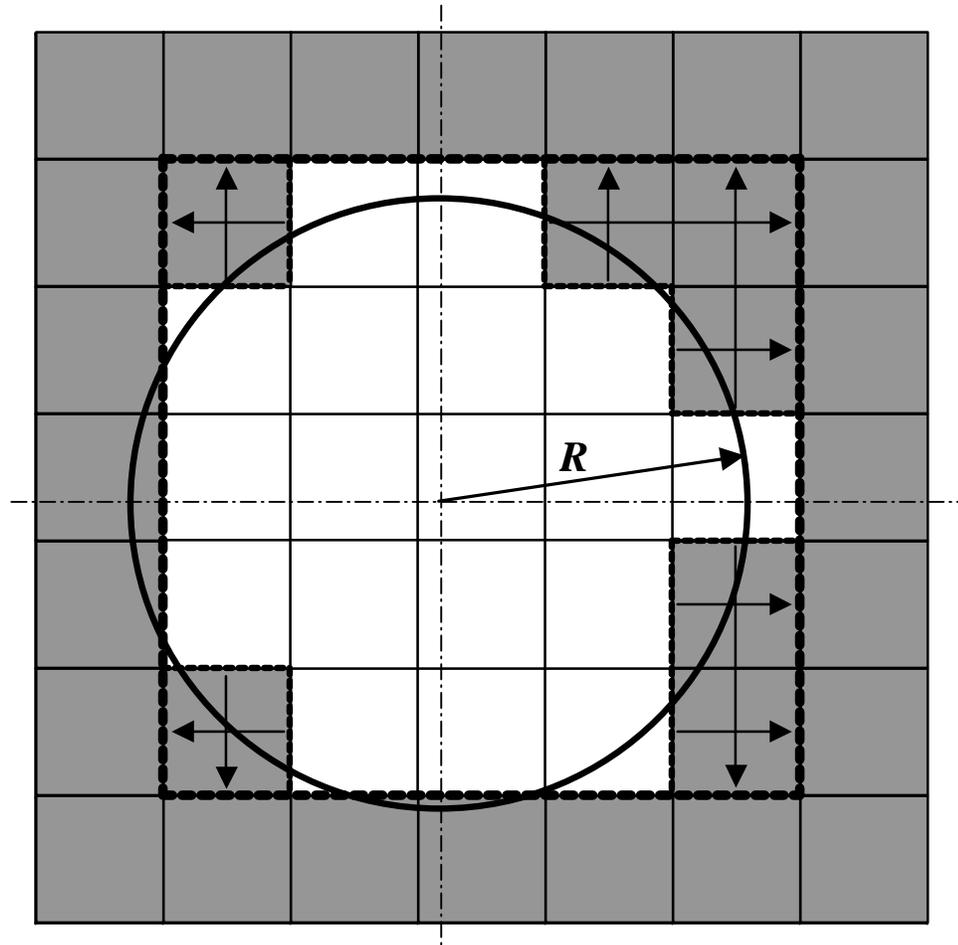


Figure 4-2. Perimeter of a circle as represented by a digital image. Because of the image voxelization the perimeter is equal to the perimeter of a square. When the pixel size is reduced, the perimeter converges to the perimeter of the square whose size is the diameter of the circle. A 27% increase in perimeter of the circle results when measured within the digital image.

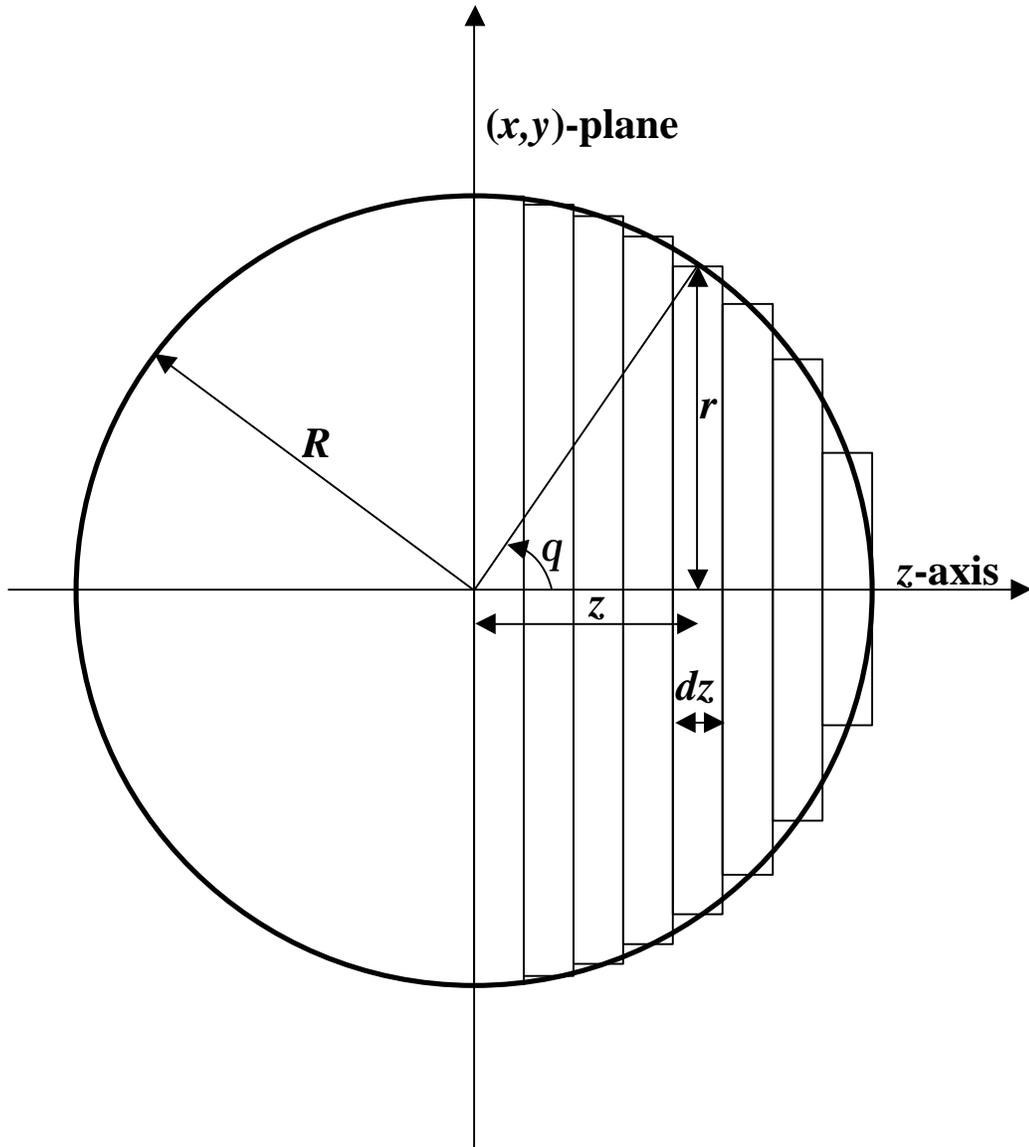


Figure 4-3. Analytical derivation of the surface area of a voxelized sphere. The sphere is divided into circular slices of thickness dz and of radius r . The surface area of the edge of each slice is calculated using the results obtained in 2D. The total surface area is obtained by integrating over the entire sphere.

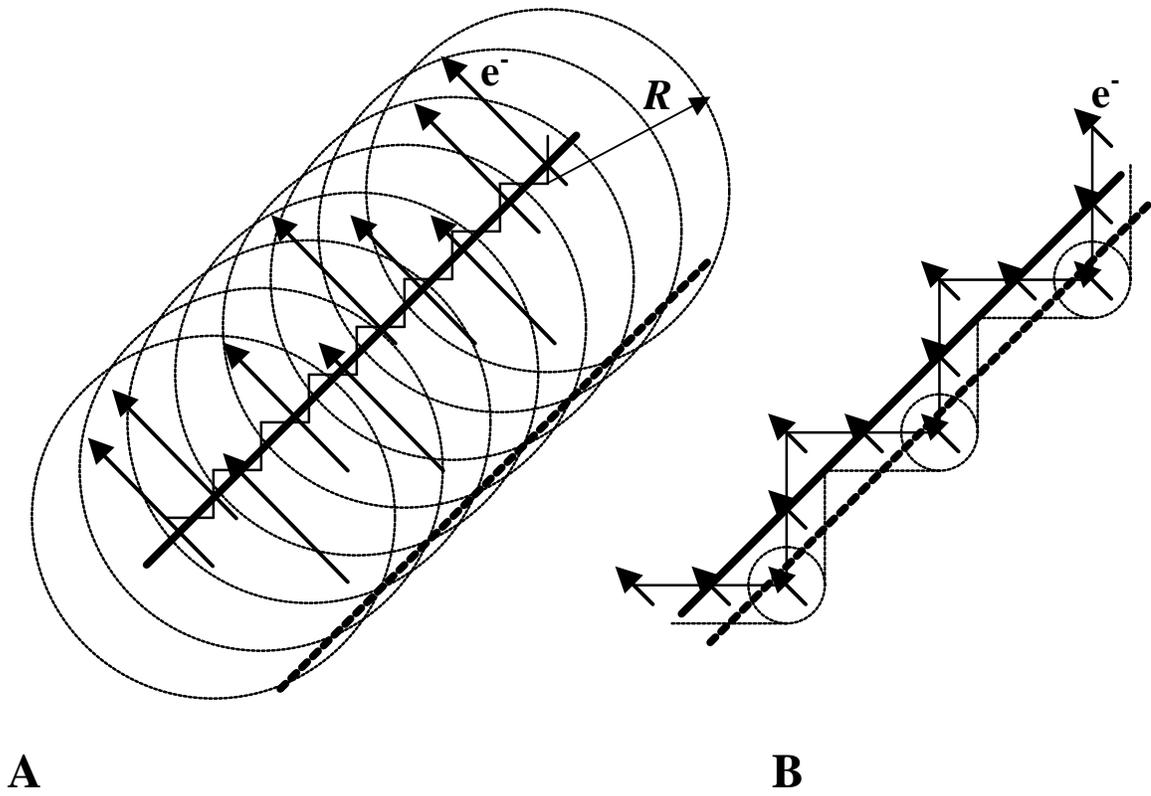


Figure 4-4. Consequence of the surface-area error on the absorbed-fraction calculation.
A) The electron range is larger than the voxel size. B) The electron range is smaller than the voxel size.

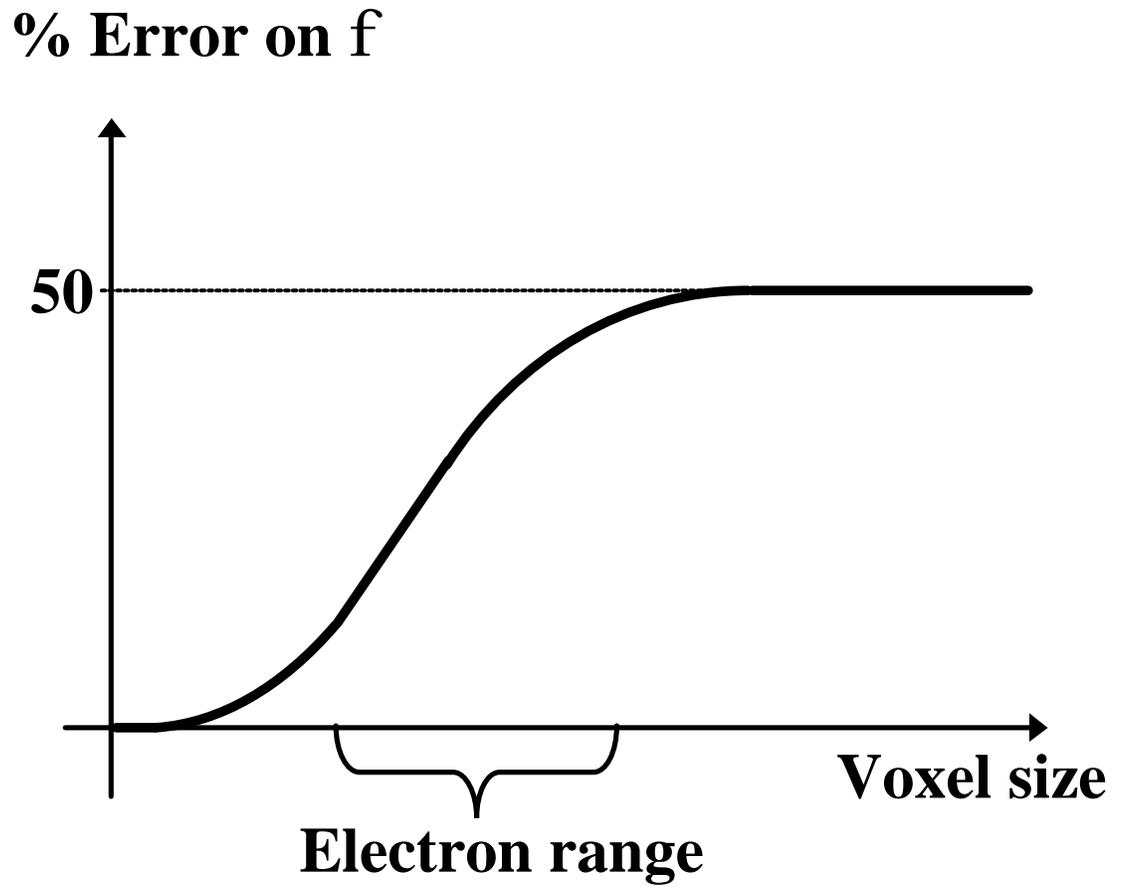


Figure 4-5. Expected evolution of the cross-absorbed-fraction overestimation as a function of the voxel size.

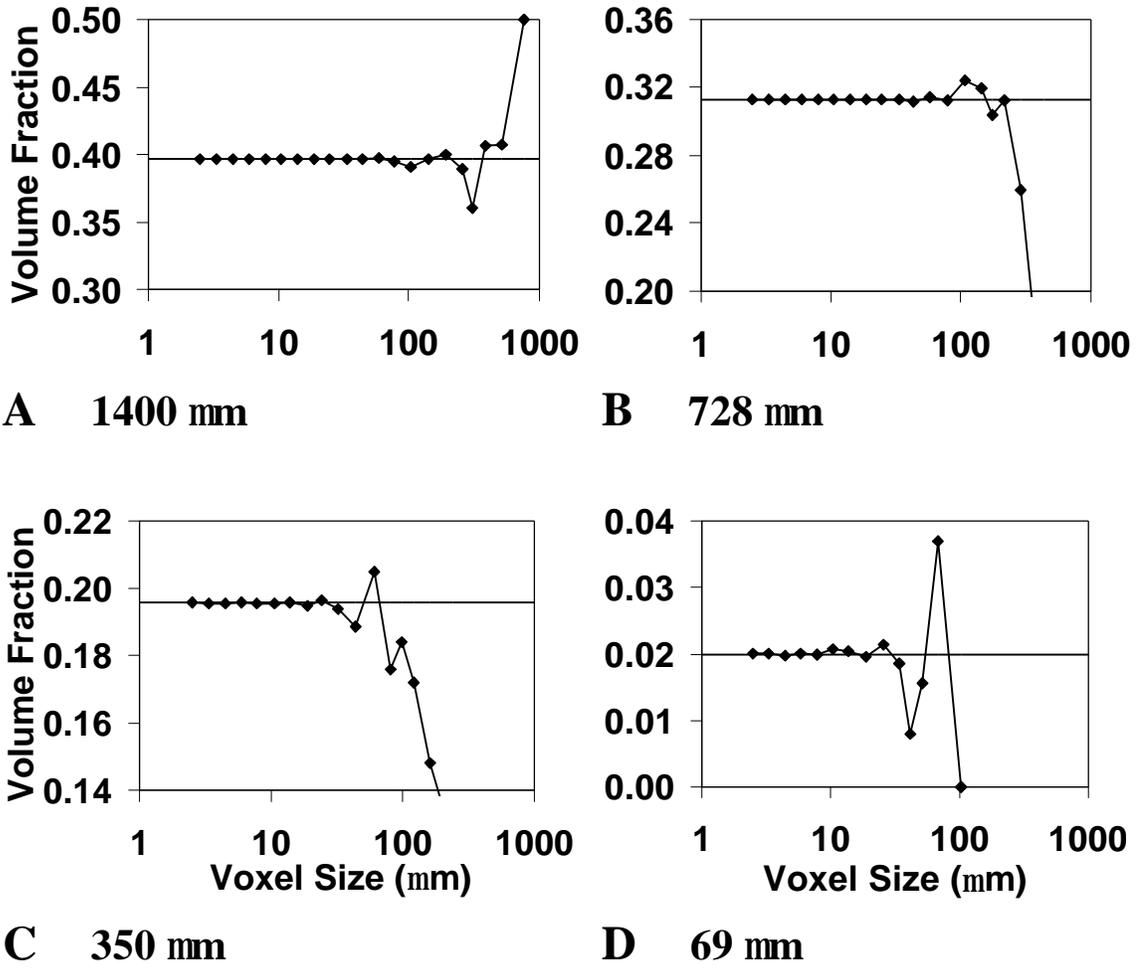


Figure 4-6. Volume fraction occupied by the sphere within the segmented image as a function of the voxel size. The horizontal solid line is the volume fraction occupied by the real sphere. The four single-sphere models represented are A) 1400, B) 0728, C) 0350, and D) 0069.

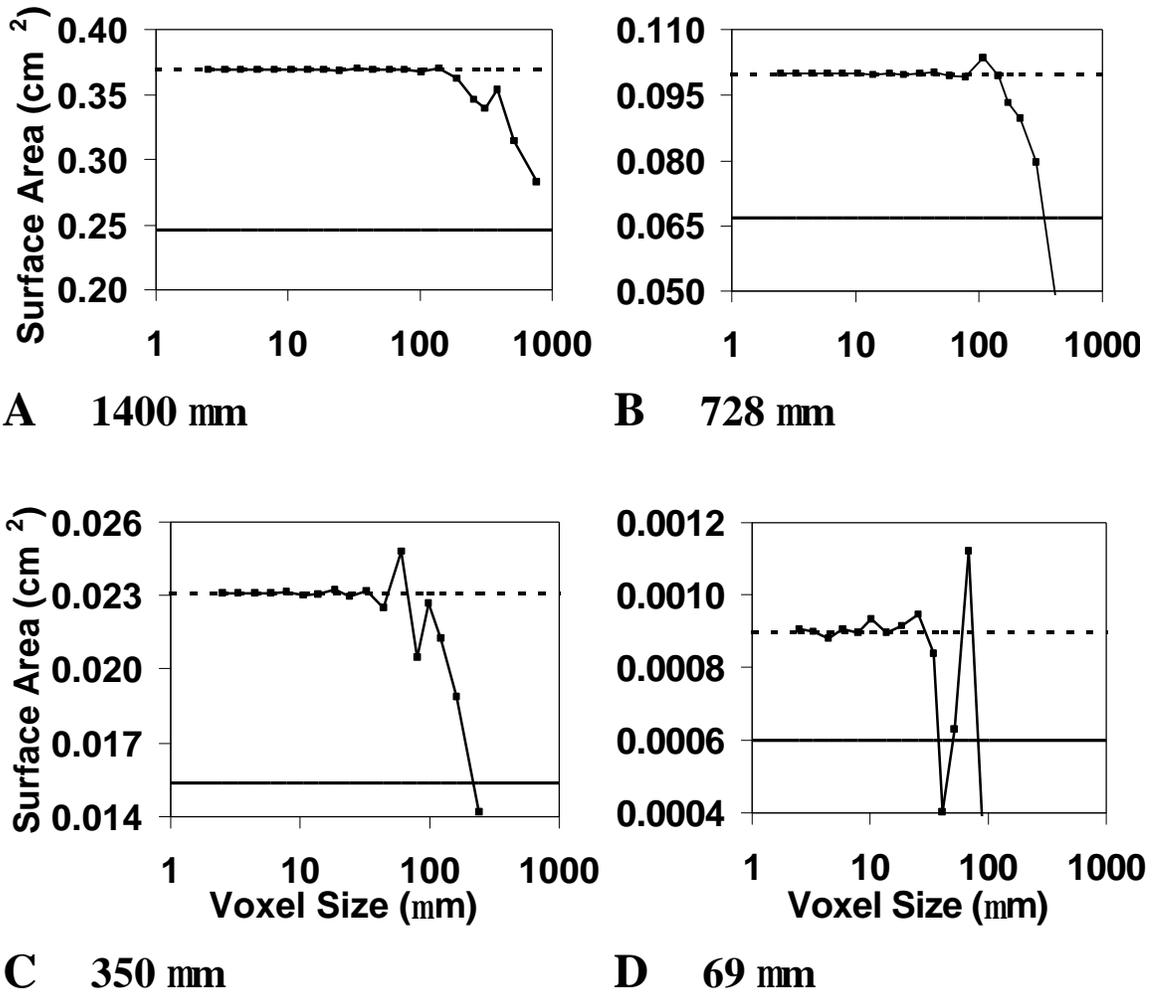


Figure 4-7. Surface area of the segmented spheres, as a function of the voxel size. The horizontal solid line is the exact surface area of the sphere. The horizontal dashed line is located at 1.5 times the exact value. The four single-sphere models represented are A) 1400, B) 0728, C) 0350, and D) 0069.

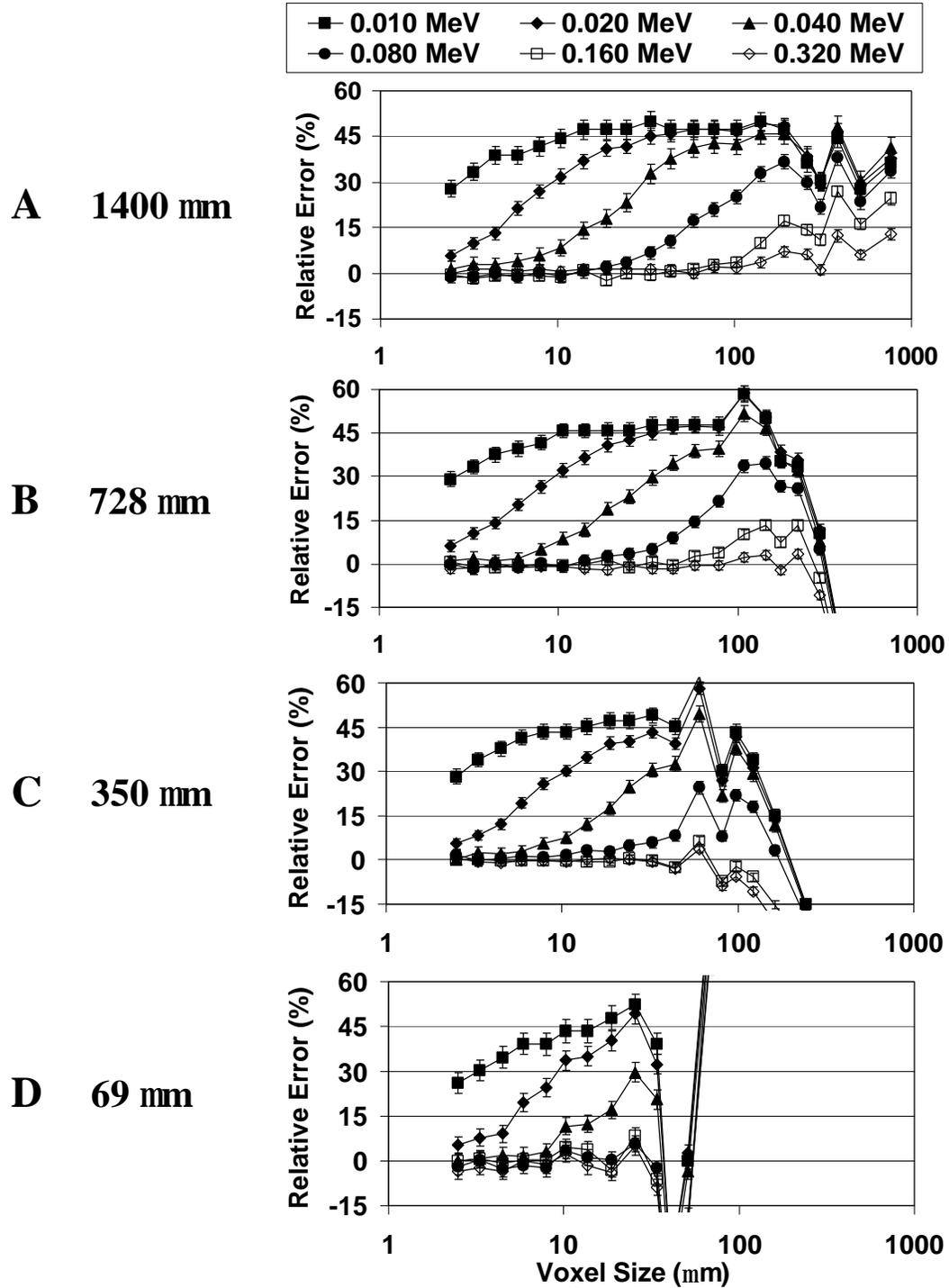


Figure 4-8. Relative error on absorbed fractions inside the sphere. The source of particles is outside the sphere (bone source irradiating marrow). The four single-sphere models represented are A) 1400, B) 0728, C) 0350, and D) 0069.

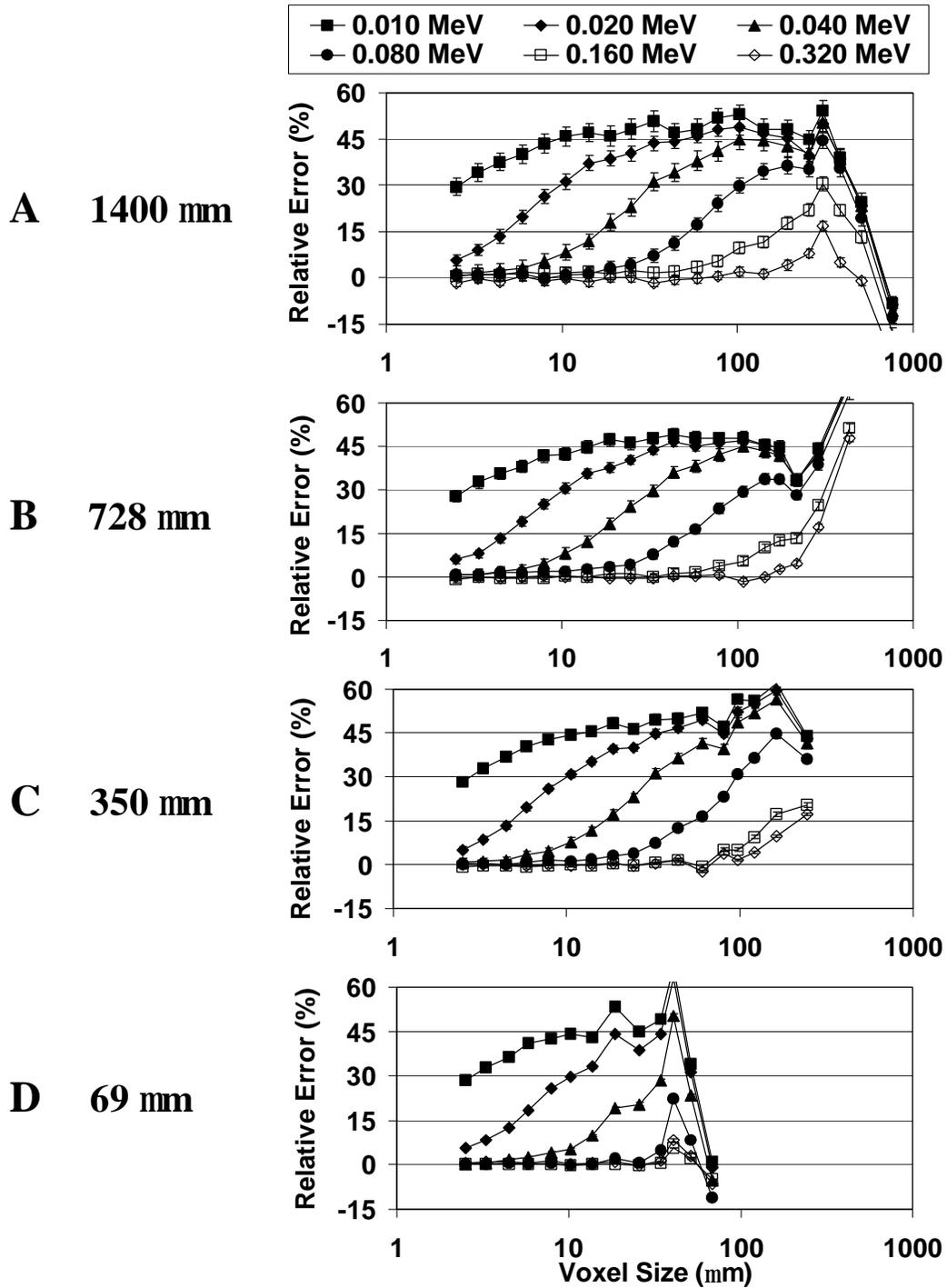


Figure 4-9. Relative error on absorbed fractions outside the sphere. The source of particles is inside the sphere (marrow source irradiating bone). The four single-sphere models represented are A) 1400, B) 0728, C) 0350, and D) 0069.

Table 4-1. Characteristics of the eight single-sphere models covering a typical range of bone-marrow cavity sizes. The name of each model is the radius of the sphere in micrometers, rounded to the closest integer. The volume fraction represents the fraction of the cube that is contained within the marrow sphere.

Model name	Sphere radius (μm)	Cube size (μm)	Sphere surface area (cm^2)	Sphere volume (cm^3)	Volume fraction (%)
1400	1400.085909	3072	2.463×10^{-1}	1.150×10^{-2}	39.65
1016	1016.085909	2304	1.297×10^{-1}	4.394×10^{-3}	35.92
0728	728.085909	1728	6.662×10^{-2}	1.617×10^{-3}	31.33
0512	512.085909	1296	3.295×10^{-2}	5.625×10^{-4}	25.84
0350	350.085909	972	1.540×10^{-2}	1.797×10^{-4}	19.57
0228	228.085909	728	6.537×10^{-3}	4.970×10^{-5}	12.88
0137	137.085909	546	2.362×10^{-3}	1.079×10^{-5}	6.62
0069	69.085909	410	5.998×10^{-4}	1.381×10^{-6}	2.00

Table 4-3. List of electron energies used in the study. The total number of histories per energy is equal to the number of runs multiplied by the number of histories per run.

Electron energy (keV)	Number of runs	Histories per run (source inside the sphere)	Histories per run (source outside the sphere)
10	100	120,000	300,000
20	100	50,000	125,000
40	100	10,000	25,000
80	100	4,000	10,000
160	100	1,600	4,000
320	100	800	2,000

Table 4-4. Electron ranges in bone marrow and cortical bone. The values have been interpolated from ICRU (1984) Publication 37, using cortical bone and water (corrected for density) for bone marrow. The densities are 1.03 g cm^{-3} and 1.92 g cm^{-3} for marrow and bone respectively.

Electron energy (keV)	CSDA range in bone marrow (μm)	CSDA range in cortical bone (μm)
10	2.4	1.4
20	8.3	4.9
40	28	16
80	95	55
160	300	180
320	900	520

CHAPTER 5
INTERACTION WITH 3D ISOTROPIC AND HOMOGENEOUS RADIATION
FIELDS: A MONTE-CARLO SIMULATION ALGORITHM¹

Introduction

Numerous situations exist in medical physics and biomedical engineering that involve the problem of a fixed object interacting with directly ionizing (charged particle) or indirectly ionizing (photon or neutron) radiation fields. Medical dosimetry routinely involves computational simulation of radiation interactions with various organs and tissues of the body. Chord-length distributions through irradiated objects have been extensively used to study these interactions, since they characterize the distance each individual particle will traverse through the irradiated object placed within the isotropic radiation field (Coleman 1969; Jokisch et al. 2001a; Jokisch et al. 1998; Jokisch et al. 2001b; Kellerer 1971).

At times, these interaction problems can be solved at a macroscopic level where the radiation field is expressed in terms of macroscopic quantities such as the photon fluence. When the geometry of the irradiated object is simple, macroscopic descriptions of the irradiation may suffice. In those cases where the geometry is complex, or if the problem itself cannot be solved via deterministic analytical solutions, Monte-Carlo methods are

¹ This chapter was accepted for publication by Computer Methods and Programs in Biomedicine and will be published in January 2003: Rajon DA Bolch WE. 2003. Interaction with 3D isotropic and homogeneous radiation fields: a Monte Carlo algorithm. Comp Meth Prog Bio 70: in press.

used. These methods are based on the random behavior of most interactions as studied at their microscopic level.

Three-Dimensional Homogeneous and Isotropic Fields

In this chapter, the term radiation field is taken in its most general meaning. It describes any macroscopic field that can be represented at the microscopic level by a spatial distribution of particles having a local density \mathbf{r} (in particles per m^3) and a local velocity \mathbf{u} (in $\text{m}\cdot\text{s}^{-1}$). The particles can be photon quanta or particulate in nature. The particle radiation field is described by its local fluence rate \mathbf{f} (also referred sometimes as the flux density)

$$\mathbf{f} = \mathbf{r}\mathbf{u} \quad (5-1)$$

The units of fluence rate are ($\# \text{ m}^{-2}\cdot\text{s}^{-1}$) where $\#$ represents the number of particles. In the most general situation, the fluence rate depends on the location in space and on the direction of the velocity. In this study, we will only refer to homogeneous and isotropic fields; consequently, the fluence rate will be constant over all defined space and will include particles coming from any direction.

In most situations, the field also has a limitation in space. This limitation has a consequence on the homogeneity and the isotropy of the radiation field. Physical models that wish to take this limitation into account are therefore more complex. When the spatial extension of the field is much larger than the size of the fixed irradiated object, an infinite field is an appropriate approximation to the real situation. This assumption is often made in field interaction applications since it simplifies greatly the physical model. In this study, we will assume an infinite radiation field, so that homogeneity and isotropy remain valid.

Monte-Carlo Methods

The key of all Monte-Carlo methods is that, at the microscopic level, the interaction of each particle that constitutes the radiation field with the fixed object can be modeled in terms of probabilistic data. The interaction of a single particle results in a series of possible events, each event having a certain probability to occur. If one wants to simulate the effect of the interaction of one particle with the irradiated object, random numbers are generated and used to select one of several possible outcomes of the interaction. Of course, all events are possible, but only one will be chosen for this specific particle. This single-particle event is generally insignificant in comparison to all interactions of the entire field. If one wants to know the macroscopic effect of all interactions, the simulation experiment must be repeated many times so that a series of random numbers can be used to select event outcomes representative of the distribution of outcomes in the actual irradiation scenario. If the number of trials is sufficiently large, the result of the entire simulation will be representative of the real event.

Random Number Generators

In some complex situations, millions of trials must be attempted before an accurate result is found. As a result, the development and improvements of Monte-Carlo techniques have largely been related to improvements in computational speed. In computer simulations, algorithms are used to generate random numbers needed for the Monte-Carlo sampling. Most of the programming languages used today provide their own random number generators. Unfortunately, some are still based on 16-bit integer arithmetic and are very inefficient. A good random number generator should be able to provide sequences of billions of random numbers without any bias of the randomness.

The state of the art can be found in the scientific literature. One suggestion is the Marsaglia and Zaman ([Marsaglia and Zaman 1991](#)) algorithm that has been used by Monte-Carlo transport codes over the past decade.

Material and Methods

The purpose of the technique presented in this chapter is to “simulate the randomness of a homogeneous and isotropic infinite 3D field of radiation when it interacts with a fixed object”. The term homogeneous refers to a uniform distribution of source particles throughout the infinite 3D space, while the term isotropic refers to a uniform distribution of directions for each individual particle throughout 4π -steradian space. In the strict definition of the problem, the probability for a particle interaction with the object is null in most cases since the particles are more likely to miss the object altogether. Consequently, the Monte-Carlo process is very inefficient and improved algorithms must be developed.

General Description of the Algorithm

The inefficiency of the technique is inherent to the infiniteness of the source of radiation combined with the distribution of particle directions over the entire 4π -steradian space. To overcome this difficulty, two solutions can be foreseen. First, one can restrict the source of radiation to a finite space around the object. Second, one can choose a direction that ultimately orients the particle in the object’s direction. Both solutions seem easy to implement at first glance, but they generate a problem. The restriction of either the space or the direction needs to be taken into account within the larger geometry of the object in order to preserve the homogeneity and the isotropy of the radiation field.

Another solution is to simulate the infiniteness of the source of particles by allowing them to start from the surface of a sphere that surrounds the object. However, unless the object itself is a sphere concentric to the surrounding source sphere, the particle direction cannot be chosen uniformly throughout 4π -steradian space. Doing this when the object is not at the center of the sphere would result in anisotropy of the radiation field since more particles would strike the object on its side closest to the inner surface of the source sphere. Here, the solid angle within which the object is seen from a point source located on the surface of the source sphere depends on the source location.

To balance the contributions of all point sources evenly, one should simulate a unidirectional field of particles (obviously oriented toward the center of the sphere) from every point on the surface of the source sphere. To do this, one can initiate these particles uniformly over the plane tangent to the sphere at that point. The simulation is therefore equivalent to an object immersed within an infinite number of homogeneous unidirectional fields having a uniform distribution of directions. Of course, if one wants the technique to be efficient, one also needs to restrict the sources to finite surfaces. To do so, one can limit each surface source to a disk that is the orthogonal projection of the sphere on that plane. Every particle leaving such a plane from outside the disk would inevitably miss the object.

Consequently, a general method to maintain the radiation field as both homogeneous and isotropic would be to combine both the restriction on space (a spherical surface source) and the restriction on direction (particles directed toward the center of the source sphere). The general algorithm can be described in two steps:

- Uniformly choosing points on the surface of a sphere that surrounds the object to be irradiated
- Simulating unidirectional fields of particles originating from disks tangent to the sphere at these points.

The only parameter of this technique is the radius of the surrounding source sphere. This radius has to be large enough so that the sphere encloses the entire object. It also needs to be small enough to maintain good efficiency of the technique. The visual description of the technique is shown in [Figure 5-1](#). The cube inside the sphere represents the irradiated object. The point P is chosen randomly on the surface of the sphere and the disk is tangent to the sphere at P . The diameter of the disk is equal to the diameter of the sphere. The grid represents the limit of the unidirectional field originating from the disk. The algorithm is now described mathematically.

Choosing a Random Point P on the Surface of the Sphere

A cubical object is located at the origin of the Cartesian coordinate system (O, x, y, z) as represented in [Figure 5-2](#). While the shape and the location of the object can vary, its location should be centered at the origin in order to optimize the efficiency of the technique. The radius R of the source sphere is chosen as small as possible but large enough so that the sample is fully enclosed inside the source sphere. The situation shown in [Figure 5-2](#) is not ideal as the source sphere should be much smaller than shown. The schematic is drawn as such only for comprehension of the problem.

Choosing a point $P (x_P, y_P, z_P)$ on the surface of the source sphere can be done by randomly choosing a direction $\bar{\Omega}$ (equivalent to (\mathbf{q}, \mathbf{j}) in a spherical coordinate system) and building the vector \overrightarrow{OP} so that it is oriented along this direction. The direction $\bar{\Omega}$ is uniformly distributed within the 4π -steradian space. This is equivalent to saying that

$\cos(\mathbf{q})$ is uniformly distributed between -1 and $+1$ and that \mathbf{j} is uniformly distributed between 0 and 2π . Therefore, if Rn_1 and Rn_2 are two random numbers uniformly distributed between 0 and 1 , the direction is given by

$$\vec{\Omega}: \begin{cases} \mathbf{q} = \arccos(1 - 2Rn_1) \\ \mathbf{j} = 2\pi Rn_2 \end{cases} . \quad (5-2)$$

The point P is then determined as

$$P: \begin{cases} x_p = R \sin \mathbf{q} \cos \mathbf{j} \\ y_p = R \sin \mathbf{q} \sin \mathbf{j} \\ z_p = R \cos \mathbf{q} \end{cases} . \quad (5-3)$$

Choosing a Random Point P' on the Surface of the Disk

The next step is to find a point P' on the source disk tangent to the source sphere at P . This can be done in three steps:

- Find a local Cartesian coordinate system (P, a, b) of the plane that contains the disk
- Find the point P' $(a_{P'}, b_{P'})$ in this local coordinate system
- Find the coordinates of P' $(x_{P'}, y_{P'}, z_{P'})$ in the initial Cartesian coordinate system using vector calculus to do the transformation.

The radius of the disk is R since it is the projection of the sphere on the plane tangent to the sphere at P .

Local Coordinate System of the Plane

One needs to find the coordinates, in the system $(\vec{x}, \vec{y}, \vec{z})$ of two vectors \vec{a} and \vec{b} that will constitute the local coordinate system (\vec{a}, \vec{b}) of the plane. Such vectors \vec{a} (x_a, y_a, z_a) and \vec{b} (x_b, y_b, z_b) must satisfy the five following conditions:

- \vec{a} belongs to the source plane, therefore it is perpendicular to \vec{OP} since the source plane is tangent to the source sphere at P

- \vec{b} also belongs to the source plane, and therefore it is also perpendicular to \overrightarrow{OP}
- \vec{a} is a unit vector
- \vec{b} is also a unit vector
- \vec{a} and \vec{b} are perpendicular.

This mathematically gives

$$\left\{ \begin{array}{l} \vec{a} \perp \overrightarrow{OP} \\ \vec{b} \perp \overrightarrow{OP} \\ \|\vec{a}\| = 1 \\ \|\vec{b}\| = 1 \\ \vec{a} \perp \vec{b} \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} x_a x_P + y_a y_P + z_a z_P = 0 \\ x_b x_P + y_b y_P + z_b z_P = 0 \\ x_a^2 + y_a^2 + z_a^2 = 1 \\ x_b^2 + y_b^2 + z_b^2 = 1 \\ x_a x_b + y_a y_b + z_a z_b = 0 \end{array} \right. . \quad (5-4)$$

This system of five equations has six unknown parameters and permits one degree of freedom for the choice of the system. This makes sense since the vector base (\vec{a}, \vec{b}) can be chosen with the vector \vec{a} having any direction around the point P , as long as both vectors remain perpendicular. To simplify the calculation, let us assume $z_a = 0$. This makes \vec{a} in the plane (\vec{x}, \vec{y}) . This explains the strange orientation of the axes a and b in

Figure 5-2. With this assumption, Equation (5-4) becomes

$$\left\{ \begin{array}{l} x_a x_P + y_a y_P = 0 \\ x_b x_P + y_b y_P + z_b z_P = 0 \\ x_a^2 + y_a^2 = 1 \\ x_b^2 + y_b^2 + z_b^2 = 1 \\ x_a x_b + y_a y_b = 0 \end{array} \right. . \quad (5-5)$$

Using Equation (5-3) within (5-5) leads to

$$\begin{cases}
x_a \sin \mathbf{q} \cos \mathbf{j} + y_a \sin \mathbf{q} \sin \mathbf{j} = 0 & \text{(I)} \\
x_b \sin \mathbf{q} \cos \mathbf{j} + y_b \sin \mathbf{q} \sin \mathbf{j} + z_b \cos \mathbf{q} = 0 & \text{(II)} \\
x_a^2 + y_a^2 = 1 & \text{(III)} \\
x_b^2 + y_b^2 + z_b^2 = 1 & \text{(IV)} \\
x_a x_b + y_a y_b = 0 & \text{(V)}
\end{cases} \quad (5-6)$$

(I) and (III) are used to find the coordinates of \vec{a} . Then (II), (IV), and (V) are used to find the coordinates of \vec{b} . The result is

$$\vec{a}: \begin{cases} x_a = \sin \mathbf{j} \\ y_a = -\cos \mathbf{j} \\ z_a = 0 \end{cases} \quad \text{and} \quad \vec{b}: \begin{cases} x_b = \cos \mathbf{j} \cos \mathbf{q} \\ y_b = \sin \mathbf{j} \cos \mathbf{q} \\ z_b = -\sin \mathbf{q} \end{cases} \quad (5-7)$$

Point P' in the Local Coordinate System

Because of the circular geometry of the problem, the polar coordinate system ($P, \mathbf{r}_P, \mathbf{j}_P$) is better adapted for its solution. The point P' is uniformly distributed within the surface of the disk centered at P . This is equivalent to saying that \mathbf{r}_P^2 is uniformly distributed between 0 and R^2 , and that \mathbf{j} is uniformly distributed between 0 and 2π . Therefore, if Rn_3 and Rn_4 are two random numbers uniformly distributed between 0 and 1, the location of the point P' on the disk is given by

$$P': \begin{cases} \mathbf{r}_P = R\sqrt{Rn_3} \\ \mathbf{j}_P = 2\pi Rn_4 \end{cases} \quad (5-8)$$

in the local polar coordinate system. The point P' is then determined in the local Cartesian coordinate system by

$$P': \begin{cases} a_{P'} = \mathbf{r}_P \cos \mathbf{j}_P \\ b_{P'} = \mathbf{r}_P \sin \mathbf{j}_P \end{cases} \quad (5-9)$$

Coordinates of P' in the Initial Coordinate System

One can use the vector relationship

$$\overrightarrow{OP'} = \overrightarrow{OP} + \overrightarrow{PP'} \Leftrightarrow \overrightarrow{OP'} = \overrightarrow{OP} + a_{p'}\vec{a} + b_{p'}\vec{b}. \quad (5-10)$$

All elements of Equation (5-10) are now known in the initial coordinate system.

Therefore, by substituting Equations (5-3), (5-7) and (5-9) into Equation (5-10), one can calculate P' ($x_{P'}$, $y_{P'}$, $z_{P'}$). Finally,

$$P': \begin{cases} x_{P'} = R \sin \mathbf{q} \cos \mathbf{j} + \mathbf{r}_P (\sin \mathbf{j} \cos \mathbf{j}_P + \cos \mathbf{q} \cos \mathbf{j} \sin \mathbf{j}_P) \\ y_{P'} = R \sin \mathbf{q} \sin \mathbf{j} - \mathbf{r}_P (\cos \mathbf{j} \cos \mathbf{j}_P - \cos \mathbf{q} \sin \mathbf{j} \sin \mathbf{j}_P) \\ z_{P'} = R \cos \mathbf{q} - \mathbf{r}_P \sin \mathbf{q} \sin \mathbf{j}_P \end{cases} \quad (5-11)$$

In Equation (5-11), R is a parameter of each specific problem and \mathbf{q} , \mathbf{j} , \mathbf{r}_P , and \mathbf{j}_P are calculated using Equation (5-2), Equation (5-8), and 4 random numbers.

Direction of the Particle

As explained previously, the direction of the particle is determined by the vector $\overrightarrow{\Omega}_{P'} = -\overrightarrow{\Omega}$. This vector is the direction of the unidirectional field. The trajectory of the particle is shown in Figure 5-2 by the long arrow that starts from the point P' and goes through the object cube. This arrow has four different parts. The second part of the arrow, as a thick dashed line, represents the trajectory of the particle across the object. The third part of the arrow, as a thin dashed line, represents the trajectory of the particle behind the object. Using $\mathbf{q}_{P'}$ and $\mathbf{j}_{P'}$ to characterize the direction in the spherical coordinate system, one can deduce

$$\overrightarrow{\Omega}_{P'} : \begin{cases} \mathbf{q}_{P'} = \mathbf{p} - \mathbf{q} \\ \mathbf{j}_{P'} = \mathbf{p} + \mathbf{j} \end{cases} \quad (5-12)$$

In computer-simulation problems, directions are better represented by their direction cosines $\cos \mathbf{a}$, $\cos \mathbf{b}$, and $\cos \mathbf{g}$ with the relationship

$$(\cos \mathbf{a})^2 + (\cos \mathbf{b})^2 + (\cos \mathbf{g})^2 = 1. \quad (5-13)$$

Calling $U_{P'}$, $V_{P'}$, and $W_{P'}$ the direction cosines of the trajectory of the particle, one can also deduce

$$\overrightarrow{\Omega}_{P'} : \begin{cases} U_{P'} = -\sin \mathbf{q} \cos \mathbf{j} \\ V_{P'} = -\sin \mathbf{q} \sin \mathbf{j} \\ W_{P'} = -\cos \mathbf{q} \end{cases} . \quad (5-14)$$

Implementation of the Algorithm

A general algorithm to simulate the randomness of a homogeneous and isotropic infinite 3D field of radiation when it interacts with a fixed object has been derived. It can be used in two different ways. The first one is the direct application of [Equations \(5-11\)](#) and [\(5-14\)](#) for as many particles as are needed to obtain the desired accuracy of the Monte-Carlo simulation using the algorithm. The algorithm is structured as shown:

Algorithm 1

get the radius R of the sphere

get the number of particles

initialize the random number generator

for each particle do

 get 4 random numbers Rn_1 , Rn_2 , Rn_3 , and Rn_4

 calculate \mathbf{q} and \mathbf{j} using [Equation \(5-2\)](#)

 calculate \mathbf{r}_P and \mathbf{j}_P using [Equation \(5-8\)](#)

 calculate the starting position of the particle using [Equation \(5-11\)](#)

calculate the direction of the particle using [Equation \(5-12\)](#) or [\(5-14\)](#)

treat the interaction of the particle with the object

end for

end Algorithm 1

This algorithm includes only one loop, whereas the general description foresees two random distributions: one for the choice of points P on the surface of the source sphere, and one for the choice of points P' on the surface of each source disk associated to each point P . This is why one may prefer a second algorithm that better simulates the idea of an infinite number of homogeneous unidirectional fields having a uniformly distributed direction around the sphere, as described previously. This second algorithm has two loops: one to select points P , and a second to select points P' over each disk. In some applications that need large numbers of particle histories, this second solution can also save CPU time since a part of the calculation depends only on the point P . The algorithm is given as follows:

Algorithm 2

get the radius R of the sphere

get the number of disks around the sphere

get the number of particles per disk

initialize the random number generator

for each disk do

get 2 random numbers Rn_1 and Rn_2

calculate \mathbf{q} and \mathbf{j} using [Equation \(5-2\)](#)

calculate the direction of the particle using [Equation \(5-12\)](#) or [\(5-14\)](#)

```

for each particle of the disk do
    get 2 random numbers  $Rn_3$  and  $Rn_4$ 
    calculate  $\mathbf{r}_p$  and  $\mathbf{j}_p$  using Equation (5-8)
    calculate the starting position of the particle using Equation (5-11)
    treat the interaction of the particle with the object
end for
end for
end Algorithm 2

```

One can notice that, by setting the number of particles per disk to unity, Algorithm 2 becomes equivalent to Algorithm 1. Algorithm 2 is less random than Algorithm 1, since the direction of the particles is chosen less often in Algorithm 2 than in Algorithm 1. Consequently, Algorithm 1 may be preferred for short series (thousands) of particles, whereas Algorithm 2 may be preferred for long series (millions or even billions of source particles).

Application to Chord-Length Distributions

In bone-marrow dosimetry, chord-length distributions through trabecular bone samples are an important issue (Jokisch et al. 1998; Jokisch et al. 2001b). The present algorithm will be a helpful tool to measure these distributions. To test the validity of the algorithm, it was first used to measure chord-length distributions through simple geometrical objects. The theoretical chord-length distributions through these simple objects are known and can be compared to the measured distributions. Two objects are tested in this study: a sphere and a cube. The theoretical distribution through a sphere of unit diameter is given as (Jokisch et al. 2001b):

$$F(L) = 2L. \quad (5-15)$$

Through a cube of unit width, the distribution is given as (Coleman 1969):

$$f(L) = \begin{cases} \frac{8L^3 - 3L^4}{3pL^3} & \text{for } 0 < L \leq 1 \\ \frac{6p + 6L^4 - 1 - 8(2L^2 + 1)\sqrt{L^2 - 1}}{3pL^3} & \text{for } 1 < L \leq \sqrt{2} \\ \frac{6p - 3L^4 - 5 + 8(L^2 + 1)\sqrt{L^2 - 2} - 24\arctan(\sqrt{L^2 - 2})}{3pL^3} & \text{for } \sqrt{2} < L \leq \sqrt{3} \end{cases} \quad (5-16)$$

The chord-length distributions of both a cube of width 1 cm and a sphere of diameter 1 cm were measured using Algorithm 2. For each object, two measurements were performed: one with 10 rays per disk and 1000 disks (10,000 rays total), and a second with 10 rays per disk and 10,000 disks (100,000 rays total). For each measurement, the chord lengths are recorded within a histogram with a bin-width of 0.01 cm. For these examples, the random number generator provided by Solaris 8 was used. It is a linear congruential algorithm using 48-bit integer arithmetic and is sufficient for generating sequences of random numbers on the order of millions.

Results and Discussion

General Algorithms

Both Algorithm 1 and Algorithm 2 have been implemented in the C language. Their codes are shown in Figures 5-3 and 5-4, respectively. For both programs the reader needs to implement three functions. The function `InitRandomGenerator` is needed for most random generators to provide a seed to the random number series. The function `GetRandomNumber` returns the next random number in the series. The function `TreatInteraction` is supposed to be the main function of the Monte-Carlo application and treats the interaction of a single particle with the irradiated object. The three functions are

not shown in [Figures 5-3](#) and [5-4](#) since they largely depend on the compiler used and the Monte-Carlo application.

Note that the technique does not guarantee that the particle will strike the object since the algorithm is independent of the object's shape. Therefore, the function `TreatInteraction` must take into account this eventuality. Note also that the technique assumes that the object is not itself a source of radiation. The particles are coming from a radiation field surrounding the object.

The constants defined for the radius of the sphere and the number of iterations can be implemented as parameters of the main program using the `argc` and `argv` variables of C. This has not been implemented in [Figures 5-3](#) and [5-4](#) in order to reduce the size of the listings.

Chord-Length Distribution Examples

Algorithm 2 was used to measure the chord-length distributions through a sphere and a cube. [Figure 5-5](#) shows the results for the sphere and [Figure 5-6](#) for the cube. In both figures, the real distribution corresponds to [Equations \(5-15\)](#) and [\(5-16\)](#), respectively. For each object, two measured distributions are shown: one with 10,000 rays fired through the object, and one with 100,000 rays fired. One can see that the distributions conform to the theoretical predictions. With 10,000 rays, the fluctuations are large, but the shape of the distributions is well preserved. With 100,000 rays, the fluctuations become smaller and the curves approach their predicted values. In the case of the cube, a loss of isotropy of the field would give a different distribution. Therefore, [Figure 5-6](#) confirms both the isotropy and the homogeneity of the simulated field.

Conclusion

Both Algorithm 1 and Algorithm 2 provide an efficient method to simulate a 3D isotropic and homogeneous field of radiation in which an object is located. The technique proposed in this chapter insures the isotropy and the homogeneity of the field and simulates the infiniteness of its source. Both algorithms can be easily implemented in any computer language, using the random number generator provided by the language compiler, or provided by the user. They can be adapted for any Monte-Carlo application by the use of the function `TreatInteraction` as explained in the previous section. Algorithm 1 is better adapted for short series of particle simulations, whereas Algorithm 2 will be preferred for longer series of particle histories. In previous studies of bone-marrow dosimetry, chord-length distributions are used to describe the path lengths of electrons emitted within the skeletal tissues ([Jokisch et al. 2001a](#); [Jokisch et al. 1998](#); [Jokisch et al. 2001b](#)). In current studies, Algorithm 2 will be used to measure chord-length distributions through the marrow cavities of trabecular bone samples by simulating an isotropic radiation field through 3D digital images of bone samples acquired via NMR microscopy.

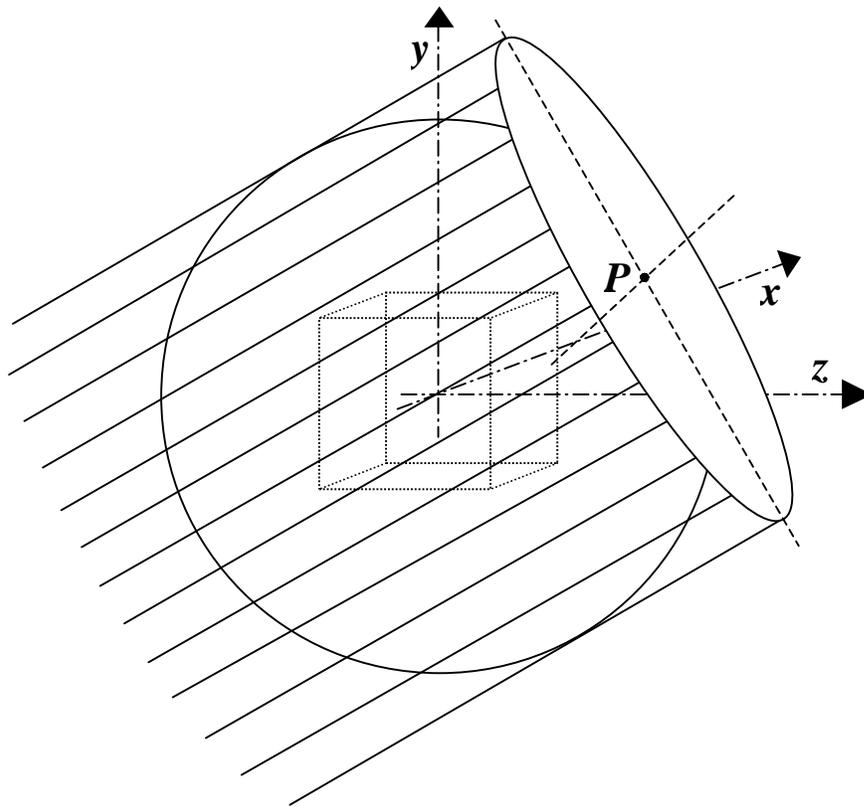


Figure 5-1. Technique used to simulate a homogeneous and isotropic infinite 3D field of radiation when it interacts with a fixed object. The cube represents the object. A point P is randomly chosen on the surface of the source sphere. A unidirectional radiation field is then simulated from the source disk tangent to the source sphere at P .

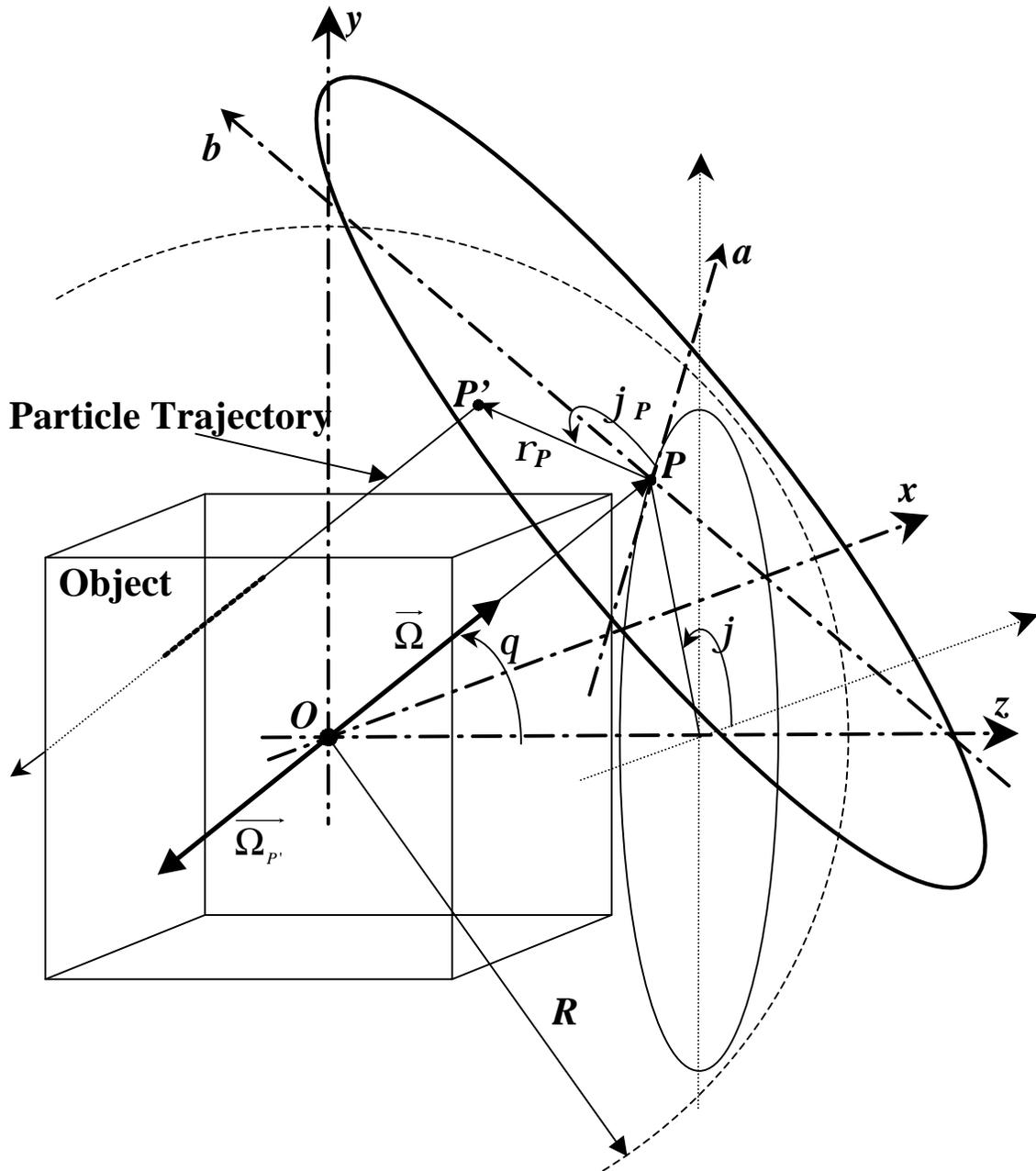


Figure 5-2. Geometric construction used to derive the trajectory of the particles. (O, x, y, z) is the absolute Cartesian reference. (P, a, b) is the Cartesian reference local to the plane of the disk. \mathbf{q} and \mathbf{j} define the location of the point P on the surface of the source sphere. \mathbf{r}_P and \mathbf{j}_P define the location of the point P' on the surface of the source disk tangent to the source sphere at P .

```

/*****
/*
/*   Algorithm1.c
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Parameters of the main program */
#define SPHERE_RADIUS      5.0
#define NB_PART            1000000

/* local functions to be implemented by the reader */
void InitRandomGenerator();
double GetRandomNumber();
void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W);

main()
{
    int PartNumber;
    double Rn1, Rn2, Rn3, Rn4;
    double Theta, Phi;
    double RhoP, PhiP;
    double XPPrime, YPPrime, ZPPrime;
    double UPPrime, VPPrime, WPPrime;

    /* get the radius R of the sphere */
    /* get the number of particles */
    /* initialize the random number generator */
    InitRandomGenerator();
    /* for each particle do */
    for ( PartNumber=0; PartNumber<NB_PART; PartNumber++ ) {
        /* get 4 random numbers Rn1, Rn2, Rn3, and Rn4 */
        Rn1 = GetRandomNumber();
        Rn2 = GetRandomNumber();
        Rn3 = GetRandomNumber();
        Rn4 = GetRandomNumber();
        /* calculate Theta and Phi using Eq. (2) */
        Theta = acos(1.0 - 2.0*Rn1);
        Phi = 2.0 * M_PI * Rn2;
        /* calculate RhoP and PhiP using Eq. (8) */
        RhoP = SPHERE_RADIUS * sqrt(Rn3);
        PhiP = 2.0 * M_PI * Rn4;
        /* calculate the starting position of the particle using Eq. (11) */
        XPPrime = SPHERE_RADIUS*sin(Theta)*cos(Phi)
            + RhoP*(sin(Phi)*cos(PhiP) + cos(Theta)*cos(Phi)*sin(PhiP));
        YPPrime = SPHERE_RADIUS*sin(Theta)*sin(Phi)
            - RhoP*(cos(Phi)*cos(PhiP) - cos(Theta)*sin(Phi)*sin(PhiP));
        ZPPrime = SPHERE_RADIUS*cos(Theta)
            - RhoP*sin(Theta)*sin(PhiP);
        /* calculate the direction of the particle using Eq. (14) */
        UPPrime = -sin(Theta) * cos(Phi);
        VPPrime = -sin(Theta) * sin(Phi);
        WPPrime = -cos(Theta);
        /* treat the interaction of the particle with the object */
        TreatInteraction(XPPrime, YPPrime, ZPPrime, UPPrime, VPPrime, WPPrime);
    } /* end for */
} /* end Algorithm 1 */

```

Figure 5-3. C program that implements Algorithm 1.

```

/*****
/*
/*   Algorithm2.c
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Parameters of the main program */
#define SPHERE_RADIUS      5.0
#define NB_PART_PER_DISK  100
#define NB_DISK            10000

/* local functions to be implemented by the reader */
void InitRandomGenerator();
double GetRandomNumber();
void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W);

main()
{
    int DiskNumber;
    int PartNumber;
    double Rn1, Rn2, Rn3, Rn4;
    double Theta, Phi;
    double RhoP, PhiP;
    double XPPrime, YPPrime, ZPPrime;
    double UPPrime, VPPrime, WPPrime;

    /* get the radius R of the sphere */
    /* get the number of disks around the sphere */
    /* get the number of particles per disk */
    /* initialize the random number generator */
    InitRandomGenerator();
    /* for each disk do */
    for ( DiskNumber=0; DiskNumber<NB_DISK; DiskNumber++ ) {
        /* get 2 random numbers Rn1 and Rn2 */
        Rn1 = GetRandomNumber();
        Rn2 = GetRandomNumber();
        /* calculate Theta and Phi using Eq. (2) */
        Theta = acos(1.0 - 2.0*Rn1);
        Phi = 2.0 * M_PI * Rn2;
        /* calculate the direction of the particle using Eq. (14) */
        UPPrime = -sin(Theta) * cos(Phi);
        VPPrime = -sin(Theta) * sin(Phi);
        WPPrime = -cos(Theta);
        /* for each particle of the disk do */
        for ( PartNumber=0; PartNumber<NB_PART_PER_DISK; PartNumber++ ) {
            /* get 2 random numbers Rn3 and Rn4 */
            Rn3 = GetRandomNumber();
            Rn4 = GetRandomNumber();
            /* calculate RhoP and PhiP using Eq. (8) */
            RhoP = SPHERE_RADIUS * sqrt(Rn3);
            PhiP = 2.0 * M_PI * Rn4;
            /* calculate the starting position of the particle using Eq. (11) */
            XPPrime = SPHERE_RADIUS*sin(Theta)*cos(Phi)
                + RhoP*(sin(Phi)*cos(PhiP) + cos(Theta)*cos(Phi)*sin(PhiP));
            YPPrime = SPHERE_RADIUS*sin(Theta)*sin(Phi)
                - RhoP*(cos(Phi)*cos(PhiP) - cos(Theta)*sin(Phi)*sin(PhiP));
            ZPPrime = SPHERE_RADIUS*cos(Theta)
                - RhoP*sin(Theta)*sin(PhiP);
            /* treat the interaction of the particle with the object */
            TreatInteraction(XPPrime, YPPrime, ZPPrime, UPPrime, VPPrime, WPPrime);
        } /* end for */
    } /* end for */
} /* end Algorithm 2 */

```

Figure 5-4. C program that implements Algorithm 2.

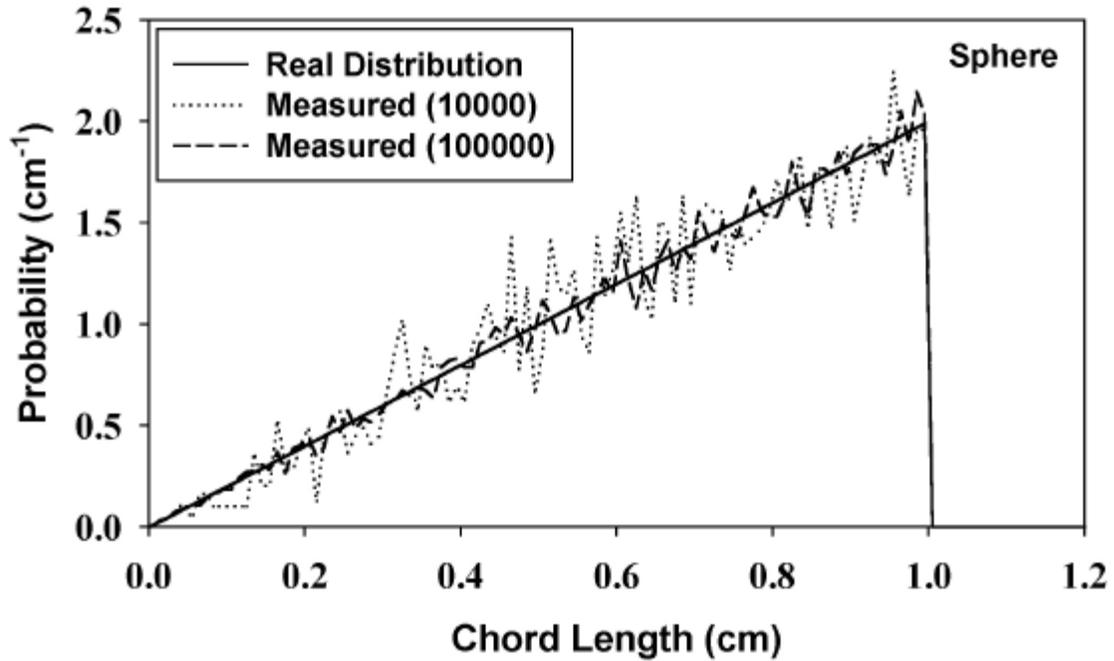


Figure 5-5. Chord-length distributions measured through a sphere of unit diameter. Two measurements were performed: one using 10,000 rays across the sphere, and a second using 100,000 rays. The measurements are compared with the mathematically derived distribution.

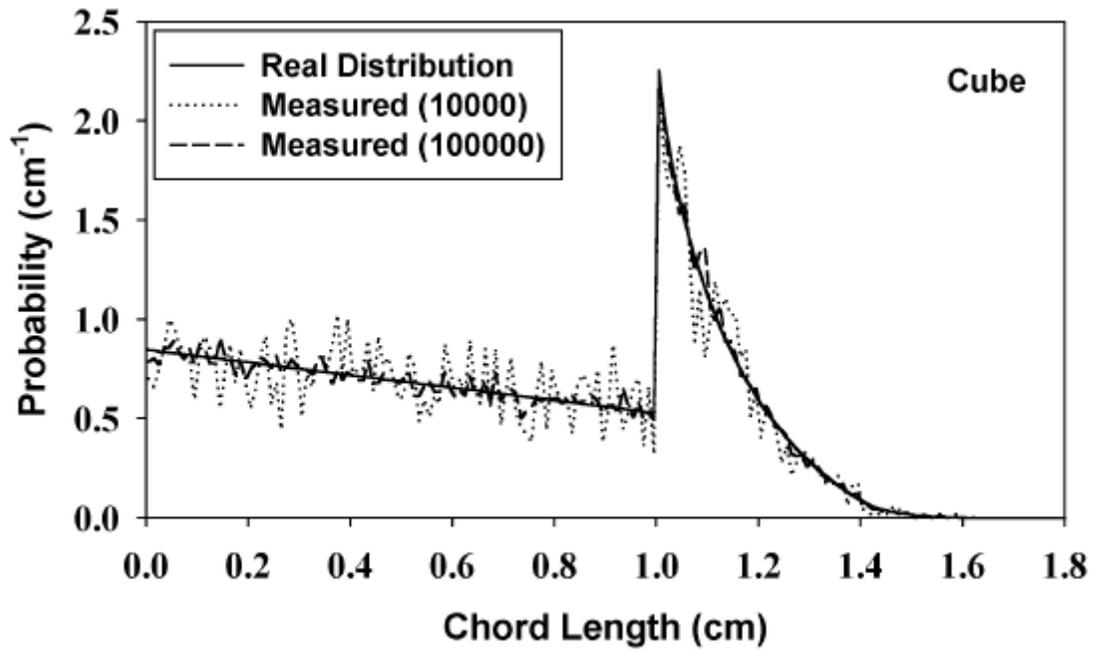


Figure 5-6. Chord-length distributions measured through a cube of unit width. Two measurements were performed: one using 10,000 rays across the cube, and a second using 100,000 rays. The measurements are compared with the mathematically derived distribution.

CHAPTER 6
VOXEL EFFECTS WITHIN DIGITAL IMAGES OF TRABECULAR BONE AND
THEIR CONSEQUENCES ON CHORD-LENGTH DISTRIBUTION
MEASUREMENTS¹

Introduction

The active bone marrow is widely recognized as one of the more radiosensitive tissues in the human body. As a result, the radiation dosimetry of bone marrow is an important component to radionuclide treatment planning (Sgouros 1993; Sgouros et al. 2000; Siegel et al. 1990). High-activity administrations thus demand interventional procedures, while low-activity administrations avoid marrow toxicity but at the potential cost of sub-optimal therapy. The accuracy of dosimetric models of bone marrow, as well as their correlations with biological effects, are thus of great clinical importance. In the adult, hematopoietically active bone marrow is associated with trabecular (or cancellous) bone. This bone type consists of a complex lattice of bone spicules that delimit marrow cavities. Figure 6-1 shows the microstructure of a small section of trabecular bone taken from a human femoral head. The image is a reconstruction of a nuclear magnetic resonance (NMR) microscopy image at a voxel resolution of 88 μm . The average thickness of the bone trabeculae is $\sim 300 \mu\text{m}$ and the average size of the marrow cavities is $\sim 1000 \mu\text{m}$.

¹ This chapter was published in *Physics in Medicine and Biology* in May 2002: Rajon DA, Jokisch DW, Patton PW, Shah AP, Watchman CJ, and Bolch WE. 2002. Voxel effects within images of trabecular bone and their consequences on chord-length distribution measurements. *Phys Med Biol* 47: 1741-1759.

Current models of bone-marrow dosimetry continue to use valuable 2D microstructural data acquired approximately thirty years ago at the University of Leeds. These models are based on chord-length distributions acquired directly within bone samples using optical scanning systems. At present, newer 3D techniques of high-resolution NMR microscopy have been applied to the study of trabecular microstructure and its associated radiation dosimetry. Chord-length distributions can be measured directly through these 3D images in order to provide new data for the dosimetric models. The goal of the present study is to better understand the effects of the image digitization and to analyze their consequences on chord-length distribution measurements as needed for skeletal dosimetry models.

Background

Chord-Length Distributions within Trabecular Bone Samples

Spiers (1966b; 1967; 1968) and colleagues at the University of Leeds conducted some of the very first studies on bone-marrow dosimetry. One of their primary achievements was the development of a dosimetry model based on measurements of chord-length distributions through trabecular regions within different skeletal sites (Beddoe 1976b; Beddoe 1977; Whitwell and Spiers 1976). These measurements are still the reference standard upon which all current-day skeletal dosimetry models are based (Bouchet et al. 2000; Eckerman and Stabin 2000).

Chord-length distributions are probability density functions that characterize the statistical distribution of chord lengths through an object. Chord lengths themselves are defined by the intersection of straight lines with the object's boundaries. There are different techniques of measuring these distributions depending on the origin and the

direction of the lines (Eckerman et al. 1985; Kellerer 1971; Spiers and Beddoe 1977). Even though no chord-length technique completely describes the behavior of electrons traveling within trabecular bone, mean-free-path randomness (or μ -randomness) has been used extensively as it is a good approximation for long-range particles and is relatively easy to measure. Consequently, the present study will refer only to the μ -randomness. Figure 6-2 represents an example of both a marrow cavity distribution and a bone trabecula distribution within a human cervical vertebra. The distributions were obtained from radiographs of thin slices of human bone samples through the use of an optical scanning system (Beddoe 1978; Darley 1968; Whitwell 1973). With careful selection of the sample orientation, omni-directional 3D chord distributions can be inferred from these 2D measurements (Beddoe et al. 1976).

Spiers' technique assumes that the electrons travel in straight lines within both the marrow cavities and the bone trabeculae, allowing a direction change at the bone-marrow and marrow-bone interfaces. In the model of Eckerman and Stabin (2000), this same approach is also applied. In the model of Bouchet et al. (1999b), 3D trajectories are permitted during electron transport in both bone and marrow.

Each of the models above continues to rely upon the Spiers chord-length distributions, as no comparable data are available in the literature. Recently, NMR microscopy of trabecular bone has been applied to the study of electron energy deposition in skeletal regions of active marrow. In this work, 3D NMR microscopy images are directly coupled to 3D radiation transport simulation codes (Jokisch et al. 2001a; Patton et al. 2002a). These same images, however, can additionally be used to acquire 3D chord-length distributions (Jokisch et al. 1998; Jokisch et al. 2001b) as needed for

chord-based dosimetry models of the skeleton (Bouchet et al. 1999b; Eckerman and Stabin 2000; Whitwell and Spiers 1976). For μ -randomness distributions, a Monte-Carlo technique is used to simulate an isotropic field of straight trajectories around the image, and the chords are recorded along these trajectories and used to build probability density distributions for both marrow and bone regions.

Voxel Effects on Chord-Length Distribution Measurements

The digital images used to measure the distributions are made of rectangular voxels. Consequently, the interface between bone and marrow is represented by a jagged surface that follows the shape of the voxels, which is in contrast to curved surfaces of the real structure. This inherent characteristic of digital images has two important consequences as explained by Jokisch et al. (2001b). First, a particle following a straight trajectory will travel through the bone and marrow regions by steps whose size are determined by the resolution of the image. Even though these steps are different depending on the direction of the trajectory, some chord lengths will have a higher probability to be observed during the random process. Jokisch et al. (2001b) termed this phenomenon “pixel (or voxel) effect A”. A second consequence is due to the 90-degree angles between the sides of the voxels. If a particle travels close and parallel to an interface in the real structure, it will remain on the same side of the interface. Within the digital image, however, the trajectory will alternatively be in bone and marrow, creating several very small chords. The result is an overestimation of the frequency of these short chords, which Jokisch et al. (2001b) denotes as “pixel (or voxel) effect “B”.

Figure 6-3 represents a 2D chord-length distribution as measured by Jokisch et al. (2001b) through one slice of a 3D image of a human thoracic vertebra. The 3D image

resolution was $59 \times 59 \times 78 \mu\text{m}^3$. Pixel effect “A” is responsible for the spikes that appear along the distribution at every voxel size, and pixel effect “B” is responsible for the overestimation of the frequency of the short chord lengths. The chord lengths were recorded through a 3D NMR image, but with all trajectories in the same plane perpendicular to one of the three axes. This choice was made to simplify the analytical study of the pixel effects. For a distribution recorded in 3D and with trajectories isotropically distributed around the sample, pixel effect “A” is expected to give broader spikes (Jokisch et al. 2001b), but pixel effect “B” will remain an important issue. In the present study, all distributions are acquired in 3D.

Minimum Acceptable Chord (MAC) Selection Criteria

Voxel effect “A” is not of real consequence since it can be removed by averaging the histogram. A bin-width equal to the largest resolution of the image smoothes the distribution and removes the presence of the distribution spikes (Jokisch et al. 2001b).

Voxel effect “B” is more pertinent since it creates short chords that do not exist in the real sample and ignores other longer chords that are present. An overall result is a shift of the entire distribution toward the shorter chords, and a significant reduction of the distribution mean. This is problematic in bone dosimetry, since the deposition of energy in either bone or marrow is based on the path length of the particle in that medium. Jokisch et al. (2001b) developed two methods to remove voxel effect “B”: Minimum Acceptable Chord (MAC) criteria 1 and 2. Both methods are based on the principle that a chord, in order to be recorded and added to the histogram, must be longer than some minimum value. MAC-1 and MAC-2 differ in the way this minimum chord is calculated. With MAC-1, the chord must be long enough to cross from one side of the voxel to the

opposite side in at least one dimension. With MAC-2, the chord must be long enough to cross from one side of the voxel to the opposite side in all three dimensions. The minimum chord in both methods depends on the direction of the trajectory and on the voxel size.

To test these techniques, Jokisch et al. (2001b) studied a voxelized spherical object. In this study, the exact chord-length distribution through a sphere, as well as the mean chord length were derived analytically and used as a reference to compare the different techniques. These analytical results are reproduced in the following equations:

$$f(l) = \frac{l}{2r^2}, \quad \text{and} \quad (6-1)$$

$$\bar{l} = \frac{4r}{3}. \quad (6-2)$$

In Equations (6-1) and (6-2), l represents the chord length and r is the radius of the sphere. The three-dimensional MAC-2 method was shown to give good results and this technique was recommended for future investigation in bone sample images.

Mathematical Model of Trabecular Bone

For dose calculation from a 3D image, one is interested in the minimum resolution required so that the image faithfully characterizes the trabecular microstructure. To this end, a mathematical model of trabecular bone was constructed and used as a benchmark for various calculations. This Model has been described in Chapter 3. The idea was the same as for the voxelized sphere of Jokisch et al. (i.e., does a calculation within an image of the model give the same result as an identical calculation within the model itself?). The mathematical model was built so that the chord-length distributions through the sample

were comparable to those measured by Spiers and colleagues within the cervical vertebral body.

The mathematical model is a $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ cube filled with spheres representing individual marrow cavities. The space between the spheres thus represents the bone trabeculae. The radius distribution for the spheres was chosen so that the entire sample satisfies the chord-length distribution criteria of Spiers' measurements. The sample was then segmented into images made of cubical voxels at varying image resolutions. The segmentation was intended to simulate a digital imaging system by calculating the volume fraction of marrow contained within each voxel. [Figure 6-4](#) displays a slice of the mathematical model seen at 4 different image resolutions. The resolutions given in [Figures 6-4-A to 6-4-D](#) are 200, 80, 40, and 16 μm , respectively.

The first use of the mathematical model was to assess the effect of the voxelization on the dose calculation when coupling the image to a Monte-Carlo radiation transport code (see [Chapter 3](#)). The study concluded that, for very low-energy electrons, some voxel effects could lead to an overestimation of the dose calculation for cross-region irradiation (e.g., bone source irradiating marrow). For some radionuclides, the overestimation in the cross-dose could be as high as 25%. A second study, based on single sphere models of marrow cavities and detailed in [Chapter 4](#), showed that this overestimation was a consequence of a systematic overestimation of the surface area of the bone-marrow interface. In the case of a single sphere, the surface area as measured by following the cubical shapes of the image voxels is 50% higher than the true surface area of the sphere. Furthermore, improving the image resolution would not reduce this effect, as it does not change the fact that the image is still composed of cubical voxels.

Material and Methods

The study described in this chapter was conducted in two parts. The first part explored the use of the MAC criteria of Jokisch et al. as applied to the mathematical model of trabecular bone developed in [Chapter 3](#). The second part extended the study of Jokisch et al. on chords across a single voxelized sphere to include a larger range of voxel sizes. Throughout the present study, only the marrow chord-length distributions are addressed. Because of the symmetry of the voxel effects, the consequences on the bone chord-length distributions are expected to be qualitatively similar.

To correctly measure a μ -random chord-length distribution through a 3D object, accurate techniques are needed to simulate an isotropic field of trajectories about that object. A general algorithm has been developed for this purpose and was described in [Chapter 5](#). The algorithm was used as the framework of all computer programs developed in this chapter.

Chord-Length Distributions through the Mathematical Model

The mathematical bone model was used to assess the accuracy of the MAC methods. First, the chord-length distribution was measured through the non-voxelized version of the model. Marrow chord lengths (intersections of the rays with the interior of the marrow spheres) were recorded and assembled into a normalized histogram representing the model's marrow chord-length distribution. This distribution served as the reference distribution for subsequent investigations.

To verify the resulting marrow chord-length distribution, an analytical chord-length distribution through a field of spheres was derived and compared with the measured distribution. To understand the comparison, a brief review of the mathematical

model is needed (see also [Chapter 3](#)). First, a 2.2 x 2.2 x 2.2 cm³ cube of osseous tissue was filled with 28,200 marrow spheres. The centers of each sphere were randomly chosen within the bone cube. The radius of each marrow sphere follows a specific distribution that insures the mathematical model to be representative of a true bone sample. Next, the mathematical model was cut inside this field of spheres to a size of 1.6 x 1.6 x 1.6 cm³. The above dimensions were chosen to ensure that the sample would be homogeneous, even to its edges. Finally, all spheres that do not overlap the final 1.6 x 1.6 x 1.6 cm³ cube were removed, as they are not part of the simulated sample. A total of 11,605 spheres remained within the model, permitting more efficient computer simulations.

A consequence of this construction is that some of the spheres are not fully contained inside the mathematical model and are cut by the edges of the cube. Therefore, the analytical expression of the chord-length distribution within the exact model is complex and will not be derived. On the other hand, the distribution within an infinite field of spheres (not cut by the edges of the cube), yet having the same radius distribution as in the mathematical bone model, can be derived as followed.

Consider an infinite 3D field of spheres with the same radius distribution as the mathematical bone model of [Chapter 3](#):

$$P(r) = P_m e^{-P_m r}, \quad (6-3)$$

where $P(r)$ is a probability density function with units of μm^{-1} , r is the radius of the sphere (in μm), and P_m is a parameter set to $0.0044 \mu\text{m}^{-1}$ for the mathematical bone model (see [Chapter 3](#)). This field of fixed spheres is inserted into an isotropic 3D field of rays that intersect the spheres. Because of the symmetry of a sphere and also because of the infiniteness of the field of spheres, one can assume the field of rays to be unidirectional

with a uniform fluence F (in μm^{-2}) instead of isotropic. Let l be a chord length (in μm) obtained by the intersection of a ray with a single sphere. The chord-length distribution within a single sphere of radius r is (Jokisch et al. 2001b)

$$f(r,l) = \begin{cases} \frac{l}{2r^2} & \text{if } l < 2r \\ 0 & \text{if } l > 2r \end{cases}, \quad (6-4)$$

where $f(r,l)$ has unit μm^{-1} . The number of rays that cross a sphere of radius r is obtained by multiplying the cross section of the sphere by the fluence of the field of rays. That is

$$N(r) = F \pi r^2. \quad (6-5)$$

Multiplying the number $N(r)$ by $f(r,l)$ gives the number of rays that cross a sphere of radius r and results in a chord length l .

$$N(r,l) = N(r)f(r,l) = \begin{cases} \frac{F \pi l}{2} & \text{if } l < 2r \\ 0 & \text{if } l > 2r \end{cases}. \quad (6-6)$$

$N(r,l)$ is the number of rays per μm of the chord length l . Its unit is μm^{-1} . $N(r,l)$ is defined for one specific sphere of radius r . To make it defined over the entire field of spheres, one needs to take into account the radius distribution $P(r)$ and to integrate over all radii. It becomes a function of l only:

$$N(l) = \int_0^\infty P(r)N(r,l)dr. \quad (6-7)$$

Since $N(r,l)$ is null for r such as $l > 2r$, Equation (6-7) becomes

$$N(l) = \int_{\frac{l}{2}}^\infty P(r)N(r,l)dr = \frac{F \pi l}{2} \int_{\frac{l}{2}}^\infty P_m e^{-P_m r} dr$$

$$\Leftrightarrow N(l) = \frac{FPl}{2} e^{-\frac{lP_m}{2}}. \quad (6-8)$$

In Equation (6-8), $N(l)$ represents the non-normalized density of chord lengths l within the entire field of spheres. To finally obtain the chord-length distribution, one needs to normalize $N(l)$ as

$$f(l) = \frac{N(l)}{\int_0^{\infty} N(l)dl}$$

$$\Leftrightarrow f(l) = \frac{lP_m^2}{4} e^{-\frac{lP_m}{2}}. \quad (6-9)$$

Equation (6-9) is the chord-length distribution within an infinite field of spheres having the radius distribution of the mathematical bone model. The mean chord length is

$$\bar{l} = \int_0^{\infty} lf(l)dl.$$

$$\Leftrightarrow \bar{l} = \frac{4}{P_m}. \quad (6-10)$$

For the mathematical model, with a value of $P_m = 0.0044 \mu\text{m}^{-1}$, the mean chord length is

$$\bar{l} = 909 \text{ } \mu\text{m}. \quad (6-11)$$

The distribution represented by Equation (6-9) has a mean chord length \bar{l} of 909 μm . This value of the mean marrow chord length can be compared with the value predicted by the Cauchy's theorem mentioned in Chapter 3; they are the same, and thus the derivation of Equation (6-9) is confirmed.

The expression in Equation (6-9) represents the chord-length distribution within the initial 28,200 spheres and cannot be used directly to check the reference distribution measured through the model. Instead, one can measure the chord-length distribution of

the initial field of 28,200 spheres. This second measurement uses essentially the same technique as the measurement through the final model; the difference is that all chords are recorded, whereas only the chords that start and end within the limit of the $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ cube are recorded in the case of the mathematical model. Consequently, a second computer program was created, almost identical to the first one. If the distribution recorded from this second measurement is found to be the same as the distribution given by [Equation \(6-9\)](#), one can assume that both programs are accurate as there is very little difference between them. For each program, a total of 5 million rays were fired through the field of marrow spheres. The chord lengths were recorded into a histogram having a bin-width of $2 \text{ }\mu\text{m}$ and a maximum chord length of $5000 \text{ }\mu\text{m}$.

The mathematical model was segmented into 19 3D digital images (see [Chapter 3](#)). For the current work, only 4 of these images were used. Their characteristics are summarized in [Table 6-1](#). The chord-length distributions were measured through these images using 4 different techniques referred to in the remainder of this chapter as PURE, GROUP, MAC-1, and MAC-2. The PURE technique corresponds to the normal measurement, with a fine bin-width of $2 \text{ }\mu\text{m}$, and without the use of the MAC techniques developed by Jokisch et al. It should show both voxel effects “A” and “B”. The GROUP technique uses a bin-width equal to the voxel size in order to eliminate voxel effect “A”. The MAC-1 and the MAC-2 techniques use a bin-width equal to the voxel size and the methods developed by Jokisch et al. to reduce voxel effect “B”.

A third computer program was constructed to perform these measurements. Columns 3, 4 and 5 of [Table 6-1](#) show, respectively, the number of rays used for PURE technique, the number of rays used for the GROUP, MAC-1, and MAC-2 techniques, and

the bin-width (equal to the voxel size) used for each technique. For the PURE technique, the bin-width is 2 μm for each image, as well as for the mathematical bone model. The numbers of rays were chosen in order to achieve reasonable accuracy, and to limit the computer simulation time.

Chord-Length Distributions through Single-Sphere Models

In their study, Jokisch et al. used a single-sphere model of radius 500 (arbitrary units). They next voxelized and segmented the model into an image with a voxel size of 10. In the present work, the same study was repeated with a 500- μm -radius sphere, but instead of using a unique voxel size, an entire range of image resolutions was considered. The effect of the MAC selection criteria (see Results and Discussion of this chapter) depends on the ratio between the voxel size and the size of the object. For electron dosimetry, a good way to characterize the size of an object is to use the mean chord length through this object. Therefore, each image of the 500- μm -radius sphere will be characterized by the ratio between the mean chord length and the voxel size. The mean chord length through a sphere is given by [Equation \(6-2\)](#). Therefore, the experiment in Jokisch et al. corresponds to a ratio of 66.7. For a typical bone sample, the mean chord length is on the order of 1000 μm . The most frequent resolution currently used to acquire a bone image at the University of Florida is 88 μm . The ratio is 11.36, which is far below the ratio used by Jokisch et al. To explore the effect of both the MAC-1 and the MAC-2 methods, the mean-chord-to-voxel-size ratio used in the current study ranged from ~ 4 to ~ 65 . The details of each segmented image are shown in [Table 6-2](#). The ratios used are shown in Column 3. The image having a voxel size of 10 μm corresponds to that used in Jokisch et al.

The three techniques, GROUP, MAC-1, and MAC-2, were used with a program almost identical to the one used for the segmented images of the mathematical model. The PURE technique was not used, since it does not give important information, except to show voxel effect “A”. Columns 4 and 5 of [Table 6-2](#) show, respectively, the number of rays used for the GROUP, MAC-1, and MAC-2 techniques and the bin-width (equal to the voxel size) used for each.

Results and Discussion

Mathematical Model of Trabecular Bone

The chord-length distribution for the marrow cavities throughout the mathematical bone model was measured and is shown in [Figure 6-5](#). In both [Figures 6-5-A](#) and [6-5-B](#), the theoretical distribution corresponds to the analytical expression given by [Equation \(6-9\)](#) and is for an infinite field of spheres, as explained previously. The measured distributions depend on how the measurement was performed. In [Figure 6-5-A](#), the distribution was measured through the exact sample, without those marrow spheres found outside the edges of the $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ bone cube. One can see that this distribution is slightly different than the theoretical one. In [Figure 6-5-B](#), however, the measured distribution was recorded through the original field of 28,200 spheres without the limitation in space. This later distribution perfectly matches the theoretical one, which explains why only the measured distribution is seen. As such, one can assume that the programs used to measure the distributions perform properly. As a consequence, the measured distribution shown in [Figure 6-5-A](#) is the reference chord-length distribution through the mathematical model and will be used to check the measurements through the images.

The chord-length distributions were then measured through the segmented images. A bin-width of 2 μm was first chosen for the PURE method, where neither of the two MAC criteria was applied. [Figure 6-6](#) shows a comparison between the measured distribution and the exact, or reference, distribution for the 4 voxel sizes shown in [Figure 6-4](#) (from 200 μm to 16 μm). Both voxel effect “A” and voxel effect “B” are seen in [Figure 6-6](#), as expected. Voxel effect “A” is mostly visible at large voxel sizes. For the smaller voxel sizes, the effect is hidden by statistical fluctuations and the thickness of the curve. Voxel effect “B” is distributed over the first voxel size. For resolutions at or below 80 μm , it generates a sharp peak at the beginning of the distribution. The mean chord lengths are listed in [Table 6-3](#) in the second column labeled ‘PURE’, and can be compared with the exact value. One can see in [Figure 6-4-A](#) that, at poor image resolution, most of the spheres are joined and the image will generate large chords that do not exist within the exact model. This situation compensates for the frequency increase of short chords generated by voxel effect “B”. Therefore, at 200 μm the mean chord length is close to the exact value; however, the shape of the curve clearly shows that the voxel effects alter the measured distribution. When the resolution is improved, the spheres are no longer connected to one another and voxel effect “B” underestimates the mean chord length by up to 35% at 16 μm .

To eliminate voxel effect “A”, the bin-width of the histogram is set to the voxel size of the image. To reduce voxel effect “B”, both MAC-1 and MAC-2 methods were used. The three different situations, called GROUP, MAC-1, and MAC-2 respectively, are compared with the exact distribution in [Figure 6-7](#). The mean chord lengths are presented in [Table 6-3](#) and are also shown in [Figure 6-7](#) in parenthesis within the figure legends.

Note that for these comparisons, the reference curves use the same bin-width as the corresponding image (columns 5 of [Table 6-1](#)). This explains the shape of the reference curves and also that the mean chord length is slightly different at poor resolution (200 μm) because of the averaging over the bins. The mean chord lengths listed under columns PURE and GROUP of [Table 6-3](#) are very close. Averaging the histogram over a larger bin-width removes voxel effect “A”, but it does not change the mean of the overall distribution that remains ~30% lower than the true value for all images with a resolution below 80 μm .

On the other hand, using either the MAC-1 or MAC-2 criteria removes some short chords and increases the mean. At 200 μm , the GROUP technique gives a mean that is the closest to the exact value. However, this is a consequence of both the poor resolution that increases the chord lengths (because of connections between the spheres) and voxel effect “B” that increases the frequency of the short chords. At this resolution, application of the MAC techniques reduces voxel effect “B” and the connections between the spheres become dominant. At lower resolution (80 μm and under), the GROUP technique gives a sharp peak at small chords: the same as the peak seen with the PURE technique in [Figure 6-6](#). The peak is not completely shown in [Figure 6-7](#) in order to better view the remainder of the plot. The MAC-1 technique reduces significantly the peak and is best suitable for a voxel size of 80 μm , for which the mean chord length is improved over those given by the GROUP method. But as the voxel size gets smaller, MAC-1 does not seem sufficient to remove the peak, and the mean becomes lower, down to 792 μm at 16 μm resolution. Therefore, the best improvement for the MAC-1 method is for a resolution between 80 μm and 40 μm .

The MAC-2 method is shown to remove the low-chord peak within each of the 4 images; however, it is additionally shown to also remove portions of the real distribution. The truncated portion is small at good resolution (16 μm) where the “Reference” curve is close to the MAC-2 curve, but is large at poor resolutions. Comparing the means, one can see that, as the voxel size is decreased from 200 μm to 16 μm , the mean decreases from 1551 μm to 920 μm . All these values overestimate the true mean chord length: 859 μm , but it appears that the mean would converge to the exact value for voxel sizes significantly smaller than 16 μm .

Single-Sphere Models

To better understand why the effectiveness of the MAC methods varies with the voxel size when applied to the mathematical bone model, the chord-length distributions through a series of segmented 3D images of a single sphere were measured using all three methods: GROUP, MAC-1, and MAC-2. [Figure 6-8](#) shows the comparison of the results with the theoretical distribution (the “reference” curve) given by [Equation \(6-1\)](#). Each plot represents a different voxel size. The number in parenthesis just under the voxel size is the ratio between the mean chord length and the voxel size. It ranges from 4.17 to 66.7. The mean chord lengths are presented in [Table 6-4](#) and are also shown in [Figure 6-8](#) within the figure legends.

The first interesting result is that the GROUP technique gives about the same mean for each ratio. Since the model is made of a single sphere, there is no increase of the chord length at poor resolutions, as was observed with the mathematical bone model because of the connection between spheres. For each ratio, voxel effect “B” introduces a

sharp peak at small chords (steeper at smaller voxel sizes) that reduces the mean chord length from 667 μm (the exact value) to about 450 μm .

Both the MAC-1 and MAC-2 techniques remove short chords introduced by voxel effect “B” and increase the mean chord length. MAC-1 removes, or at least reduces, the peak at large voxel sizes (small ratios) where it gives good results. But at large ratios, it is not sufficient to reduce the peak correctly, and the mean chord length remains low, around 610 μm . The best results are obtained at small ratios. With MAC-2, the mean chord length is gradually reduced from 825 to 678 μm when the ratio is increased from 4.17 to 66.7. Looking at the shape of the curves, one can see that, at 16 μm , a good match to the “Reference” curve is seen. However, as the ratio decreases, the shape of the curve worsens. Too many short chords are removed, and the technique tends to shift the distribution toward the larger chords. Consequently, MAC-2 gives good results for very large ratios (high resolution), but is too invasive at small ratios.

To compare these results with the mathematical model results, one needs to also use the ratio between the mean chord length of the model and the voxel size (numbers in parenthesis under the voxel size in [Figure 6-7](#)). In so doing, the MAC-1 technique is shown to be better for the mathematical model at a ratio of ~ 11 for 80 μm , while for the MAC-2 technique the higher the ratio, the better the results (at 16 μm , the ratio is ~ 54). These values are consistent with the values found for the single sphere model. Therefore, for a defined geometry of the object, the MAC selection criteria depend on the resolution of the image.

Both the MAC-1 and MAC-2 techniques remove the chords that are shorter than a Minimum Acceptable Chord. This Minimum Acceptable Chord, as explained previously

and detailed by Jokisch et al. (2001b), is a function of the voxel size and of the direction of the ray. With the MAC-1 technique, the Minimum Acceptable Chord is always shorter than the main diagonal of the voxel. With the MAC-2 technique, on the other hand, the chord must cross at least one voxel in any of the 3 dimensions in order to be recorded. Therefore, the Minimum Acceptable Chord can be very large if the direction of the ray is at a small angle from one of the three axes. As an example, one can imagine a ray fired along the x-axis and right at the center of a sphere. Such a ray would give a chord length equal to the diameter of the sphere. Since the ray is along the x-axis, it will never cross a voxel in the y and the z direction, and MAC-2 will remove this chord from the distribution even though it is not a consequence of voxel effect “B”. In this example, the ray was along the x-axis, but for a ray slightly tilted from the axis, the consequence would be the same. The maximum angle between the direction of the ray and the axis for which the chord will be removed depend on the voxel size. The smaller the voxel, the smaller the angle must be and, as a consequence, fewer chords would be removed by MAC-2. This explains why MAC-2 becomes more and more effective as the resolution is improved.

Conclusion

When recording chord-length distributions through 3D digital images, two voxel effects are identified. Voxel effect “A” is responsible for spikes that appear over the entire distribution, but is easily removed by using a histogram bin-width equal to the image resolution. Voxel effect “B” produces an increase in the frequency of short chords. This increase shifts the entire distribution and reduces significantly the mean chord length. Two Minimum-Acceptable-Chord methods, MAC-1 and MAC-2, were created to reduce voxel effect “B” (Jokisch et al. 2001b). It has been shown in the present chapter that MAC-1

does not completely remove the effect. Even though, at some resolution, the mean chord length can be close to the real value, the shape of the distribution remains different from the exact distribution. The peak seen with the GROUP technique (bin width equal to the voxel size) is reduced, but is still present in most of the plots in [Figures 6-7](#) and [6-8](#). The MAC-2 technique removes the peak completely, and is therefore seen as an improvement over the MAC-1 technique. This conclusion is similar to the conclusion made by Jokisch et al. ([2001b](#)).

It has been shown that the MAC-2 technique correctly removes the short chords introduced by voxel effect “B”, but it also removes some chords that are true features of the object. At larger voxel sizes, the problem becomes more significant. Consequently, the effectiveness of the MAC methods depends on the ratio between the mean chord length of the object and the voxel size. The larger this ratio, the better the method. The previous conclusion made by Jokisch et al. (that MAC-2 should be used) is only applicable to large mean chord-to-voxel-size ratios. If the object size (its mean chord length) is large compared to the voxel size, the undesirable effect of the MAC-2 method has little consequence on the overall distribution. However, when the ratio is small, the effect is the one shown at poor resolution in both the study of the mathematical bone model and of the single-sphere model. The distribution overestimates the frequency of the larger chords, and the mean chord length is also overestimated. The current resolution used for NMR bone imaging is $\sim 80 \mu\text{m}$. At this resolution, the MAC-2 method overestimates the mean chord length of the mathematical model by $\sim 30\%$ ([Figure 6-7](#)). Note that an overestimation of the mean chord length is a legitimate consequence of the image voxelization. All features of an object with a size of the same order as the voxel size

cannot be seen within the image. Therefore, the chords through these features should not be recorded within the distribution. As a consequence, the voxelization process of the imaging system is expected to slightly overestimate the mean chord length. However, the ~30% overestimate found after application of the MAC-2 method for a typical bone image (the ratio between the mean chord length and the voxel size is ~12) cannot be explained only by the voxelization process.

The main consequence of this study is that the impact of the voxel effects on chord-length distribution measurements are complex and one needs to be careful when trying to use digital techniques to represent bone samples. Spiers' research group performed their study with an optical scanning system using radiographic images (Darley 1968). When looking at Spiers results, (for example the ones in Figure 6-2), it appears that these distributions also show an increase in the frequency of short chords. Both curves in Figures 6-2-A and 6-2-B have a shape that resembles distributions affected by voxel effect "B". Furthermore, all theoretical probability distributions, that are based on smooth geometries that have been derived analytically, approach zero when the chord length goes to zero. This is true for a circle in 2D, for a sphere in 3D, and also for the mathematical trabecular bone model as shown by Equation (6-9) and in Figure 6-5. On the other hand, distributions that do not approach zero at zero chord lengths are the ones derived through a square or a cube (Jokisch et al. 2001b). Spiers' distributions do not approach zero and were probably affected by voxel effect "B" when they were measured. Consequently, there is currently no technique to measure chord-length distributions through trabecular bone that is not affected by voxel effect "B" to some extent. Current models of bone-marrow dosimetry are based on the Spiers' measurements that probably

underestimate the true mean chord length through the marrow cavities by the same amount as seen for the mathematical bone model: ~30% for both the PURE and GROUP techniques below 80 μm . At high electron energies, the mean chord length is an important parameter for bone-marrow dosimetry (Bouchet et al. 1999b; Jokisch et al. 2001b). As a consequence, a ~30% error on the mean chord length can have an impact on dose calculations currently performed using chord-based models.

The measurement of chord-length distributions is not the only problem associated with voxel effects within digital images. A previous study on the direct coupling of NMR images with Monte-Carlo transport codes was also confronted with voxel effects (see Chapters 3 and 4). The problem was related to a systematic overestimation of the surface area of spherical objects when they are digitalized with an imaging system. Even though the voxel effects reported in these studies are different than the ones related to chord-length distributions, they have the same origin. They are due to the use of rectangular voxels to represent smooth surfaces of objects. Improving the resolution will not eliminate the problem, since the voxels are still rectangular with 90-degree angles between their sides, and the surfaces will still appear jagged throughout the image.

Consequently, new methods are needed to better represent the bone microstructure acquired from a digital image. For chord-length distributions, an improvement of the MAC methods can be investigated in order to take into account the dependence of the method on the mean chord length-to-voxel-size ratio. However, this can be a complex study since the mean chord length can only be deduced from the distribution itself. Since the surface area is also affected by a voxel effect (Chapter 3 and 4), and because the object is not convex, the Cauchy's theorem cannot be used to deduce

the mean chord length from the surface area and volume. Furthermore, solving the chord-length distribution problem will not solve other problems related to voxel effects. Since all voxel effects have their origin in the rectangular shape of the voxels, another solution would be to change the current jagged bone-marrow interface into a smoother surface. Studies have already been conducted to construct polygonal surfaces from 3D digital images ([Lorensen and Cline 1987](#); [Muller et al. 1994](#)). The idea is to separate each voxel that belongs to the interface into two regions according to the gray-level values at neighboring voxels. Using this technique, the quality of the surface representation is improved with the resolution of the image. The surface become smoother since the angles between the different polygons increasingly approach 180 degrees. The polygonal surface can be used instead of the original image to measure the chord-length distributions. The voxel effects themselves would be reduced, and more accurate results would be achieved. New chord-length distributions could be measured and better dosimetric models could be derived from the resulting distributions. Finally, this technique will allow the acquisition of chord distributions when considering forms of randomness other than μ -randomness. Examples include “radial” or r-randomness that has been suggested to be more appropriate for simulations of beta particle emissions originating from within complex structures such as trabecular bone ([Spiers and Beddoe 1977](#)).

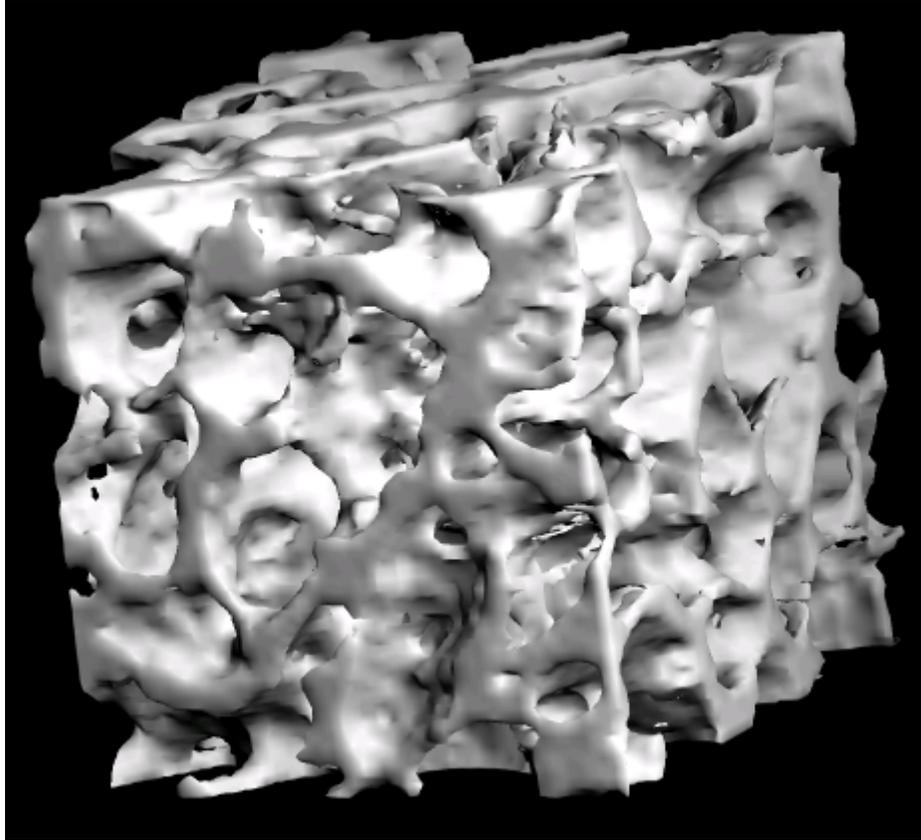


Figure 6-1. Cubical sample of trabecular bone reconstructed from a 3D NMR image obtained at 4.7 tesla. The image was taken over an 11 hour and 10 minute acquisition time (TR = 600 ms, TE = 9.1 ms, spectral width: 123,457 Hz, 2 averages, matrix: 512 x 256 x 256, field of view: 4.5 x 2.25 x 2.25 cm³). The sample was sectioned from the right femoral head of a 51-year male. The sample size is 5.6 x 5.6 x 5.6 mm³. The image resolution is 88 x 88 x 88 μm^3 .

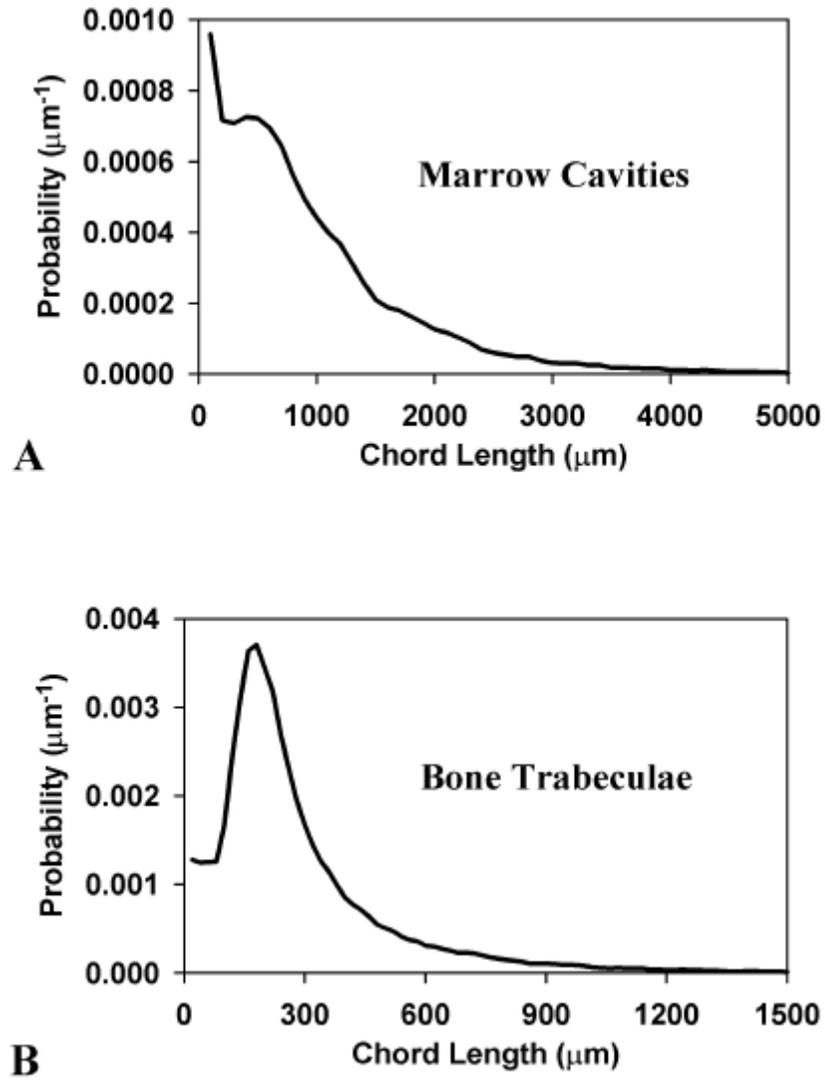


Figure 6-2. Chord-length distributions measured by Spiers and colleagues within a thin slice of a human cervical vertebra. Data from study by Whitwell (1973). A) Marrow cavity distribution. B) Bone trabecula distribution.

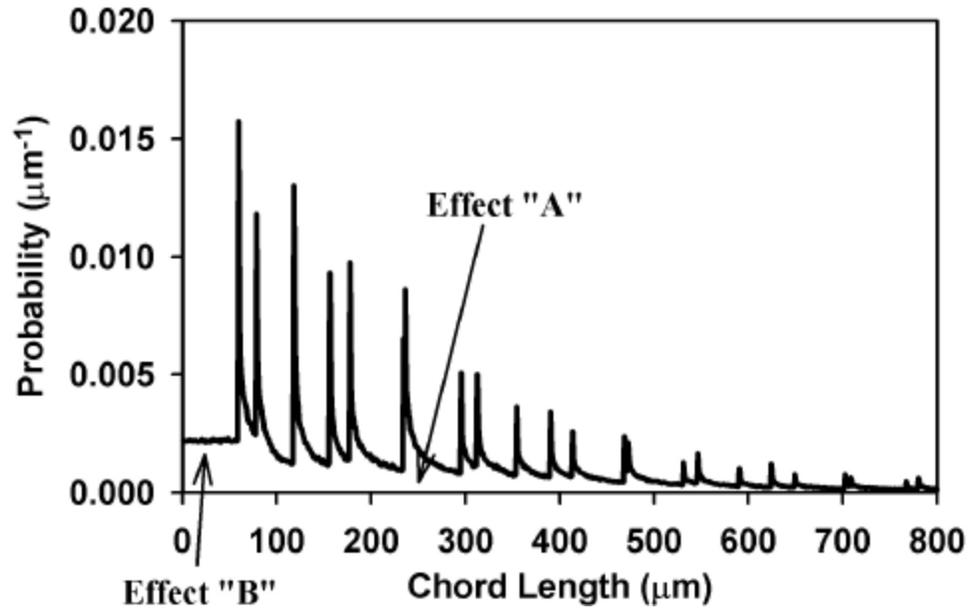


Figure 6-3. Bone trabecula chord-length distribution measured through a NMR image of a human sample of trabecular bone. The histogram was recorded over a 2D slice of the image. The bin-width used to record the histogram is 1 μm . The in-plane image resolution is 59 x 78 μm^2 . Both pixel effects "A" and "B" are shown.

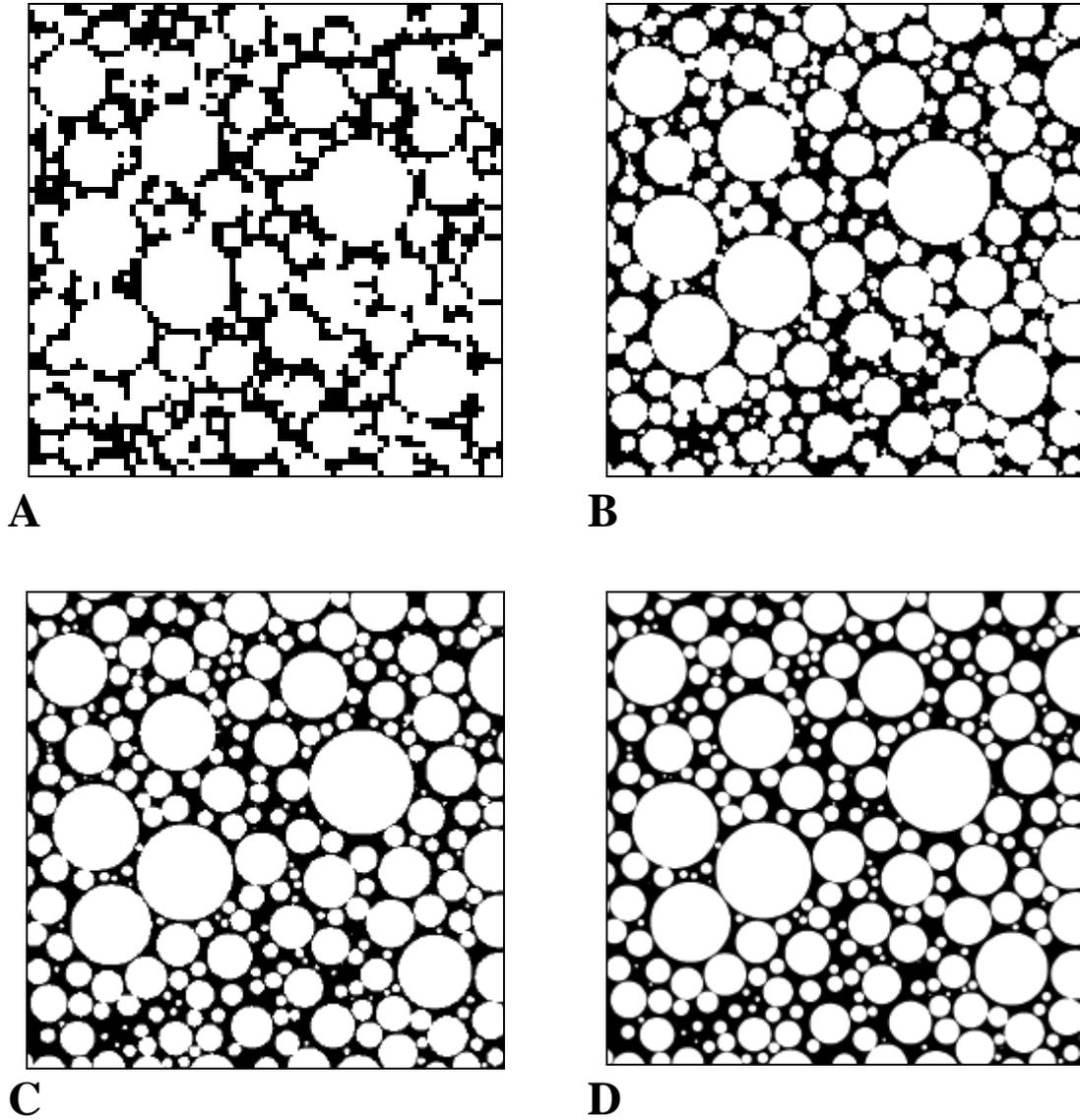


Figure 6-4. Segmentation of the mathematical model of trabecular bone. The 4 pictures simulate different image resolutions, but represent the same slice through the mathematical model. The voxel sizes are A) 200 μm , B) 80 μm , C) 40 μm , and D) 16 μm .

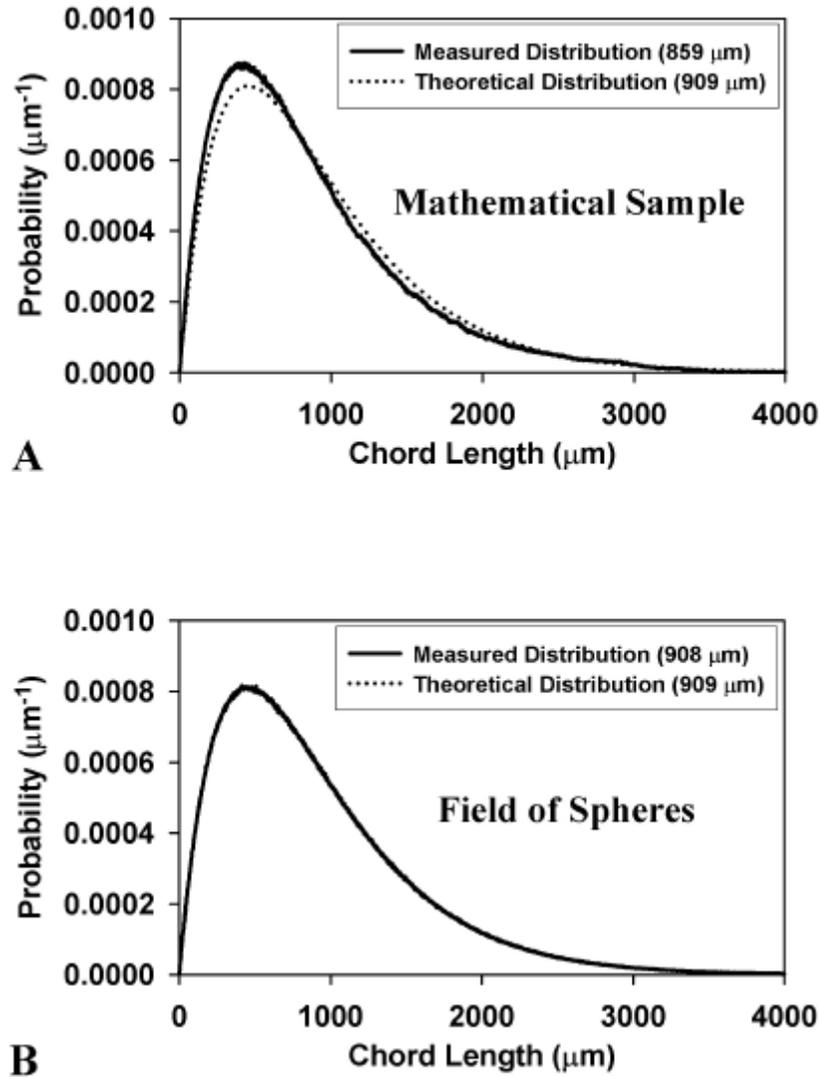


Figure 6-5. Comparison between the theoretical and the measured distribution through the mathematical model. The theoretical distribution is for an infinite field of spheres that has the same radius distribution as used to create the mathematical bone model. A) The measured distribution is recorded through the mathematical model. B) The measured distribution is recorded through a limited field of spheres that has the same radius distribution. These curves coincide perfectly on B). The numbers in parenthesis are the mean chord lengths for each distribution.

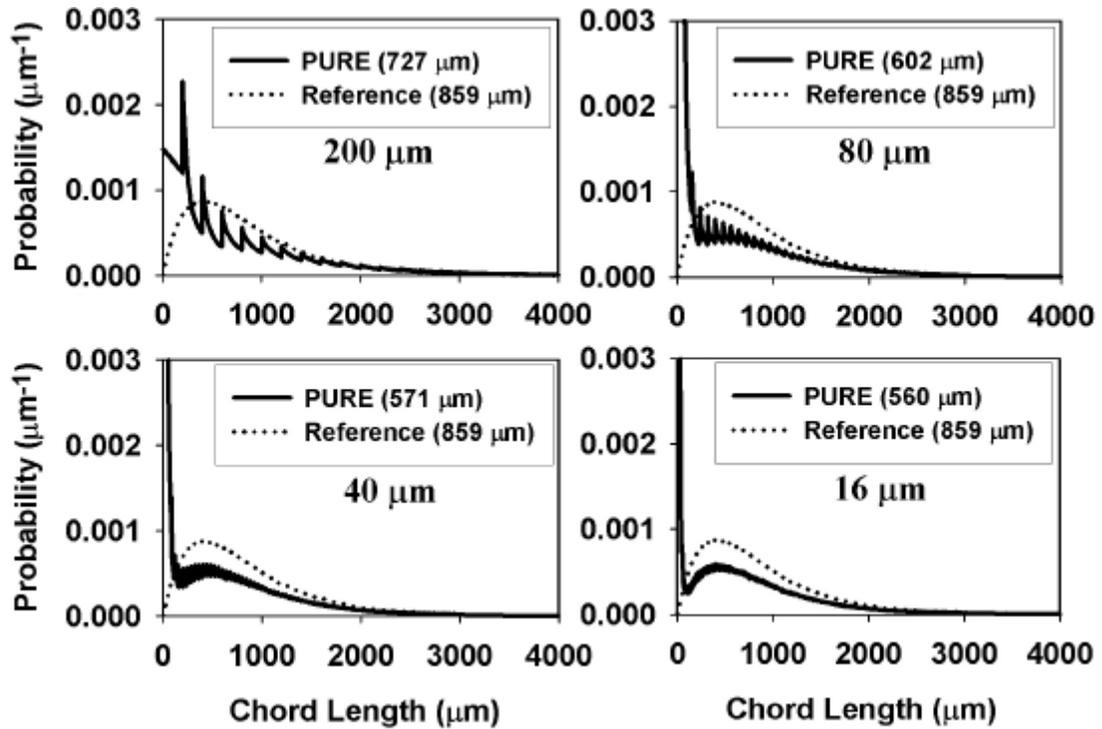


Figure 6-6. Three-dimensional chord-length distributions through segmented images of the mathematical model. The 4 resolutions represented in Figure 6-4 are used. Each distribution is obtained with a 2- μm -bin-width histogram and compared with the exact (“Reference”) distribution. The numbers in parenthesis are the mean chord lengths for each distribution.

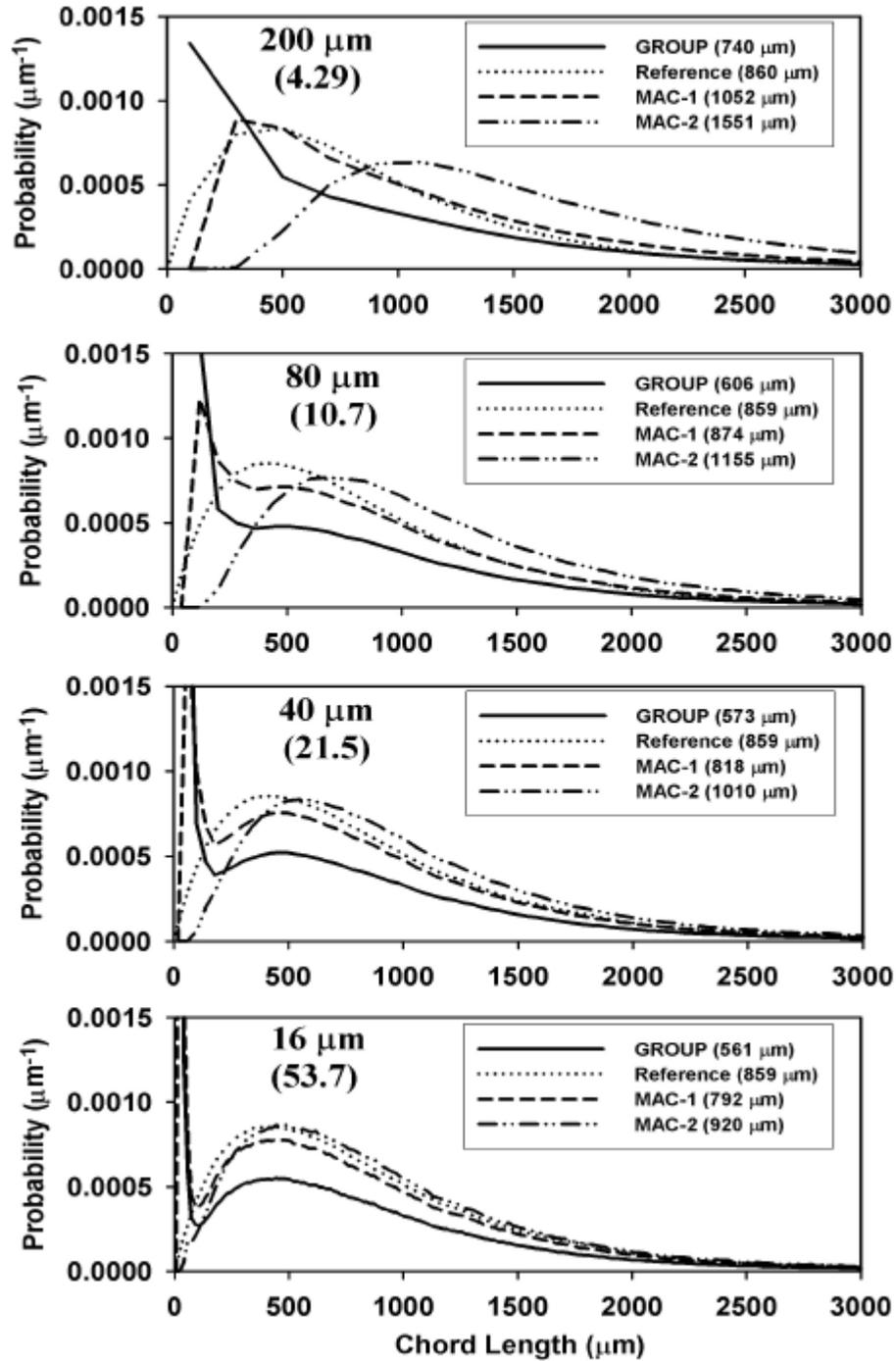


Figure 6-7. Distributions measured through the mathematical model using different techniques. The 4 resolutions represented in Figure 6-4 are used. The numbers in parenthesis inside the legend captions are the mean chord lengths for each distribution. The numbers in parenthesis under the resolution are the ratios between the mean chord lengths (of the reference) and the voxel sizes.

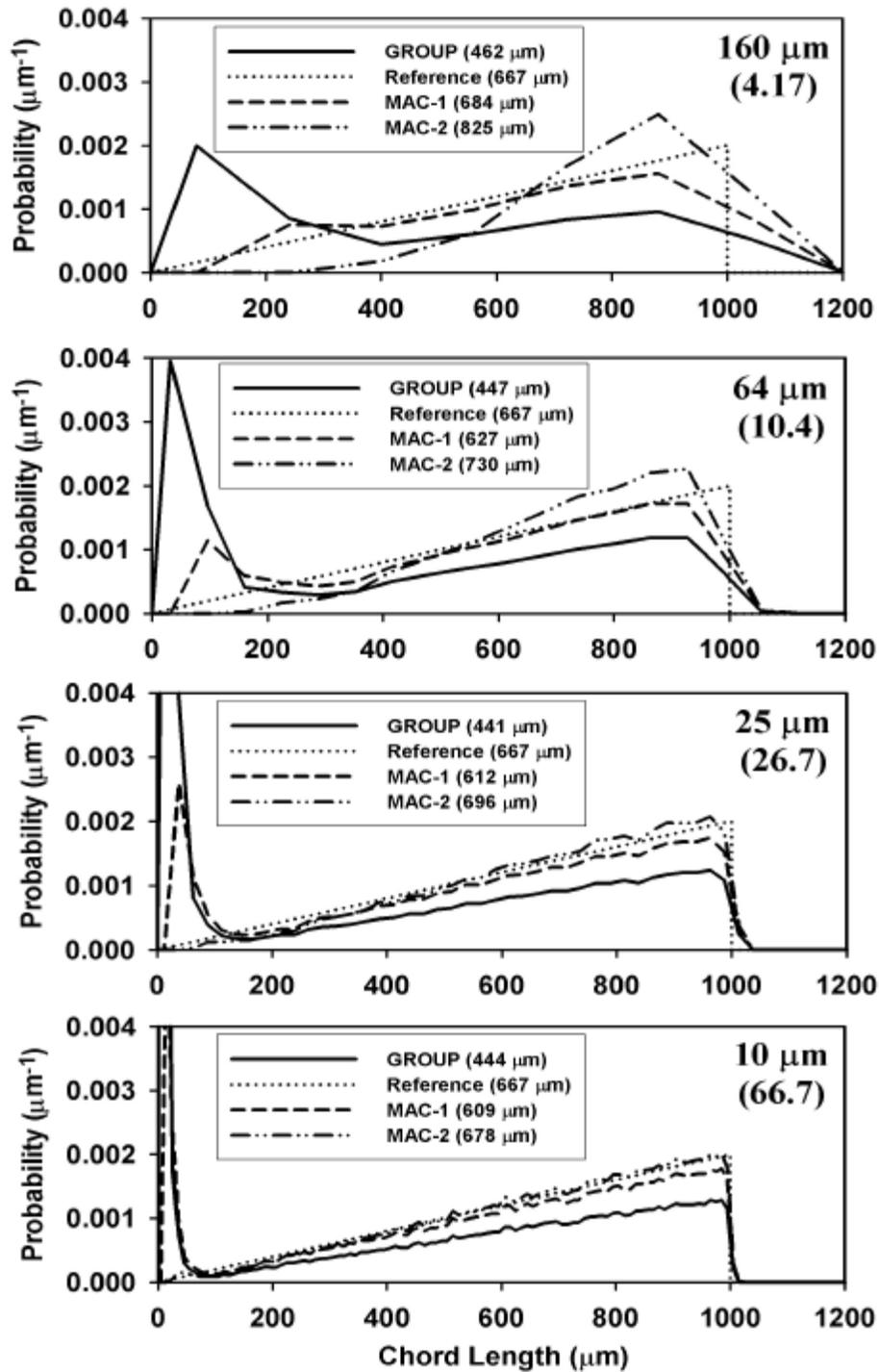


Figure 6-8. Distributions through a single sphere of radius $500 \mu\text{m}$ using different techniques. Four resolutions are showed. The numbers in parenthesis inside the legend captions are the mean chord lengths for each distribution. The numbers in parenthesis under the resolution are the ratios between the mean chord lengths (of the reference) and the voxel sizes.

Table 6-1. Characteristics of the 4 segmented images and of the mathematical trabecular bone model used in the chord-length distribution study. For the PURE technique, the bin-width of the histogram is 2 μm for both the reference mathematical bone model and the simulated digital images.

Voxel size (μm)	Voxels per dimension	Rays fired for the PURE technique (thousands)	Rays fired for the GROUP and MAC techniques (thousands)	Bin-width for the GROUP and MAC techniques (μm)
200	80	4000	100	200
80	200	2500	250	80
40	400	1500	300	40
16	1000	1000	500	16
Model		5000		

Table 6-2. Characteristics of the 4 segmented images of the single-sphere marrow cavity model used in the chord-length distribution study

Voxel size (μm)	Voxels per dimension	Ratio of mean chord/voxel size	Rays fired for the GROUP and MAC techniques (thousands)	Bin-width for the GROUP and MAC techniques (μm)
160	10	4.167	50	160
64	25	10.42	150	64
25	64	26.67	400	25
10	160	66.66	1000	10

Table 6-3. Mean chord lengths for the 4 segmented images and the mathematical bone model using the different techniques to record the chord-length distributions.

Voxel size (μm)	Mean Chord Length (μm)			
	PURE	GROUP	MAC-1	MAC-2
200	727	740	1052	1551
80	602	606	874	1155
40	571	573	818	1010
16	560	561	792	920
Model	859			

Table 6-4. Mean chord lengths for the 4 segmented images of the single-sphere marrow cavity model using the different techniques to record the chord-length distributions. The reference value for the sphere is 667 μm .

Voxel size (μm)	Ratio mean/voxel size	Mean Chord Length (μm)		
		GROUP	MAC-1	MAC-2
160	4.167	462	684	825
64	10.42	447	627	730
25	26.67	441	612	696
10	66.66	444	609	678

CHAPTER 7
MARCHING-CUBE ALGORITHM: REVIEW AND TRILINEAR-INTERPOLATION
ADAPTATION FOR IMAGE-BASED DOSIMETRIC MODELS¹

Introduction

Internal dosimetry plays an important role in the evaluation of risks associated with occupational exposures to radionuclides and normal organ toxicity during radionuclide therapies for cancer treatment. Current techniques for assessing radiation absorbed dose to internal organs can be traced to methods originally published in 1968 by the Society of Nuclear Medicine's Medical Internal Radiation Dose (MIRD) Committee (Loevinger and Berman 1968a; Loevinger and Berman 1968b). The method requires separate determinations of \tilde{A}_s (the integral number of nuclear decays that occur within source organs/tissues of the body) and the S value (the absorbed dose to the target organ/tissue per decay in the source) (Loevinger et al. 1991). S values, in turn, require assessments of the absorbed fraction, $f_{(s \leftarrow t)}$, defined as the fraction of emitted radiation energy in the source tissue that is deposited in the target tissue. To calculate values of absorbed fraction, current dosimetric techniques make use of Monte-Carlo radiation transport codes coupled with mathematical models of human internal organ anatomy.

Stylized mathematical models of the human body have traditionally been developed using 3D surface equations to delineate organ boundaries during radiation transport

¹ This chapter has been submitted to Computerized Medical Imaging and Graphics: Rajon DA and Bolch WE. submitted. Marching Cube algorithm: review and trilinear interpolation adaptation for image-based dosimetric models. Comp Med Imag Graph: submitted.

simulations ([Bouchet et al. 1999a](#); [Cristy 1980](#); [Cristy and Eckerman 1987](#); [Poston et al. 2002](#); [Stabin et al. 1995](#)). As these models have become more and more complex in their geometric descriptions, their coupling to Monte-Carlo codes has become more challenging. Furthermore, their relation to real patient anatomy is uncertain. Another challenging problem is to create models of microscopic anatomy. In the dosimetry of the bone marrow, the difficulty is to simulate the individual segments of bone trabeculae that surround the bone-marrow cavities. Early studies demonstrated that measurements of chord-length distributions across both the marrow cavities and the bone trabeculae could solve this problem ([Darley 1968](#)). Using these early measurements of chord-length distributions and the latest advances in Monte-Carlo radiation transport, it has been possible to calculate the deposition of energy within the marrow cavities for realistic distributions of radionuclides localized in skeletal tissues ([Beddoe 1976b](#); [Beddoe 1978](#); [Bouchet et al. in press](#); [Bouchet and Bolch 1999](#); [Whitwell and Spiers 1976](#)).

Current image techniques allow computed tomography (CT) or magnetic resonance imaging (MRI) to achieve images of human anatomy needed for radiation transport studies. In medical imaging, each voxel is assigned a gray-level that can be used either for visualization or for segmentation of the image into organs or tissues. Such segmented images are used as input models for energy deposition calculation instead of the current geometric models. The computer simulation is simplified since the calculation is reduced to the intersection of particle trajectories with rectangular voxels. This simplicity improves the calculation speed and the reliability of the codes. Furthermore, the image better represents the shape of the organs, since they are no longer approximated by equations. The technique has already been applied to an entire human body with excellent

results (Chao et al. 2001a; Chao et al. 2001b; Xu et al. 2000; Zankl and Wittmann 2001). Current imaging techniques additionally allow the acquisition of 3D visualization of the microstructure of trabecular bone within ex-vivo samples via either micro-computed tomography (microCT) (Muller et al. 1994; Ruegsegger et al. 1996) or nuclear magnetic resonance (NMR) microscopy (Jokisch et al. 1998; Majumdar et al. 1999; Patton et al. 2002a; Wehrli et al. 1998) at high resolution ($<100 \mu\text{m}$). Another application of 3D microimaging of trabecular bone is the measurement of chord-length distributions directly within the skeletal samples and thus to provide new data for the chord-length-based models of bone-marrow dosimetry (Jokisch 1997; Jokisch et al. 2001b).

The 3D-image-based technique eliminates both the complexity of the computer codes and the approximation of the organ shapes. However, the rectangular voxels have geometric consequences known as voxel effects. First, when chord-length distributions are measured through a digital image, repetitive spikes appear along the distributions, and an important overestimation of the short-chord frequency is observed (Jokisch et al. 2001b) (see also Chapter 6). Second, the voxelization process does not preserve the surface area of the boundary between two segmented tissues (even though it preserves the volume). This overestimate of the surface area has important consequences when the image is used with a Monte-Carlo transport code to calculate radiation energy deposition near the tissue interface (see Chapters 3 and 4). Some attempts have been proposed to reduce these effects, but their study using geometrical models (e.g., spheres and distributions of spheres) proved the task to be difficult (Jokisch et al. 2001b) (see also Chapter 6).

The conclusion of these previous studies was that a new technique was needed that would transform the voxel-based model into a surface-based model (see [Chapters 4 and 6](#)). Many algorithms have already been proposed for 3D image rendering. Cuberilles ([Chen and Herman 1985](#); [Herman and Liu 1979](#); [Herman and Udupa 1983](#)) and ray casting ([Farrell 1983](#); [Hohne and Bernstein 1986](#)) were the most frequently reported techniques during the early 1980s. In 1987, Lorensen and Cline ([1987](#)) proposed the Marching-Cube (MC) algorithm that has become a standard for isosurface generation. Several problems were discovered after it first appeared, and many adaptations have been proposed to improve the method. Van-Gelder and Wilhelms ([Van-Gelder and Wilhelms 1994](#)) have given a detailed mathematical description of many of these problems within the original algorithm.

The purpose of the present study is to propose two implementations of the MC algorithm for image-based dosimetry models. One will use the traditional MC algorithm with triangular surface generation. The second will calculate an isosurface based on the trilinear interpolation of the image gray-level. This later approach proposes a single surface equation per cube that simplifies the exploration of the image by a computer program. To introduce these techniques, we will first give a simplistic, but detailed, review of the MC algorithm. Next, our approaches will be described and tested using a mathematical model of a trabecular bone sample. Finally, we will discuss the adaptation of the techniques to anthropomorphic models that contain several isosurfaces.

Marching-Cube Algorithm

The MC algorithm is commonly divided into two steps:

- Construction of the isosurface corresponding to an isovalue of gray-levels

- Calculation of the gray-level gradient vectors along the isosurface.

The first step produces a set of triangles that forms the isosurface. The gradient vectors of the second step are used to produce Gouraud-shaded images that better display the perception of a 3D object on a 2D screen. To use the isosurface for computer calculation of radiation transport, only the first step is needed, and thus we will limit the present study to only the construction of the isosurface. Of course, the image rendering will be used for visualization purposes in the remainder of this text, but this task will be relegated to IDL 5.5 (RSI, 2001): a standard personal-computer software for medical image processing and display.

Original MC Algorithm

The study of a simple 2D case will serve to introduce the algorithm. [Figure 7-1-A](#) represents a small image in which each pixel is assigned a gray-level from 0 to 3. For this gray-level range, the isocontour will be defined by the isovalue 1.5. In [Figure 7-1-B](#), a grid, that delineates the Marching Squares, has been superimposed on the image. Each square has its vertices located at the center of four adjacent pixels of the initial image with each vertex assigned a black dot if the pixel gray-level value exceeds the isovalue. An edge joining two adjacent square vertices will be crossed by the isocontour if one of its two vertices, and only one, contains a dot. In [Figure 7-1-C](#), all such square edges are tagged with a small circle located at the center of the edge. Next, the circles are joined together by a set of segments that forms the isocontour. Finally, the location of each small circle is interpolated along the edge, using the gray-levels of the two edge vertices. The final result is represented in [Figure 7-1-D](#).

The algorithm proposed by Lorensen treats one square at a time. The four vertices of the square are used to decide whether or not the isocontour crosses the square, and to calculate the intersection points. Once a square is treated, the algorithm marches to the adjacent square until the entire image is covered. With four vertices that can be above or below the isovalue, 16 different square configurations are possible. The segmentation of these configurations is shown in [Figure 7-2-A](#). The configuration numbers (from 0 to 15) represent the binary state of the four vertices. One can notice that configurations 5 and 10 are crossed twice by the isocontour. A computer program can easily calculate the configuration number for each Marching Square and a look-up table is used to determine which edges will be joined to form the isocontour within the marching square. Next, the interpolation is performed along the edges to calculate the exact location of the intersection points of the segments. Of course, with the same edge being involved in two marching squares, the program can be optimized.

In 3D, the Marching Cubes are made of eight vertices located at the centers of eight adjacent voxels. Therefore, there are 256 possible cube configurations. Each configuration can be intersected up to four times by the isosurface. Each intersection is a polygon, whose vertices are interpolated along several of the 12 edges of the cube. The edges of a polygon are segments that lie within the faces of the cube in the same way that the segments of the 2D isocontour lay within the Marching Squares. Therefore, [Figure 7-2-A](#) also gives the 16 different ways a face of a Marching Cube will be intersected by the isosurface. Since each intersection point is interpolated independently, all polygons joining more than three intersection points are mostly non-planar. As a consequence, the polygons need to be divided into triangles to be handled in equation form by a computer

program using the image (e.g., for radiation transport simulations or chord-length measurements).

The triangulation of the 256 possible configurations is a tedious and error-prone task. Fortunately, a large number of symmetries can be found within Marching Cubes of the image. In the 2D situation of [Figure 7-2-A](#), configurations 1, 2, 4, and 8 can be derived from one another by basic rotations and then grouped into a single pattern numbered 1 as shown in [Figure 7-2-B](#). Furthermore, configurations having three or four dots are the binary complements of configurations with one or zero dots and thus there is no need to create new patterns for these configurations, as they will be segmented as their complements. Therefore, the study is reduced to the four patterns of [Figure 7-2-B](#). Note that patterns 2 and 3 include their own complements, since they have two dots each.

In 3D, Lorensen and Cline ([1987](#)) studied rotations and complementation in the cube, and proposed 15 patterns from which one can derive all 256 configurations. These patterns are presented in the four first rows of [Figure 7-3](#). Note that pattern 11 and 14 are symmetric by reflection, but Lorensen and Cline only considered rotations ([Lorensen and Cline 1987](#); [Van-Gelder and Wilhelms 1994](#)) and decided to treat them independently. The two last rows of [Figure 7-3](#) represent the complement patterns of patterns 0 to 7. They are not considered in Lorensen's algorithm since they are treated as pattern 0 to 7. Note that patterns 8-14 all have four dots and include configurations that are complement by pairs; consequently, they do not have a complement pattern. The entire study of the triangulation of the isosurface is hence reduced to the study of the 15 patterns.

Overall, the MC algorithm is easy to implement. Marching over the entire image, each cube is treated one at a time using only eight voxels from which is calculated a

configuration number. This number is used as an index to a look-up table that contains the number of triangles and the edges of the marching cube on which each triangle is connected. Finally, the gray-levels of the 8 voxels are used to interpolate each intersection point along its edge. The 256-configuration look-up table is the most critical and tedious part of the algorithm, but its building is facilitated by rotations and complementation. Another solution is to create a look-up table for the 15 patterns only, and let the computer program do the rotations and complementation within the cube. This technique, however, does not reduce the complexity of the algorithm since the rotations and complementation have to be treated anyway, and the program still needs a configuration look-up table to determine the pattern of each configuration. In addition, the computer code that performs these operations would be more complex, error prone, and slower than codes that rely only on look-up table values.

As one can see in [Figure 7-3](#), the same intersection point belongs to several triangles of the same marching cube. In addition, the same cube edge is shared between 4 adjacent cubes. As a result, the interpolation process can be optimized and the MC triangulation is often divided into two steps:

- Creation of a vertex table by calculating the three coordinates of each intersection point
- Creation of a triangle table that contains three pointers to three vertices of the first table.

Ambiguity Problem

In 1988, Düurst ([1988](#)) first reported that the MC algorithm might produce holes in the isosurface. These holes have their origin in what is referred to as “the ambiguity problem.” Let us consider pattern 3 of the 2D situation ([Figure 7-2-B](#)). Both

configurations 5 and 10 belong to this pattern and divide the square into two dotted regions that isolate a single non-dotted region. For both configurations, the surface-area segmentation within the square is 25% above the isovalue and 75% below, whereas the symmetry of pattern 3 would instead suggest a 50% - 50% repartition. Furthermore, why should pattern 3 always propose a segmentation that separates two dotted regions by one non-dotted region? This choice tends to unfairly reduce the size of the dotted regions. It should be possible to also propose a reverse geometry shown in [Figure 7-4-A](#).

The ambiguity for the choice between the direct and the reverse segmentation of a square can lead to a totally different result for images with small features. This problem was addressed by Zhou et al. (1994) and referred as the type A small feature problem. [Figure 7-4-B](#), illustrates the worst-case scenario: the image is alternately made of dotted and non-dotted voxels. The direct segmentation isolates small dotted regions within an infinite non-dotted field, whereas the reverse segmentation isolates small non-dotted regions within an infinite dotted field. The symmetry of the initial image suggests a symmetric segmented image with identical shapes for the dotted and non-dotted regions. In 2D, either alternatively or randomly choosing between the direct or the reverse segmentation for each square can solve the problem. The results are shown in [Figure 7-4-B](#). The alternate segmentation gives symmetrical stripes, preserves the surface-area, and proposes symmetric shapes. However, the result shows an anisotropy not suggested by the initial image. Consequently, the random segmentation would be preferred.

The example in [Figure 7-4-B](#) is an extreme case with perfect symmetry between the dotted and the non-dotted regions. In most images, this symmetry does not exist in the real structure, and both the alternate and the random segmentations are not suitable. The

decision between direct and reverse segmentation should be done independently for each square and using the four vertices of the square. Such a solution has been proposed by Nielson and Hamann (1991) and will be discussed later.

Hole Problem

In 3D, the ambiguity problem appears each time a face of a cube is ambiguous. In [Figure 7-3](#), one can see that six of the regular patterns and three of the complement patterns have ambiguous faces. In 2D, both the direct and the reverse segmentation can coexist within the same image because the choice between direct and reverse only changes the segments between the intersection points, but not their location along the edges. In 3D, an ambiguous face is shared by one cube that may use the direct segmentation and one cube that may use the reverse one. This situation would lead to intersecting segments that do not match properly. Therefore, in 3D an alternate or a random segmentation such as the one proposed in [Figure 7-4-B](#) couldn't produce a continuous isosurface.

Lorenson and Cline (1987) proposed to treat the complement configurations with the same pattern design. For example, in [Figure 7-3](#), pattern 3 and pattern 3c are designed with the same triangles. However, the bottom face of pattern 3 is of the direct type whereas the bottom face of pattern 3c is of the reverse type. When a pattern 3c is flipped over and put underneath a pattern 3, the result is the discontinuity shown on the left panel of [Figure 7-5-A](#). The problem appears each time a complement pattern is connected to a regular pattern by an ambiguous face. [Figure 7-5-A](#) also shows a pattern 10 on top of a pattern 6c (center) and a pattern 6 on top of a pattern 3c (right). The result of this unsolved ambiguity problem is a hole in the isosurface such as shown in [Figure 7-6-A](#). [Figure 7-6-A](#) is a 22 x 22 x 22 NMR microscopy image of a small segment of the

trabecular bone microstructure. The left image is the direct display of the triangles produced by the MC algorithm using the patterns of [Figure 7-3](#). On the right picture, the rendered image is shown as generated by IDL 5.5 using the triangle list.

Image Size and Processing Time

Another concern with the MC algorithm is the amount of data to be generated. Each intersection point needs three coordinates (e.g., twelve bytes on most computer systems). For each triangle, three integers are required to access the summits in the vertex table (again, twelve bytes are required). The image shown in [Figure 7-6-A](#) has 2,110 vertices and 4,214 triangles. That is 75,888 bytes required to store the image. The initial image with one byte per voxel occupies only 10,648 bytes. The MC algorithm has in effect multiplied the size of the image by a factor of seven. Of course, this multiplication factor would be reduced with a smaller voxel size since fewer cubes would be on the isosurface. However, the resolution used for [Figure 7-6](#) (between 4 and 10 times smaller than the smallest features of the object size) is a typical ratio to obtain a good definition of the object. Furthermore, the multiplication by seven has been observed on most trabecular bone images tested. Therefore, an ordinary 256 x 256 x 256 image would create about 3.3 million summits and 6.6 millions triangles. This represents more than 110 mega-bytes.

A second technical concern is the time required to create the MC image and also to process the MC image. The construction of the isosurface shown in [Figure 7-6](#) takes about 0.1 second on a 750 MHz Unix machine. The look-up table indexed by the configuration numbers provides a fast access to the designs and allows generating about 40,000 triangles per second. The 256 x 256 x 256 image can be treated in less than 3 minutes. The processing of a MC image is a greater concern. Being able to do fast

mathematical transformations with an image containing millions of triangles is a difficult challenge. One can see in [Figure 7-6](#) that many triangles are generated for the flat surface that represents the right edges of the sample (about 250 triangles). This surface could easily be modeled with less than 20 triangles. Treating the image one cube at a time reduces the complexity of the algorithm but leads to the impossibility of adapting the size of the triangles to the local geometry.

Marching Tetrahedrons

The simplest solution to solve the hole problem is to eliminate ambiguous faces. A tetrahedron has four vertices and only 16 possible configurations. To segment the triangular faces, a unique connection is required to join two edges. This extreme simplicity leads to only 3 patterns and 2 complement patterns. They are represented in [Figure 7-7-A](#). There is no ambiguous face and only 0, 1, or 2 triangles constitute the isosurface within the tetrahedron.

Many propositions have been made to break up the Marching Cubes into a set of tetrahedrons ([Bloomenthal 1988](#); [Chan and Purisima 1998](#); [Doi and Koide 1991](#); [Ning and Bloomenthal 1993](#); [Payne and Toga 1990](#); [Shirley and Tuckman 1990](#); [Treece et al. 1999](#); [Zhou et al. 1995](#)). [Figure 7-7-B](#) proposes 2 solutions to fit 5 tetrahedrons into a cube. One can see in [Figure 7-7-B](#) that two opposite faces of a cube have a different break up. Therefore, both breaking-up patterns proposed in [Figure 7-7-B](#) must be used alternatively to avoid any discontinuity within the isosurface. Furthermore, as a cube vertex can be owned by one or four tetrahedron of the cube, alternating the two segmentation methods within the same image will force a vertex to be owned by eight or by thirty-two tetrahedrons (one or four of each of the eight adjacent marching cubes). The consequence

is that an isolated dot would be surrounded by an isosurface with one of the two different topologies shown in [Figure 7-7-C](#) and the isosurface would not respect the homogeneity of the initial image.

In [Figure 7-7-D](#), the cube is broken up into six tetrahedrons. Since two opposite faces of the cube have the same design, the same break-up pattern can be applied to the entire image. No discontinuity would be found within the isosurface, but an isolated dot would be surrounded by an isosurface like the one shown in [Figure 7-7-E](#) and the isosurface would not respect the isotropy of the initial image.

Other solutions have been proposed to break up the Marching Cube into tetrahedrons so that the resulting isosurface respects the homogeneity and the isotropy of the initial image. They involve twenty-four or even forty-eight tetrahedrons per cube. These solutions tend to produce smoother isosurfaces than the initial MC algorithm and they solve the ambiguity problem. Nevertheless, they generate a large number of triangles. Therefore, one may prefer the extended MC algorithm described next.

Extended MC Algorithm

The hole problem is a consequence of the reverse design implicitly proposed to the ambiguous faces of the complement patterns 3c, 6c, or 7c, when they are treated as patterns 3, 6 or 7. To solve the problem, one should consider these complement patterns as independent patterns having their own segmentation that would give the ambiguous faces a direct design. This has been proposed by different research groups ([Baker 1989](#); [Herman and Udupa 1983](#); [Montani et al. 1994b](#); [Udupa and Ajjanagadde 1990](#); [Zhou et al. 1994](#)) and is shown in [Figure 7-8](#). All ambiguous faces of the 23 patterns have a direct segmentation and no hole will be created. [Figure 7-5-B](#) shows how these new patterns

connect for the three examples proposed in [Figure 7-5-A](#). [Figure 7-6-B](#) also shows how the hole of [Figure 7-6-A](#) is resolved.

The hole problem is solved, but the technique will systematically overestimate the volume of the objects below the isovalue, as explained previously. For that reason, one may think of a series of patterns that design all ambiguous faces with the reverse segmentation. This new series of patterns is proposed in [Figure 7-9](#). All faces of the 23 patterns have a reverse segmentation and the hole problem is also solved. [Figures 7-5-C](#) and [7-6-C](#) show how the holes are resolved with the reverse segmentation.

Lately, a new disambiguation method has been proposed by Delibasis et al. (2001). Instead of a look-up table, the algorithm evaluates the isosurface design for each cube. Visiting the cube edges that are crossed by the isosurface allows following the contour of the polygons. The technique reproduces all 23 patterns of the direct segmentation of [Figure 7-8](#). The authors did not address the reverse segmentation, but a slight modification of their algorithm would generate the reverse patterns of [Figure 7-9](#). Delibasis method eliminates the need for a look-up table, but necessitates a complex algorithm to follow the polygon edges within the cube. Such a complexity can be tedious and error prone. In addition, the resulting program will be slower than a direct access to a configuration look-up table. Furthermore, the method produces non-planar and non-triangulated polygons and only a program that can subdivide the polygons into triangles will be able to manipulate the image.

Facial Deciders

Solving the hole problem by the Extended MC algorithm does not solve the ambiguity problem. Why should we use the direct patterns of [Figure 7-8](#) instead of the

reverse patterns of [Figure 7-9](#)? In [Figure 7-6](#), one can see that the direct solution creates a slight hollow instead of the hole, whereas the reverse solution creates a slight protrusion. If the same choice is made for the entire image, the result can be a deleterious in the case of small objects. Furthermore, the two series of patterns proposed in [Figures 7-8](#) and [7-9](#) cannot be merged within the same image. For better image representation, the decision between direct and reverse segmentation should be made for each face and not at the marching cube level.

The average of the four gray-levels of an ambiguous face can be used to decide whether the entire face is more above or below the isovalue ([Wyvill et al. 1986](#)). If the average is below the isovalue, the direct segmentation should be used for the face. If it is above, the reverse segmentation should be used. Nielson and Hamann ([1991](#)) proposed another facial decider that is based on a bilinear interpolation of the gray-level over the ambiguous face. The interpolation gives a function $B(x,y)$ that represents the gray-level over the face. The gray-level is equal to the isovalue along a curve that is a hyperbola with two branches that isolate either the two dotted vertices or the non-dotted vertices. Nielson and Hamman showed that comparing the function $B(x,y)$ at the intersection of the hyperbola asymptotes with the isovalue will tell which edges of the face are connected by the hyperbola, thus allowing one to decide between direct or reverse segmentation.

The facial deciders may propose a different segmentation for two ambiguous faces within the same marching cube, but none of the patterns shown in [Figures 7-3](#), [7-8](#), or [7-9](#) proposes such a situation and new designs need to be created. A configuration with two ambiguous faces would have four different designs. Pattern 13, with six ambiguous faces, has $64 (2^6)$ possible designs. Nielson and Hamann ([1991](#)) showed that some designs could

not be resolved by using only the twelve edges of the marching cube. Instead, they proposed to add a vertex that is interior to the marching cube and whose location depends on the eight vertices of the cube. [Figure 7-10](#) shows the four triangulations proposed by Nielson and Hamann using this extra vertex for pattern 10.

The ambiguity problem is solved with both facial deciders, but they require extra configurations, and for most of them more triangles than with the extended MC algorithm. Once more, the problem of the MC algorithm is to stay at the cube level and the size of the triangles only depends on the voxel size whereas they should depend on the size of the local geometry.

Optimizing the Image Size

The MC algorithm produces triangles that can be very small. In [Figure 7-6](#), some triangles are barely visible and could be treated as a single pixel of the resulting image. This idea was developed by the Dividing-Cube technique ([Cline et al. 1988](#); [Crawford et al. 1988](#)). As a triangle is far more costly to project on a computer screen than a single point, each cube of the original MC algorithm can be divided into sub-cubes until the projection of the cubes are represented by single pixels of the image. Another wasteful consequence of the MC algorithm is the impossibility to adapt the size of the triangles to the object size. To overcome this constraint, Montani et al. ([Montani et al. 1994a](#); [Montani et al. 2000](#)) proposed the Discrete MC algorithm. The idea is to merge adjacent cubes that have coplanar polygons (before interpolation) into a unique cell to create larger polygons. Then the polygon vertices are interpolated along the edges of the new cell. A large flat surface as the one seen on the sides of the bone sample of [Figure 7-6](#) would be merged into fewer triangles. This technique was proved to reduce the number of triangles

by a factor 3 on common images (Montani et al. 2000). The Marching Triangles (Hartmann 1998; Hilton et al. 1996) is another method that generates a Delaunay triangulation by marching over the surface. It creates a mesh whose density does not depend on the voxel size. The resulting image also reduces the number of triangles by a factor 3 for common images.

These techniques that reduce the amount of triangles are important for rendering purposes, but they have a common drawback. A computer program that seeks to travel through the image (e.g., as in tracing the path of a radiation particle) needs to calculate distances to the next isosurface intersection from the current location. For this, it would be too costly to explore the entire triangle list of the image. Instead, the initial marching cube that contains the current location provides the isolation of a few triangles. The exploration is limited to only these triangles and the program can march to the next cube and continue the process. With the MC techniques that provide a list of triangle that are not related to the initial cube grid, this localization information is lost and the program would become slower. Therefore, our adaptation will focus on techniques that preserve the initial marching-cube grid.

Material and Methods

As mentioned earlier, our goal is to minimize the effects caused by the rectangular shape of the voxels within 3D images of trabecular bone samples as applied to skeletal dosimetry studies. The first objective is to preserve the surface area of the boundary between the bone trabeculae and the marrow cavities. The second is to eliminate the voxel effects when measuring chord-length distributions through these images. We want to apply a MC-like technique to create an improved interface between bone and marrow.

To test our methods, a mathematical model of trabecular bone sample was created and will be discussed first.

Mathematical Bone Sample

The mathematical bone sample used in the present study is based on a first model created for dosimetry calculation studies (see [Chapter 3](#)). This first model had two major drawbacks. First, the real area of the interface between bone and marrow could only be estimated through rough approximation. As we want to be able to assess how the MC techniques preserve the surface area, we need to know the exact value of this area through the model. Second, some of the simulated bone trabeculae were so thin that even at high image resolutions, connections were present between the spherical marrow cavities. Real samples of trabecular bone do not have extremely thin trabeculae (in comparison), and this artifact would make the surface area of the model more difficult to converge to its exact value.

To avoid these problems, we created another model made of a $1.2 \times 1.2 \times 1.2 \text{ cm}^3$ cube of osseous tissue filled with 3,400 spheres containing bone marrow (i.e., simulating the marrow cavities). The spheres have a normalized radius distribution described in [Chapter 3](#) and given by:

$$P(r) = P_m e^{-P_m r}. \quad (7-1)$$

In [Equation \(7-1\)](#), r is the radius of the spheres and $P_m = 44 \text{ cm}^{-1}$ is a parameter chosen so that the chord-length distribution within the entire sample is close to a typical marrow cavity chord-length distribution measured for the cervical vertebrae ([Whitwell 1973](#)). The locations of the spheres are randomly chosen within the cube but they must satisfy the three following conditions:

- No sphere can extend beyond the limits of the cube
- No sphere can overlap another sphere of the model
- A sphere must be at least at 5% its radius from any other sphere and from the cube edges.

The two first conditions allow the exact calculation of the surface area of the interface and the third condition preserves a minimum thickness between the spheres to avoid the artificial connections of marrow cavities mentioned above. A 2D cross section of the model is presented in [Figure 7-11](#). The exact surface area of the spheres has been calculated and is

$$S_{spheres} = 44.0916655 \text{ cm}^2. \quad (7-2)$$

The normalized chord-length distribution within the sample was derived in [Chapter 6](#).

With l representing a chord length across one of the spheres, this distribution is

$$f(l) = \frac{lP_m^2}{4} e^{-\frac{lP_m}{2}} \quad (7-3)$$

The mean of this distribution has also been calculated and is:

$$\bar{l} = \frac{4}{P_m} = 909 \mu\text{m}. \quad (7-4)$$

[Equations \(7-2\)](#), [\(7-3\)](#), and [\(7-4\)](#) will be used as a benchmark for the present study.

A simulation of an imaging technique was used to create artificial 3D images of the mathematical bone sample. For different resolutions, ranging from 1000 μm to 24 μm , a grid structure is placed over the sample to delimit the image voxels. A gray-level (from 0 to 255), that represents the volume fraction of the voxel that is inside the spheres, is then assigned to each voxel. For voxels partially overlapping one or several spheres, a Monte-Carlo technique using 20,000 points per voxel is used to assess the volume

fraction. The characteristics of the images are presented in [Table 7-1](#). Column 1 is the voxel size and column 2 is the image size. With a gray-level based on the volume fraction and ranging from 0 to 255, a natural isovalue to segment the image into bone and marrow regions is shown to be 127.5.

To quantify the improvement of the MC technique over the voxel technique (using the initial gray-level image), the surface area of the interface was measured within the segmented images. The measurement is simple: for each voxel, each time one of the six adjacent voxels is not on the same side of the isovalue, the area of the common face is added to the isosurface area. The final result is thus divided by two, as each cube face is counted twice. The cavity chord-length distributions were also measured within these voxel images for comparison with the MC techniques. The technique used is the same as described in [Chapters 5](#) and [6](#). The bin-width to build the histogram is 6 μm and 2 millions rays were fired around the sample to perform the measurements. These characteristics were the same for every chord-length distribution histogram built within the present chapter.

Direct and Reverse Extended MC Algorithm

For rendering purposes, the hole problem of the initial MC algorithm can be accepted to a certain extent. With a high-resolution image, the holes would be very small and would not alter the general perception of the object for a human viewer. For a computer program trying to find its way through the image, the crossing of a hole would make it unable to remember on what side of the isosurface it then travels. Therefore, holes are not acceptable for applications to radiation transport studies within the image. On the other hand, the reduction of the number of triangles is not the priority. We prefer

to keep the cubical grid generated by the MC algorithm so that the program can explore the cubes one at a time and limit its search of the intersection with the isosurface to the few triangles that belong only to the local marching cube. As a consequence, the extended MC algorithm described previously is what is needed for our applications. However, we want to try both the direct and the reverse segmentations and see if they produce a significant difference. The facial deciders would create more triangles within each cube, would slow down the program, and may not provide a significant improvement in the final results.

For image rendering, the MC algorithm generates two lists: a vertex list with three coordinates per vertex and a triangle list with three pointers to the vertex list per triangle. A synopsis of these two tables is presented in [Figure 7-12](#) along with two other tables that are described now. A program that wants to explore the image one marching cube at a time also needs to remember which triangles belong to which marching cube. Therefore, a list of the isocubes (cubes on the isosurface) is also required. For each isocube, a pointer (“1stTr” in [Figure 7-12](#)) to the first triangle of the cube as well as the number of triangles within the cube (“Tr” in [Figure 7-12](#)) are needed. Each cube also needs an identification number (“No” in [Figure 7-12](#)) that is calculated from its location (I, J, K) within the image. As the isocube list is sorted by identification number, a binary search will provide a fast access to the list. For the non-isocubes, one only needs to remember on which side of the isosurface they are located. Since an image with a high resolution would have long chains of such marching cubes, a look-up table with one entry per chain is sufficient. Each entry gives access to the identification number of the first cube of the chain (“No” in [Figure 7-12](#)), the number of cubes in the chain (“Nb” in [Figure 7-12](#)), and the status of the

chain: above or below the isovalue (“St” in [Figure 7-12](#)). A binary search also provides a fast access to the table. A program traveling one cube at a time within the image would need to calculate the cube identification number and access the isocube list to check if the cube is on the isosurface. If it is not, it can access the non-isocube list. If it is, the isocube provides access to the triangle list and then to the vertex list.

The direct and the reverse techniques are referred to as DIR and REV in the remainder of this text. They were applied to the series of images of the mathematical bone sample. The surface area of the isosurface was calculated by summing the areas of each triangle of the isosurface. The C++ program that creates the triangle list and calculates the surface area of the bone-marrow interface is listed in [Appendix E](#). It uses the look-up tables that are in [Appendix A](#). Three sets of look-up tables are provided. The first one (LOR) uses the initial Lorensen patterns and is only used in this dissertation to display [Figure 7-6-A](#) that shows the hole problem. The two others: (DIR) and (REV) implement the direct and the reverse technique respectively. The program of [Appendix E](#) also uses some of the C++ tools that are listed in [Appendices B, C, and D](#).

The cavity chord-length distributions were also measured for comparison with [Equations \(7-2\) and \(7-3\)](#). The C++ program that measures the chord-length distributions using the MC method is listed in [Appendix F](#). It uses some of the tools that are listed in [Appendices B and D](#).

Trilinear-Interpolation Isosurface

The MC algorithm proposed above creates four tables ([Figure 7-12](#)) that describe the isosurface within the image. These tables are ~10 times larger than the image itself if one includes the two cube tables. Furthermore, a Monte-Carlo program that travels cube

by cube within the image needs to access these tables and to calculate the intersection of a trajectory with the triangles of each marching cube. That is not a complex geometrical problem, but it can be time-consuming since there is up to five triangles per cube and the program needs to calculate the intersection with the plane defined by each triangle, as well as to check if the intersection is within the limit of the triangle.

A better solution would be to use a unique equation of the isosurface derived from the initial image and to calculate the intersection of the trajectory with this surface. Previous research has suggested the idea of using a trilinear interpolation of the eight gray-levels of the cube vertices to create a scalar field of gray-level throughout the cube (Matveyev 1994; Natarajan 1994; Van-Gelder and Wilhelms 1994). The bilinear interpolation used by Nielson and Hamann (1991) in their asymptotic decider is a restriction of the trilinear interpolation to each ambiguous face of the cube. When equating the resulting 2D gray-level field to the isovalue, Nielson and Hamann found the equation of a hyperbola. By extending the technique to the entire marching cube, the result is the equation of a hyperboloid. This method only requires memory for the initial gray-level image, eliminates the look-up tables used to generate the MC triangles, avoids the time-consuming manipulation of the triangles, and proposes a better representation of the isosurface than that given by collections of flat triangles.

For the marching cube of unit size represented in Figure 7-13, the trilinear interpolation of the gray-level is

$$\begin{aligned}
 B(x, y, z) = & B_0(1-x)(1-y)(1-z) + B_1x(1-y)(1-z) \\
 & + B_2xy(1-z) \quad \quad \quad + B_3(1-x)y(1-z) \\
 & + B_4(1-x)(1-y)z \quad \quad + B_5x(1-y)z \\
 & + B_6xyz \quad \quad \quad \quad \quad + B_7(1-x)yz
 \end{aligned} \tag{7-5}$$

In Equation (7-5), $B(x,y,z)$ is the gray-level at any point within the cube and the B_n values are the gray-levels at the eight vertices. If Iso is the isovalue that will segment the image, the isosurface is represented by

$$B(x, y, z) = Iso . \quad (7-6)$$

This hyperboloid has asymptotic surfaces that separate the infinite space into eight regions. Four of these regions contain the four lobes of the hyperboloid. Depending on the location of the center of symmetry relative to the marching cube, one, two, three, or four of these lobes can intersect the cube. Some examples are represented in Figure 7-14 for different vertex patterns. In Figure 7-14, the numbers in parentheses are the configuration numbers calculated from the vertex numbering proposed in Figure 7-13. Figure 7-14-A represents a pattern 1. Note that the linear interpolation of the connecting points along the edges of the cube is a restriction of the trilinear interpolation. Therefore, the connecting points of the triangle proposed by Lorensen are the same as the connecting points shown in Figure 7-14-A. Figure 7-14-B represents another example of the same configuration with different values for the vertex gray-levels. B_2 is chosen farther from the isovalue than the other vertices and the surface is pushed away from the vertex. Figure 7-14-C represents a pattern 8. It can be very different from a flat polygon. Figures 7-14-D, 7-14-E, and 7-14-F represent respectively patterns 9, 5c, and 11. They can be compared with their equivalent triangulated patterns in Figure 7-8.

It can be shown that the trilinear-interpolation technique generates the 23 patterns of the MC algorithm. But how does the trilinear interpolation solve the ambiguity problem? Figure 7-15-A represents a pattern 3 designed with 2 triangles. When the gray-levels B_0 and B_2 are changed to a value farther from the isovalue than B_1 and B_3 , the

result is shown in [Figure 7-15-B](#): the two surfaces converge closer and closer together. In [Figure 7-15-C](#), they join into a single surface. The direct design of the ambiguous face of [Figure 7-15-B](#) (the bottom face in this case) has become a reverse design in [Figure 7-15-C](#). Going a little farther, [Figure 7-15-D](#) is equivalent to the reverse design of pattern 3 in [Figure 7-9](#). [Figures 7-15-E](#) and [7-15-F](#) show the same change for a pattern 6. It is thus shown that the trilinear technique automatically handles the ambiguity problem.

How the trilinear interpolation allows a facial decision is shown in [Figure 7-16](#) that represents the four possible designs of a pattern 10. By changing the values of the vertex gray-levels, one can obtain the four different situations. The curves that represent the intersection of the surface with the ambiguous faces are the hyperbolae used by Nielson and Hamann (1991) to decide between direct and reverse methods. [Figure 7-16](#) can be compared with [Figure 7-10](#). It can be shown that all additional configurations introduced by Nielson and Hamann are automatically handled by the trilinear-interpolation technique.

Finally, one may worry about how these surfaces will connect between the cubes. Since the bilinear interpolation at a face is nothing less than the restriction of the trilinear interpolation to the face, two adjacent cubes will converge to the same hyperbola at the connecting face. Therefore, the surface is continuous. [Figure 7-17-A](#) shows how a pattern 3 is connected to a pattern 3c. The surface is continuous and smooth. The smoothness of the surface is not a guaranty since the two cubes have only four vertices in common. [Figure 7-17-B](#) shows a slight angle between the two surfaces of a pattern 2 on top of a pattern 5c. In [Figure 7-17-C](#), four cubes are represented together. The surface is not completely smooth, but it is continuous. For all the configurations presented in

Figures 7-14, 7-15, 7-16, and 7-17, the vertex gray-levels used to produce the isosurfaces are reported in Table 7-2.

The trilinear interpolation suffers a few singularities. First, if one vertex of the cube were equal to the isovalue, the surface would resume to a single point at this vertex. This is avoided by the use of a non-integer isovalue, like 127.5 for the mathematical bone sample. Second, symmetric configurations of the vertex gray-levels would lead to a surface that would no longer be a hyperboloid. As an example, one can imagine a pattern 13 with all its dotted vertices equal to 255 and all its non-dotted vertices equal to 0. This perfect symmetry would transform the surface into three planes that intersect at the center of the cube and divide it into eight sub-cubes: four of them being above the isovalue and four being below. These non-hyperboloid situations are not critical for a computer program as will be seen in the next section.

Because of the complexity of Equation (7-6), we will not try to evaluate the surface area of the isosurface within each marching cube. Our study will be limited to the chord-length distribution measurement to show that the trilinear technique is suitable for image-based computer calculations.

Intersection of a Straight Line with a Hyperboloid

In both the particle transport code and the chord-length distribution measurements, one is interested in the intersection of a straight line (i.e., chord) with the hyperboloid surface given by Equation (7-6). The parametric equation of a straight line characterized by a point (X_0, Y_0, Z_0) and a direction given by the direction cosines U , V , and W such as $U^2 + V^2 + W^2 = 1$ can be substituted into Equation (7-6) to find the intersection points. Equations (7-5) and (7-6) are given for a cube of unit size. Therefore, the parametric

equation of the straight line must be corrected to take into account the cube size. With V_x , V_y , and V_z being the voxel sizes of the initial image, the straight line is characterized by

$$\begin{cases} x = \frac{X_0 + DU}{V_x} \\ y = \frac{Y_0 + DV}{V_y} \\ z = \frac{Z_0 + DW}{V_z} \end{cases} . \quad (7-7)$$

In Equation (7-7) D is the parameter of the straight line, that is the distance from a point (x, y, z) of the line to (X_0, Y_0, Z_0) . If D is negative, the point is in the backward direction.

Using Equation (7-7) into Equation (7-6) gives a cubic equation with D as unknown.

That is

$$aD^3 + bD^2 + cD + d = 0. \quad (7-8)$$

with

$$\begin{aligned} a &= UVW\Delta_B \\ b &= UV(\Delta_B Z_0 - \Delta_z V_z) + VW(\Delta_B X_0 - \Delta_x V_x) + WU(\Delta_B Y_0 - \Delta_y V_y) \\ c &= U(\Delta_B Y_0 Z_0 - \Delta_y Z_0 V_y - \Delta_z Y_0 V_z + \Delta_{yz} V_y V_z) \\ &\quad + V(\Delta_B Z_0 X_0 - \Delta_z X_0 V_z - \Delta_x Z_0 V_x + \Delta_{zx} V_z V_x) \\ &\quad + W(\Delta_B X_0 Y_0 - \Delta_x Y_0 V_x - \Delta_y X_0 V_y + \Delta_{xy} V_x V_y) \\ d &= (Iso - B_0)V_x V_y V_z + \Delta_B X_0 Y_0 Z_0 + \Delta_{xy} Z_0 V_x V_y + \Delta_{yz} X_0 V_y V_z + \Delta_{zx} Y_0 V_z V_x \\ &\quad - (\Delta_x Y_0 Z_0 V_x + \Delta_y Z_0 X_0 V_y + \Delta_z X_0 Y_0 V_z) \end{aligned} \quad (7-9)$$

and

$$\begin{aligned} \Delta_B &= B_0 + B_2 + B_5 + B_7 - (B_1 + B_3 + B_4 + B_6) \\ \Delta_x &= B_0 + B_7 - (B_3 + B_4) \\ \Delta_y &= B_0 + B_5 - (B_1 + B_4) \\ \Delta_z &= B_0 + B_2 - (B_1 + B_3) \\ \Delta_{xy} &= B_0 - B_4 \\ \Delta_{yz} &= B_0 - B_1 \\ \Delta_{zx} &= B_0 - B_3 \end{aligned} \quad (7-10)$$

This equation is solved analytically using traditional complex number calculation (Press et al. 1992). In some situations involving symmetry of the vertex gray-levels, the

cubic equation can reduce to a quadratic equation or even a linear equation. Therefore, 0, 1, 2, or 3 real solutions are expected. Among them, the smallest positive value of D (if there is one) is the distance from the point (X_0, Y_0, Z_0) to the surface when traveling toward the direction (U, V, W) . Then, the intersection point is given by

$$\begin{cases} x = X_0 + DU \\ y = Y_0 + DV \\ z = Z_0 + DW \end{cases} . \quad (7-11)$$

This technique was used to measure the chord-length distribution through the images of the mathematical bone sample. The results were then compared with the extended MC technique and with [Equations \(7-3\) and \(7-4\)](#).

When solving a cubic equation with a computer code, even with double precision variables, round-off errors occur, and it is not possible to guarantee the exactness of the solution. When the intersection of the trajectory with the isosurface is measured from a point close to the surface, the calculation returns a very small distance D that may be negative whereas it should be positive (or vice-versa). The consequence is that the intersection is missed (or that an extra one is created) and the program loses control of where the particle or chord travels. In Monte-Carlo calculations that treat millions of trajectories, a systematic detection of the problem showed that about 0.1% of the trajectories suffered this problem, and thus we decided to abandon these trajectories instead of trying to correct them. The 0.1% does not mean that the result is affected by the same amount. It means that it is calculated from only 99.9% of the sampling initially chosen. Therefore, the consequence on the result is probably orders of magnitude smaller than this percentage.

The chord-length distribution measurement that implements the intersection of the hyperboloid surface was performed with the C++ program listed in [Appendix G](#). It uses some of the tools that are listed in [Appendix B](#).

Results and Discussion

Surface-Area Measurement

The total surface area of the marrow spheres within the mathematical bone sample was measured within the voxel-based image as well as in images with both the direct and the reverse techniques of the MC algorithm. The results are listed in columns 3, 4, and 5 of [Table 7-1](#). The convergence of the area to the exact value is shown in [Figure 7-18](#). The solid straight line is the exact value given by [Equation \(7-2\)](#). The dotted straight line corresponds to the exact value multiplied by 1.5. One can see, as predicted in previous studies (see [Chapters 3](#) and [4](#)), that the measurement of the surface area through the voxel-based images overestimates the true value by 50% at high resolution (small voxel size). On the other hand, both the direct (DIR) and the reverse (REV) techniques converge to the exact value. At 75 μm , which is about what is currently used for NMR microscopy imaging of trabecular bone, the approximation is within 2%. At 30 μm , it is down to only 0.5%.

The DIR and the REV techniques differ slightly as shown in [Figure 7-18](#). At large voxel sizes, the space between the spheres of the model (representing the bone trabeculae) is thin compared to the voxel size. Therefore, an ambiguous face with a direct segmentation will generate two long segments next to each other. A reverse segmentation will generate two short segments located at two opposite corners of the voxel. This explains why the REV surface area is smaller than the DIR surface area at large voxel

sizes. At small voxel sizes, the situation is inverted and the REV surface area becomes larger. The two curves intersect at about 400 μm which is roughly the average thickness of the space between the spheres, as can be estimated in [Figure 7-11](#).

Chord-Length Distributions

The chord-length distributions were measured through the voxel-based images and the MC-based images using the three techniques described previously (DIR, REV, and trilinear interpolation). The results are shown in [Figure 7-19](#) for four voxel sizes: 375 μm , 150 μm , 60 μm , and 24 μm . In [Figure 7-19](#), the solid-line curves represent the exact distribution calculated from [Equation \(7-3\)](#). The numbers in parentheses within the legend captions are the mean chord lengths for each distribution. The measurements through the voxel-based images clearly show the voxel effects discussed in the previous chapters. At large voxel sizes, the curve presents spikes every voxel size. At small voxel sizes, these spikes tend to disappear, but a sharp peak (not entirely shown in [Figure 7-19](#)) remains present at the beginning of the distribution (short chords). The voxel effects increase the frequency of the short chords and the mean chord length is reduced by 30% at image resolutions of both 60 μm and 24 μm .

At large voxel sizes (e.g., 375 μm), the three MC-based techniques do not give better results than the voxel-based technique. They are even worse if one compares the mean chord lengths. A poor image resolution, even using a good interface-definition technique would not be able to give good results. At 150 μm , the shape of the three MC-based distributions begin to converge with true distribution. The mean chord lengths are all within 5% of the exact mean. Below 60 μm , they are within 1% of the exact mean, and the curves match almost perfectly, except for a residual voxel effect discussed below.

Let us first compare the two polygon techniques (DIR and REV). They both show a voxel effect that overestimates the frequency of short chords. The effect extends from 0 to a chord length about twice the voxel size. The angles between the triangles are not flat. Even though they tend to become flat as the resolution improves, there is still a small angle and these zigzag-like surfaces will always increase the frequency of short chords for the trajectories traveling close and parallel to the boundary. The voxel effect seems to be different between DIR and REV, especially at resolutions of 150 μm and 60 μm . As discussed previously, and as can be seen in [Figure 7-4-B](#), the direct technique better suits the geometry composed of dotted cavities surrounded by a non-dotted network. That is exactly the situation of the mathematical bone sample where the spheres are the dotted regions. Furthermore, [Figures 7-5-B](#) and [7-5-C](#) will help explain the differences of short chord frequencies. The three direct examples of [Figure 7-5-B](#) look more like a sharp piece of bone (non-dotted region) that penetrates into a block of marrow (dotted region). This sharpness will produce an excess of short bone chords. In [Figure 7-5-C](#), on the other hand, the reverse segmentation of the same examples look more like a sharp piece of marrow that penetrates a block of bone, and will produce an excess of short marrow chords. That is exactly what happens within the mathematical bone sample. The reverse segmentation produces more short marrow chords than the direct method. This explains the difference at the 150- μm and 60- μm image resolutions between the DIR and REV techniques. The asymptotic decider discussed previously would probably give a voxel effect between the DIR and the REV techniques. At 24 μm , the voxel size becomes so small compared to the object size that the ambiguous faces become rare and there is little differences noted between the DIR and the REV curves.

The trilinear-interpolation technique should eliminate the angles between the triangles, at least within region interior to the marching cubes. [Figure 7-17](#) shows that there are still angles at the cube interfaces and that a small voxel effect is expected. However, in [Figure 7-19](#), it seems that the effect is worst (even though it remains small) than with the traditional MC techniques (DIR or REV). This observation is attributed to the concavity of the isosurface. In [Figure 7-14-A](#), the surface is generated from a configuration 4 that has one vertex set to 255 and seven vertices set to 0. The dotted vertex can be seen as the center of curvature of the surface. Configuration 251 is the complement of configuration 4. With one vertex set to 0 and seven vertices set to 255, it would generate the exact same surface as configuration 4. In this case, however, the center of curvature is the non-dotted vertex. As a consequence, the concavity of the isosurface depends more on the configuration of the cubes than on the concavity of the object represented. For large voxel sizes, the object will have small features that will be well represented by the cube configurations and the concavity problem will have little consequence. For small voxel sizes at the cube size level, the isosurface is seen as a flat surface whose concavity is dictated more by the cube configuration than by the object shape. The result is that a flat surface that extends over a large number of cubes is likely to be represented by an undulated surface. These undulations generate artificial short chords that are responsible for the small voxel effects seen at high resolution with the use of the trilinear-interpolation technique.

One can see in [Figure 7-19](#) that the voxel effect introduced by the trilinear technique affects the distribution from 0 to a chord length of about two voxel sizes. At high resolution, all distributions measured within smooth surfaces should drop to zero at zero

chord length (Jokisch et al. 2001b) (see also Chapter 3). Besides, the shape of the exact distribution is almost linear at short chords. Therefore, a solution to remove this effect is to change the beginning of the distribution into a linear distribution that smoothly connects to the measured distribution at a chord length equal to two voxel sizes. This procedure was done for the four images of Figure 7-19 and they are compared to the exact distribution in Figure 7-20. Figure 7-20 represents the corrected distributions measured with the trilinear technique. At 375 μm and 150 μm , the correction is not a significant improvement. However, at 60 μm , the distribution is very close to the exact one. At 24 μm , no distinct differences are noted between the exact and the measured distributions.

Conclusion

Three new techniques based on the MC algorithm were developed to better represent the boundary between bone trabeculae and marrow cavities when 3D images of trabecular bone are coupled with computer programs for radiation transport or chord-length distribution measurements. A mathematical sample of trabecular bone was used to test these techniques. The DIR and REV techniques are based on the extended MC algorithm that solves the ambiguity problem by providing a specific design for each complement pattern. The two methods give an excellent preservation of the surface area of the boundary. The third technique is based on the trilinear interpolation of the gray-level over each marching cube. This technique eliminates the image size problem and the need for look-up tables of the DIR and the REV techniques, it solves the ambiguity problem at the marching-cube face level as does the asymptotic decider (Nielson and Hamann 1991), and it simplifies the computer calculations within the image since only one equation is used per marching cube. The surface area was not measured using this

technique, but it is expected to give similar results as given by the DIR and REV methods. The three techniques were used to measure chord-length distributions through the simulated images. When the voxel size reaches reasonable values like the one currently used to image trabecular bone samples via NMR microscopy (Jokisch et al. 2001a; Patton et al. 2002a), the distributions are in a good agreement with the reference distribution. A slight voxel effect is still present, but it can be easily removed, as it only affects the very short chords (below two voxel sizes) where a majority of distributions should be linear.

As a conclusion, one of the three methods should be used each time a computer simulation is to be performed in direct connection with a digital image. The trilinear-interpolation technique is the simplest and will be used to couple future NMR microscopy images of trabecular bone with Monte-Carlo radiation transport codes. It will also be chosen to measure new chord-length distributions through these same samples. The discrepancy found when using the initial voxel images will be reduced from more than 30% (as shown in Chapters 3 and 4) to only a few percent.

Another application of the trilinear technique is for visualization purpose. The current MC surfaces necessitate ~ 7 times the original image size to store the triangles. A visualization program based on the trilinear technique would reduce this memory problem. However, the display of each individual cube as a hyperboloid surface may pose other challenges. The first concerns the computer time required to calculate the isosurface. A second problem is the presence of singularities among hyperboloid surfaces: the asymptotic surfaces. A visualization program must detect these singularities prior to display. The examples proposed in Figures 7-14 to 7-17 have been carefully chosen so that the singularities fall outside the cube so that they are not seen within the Figures

Finally, the current technique was developed within an image containing two media separated by a single isosurface. When imaging techniques are used to show organs of the body, segmentation can be applied to separate the different body tissues. The result is an image for which the gray-levels are used to represent different regions of the body instead of a distance from the isosurface. With a 256 gray-level image, up to 256 different tissues or organs can be segmented within that image. Each half integer can be seen as the isovalue that separates two organs and each isosurface would be on a 2 gray-level scale. The connecting points along the edges of the cubes no longer need to be interpolated, but the trilinear technique can still be used to represent the isosurface. A problem is that the same cube may be crossed by several isosurfaces and a more complex algorithm would need to be developed; nevertheless, the technique is perfectly suitable for these imaging applications.

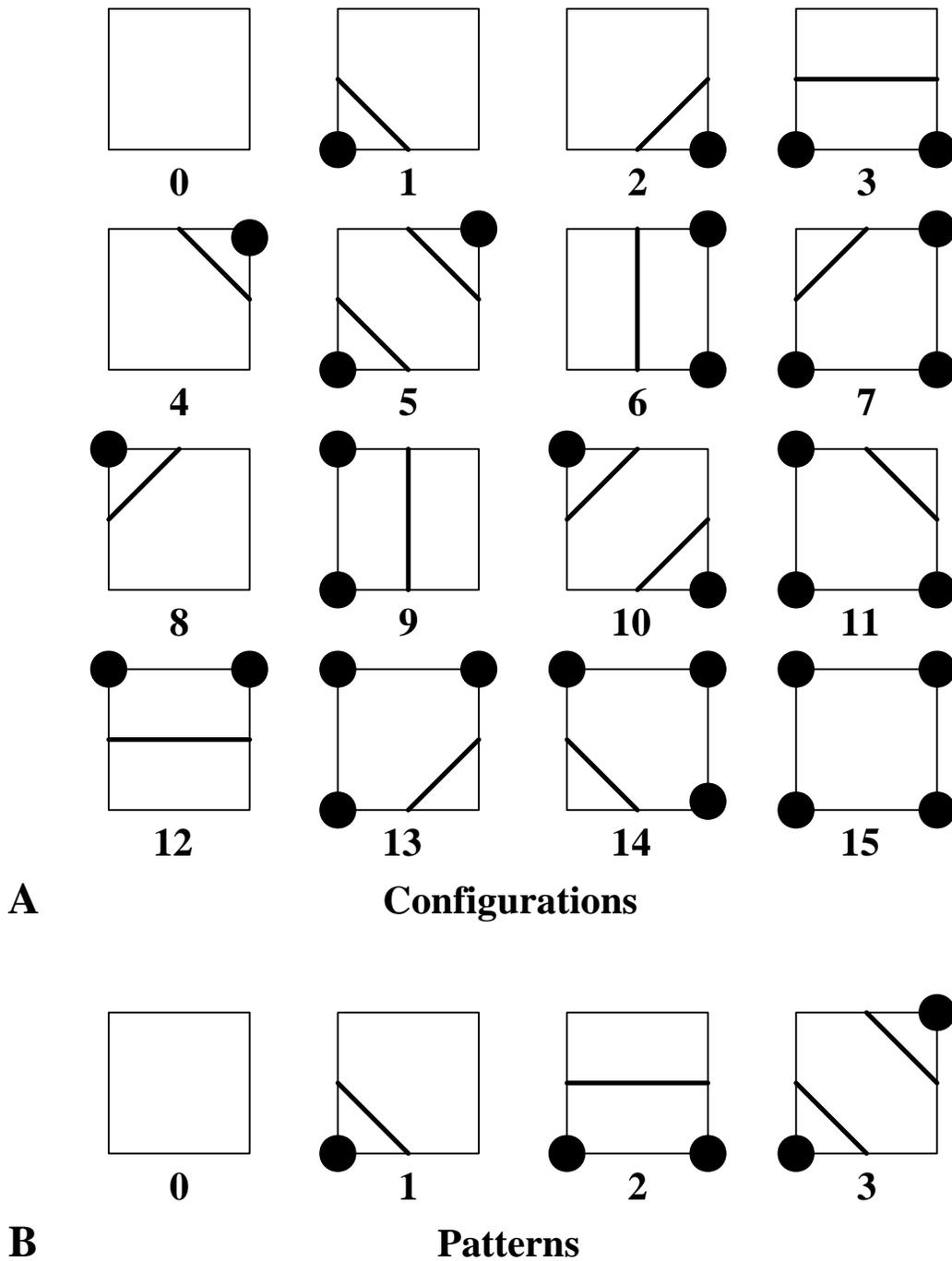
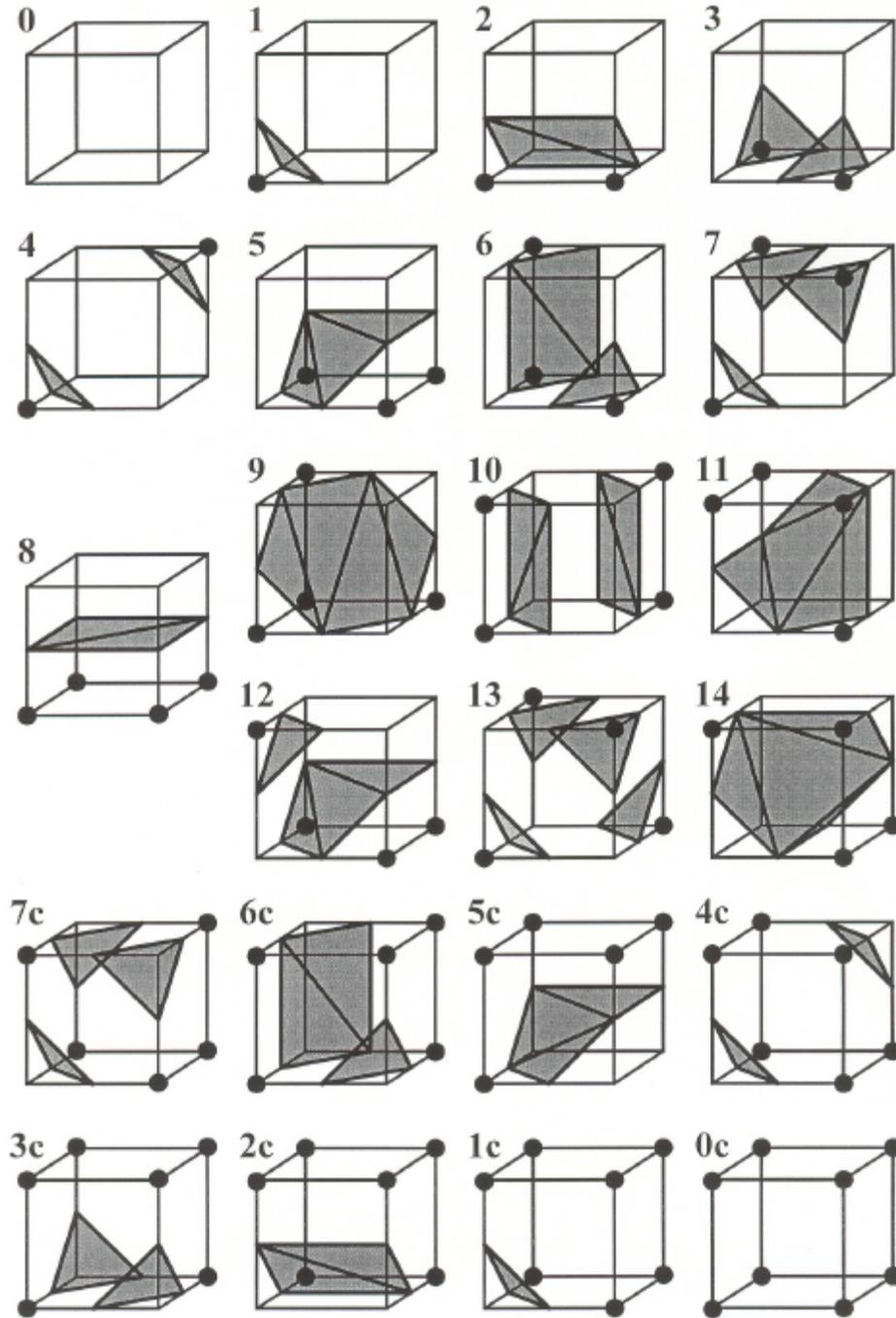


Figure 7-2. MC algorithm configurations and patterns in 2D. A) The 16 Marching-Square configurations for a 2D image. B) Applying rotations and binary complementation, the study can be reduced to only four patterns (with their complements).



Loresen Patterns

Figure 7-3. MC patterns proposed by Lorensen for 3D images. Patterns 0c to 7c are the complement of patterns 0 to 7 and are designed alike. Since patterns 8 to 14 have four vertices above the isovalue and four vertices below, they include their own complements.

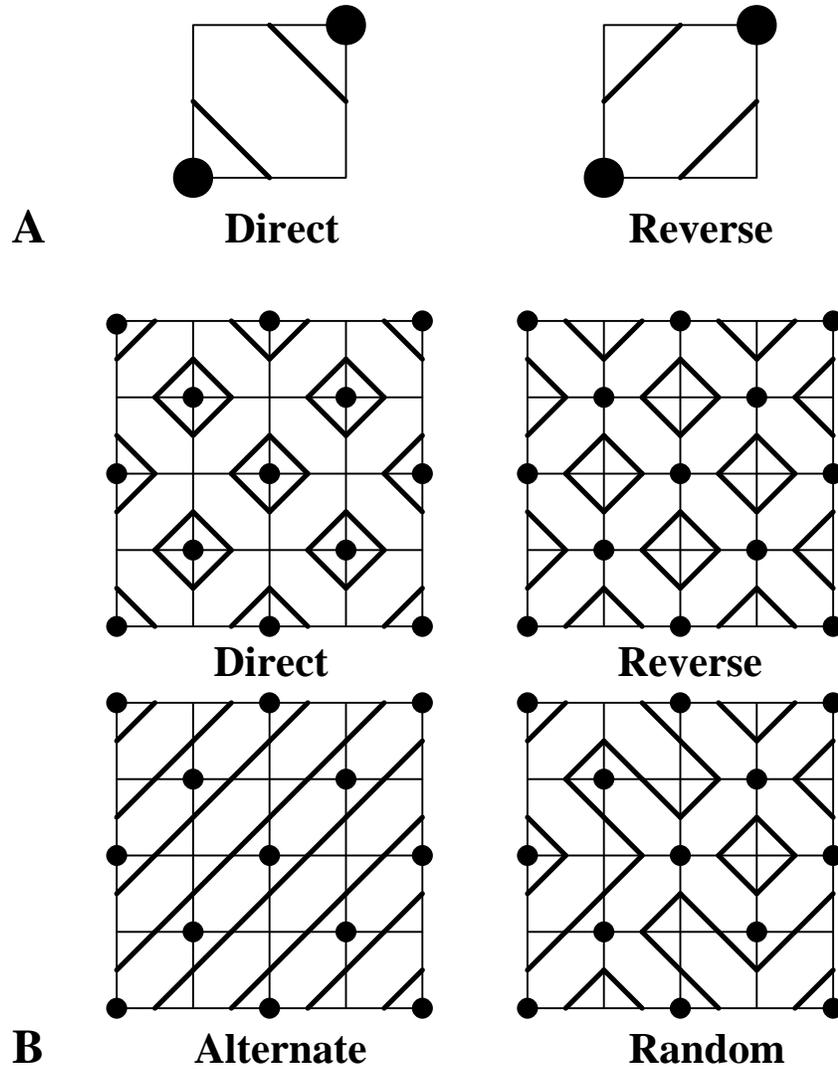


Figure 7-4. Ambiguity problem in 2D. A) The two possible designs for the ambiguous square. B) Depending on the design chosen, the same gray-level image would have a completely different segmentation. Direct: all ambiguous squares have a direct design. Reverse: all ambiguous squares have a reverse design. Alternate: the squares are alternatively designed with direct and reverse segmentation. Random: the decision is made randomly for each square.

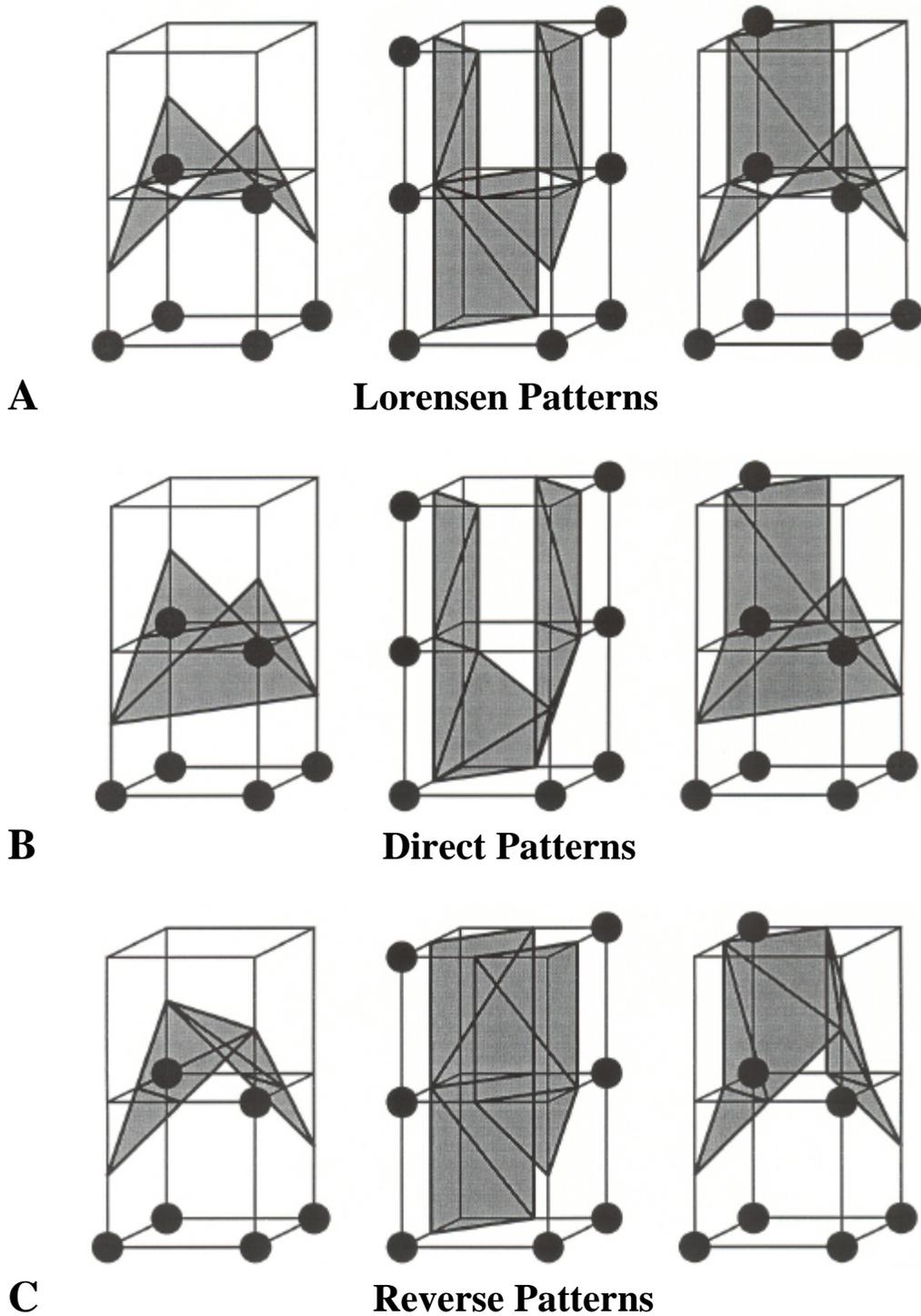


Figure 7-5. Hole problem as a consequence of the ambiguity problem. A) Three examples of the hole problem. Left: a pattern 3 on top of a pattern 3c; center: a pattern 10 on top of a pattern 6c; right: a pattern 6 on top of a pattern 3c. B) Solutions of the hole problem using the direct technique. C) Solutions of the hole problem using the reverse technique.

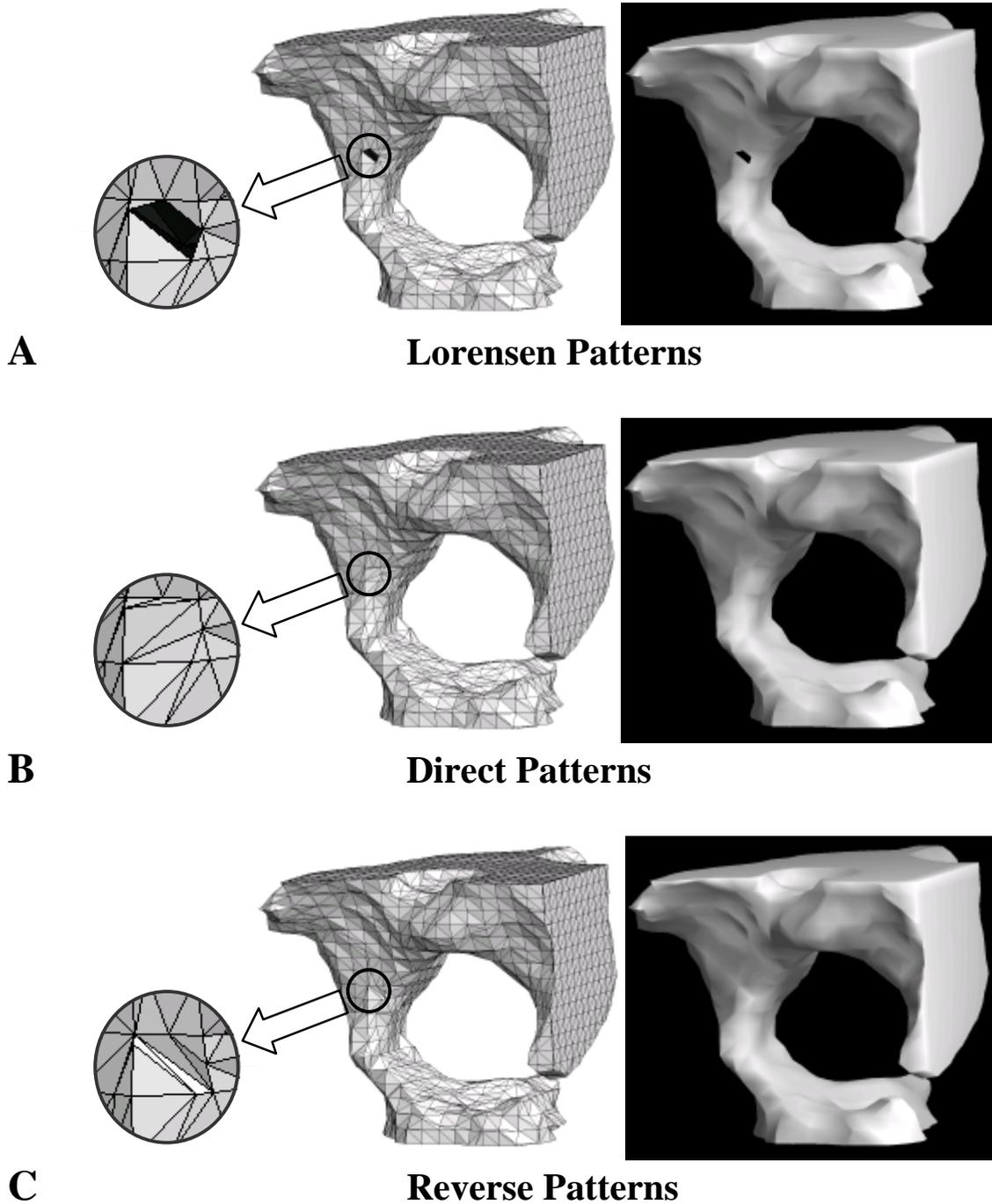


Figure 7-6. Hole problem for a $22 \times 22 \times 22$ image of a trabecular bone sample. The image resolution is $88 \times 88 \times 88 \mu\text{m}^3$. The left image is the display of the MC triangles. The right image uses IDL 5.5 (RSI, 2001) for image rendering. A) The hole is a consequence of the connection between a pattern 3 and a pattern 3c. B) Solution of the hole problem using the direct technique. C) Solution of the hole problem using the reverse technique.

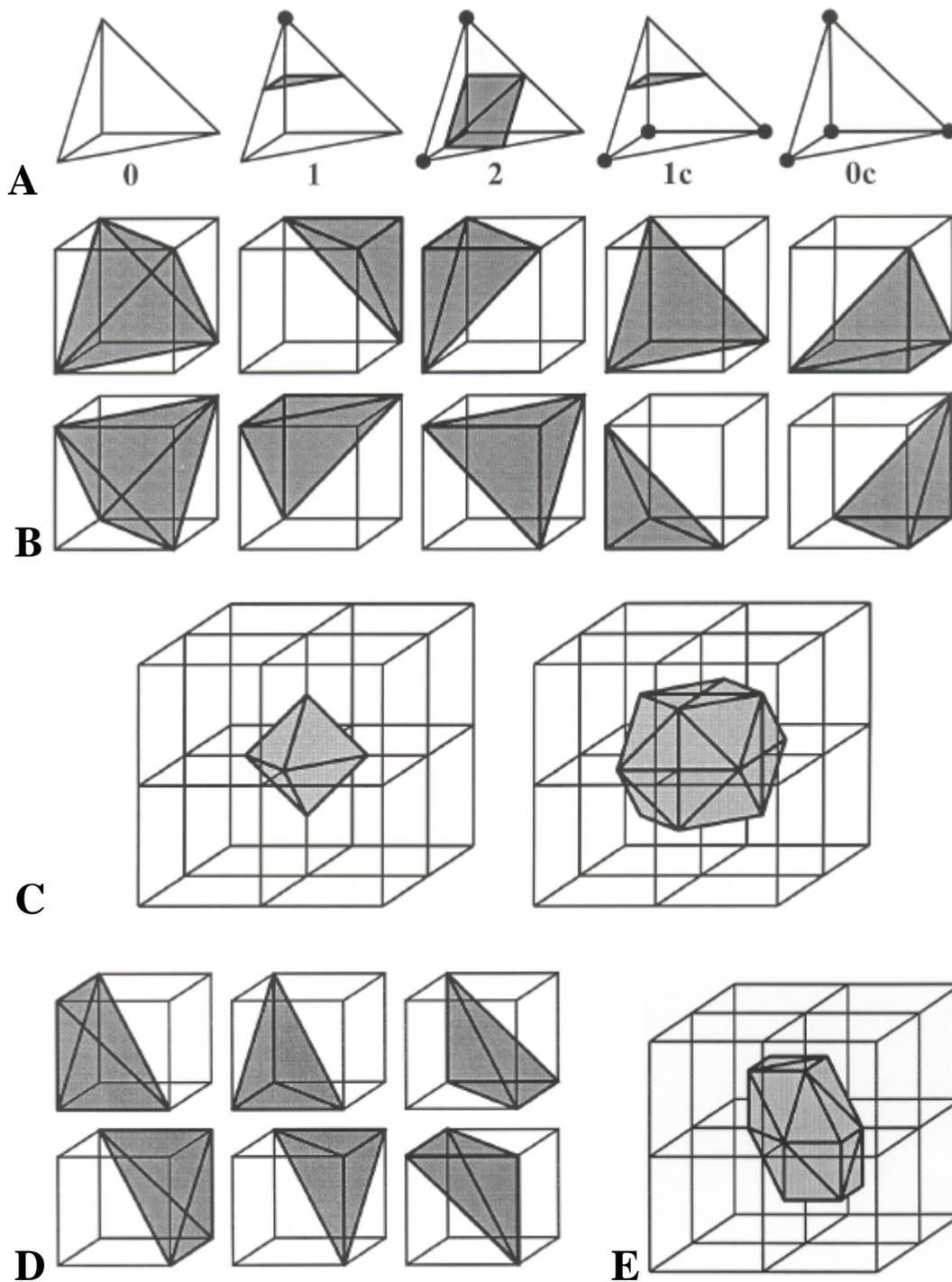
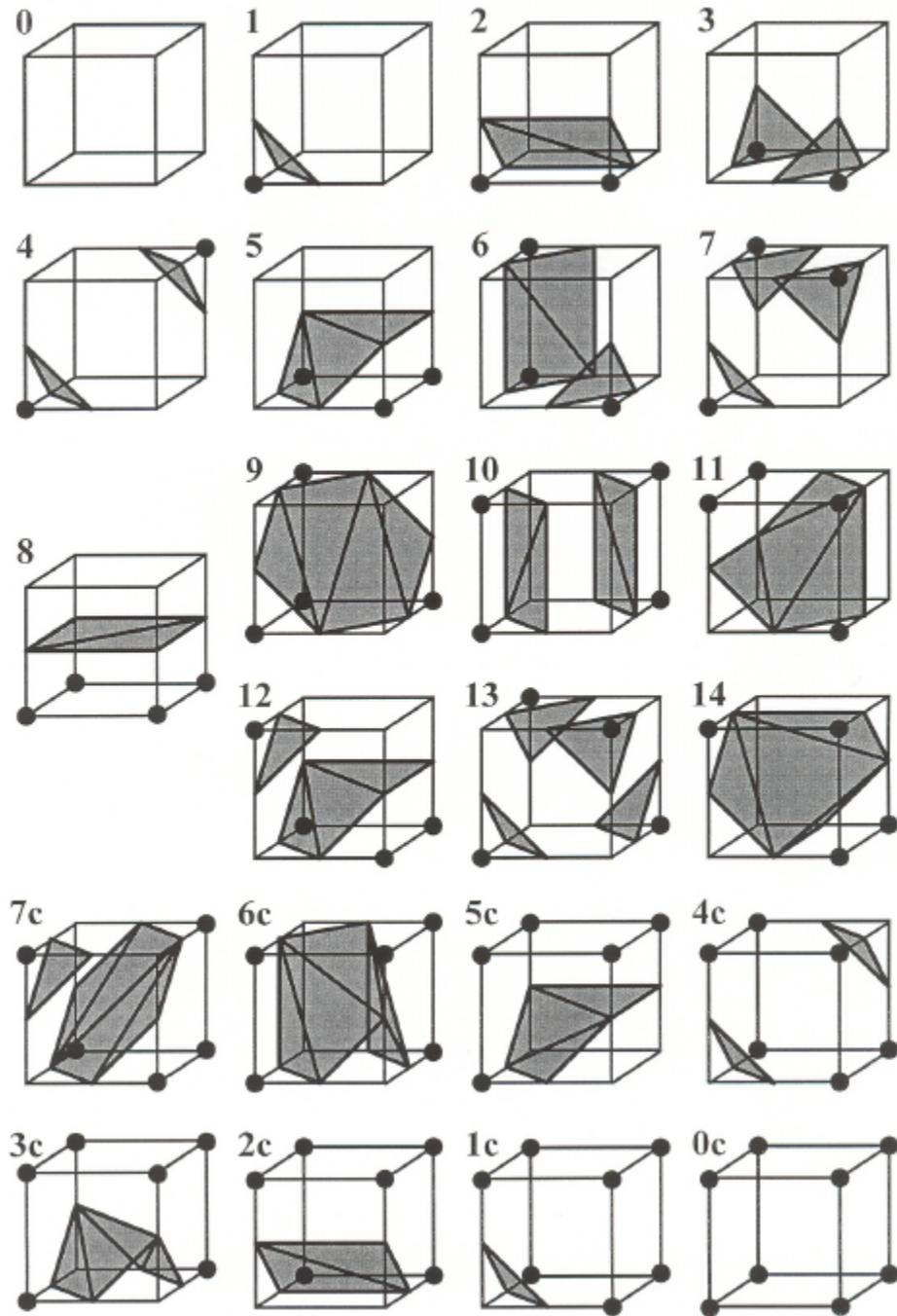
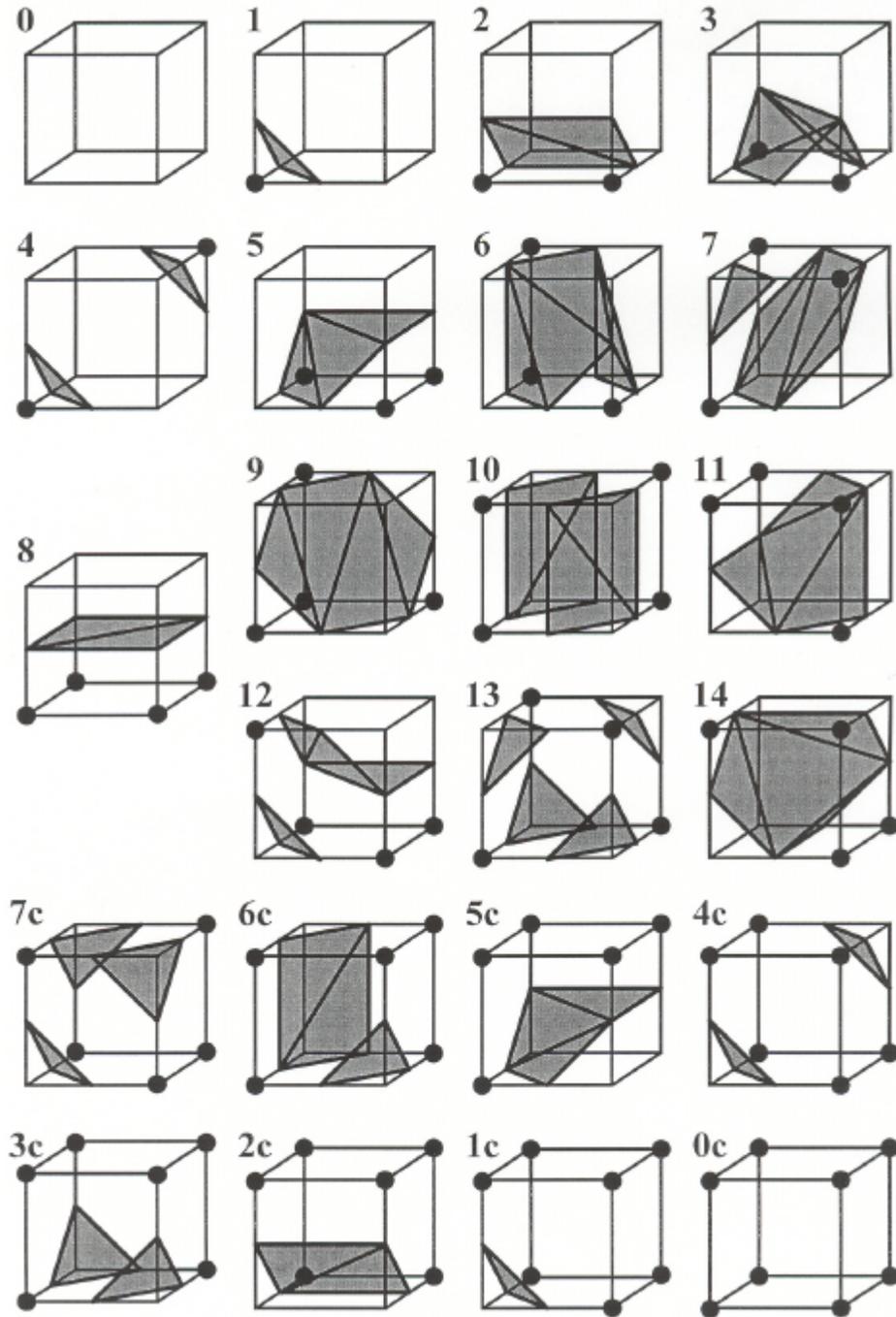


Figure 7-7. Marching-Tetrahedron method. A) The five Marching-Tetrahedron patterns. B) Two solutions to break up a marching cube into five tetrahedrons. C) Using the decomposition proposed in B), a single isolated dot can be surrounded by two different isosurfaces. D) Solution to break up a marching cube into six tetrahedrons. E) Using the decomposition proposed in D), a single isolated dot is surrounded by a non-isotropic volume.



Direct Patterns

Figure 7-8. MC patterns proposed to solve the hole problem using the direct design for ambiguous faces. Patterns 7c, 6c and 3c no longer have the same design as their complements.



Reverse Patterns

Figure 7-9. MC patterns proposed to solve the hole problem using the reverse design for ambiguous faces. Patterns 3, 6, and 7 have the same design as Patterns 3c, 6c, and 7c of Figure 7-8. Patterns 7c, 6c, and 3c have the same design as Patterns 7, 6, and 3 of Figure 7-8. Patterns 10, 12, and 13 also have a different design than their equivalent of Figure 7-8 since they also have ambiguous faces.

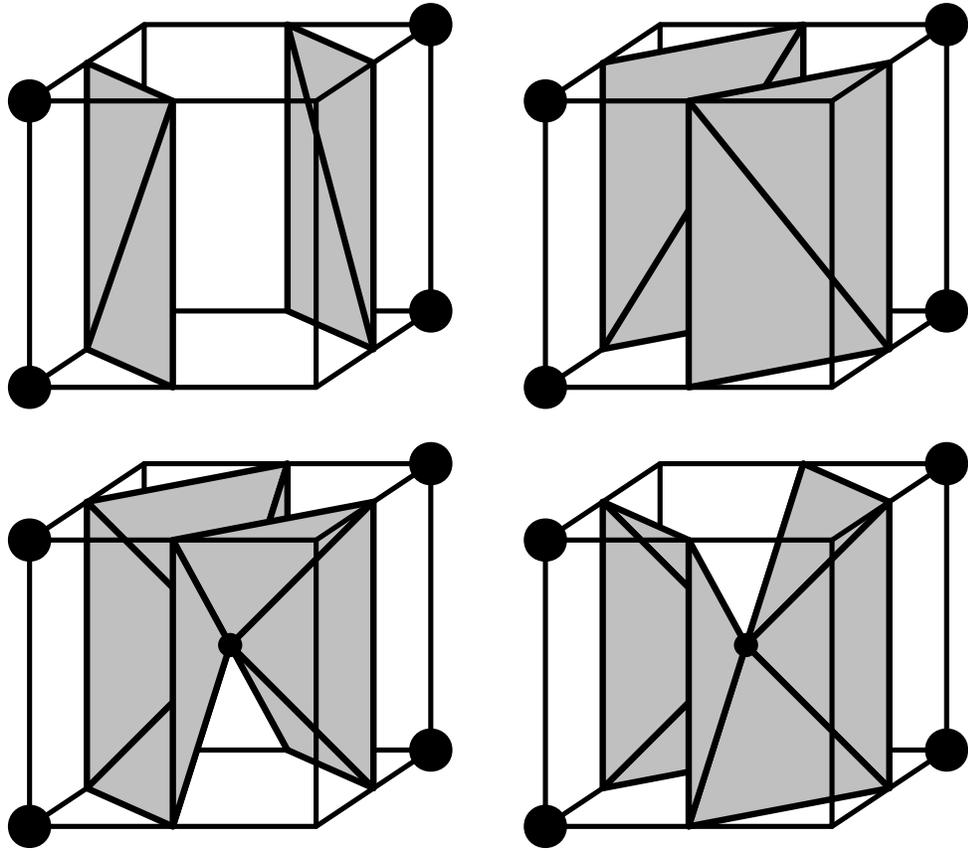


Figure 7-10. The four triangulations that solve the ambiguity problem for pattern 10. They were proposed by Nielson and Hamann (1991) in their asymptotic-decider method. An ambiguous face can have a direct design whereas the other one has a reverse design.

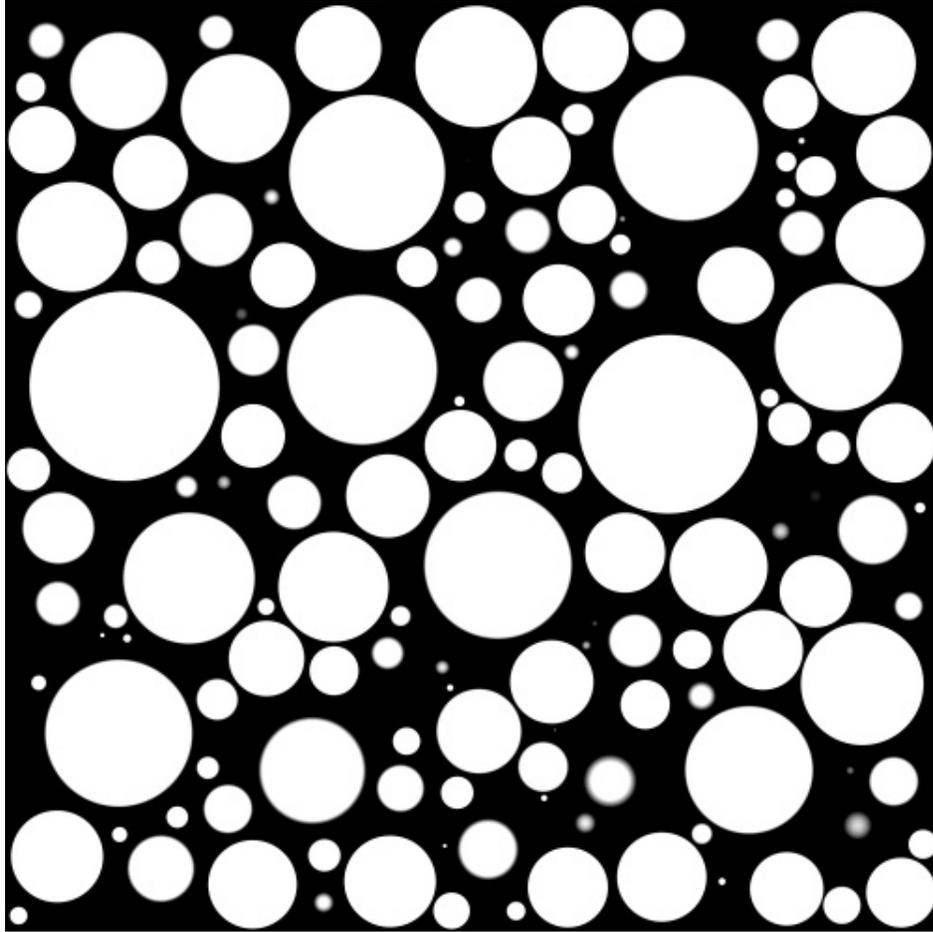


Figure 7-11. Transverse slice through the mathematical sample of trabecular bone. The cross-sectional area is $1.2 \times 1.2 \text{ cm}^2$.

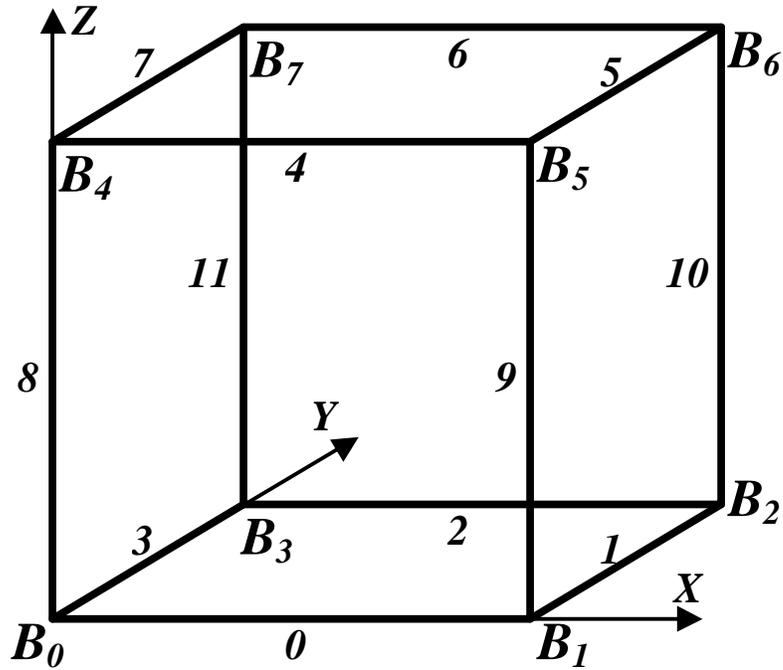


Figure 7-13. Localization of the eight vertices and twelve edges of a marching cube. The B_n values are the gray-levels of the vertices. The index n represents the vertex number as used for the calculation of the cube configuration from the binary state of each vertex. The edge numbers correspond to these used by the look-up tables to design the different configurations as explained in [Appendix A](#).

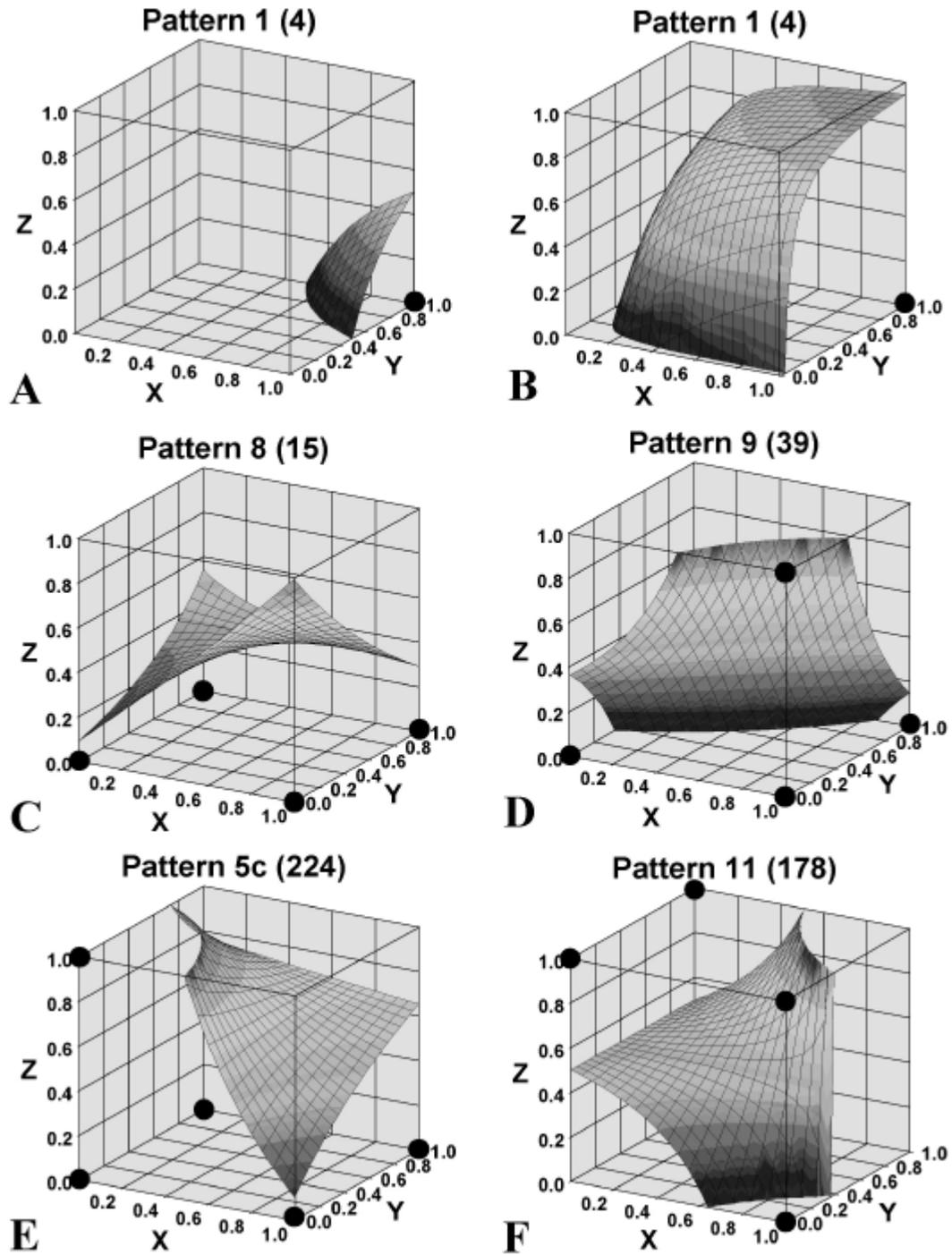


Figure 7-14. Trilinear-interpolated isosurface examples within unit cubes. The numbers in parentheses are the configuration numbers. The gray-levels used to obtain these surfaces are given in [Table 7-2](#).

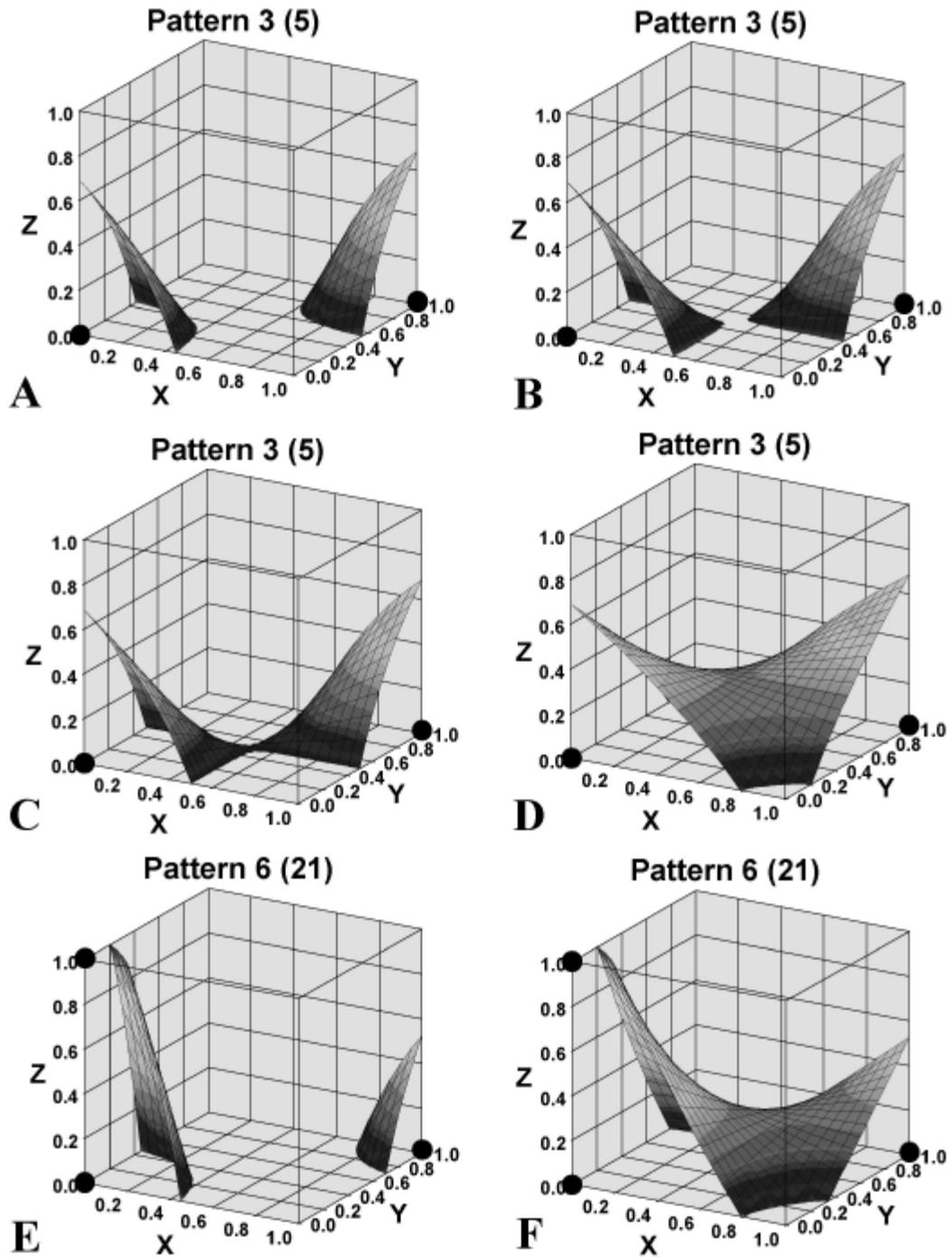


Figure 7-15. Trilinear-interpolated isosurfaces within unit cubes. A), B), C), and D) show how a pattern 3 evolves from the direct design of A) to the reverse design of D). E) Direct pattern 6. F) Reverse pattern 6. The numbers in parentheses are the configuration numbers. The gray-levels used to obtain these surfaces are given in [Table 7-2](#).

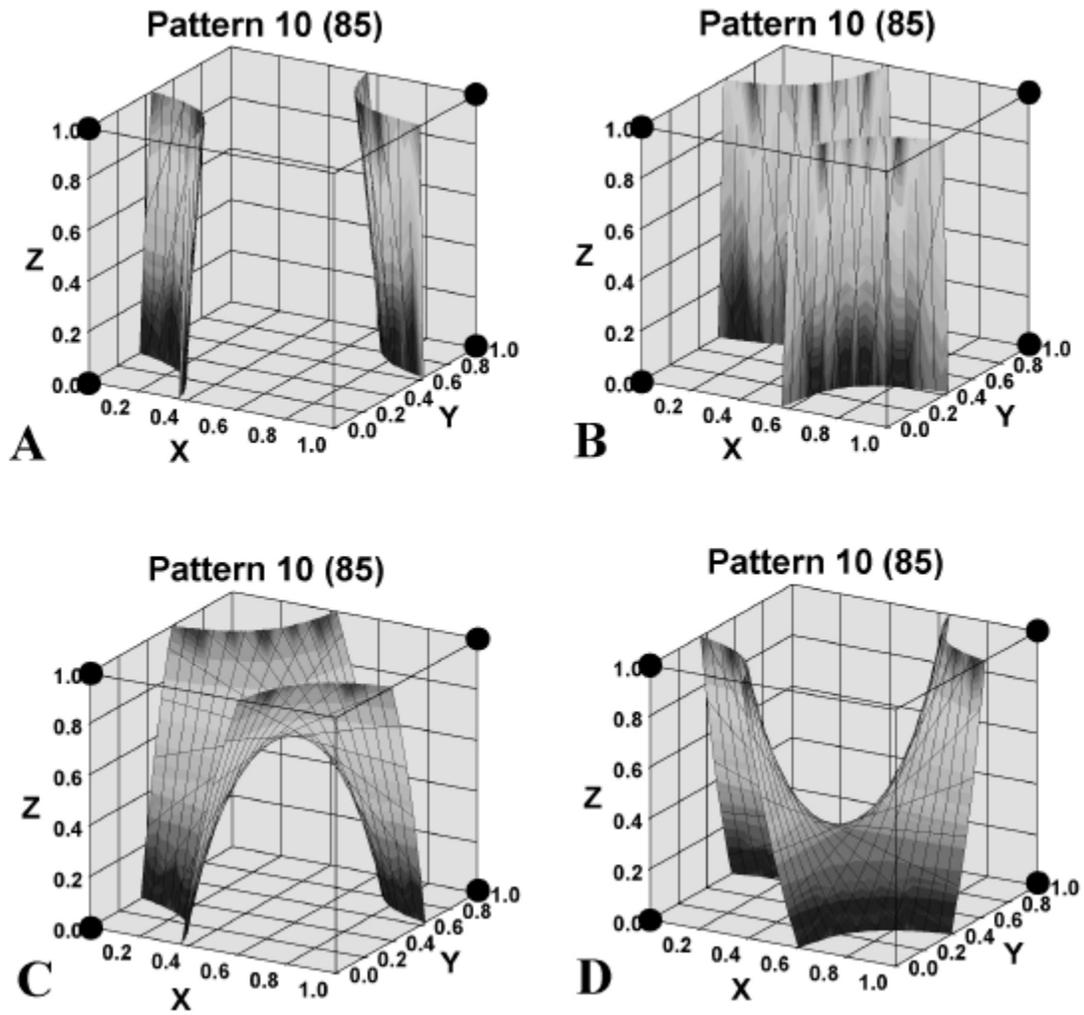


Figure 7-16. Trilinear-interpolated isosurfaces within unit cubes of pattern 10. A) Both ambiguous faces are direct. B) Both ambiguous faces are reverse. C) and D) The two ambiguous faces have a different design. The numbers in parentheses are the configuration numbers. The gray-levels used to obtain these surfaces are given in [Table 7-2](#).

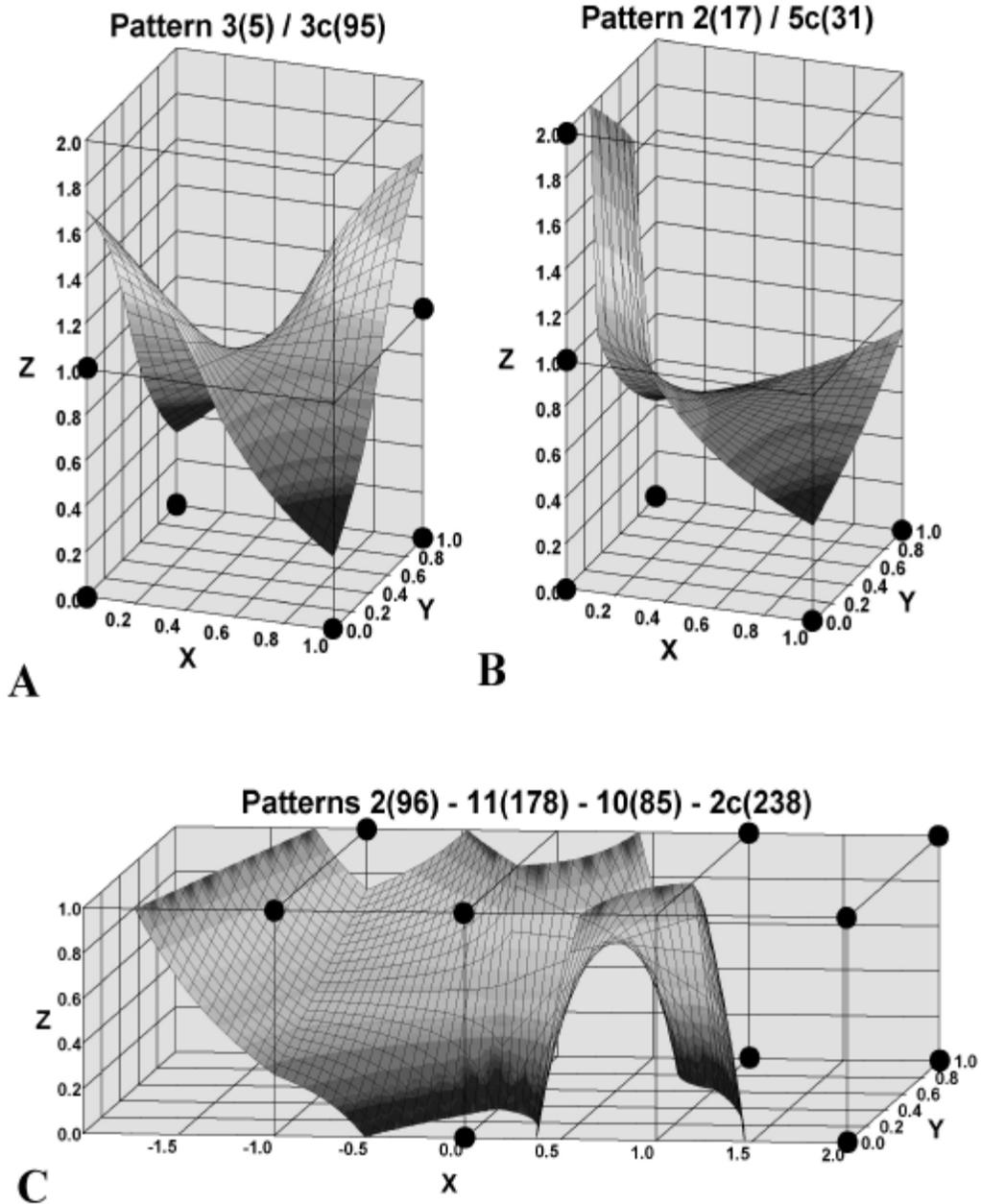


Figure 7-17. Trilinear-interpolated isosurfaces within several adjacent cubes, showing how the surface connects. A) A pattern 3 on top of a pattern 3c. B) A pattern 2 on top of a pattern 5c. C) Four cubes of patterns 2, 11, 10, and 2c, from left to right. The numbers in parentheses are the configuration numbers. The gray-levels used to obtain these surfaces are given in [Table 7-2](#).

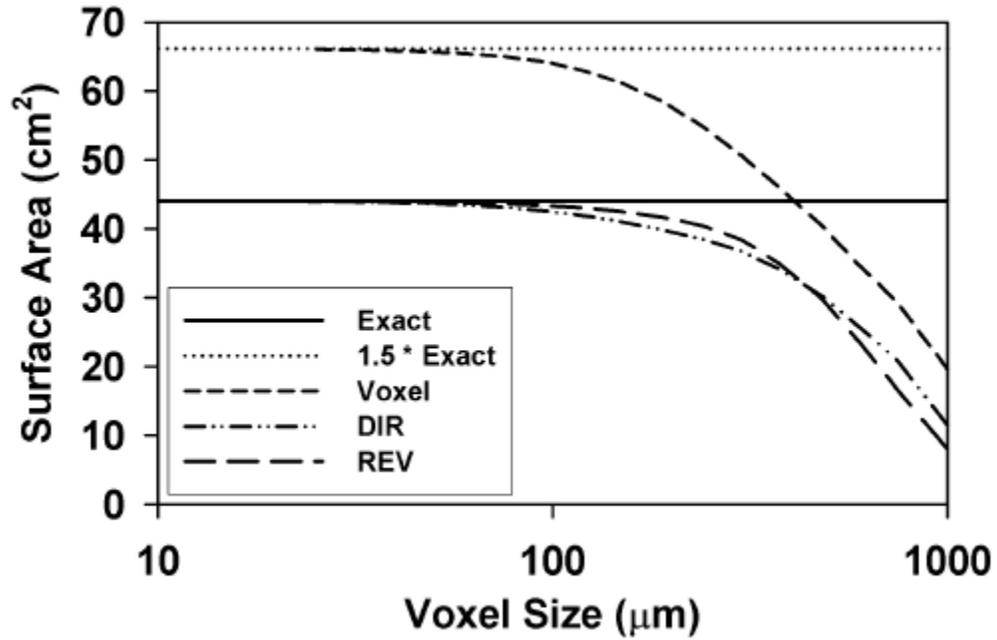


Figure 7-18. Surface area as a function of image resolution for the mathematical sample. The solid straight line is the exact value. The voxel model converges to 1.5 times the exact value. Both the DIR and the REV MC models converge to the exact value.

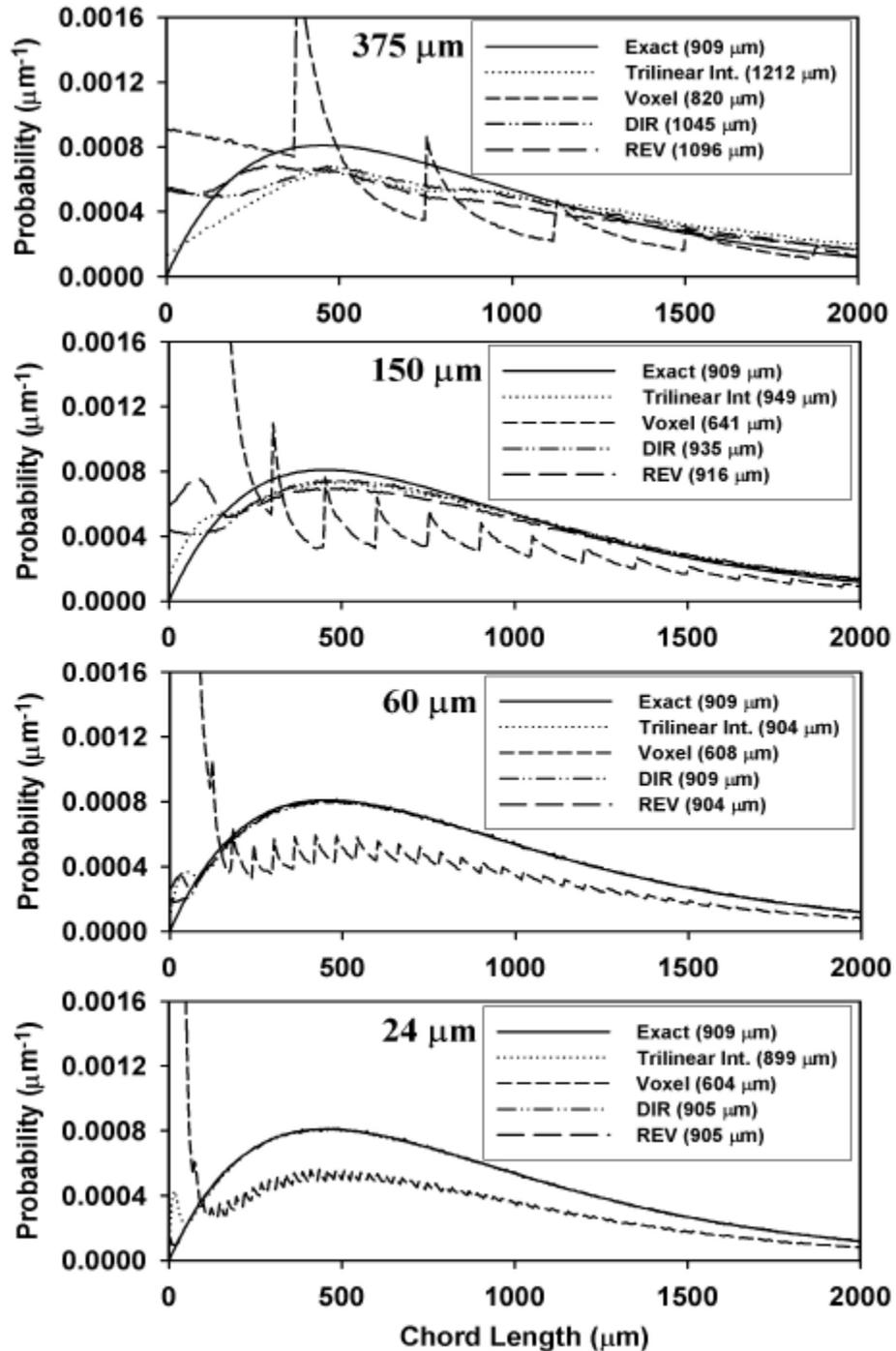


Figure 7-19. Chord-length distributions measured through the mathematical bone sample for four different voxel sizes. The solid line is the exact distribution calculated from Equation (7-3). The three MC techniques (DIR, REV, and trilinear interpolation) are compared with the measurement through the voxel image.

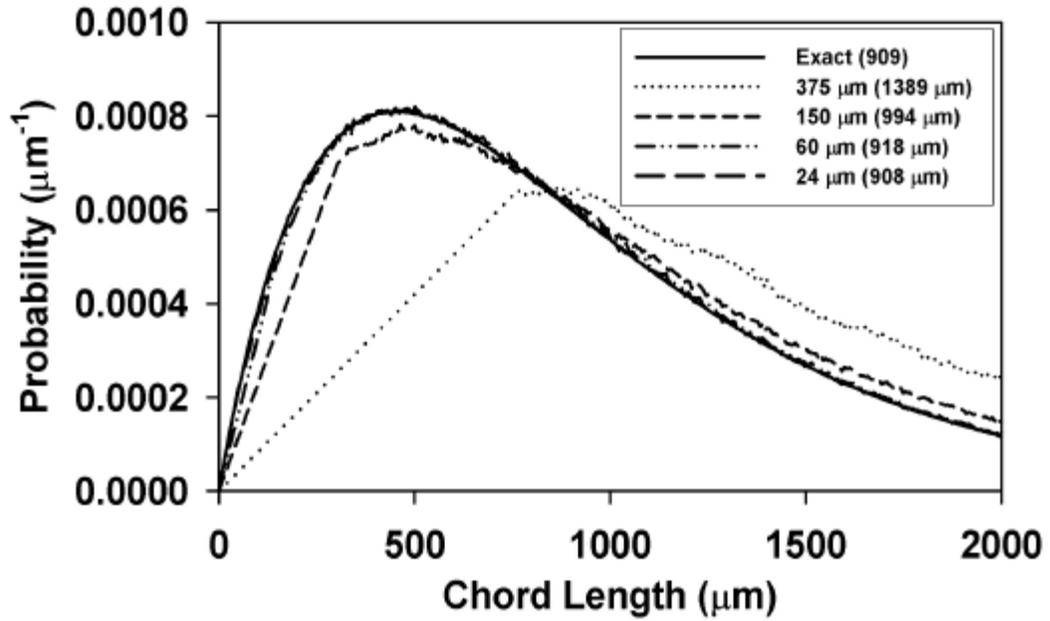


Figure 7-20. Corrected chord-length distributions measured with the trilinear technique. The measurement is done for 4 image resolutions. The voxel effect is removed by forcing the distribution to be linear at short chord lengths.

Table 7-1. Characteristics of the 17 segmented images of the mathematical bone sample. The voxel sizes range from 1000 μm to 24 μm . The isosurface area is calculated within the voxel image (column 3) and within the MC polygon image using both the DIR and REV techniques (columns 4 and 5, respectively). The exact interface area is $S_{spheres} = 44.0916655 \text{ cm}^2$.

Voxel size (μm)	Voxels per dimension	Surface-area Voxel (cm^2)	Surface-area MC-DIR (cm^2)	Surface-area MC-REV (cm^2)
1000.0	12	19.58	11.47	8.02
750.0	16	29.18	20.83	16.56
600.0	20	34.67	25.70	23.49
480.0	25	40.53	30.26	29.87
375.0	32	45.99	34.20	34.98
300.0	40	50.66	36.74	38.44
240.0	50	54.86	38.49	40.47
187.5	64	58.63	39.94	41.75
150.0	180	61.15	41.02	42.49
120.0	100	62.97	41.93	43.08
96.0	125	64.20	42.61	43.41
75.0	160	65.00	43.14	43.63
60.0	200	65.44	43.47	43.74
48.0	250	65.73	43.69	43.81
37.5	320	65.90	43.85	43.89
30.0	400	66.00	43.94	43.95
24.0	500	66.04	44.00	44.01

Table 7-2. Gray-levels used for the isosurface equation of the cubes presented in [Figures 7-14, 7-15, 7-16, and 7-17](#). The gray-levels range from 0 to 255 and the isovalue (*Iso*) is 127.5.

Figure	B ₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇
7-14-A	0	0	255	0	0	0	0	0
7-14-B	125	125	250	125	120	120	120	120
7-14-C	140	255	180	160	0	127	0	100
7-14-D	200	255	150	0	0	255	0	0
7-14-E	255	140	255	255	230	20	90	60
7-14-F	0	200	0	0	255	205	5	255
7-15-A	230	2	230	2	80	0	80	0
7-15-B	230	24	230	24	80	0	80	0
7-15-C	230	26	230	26	80	0	80	0
7-15-D	230	100	230	100	80	0	80	0
7-15-E	230	2	180	2	160	0	80	0
7-15-F	230	100	180	100	160	0	80	0
7-16-A	180	38	180	33	185	40	185	35
7-16-B	180	88	180	83	185	90	185	85
7-16-C	180	38	180	33	185	90	185	85
7-16-D	185	90	185	85	180	38	180	33
7-17-A top	230	15	230	15	80	0	80	0
7-17-A bottom	255	180	255	180	230	15	230	15
7-17-B top	190	0	110	0	175	0	0	0
7-17-B bottom	255	220	255	220	190	0	110	0
7-17-C left	0	80	0	0	80	255	175	0
7-17-C left-center	80	180	33	0	255	185	85	175
7-17-C right-center	180	38	180	33	185	90	185	85
7-17-C right	38	231	253	180	90	235	254	185

CHAPTER 8
HYPERBOLOID REPRESENTATION OF THE BONE-MARROW INTERFACE
WITHIN 3D NMR IMAGES OF TRABECULAR BONE: APPLICATIONS TO
SKELETAL DOSIMETRY¹

Introduction

Irradiation of the active bone marrow continues to be dose-limiting for many radionuclide therapies (Sgouros 1993; Sgouros et al. 2000). Consequently, efforts persist in the development of skeletal dose models that improve patient specificity (Bolch et al. 2002b; Shen et al. 1999; Shen et al. 2002). Recent physical models of skeletal dosimetry have been constructed using 3D images of human trabecular bone samples acquired from high-resolution Nuclear Magnetic Resonance (NMR) imaging systems (Figure 8-1). These images are used as a geometrical input data for radiation transport codes that calculate energy deposition events within both the bone trabeculae and the marrow cavities of the sample (Bolch et al. 2002a; Jokisch 1999; Patton et al. 2002a).

In Chapters 3 and 4 it has been shown that, at low electron energies, voxel effects within the digital image could result in overestimates of the dose to marrow when the radiation source is localized within bone trabeculae (or vice-versa). These effects have their origin in the rectangular faces of the image voxels such that the surface area of the bone-marrow interface is overestimated. Furthermore, it was shown that increasing the image resolution does not reduce these effects. Consequently, a new technique, based on

¹ This chapter has been submitted to Physics in Medicine and Biology: Rajon DA, Shah AP, Watchman CJ, Brindle J, and Bolch WE. submitted. A hyperboloid representation of the bone-marrow interface within 3D NMR images of trabecular bone: applications to skeletal dosimetry. Phys Med Biol: submitted.

the Marching-Cube (MC) algorithm, has been developed to transform the voxel-based interface into a surface represented by smooth hyperboloid surfaces. The technique has been used in [Chapter 7](#) to calculate chord-length distributions within bone samples. The voxel effects that affected these distribution measurements were reduced to an acceptable level. The goal of the present chapter is to explore whether this improved surface representation correspondingly minimizes errors in energy deposition to non-source target regions within trabecular bone.

Hyperboloid Marching-Cube Algorithm

The MC algorithm is a well-known visualization technique that transforms a gray-level image into a polygonal (i.e., triangulated) surface to represent the 3D object. The resulting surface can be processed and displayed on a computer monitor ([Düurst 1988](#); [Lorensen and Cline 1987](#); [Nielsen and Hamann 1991](#); [Treece et al. 1999](#); [Van Gelder and Wilhelms 1994](#); [Zhou et al. 1994](#); [Zhou et al. 1995](#)). In this study of trabecular bone, the MC algorithm uses the voxel intensities in an NMR microscopy image to calculate the best-estimate position of the bone-marrow interface. The interface is represented by an isosurface in the gray-level field of the image. The gray-level isovalue that corresponds to the isosurface must be predefined prior to running the algorithm by analyzing the image's gray-level histogram. Once the isovalue is determined, each segment that joins the center of two adjacent voxels is analyzed to determine if it crosses the isosurface. If it does, the exact location of the isosurface is calculated along the segment by linear interpolation of the voxel gray levels. Then, within each cube formed by the centers of eight adjacent voxels, the intersection points are connected to form polygons composed of individual triangles. The triangles placed side-by-side form a

closed surface that separates bone from marrow regions of the sample. An example of the isosurface produced by the MC algorithm is shown in [Figure 8-2](#). The result is a triangulated surface that no longer follows the rectangular shape of the voxels. Furthermore, the smoothness of the isosurface improves as the voxel size is reduced. [Chapter 7](#) showed that the surface area of the interface – whose overestimation was responsible for corresponding errors on absorbed-fraction (AF) calculations – was well preserved by the triangulated isosurface.

The MC algorithm transforms a binary image composed of voxels into one defined by triangles. A consequence of this transformation is an increase in the data size of the image. Each voxel on the isosurface is transformed into a set of triangles, with each triangle requiring 9 real numbers to store its 3 summits. The data size is multiplied by ~ 7 in the case of a typical NMR image of trabecular bone (see [Chapter 7](#)). Furthermore, the MC algorithm is subject to ambiguity problems ([Düurst 1988](#); [Nielson and Hamann 1991](#); [Treece et al. 1999](#); [Van-Gelder and Wilhelms 1994](#); [Zhou et al. 1994](#)). Solutions exist to correctly solve this problem ([Nielson and Hamann 1991](#); [Van-Gelder and Wilhelms 1994](#); [Wyvill et al. 1986](#)), but they require the defining of additional triangles that increases data storage requirements. In [Chapter 7](#), techniques have been developed to overcome both the ambiguity and data size problems. In this adaptation of the MC algorithm, the set of triangles generated within each marching cube is replaced by a hyperboloid surface whose equation is determined by trilinear interpolation of the gray levels of the eight cube vertices. Only the initial gray-level image is needed to generate the isosurface. Furthermore, the technique was shown to be simpler to implement than the original MC algorithm since only one equation is used per cube, in contrast to up to 12 non-coplanar

triangles needed with the triangulated-isosurface technique (i.e., the worst cube configuration when solving the ambiguity problem). The technique was successfully applied to chord-length distribution measurements through a mathematical model of trabecular bone. The chord distributions matched their expected shapes, with mean chord lengths being preserved within 1% for images at resolutions typical of NMR microscopy.

Because of the complexity of the equation of the hyperboloids, the surface area of the isosurface was not measured using the hyperboloid technique. Instead, it was assumed that the measurement would have closely approximated the result given for the triangulated isosurface. It was concluded that new hyperboloid MC (HMC) algorithm was well adapted for future use in coupling NMR images of trabecular bone with the Monte-Carlo radiation transport codes.

Material and Methods

To test the effectiveness of the HMC algorithm in solving the voxel-effect problem, the technique is first applied to a mathematical sample of trabecular bone similar to the one used in [Chapters 3, 4, and 6](#). Next, a real bone sample is used to compare the results obtained from the HMC technique with those obtained from the direct coupling of the NMR voxelized image. In both studies, absorbed fractions are assessed at various source electron energies, and S values are then calculated for selected low-energy beta emitters.

Revised Mathematical Model of Human Trabecular Bone

The mathematical model of trabecular bone initially developed to investigate voxel effects in [Chapter 3](#) consisted of a $1.6 \times 1.6 \times 1.6 \text{ cm}^3$ cube filled with 11,605 randomly-placed non-overlapping spheres each representing individual marrow cavities. Spaces between the spheres depict the bone trabeculae. The radius distribution of the

marrow spheres was chosen so that the chord-length distribution through the field of spheres approximated that of human cervical vertebrae as measured optically at the University of Leeds (Beddoe 1978; Whitwell 1973). Segmentation of the reference mathematical model is used to generate simulated digital images of trabecular bone at differing resolutions (see Chapter 3).

The mathematical bone sample of the current study is slightly different. First, to improve the calculation efficiency, the sample size was reduced to $1.2 \times 1.2 \times 1.2 \text{ cm}^3$. Second, in the original model, random placement of the marrow spheres did not preclude exceedingly small (i.e., tangential) sphere-to-sphere separations. This circumstance rarely happens in a human trabecular bone (i.e., extremely thin trabeculae). Furthermore, this artifact has important consequences on the topology of corresponding simulated NMR images where, even at low resolution, two neighboring marrow spheres could interconnect, greatly reducing the surface area of the bone-marrow interface. Since this surface area is the origin of voxel effects to be addressed, methods to eliminate it were implemented in the revised mathematical bone sample such that simulated bone trabecula thicknesses were never smaller than 2.5% of the radii of the marrow cavities that surround it. A cross-section of the revised mathematical bone sample is presented in Figure 8-3. A total of 4776 spheres are located within this new bone model – many of the smaller spheres do not appear, as they are smaller than the resolution of the image.

Nineteen digital images of the revised mathematical bone sample were then created simulating an imaging system with resolutions of $15 \mu\text{m}$ to $1000 \mu\text{m}$. In this process, a 3D grid is first placed over the mathematical bone sample to delineate the image voxels. For each voxel, the fraction of voxel volume inside the marrow spheres is calculated and

scaled across a gray level range of 0 to 255. The isovalue to construct the isosurface that separates bone and marrow is thus 127.5. A segmentation of the image into bone and marrow voxels is achieved by assigning all voxels with a gray level larger or equal to 128 to marrow, and all voxels with a gray level less or equal to 127 to bone. The characteristics of the 19 simulated images are presented in [Table 8-1](#). The marrow volume fraction of column 3 (Voxel Image) is calculated by counting all voxels with gray levels above the isovalue. The marrow volume fraction of column 4 (HMC Image) uses a Monte-Carlo technique to calculate the volume fraction enclosed by the isosurface. The equation of the hyperboloid is used to test if a point is located on the marrow side or on the bone side of the isosurface. The reference marrow volume fraction has been measured using a Monte-Carlo technique applied throughout the entire mathematical bone sample and is

$$V_{marrow} = 63.46^{\pm 0.05} \% . \quad (8-1)$$

In [Equation \(8-1\)](#), the error represents the statistical uncertainty of the Monte-Carlo technique. The C++ program that calculates the marrow volume fraction using the HMC method is listed in [Appendix H](#). It uses some of the tools listed in [Appendix B](#).

The surface area of column 5 (Voxel Image) is calculated by counting the voxel faces that separate a bone voxel from a marrow voxel and by multiplying the number by the area of the voxel face. For the surface area of column 6 (MC Image), we did not use the equation of the hyperboloid because of the complexity of the problem. Instead, we used the triangulated representation and summed the surface area of all the triangles to obtain the overall area of the isosurface (i.e., assuming that the hyperboloid surface area is close to the triangulated surface area). Note that the direct technique [referred to as DIR

in [Chapter 7](#)] was used to generate the triangles. As there is little difference between the direct and the reverse techniques, this choice will have little consequence in the present study. The Surface-area calculation is performed by the same C++ program that generates the triangle list and is thus listed in [Appendix E](#). The reference value of the surface area cannot be calculated accurately as some spheres are cut by the edges of the sample. Therefore, it is not possible to have a reference value for the surface area of the bone-marrow interface. As shown in [Chapter 7](#), the surface area of the triangulated isosurface converges to the correct value as the resolution is improved, whereas the surface area that follows the shape of the voxels converges to a 50% overestimation. In this study, we will estimate the reference value by graphical extrapolation of the convergence value of the triangulated surface to infinitely small voxel sizes. This method was used giving the result:

$$S_{\text{interface}} = 49.95^{\pm 0.05} \text{ cm}^2. \quad (8-2)$$

In [Equation \(8-2\)](#), the error reflects the uncertainty of the graphical estimate of the convergence value.

AF Calculations within the Mathematical Bone Model

Absorbed fractions of energy deposited within both bone trabeculae and marrow cavities were calculated for monoenergetic electron sources. Two Monte-Carlo experiments were conducted: one with the radiation source within the marrow cavities, and one with the radiation source within the bone trabeculae. As voxel effects only affect the cross-dose calculation, we were only interested in values of the cross-absorbed fraction: $f_{i,(\text{bone} \leftarrow \text{marrow})}$ and $f_{i,(\text{marrow} \leftarrow \text{bone})}$, where the index i represents the energy of the monoenergetic electrons. Since voxel effects only have consequences at low particle

energy (see [Chapters 3 and 4](#)), the highest energy considered in the study was 800 keV. The energy series, 25 keV to 800 keV, is presented in [Table 8-2](#) along with results to be detailed later.

Electron transport simulations were performed using the Monte-Carlo transport code EGSnrc ([Kawrakow 2000](#)). The two EGS4 user codes used in [Chapters 3 and 4](#), within the mathematical sample and the voxel-based images, were rewritten using EGSnrc, and will be hereafter be referred to as the “reference-code” and “voxel-code”, respectively. A third EGSnrc code was written implementing the HMC technique, and will be cited as the “HMC-code”. Note that the voxel-code uses the segmented images, whereas the HMC-code uses the gray level images since it interpolates the location of the threshold using the gray levels of two adjacent voxels. The numbers of histories were chosen to achieve a 95% confidence interval of $\pm 1\%$ for the cross-AF. The same numbers of histories were used for all three codes, since statistical fluctuations in the AF depend mostly on the electron energy and the geometry of the source, which is roughly equivalent in each case studied. The number of histories used is listed in [Table 8-2](#): column 2 and 3 for the source in marrow and source in bone, respectively. For each of the 15 energies considered, the AFs calculated with the voxel-code and with the HMC-code were compared with those calculated with the reference-code for each of the 19 voxel sizes. The three EGSnrc user codes have been developed to do the comparison. HMC-code is listed in [Appendix I](#) as well as the corresponding configuration file, an example of its input file, and an example of its output file.

Finally, tissue compositions for both adult cortical bone and active bone marrow (100% cellularity) were taken from ICRU (1992) Report 46. Tissue densities were assigned at 1.92 g cm^{-3} and 1.03 g cm^{-3} for bone and marrow, respectively.

S-Value Calculations within the Mathematical Bone Model

The absorbed fractions calculated with the three EGSnrc codes were then used to calculate S values (dose per unit cumulated activity) (Howell 1994; Loevinger et al. 1991) for four low-energy beta-emitting radionuclides chosen for their interest in skeletal dosimetry (^{131}I , ^{33}P , ^{153}Sm , and $^{117\text{m}}\text{Sn}$). Table 8-3 shows the radiological characteristics of the four radionuclides selected along with results to be discussed later. S values were calculated using the decay data from Eckerman et al. (1993) and the MIRD schema. Both marrow and bone sources of beta particles and electrons were considered, even though some radionuclides may be associated with radiochemicals that exclusively localize on bone surfaces (e.g., ^{153}Sm and $^{117\text{m}}\text{Sn}$). The masses of both the bone and marrow targets were calculated using the marrow volume fractions of Table 8-1 and the mass densities given by ICRU (1992) Report 46. S values calculated from the AFs measured with both the voxel-code and HMC-code were compared to those derived using AFs given by the reference-code.

Application to NMR Microscopy Images of Human Trabecular Bone

To estimate the effects of the new HMC technique on a real bone sample, AF calculations were performed using a NMR microscopy image of a human L₄ lumbar vertebra. The sample was harvested from a 51-year male cadaver who died in a cardiovascular accident. A piece of trabecular bone $1.5 \times 1.5 \times 2.5 \text{ cm}^3$ in size was sectioned from L₄ vertebral body. The bone marrow was digested using a 5.25% sodium

hypochlorite solution and the marrow cavities were filled with Gd-doped water. The sample was then imaged at 4.7 T using a conventional 3D spin-echo sequence over 14 h. The resulting 256 x 256 x 512 voxel image had a cubical voxel resolution of 88 μm . A region of interest of 102 x 241 x 79 voxels was then selected for subsequent image analysis as shown in [Figure 8-4-A](#). A median filter was applied to reduce image noise as shown in [Figure 8-4-B](#). Next, the gray-level histogram was analyzed to determine a threshold value between bone and marrow using methods described previously by Jokisch et al. (1998). The threshold value found for this image was 13.5 on the 0-255 gray-level scale. The image was then segmented for use with the voxel code as shown in [Figure 8-4-C](#). This bone sample is the same one used to extract the small trabecular regions displayed in both [Figures 8-1](#) and [8-2](#).

Absorbed fractions of energy were calculated within the image using the EGSnrc Monte-Carlo transport code. Both the voxel-code and the HMC-code were used with the same 15 energies used for the mathematical sample. Note that the voxel-code uses the segmented image ([Figure 8-4-C](#)), while the HMC-code uses the filtered image ([Figure 8-4-B](#)). The two sources – bone and marrow – were simulated and the number of histories was the same as used previously. The results obtained from the voxel-code were then compared with the results obtained from the HMC-code. S values were also calculated for comparison using the same radionuclides discussed in the previous section. Target masses were calculated from the marrow volume fraction within the sample and the densities provided by ICRU (1992) Report 46. They are $m_{\text{marrow}} = 0.958$ g and $m_{\text{bone}} = 0.746$ g for marrow and bone, respectively.

Results and Discussion

In both this and the next sections, the cumulative surface areas of all bone voxel surfaces that adjoin the neighboring marrow cavities will be referred to as the “voxel representation” of the bone-marrow interface. Similarly, the surface that follows the triangles generated by the MC algorithm and the surface that follows the hyperboloid surfaces generated by the HMC algorithm will be referred to as the “MC representation” and “HMC representation”, respectively, of that same bone-marrow interface.

Volume Fraction and Surface Area

Marrow volume fraction as measured using the voxel representation and the HMC representation of the mathematical bone sample were compared with the reference value. The results are shown in [Figure 8-5-A](#). Both the voxel and the HMC representations converge to the reference value as the voxel size is reduced. The voxel representation converges faster but both remain within 2% at voxel sizes below 250 μm . At large voxel sizes, the marrow volume fraction is overestimated as marrow cavities are artificially interconnected.

[Figure 8-5-B](#) shows the same comparison for the surface area of the interface. As explained previously, the MC representation is used instead of the HMC representation, assuming that both techniques would give the same result. The results of [Figure 8-5-B](#) clearly confirm that the voxel representation converges to a value that is 50% larger than the reference value, whereas the MC representation correctly converges to the true value. At poor image resolution (large voxel sizes), both tend toward zero, as the volume fraction is no longer preserved. These results are in agreement with the results from

Chapter 7, and provide justification for the use of the MC techniques in skeletal transport models.

Absorbed Fractions of Energy

The AF of energy was calculated for the 15 different energies listed in Table 8-2 and for both marrow and bone trabecula radiation sources. First, the reference-code was used to calculate reference values of the cross-absorbed fractions $f_{(\text{bone} \leftarrow \text{marrow})}$ and $f_{(\text{marrow} \leftarrow \text{bone})}$ within the mathematical sample. The results are listed in columns 4 and 5 of Table 8-2, respectively. At low energies, the AF is very small as particles are absorbed primarily in their respective source region. At intermediate energies, the AF increases and approaches a constant value. At even higher energies (approaching 800 keV), they are expected to decrease as more and more energy escapes at the sample boundaries.

Second, the voxel-code was executed for each of the 15 energies and for each of the 19 images of the mathematical bone sample. Figures 8-6-A and 8-6-B show the evolution of the relative error for $f_{(\text{bone} \leftarrow \text{marrow})}$ and $f_{(\text{marrow} \leftarrow \text{bone})}$, respectively, as a function of the voxel size (for clarity, only 7 are shown). The relative errors are calculated using the expression:

$$\Delta E r_f = \left(\frac{f_{\text{vox}} - f_{\text{ref}}}{f_{\text{ref}}} \right) \times 100\% , \quad (8-3)$$

where f_{vox} is the absorbed fraction calculated using one of the voxel representations of the bone-marrow interface and f_{ref} is the absorbed fraction calculated using the reference-code.

The two plots are fairly identical as the voxel effects are symmetric for cross-region irradiations. The results confirm conclusions made in Chapters 3 and 4. First, at high

particle energies, $f_{(\text{bone} \leftarrow \text{marrow})}$ and $f_{(\text{marrow} \leftarrow \text{bone})}$ follow the curves of the volume fraction of the target. Second, at low particle energies, the curves are influenced by the surface area of the bone-marrow interface. At large voxel sizes (above 300 μm), the surface area is underestimated and so is the AF. As the resolution is improved, the voxel-code overestimates the AF by almost 40% at 50 keV. At very high image resolution (small voxel sizes), the low-energy curves converge to zero, but the convergence occurs at voxel sizes well below those obtainable in standard 3D NMR microscopy. Note that error bars are not provided in [Figure 8-6](#), as coefficients of variation on the AF are held to 1% and less.

Third, the HMC-code was executed with the same series of energies and image resolutions. The results are given in [Figures 8-7-A](#) and [8-7-B](#) showing the evolution of the relative error for both $f_{(\text{bone} \leftarrow \text{marrow})}$ and $f_{(\text{marrow} \leftarrow \text{bone})}$, respectively, as a function of the voxel size. Relative errors are calculated using the expression:

$$\Delta E r_f = \left(\frac{f_{HMC} - f_{ref}}{f_{ref}} \right) \times 100\%, \quad (8-4)$$

where f_{HMC} is the absorbed fraction calculated using the HMC representation of the interface for each of the 19 images.

At high electron energies (500 keV and above), the curves of [Figure 8-7](#) are similar to corresponding curves shown in [Figure 8-6](#) as the AF continues to track with the volume fraction of the target. At low electron energies, however, distinct differences are noted between the voxel and HMC representations of the bone-marrow interface. First, at large voxel sizes (above 400 μm), the AF curves in [Figure 8-7-B](#) increase with increased voxel size, whereas they decrease in [Figure 8-6-B](#). One can notice in [Figure 8-5-A](#) that the

marrow volume fractions are also different between the two models at large voxel sizes. At 400 μm , the HMC volume fraction is largely overestimated, whereas the voxel volume fraction is close to its reference value. Therefore, in [Figure 8-7-B](#), the volume fraction influence is stronger than in [Figure 8-6-B](#). This explains why the AF (at low particle energies) continues to decrease with increased voxel size in [Figure 8-6-B](#) (influenced only by the surface area) whereas it first decreases to around 100 μm and then increases above 400 μm in [Figure 8-7-B](#), because the volume fraction influence overcomes the surface area influence. In [Figure 8-7-A](#), the same behavior is not seen because both influences – surface area and volume fraction of the target – work in the same direction to underestimate the AF. This is why, at low particle energies and large voxel sizes, the curves of [Figures 8-7-A](#) and [8-6-A](#) appear very similar.

The second consequence for low-energy particles is that the HMC calculation no longer overestimates the AF at intermediate voxel sizes since the HMC algorithm preserves the surface area of the bone-marrow interface. This is the more important result of the study: that the HMC representation allows the AF for cross-irradiations to converge to its reference value as the image resolution is improved. This convergence is within 10% below 150 μm and within 3% below 60 μm .

Radionuclide S Values

The AF results discussed above were used to calculate the S values for the four radionuclides listed in [Table 8-3](#). First, the reference AFs were used to calculate the reference S values $S_{(\text{bone}\leftarrow\text{marrow})}$ and $S_{(\text{marrow}\leftarrow\text{bone})}$ within the mathematical bone sample. The results are given in columns 6 and 7 of [Table 8-3](#), respectively.

Second, the AFs calculated from the voxel-code were used to calculate the voxel S values for each radionuclide. The results are compared with the reference S values. [Figures 8-8-A](#) and [8-8-B](#) show the evolution of the relative error for $S_{(\text{bone} \leftarrow \text{marrow})}$ and $S_{(\text{marrow} \leftarrow \text{bone})}$, respectively, as a function of voxel size. The relative errors are calculated using the expression

$$\Delta Er_S = \left(\frac{S_{\text{vox}} - S_{\text{ref}}}{S_{\text{ref}}} \right) \times 100\% , \quad (8-5)$$

where S_{vox} is the S value calculated using AFs calculated from the voxel representation of the interface within each of the 19 simulated images, and S_{ref} is the S value calculated using the reference AFs.

For both ^{131}I and ^{153}Sm , the convergence is good in both [Figures 8-8-A](#) and [8-8-B](#) since a sizable portion of the beta spectrum is above 200 keV. These high-energy particles have an AF that is in a good agreement with the reference value and they contribute most of the energy deposition by the radionuclide within the target region. For very low energy emitters, these discrepancies will have a larger consequence as can be seen for ^{33}P and $^{117\text{m}}\text{Sn}$ in both [Figures 8-8-A](#) and [8-8-B](#). The relative error is about 10% at 150 μm and is about 7% at 75 μm . At large voxel sizes, all S values decrease as all low-energy AFs decrease with increasing voxel as shown in [Figure 8-6](#).

Third, the AFs calculated from the HMC-code were used to calculate the HMC S values for each radionuclide. The results are again compared with the reference S values. [Figures 8-9-A](#) and [8-9-B](#) show the evolution of the relative error for both $S_{(\text{bone} \leftarrow \text{marrow})}$ and $S_{(\text{marrow} \leftarrow \text{bone})}$, respectively, as a function of the voxel size. The relative errors are calculated using the expression

$$\Delta Er_S = \left(\frac{S_{HMC} - S_{ref}}{S_{ref}} \right) \times 100\% , \quad (8-6)$$

where S_{HMC} is the S value calculated using the AFs calculated from the HMC representation of the bone-marrow interface with each of the 19 simulated images.

At small voxel sizes, all four radionuclides show a convergence of their S values to their corresponding reference values. The error is less than 1% at resolutions better than 75 μm . At larger voxel sizes (between 75 μm and 400 μm), ^{33}P and $^{117\text{m}}\text{Sn}$ undergo a drop that is a consequence of the similar drop for the low-energy AFs in both [Figures 8-7-A](#) and [8-7-B](#). At even larger voxel sizes, the ^{33}P and $^{117\text{m}}\text{Sn}$ S-value curves both increase again. This is expected for $S_{(\text{marrow} \leftarrow \text{bone})}$ in [Figure 8-9-B](#) since the corresponding AFs also increase with increasing voxel size ([Figure 8-7-B](#)), but not for $S_{(\text{bone} \leftarrow \text{marrow})}$ in [Figure 8-9-A](#) for which the corresponding AFs all decrease at large voxel sizes ([Figure 8-7-A](#)). The explanation of this mismatch between [Figures 8-7-A](#) and [8-9-A](#) comes from the repartition of the volume fraction between bone and marrow. As seen in [Figure 8-5-A](#), the bone volume fraction is underestimated above 250 μm , whereas it drops dramatically to zero at very large voxel sizes (not shown in [Figure 8-5-A](#)). At 750 μm , the bone volume fraction has been reduced by a factor 8. Therefore, the S value is multiplied by this same factor. This is enough to overwhelm the decrease in the AFs of [Figure 8-7-A](#). On the other hand, the marrow volume fraction is overestimated at large voxel sizes, but the overestimation is limited to 100% of the size of the sample, which is about 1.6 times the reference value for the marrow volume fraction. Therefore, the S values of [Figure 8-9-B](#) can only be divided by less than 1.6 at large voxel sizes. This is not enough to overwhelm the increase of the AFs shown in [Figure 8-7-B](#). These same

features should also apply to [Figure 8-8-A](#), but [Figure 8-5-A](#) shows that the volume fraction discrepancy occurs at a larger voxel size in the case of the voxel representation of the bone-marrow interface than for its representation by the HMC algorithm. Consequently, the influence of the AFs is stronger than the influence of the volume fraction and the data of [Figure 8-8](#) follows from the data of [Figure 8-6](#).

We further note that the results obtained with the voxel-code are different than the ones from [Chapter 3](#). In this earlier study, the overestimation of the S value in bone when irradiated by sources in marrow was about 5% for ^{131}I and ^{153}Sm and about 25% for ^{33}P and $^{117\text{m}}\text{Sn}$; they are only 1% and 10%, respectively, in the present chapter. These differences can be attributed to two major changes on the study. First, the revised mathematical bone sample has different features than the earlier model. A different marrow volume fraction and a different convergence of the surface area of the bone-marrow interface (because of the larger thickness of the bone trabeculae) can influence the results. Second and more importantly, the current study was performed using the EGSnrc electron transport code, whereas the previous study used EGS4. EGSnrc offers important improvements in the transport of low-energy electrons over that provided in EGS4, and our study is mostly concerned with low-energy electrons. Furthermore, the EGSnrc code also improves electron transport at boundaries and our model is focused on energy deposition events at region boundaries. For 50 keV electrons, a 65% overestimation of the AFs was found in [Chapter 3](#) for a mathematical model made of spheres, whereas the surface area overestimation was less than 50%. It was shown in [Chapter 4](#) that the overestimation of the AF should be at most equal to the overestimation of the surface area: that is 50% in the case of spherical marrow cavities. The additional

15% of the 50 keV electron results had not been explained in [Chapter 3](#). In the current study, using EGSnrc, the AF relative error was always under 50%, even for the 25 keV electrons (results not shown in [Figure 8-6](#)). Consequently, the additional 15% is assumed to be a consequence of the less-than-optimal low-energy electron transport by EGS4.

Application to NMR Microscopy Images of Human Trabecular Bone

Both the voxel-code and the HMC-code were used to transport electrons within the NMR image of a real sample of human trabecular bone. [Figures 8-10-A](#) and [8-10-B](#) compare the results for $f_{(\text{bone} \leftarrow \text{marrow})}$ and $f_{(\text{marrow} \leftarrow \text{bone})}$, respectively, as a function of the electron energy. One can see that the difference between the two representations is more important at low energy, as predicted by the study of the mathematical model. To measure the relative difference between the two series of results, the results obtained with the HMC-code were used as the reference and the relative error was calculated for the voxel-code results using the expression:

$$\Delta Er_f = \left(\frac{f_{\text{vox}} - f_{\text{HMC}}}{f_{\text{HMC}}} \right) \times 100\% . \quad (8-7)$$

These relative differences are presented in [Figure 8-11](#). They show that, at low electron energies, the voxel-code overestimates the AF by almost 50% as compared to results using the HMC-code. At higher particle energies, both codes give similar results. [Figure 8-11](#) can also be compared with [Figure 8-6](#), considering that the image voxel size was 88 μm . The calculation performed within the real bone image is in a good agreement with the calculation performed for the mathematical sample. They both overestimate the AF calculated with the voxel-code for low-energy electrons.

The S values were also calculated for the real bone sample. The results are presented in [Table 8-4](#) for the same 4 radionuclides used with the mathematical model. Columns 2 and 3 give $S_{(\text{bone} \leftarrow \text{marrow})}$ for the voxel-code and the HMC-code, respectively. Column 4 shows the relative difference of the voxel-code using the HMC-code results as reference. Columns 5, 6, and 7 give similar data for $S_{(\text{marrow} \leftarrow \text{bone})}$. First, one can see that there is little difference between the S values calculated within the mathematical sample and the ones calculated within the real bone sample. For this, one can compare column 3 of [Table 8-4](#) with column 6 of [Table 8-3](#) or column 6 of [Table 8-4](#) with column 7 of [Table 8-3](#). This agreement gives confidence in the use of the mathematical bone sample for studying voxel effects representative of those present in images of real human bone. Second, the overestimates of the results by the voxel-code within the real bone sample agree with corresponding overestimates seen within the mathematical model. The overestimation of the cross-region S value is important for ^{33}P and for $^{117\text{m}}\text{Sn}$ (very low-energy beta emitters), yet is moderate for higher-energy emitters such as ^{131}I and ^{153}Sm . These comparisons between the results of the voxel-code and those of the HMC-code within the real bone sample confirm that the HMC-code provides an important reduction of interface voxel effects, as it has shown to do within the mathematical model of trabecular bone.

Conclusion

The Marching-Cube algorithm was adapted and applied to NMR microscopy images of trabecular bone samples. This adaptation allows modeling the bone-marrow interface by hyperboloid surfaces at the voxel level. The surfaces, joined together, form an isosurface in the gray-level field of the image that is a better representation of the

bone-marrow interface than a surface following the individual faces of image voxels at bone surfaces. This HMC representation was used to transport monoenergetic electrons within images of a mathematical bone sample with results compared to reference values. [Figures 8-7](#) and [8-9](#) demonstrate that both calculations of the AF and radionuclide S values converge to their respective reference values as the image resolution is improved (reduction of the voxel size). Below 75 μm , the S value is calculated with an error less than 1% even for low-energy beta emitters such as ^{33}P and $^{117\text{m}}\text{Sn}$. The voxel representation of the bone-marrow interface was also used to transport the same monoenergetic electrons in order to be compared with the HMC representation. Comparing [Figures 8-6](#) with [Figure 8-7](#) on one hand, and [Figure 8-8](#) with [Figure 8-9](#), show that the HMC representation is an improvement over the voxel representation and that it reduces significantly voxel effects when transporting particles within 3D images of bone samples.

A real bone sample was also used to compare the two representations and confirm the results obtained with the mathematical bone sample. The HMC algorithm provides a good representation of the bone-marrow interface that reduces the AF and S-value overestimations due to the voxel effects. The difference between the results of the voxel and the HMC representation is on the same order than when the two codes are applied to the mathematical sample. Therefore, the application of the HMC algorithm should be used as a representation of the bone-marrow interface within images of trabecular bone samples when coupled with a Monte-Carlo transport code.

The development of the technique described here was motivated by the large values of the AF relative errors seen when the transport is made within an image

with voxel sizes on the same order as the electron range (see [Chapters 3 and 4](#)). For bone-marrow dosimetry via NMR microscopy, the image resolution used is between 50 μm and 100 μm , which correspond to a significant voxel effect for energies up to 150 keV ([Figure 8-6](#)). At this energy, the electron range is about 3 times the resolution of the image. In [Chapter 4](#), using single sphere models, it was concluded that the voxel effect covered a voxel-size range that will evolved has a function of the electron range within the source of radiation. Therefore, for larger voxel sizes that the ones used in bone-marrow dosimetry, the voxel effects are expected to be significant at higher energies. Anthropomorphic models of the human body are frequently used to assess the dosimetric parameters at the organ level ([Poston et al. 2002](#)). New research use voxel-based models acquired through CT or NMR imaging to determine the organ geometry ([Chao et al. 2001a](#); [Chao et al. 2001b](#); [Xu et al. 2000](#); [Zankl and Wittmann 2001](#)). The voxel size used for these images is on the order of 0.5 to 1 mm. Applying thus “3 times” rule of thumb deduced above, the voxel effects are expected to be significant for electron ranges up to 3 mm. That is for energies up to 700 keV. Therefore, consequences similar to the ones described here should be expected for cross-doses over organs next to each other for electron energy up to 700 keV. The application of the HMC algorithm would be a solution to solve these problems.

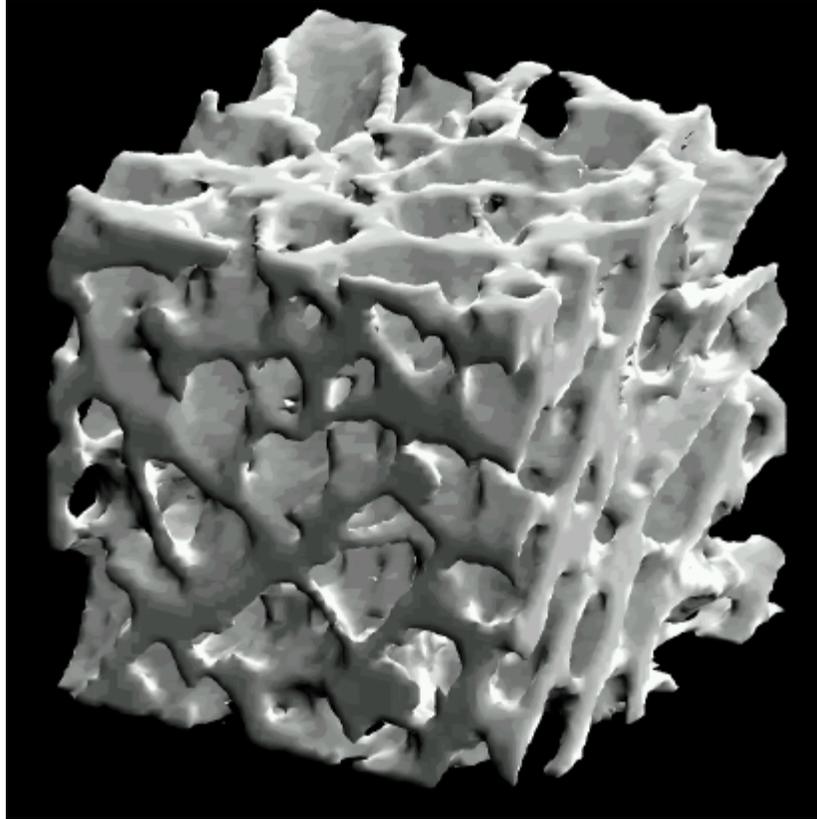


Figure 8-1. Cubical sample of a human lumbar vertebra. The 3D visualization is a reconstruction from a 3D NMR image obtained at 200-MHz proton resonance frequency (4.7 tesla). The image was taken over an 11 hour and 10 minute acquisition time (TR = 600 ms, TE = 9.1 ms, spectral width: 123,457 Hz, 2 averages, matrix: 512 x 256 x 256, field of view: 4.5 x 2.25 x 2.25 cm³). The sample size is 5.6 x 5.6 x 5.6 mm³. The image resolution is 88 x 88 x 88 μm^3 .

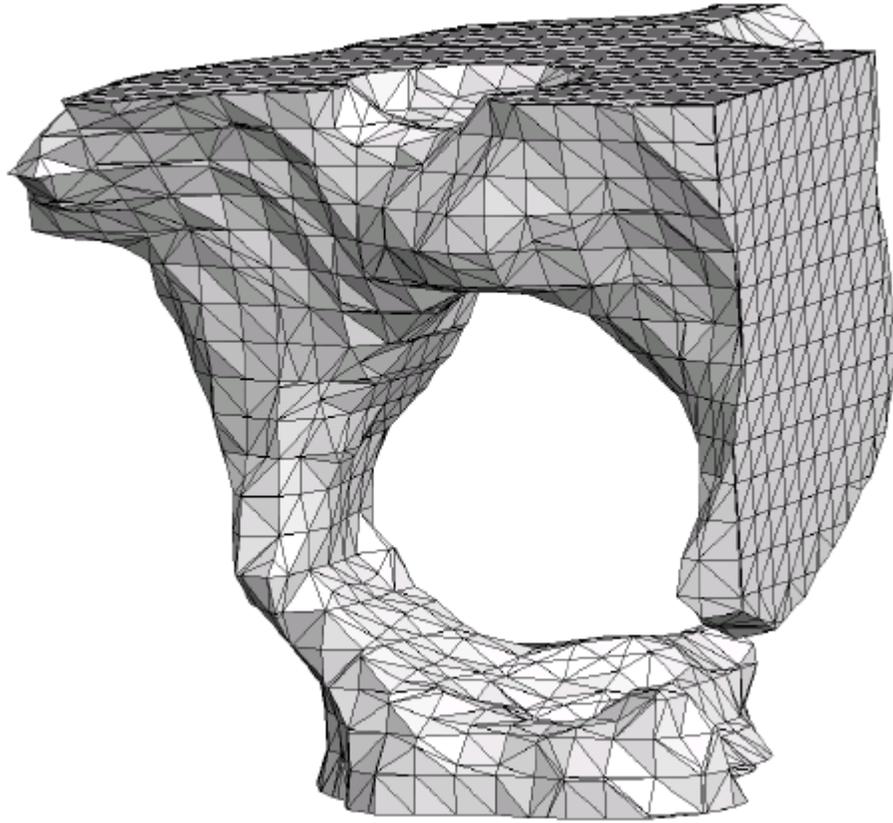


Figure 8-2. 3D representation of the triangulated bone-marrow interface obtained from the MC algorithm. The initial image is a 22 x 22 x 22 voxel segment of bone extracted from the lumbar vertebra of [Figure 8-1](#).

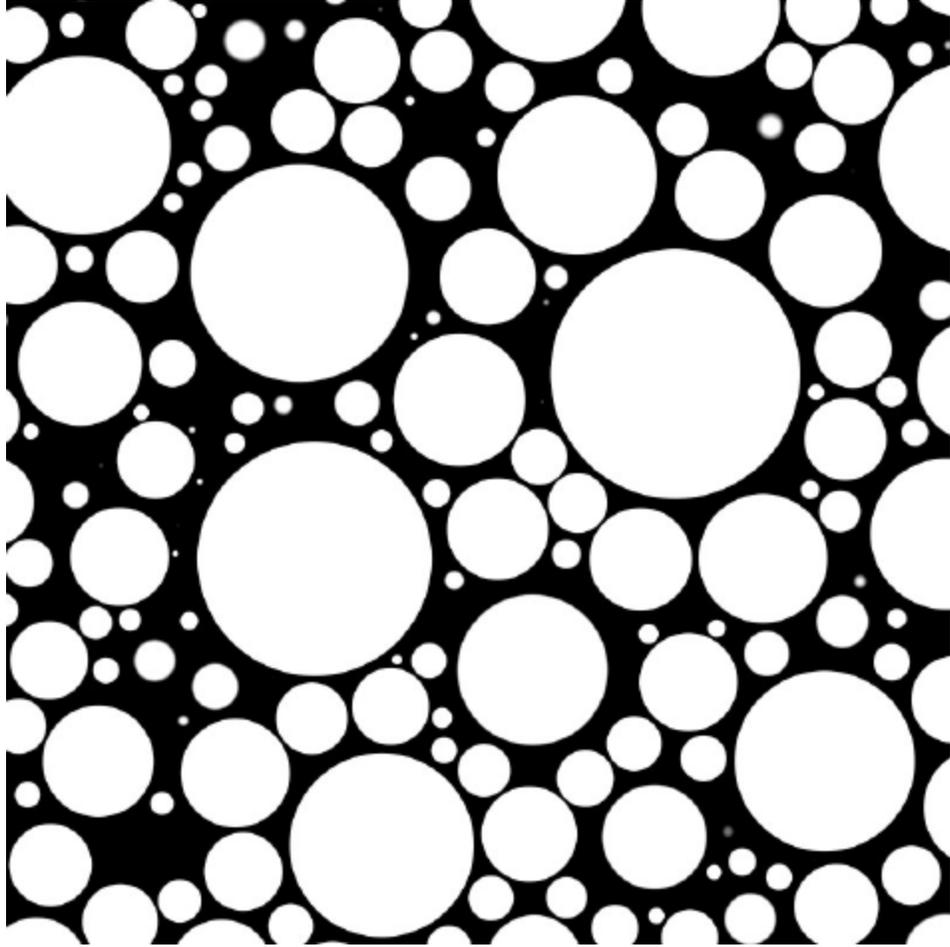


Figure 8-3. Transverse slice, $1.2 \times 1.2 \text{ cm}^2$, through the revised mathematical sample of trabecular bone. The spheres represent the marrow cavities with the spaces between them depicting the bone trabeculae.

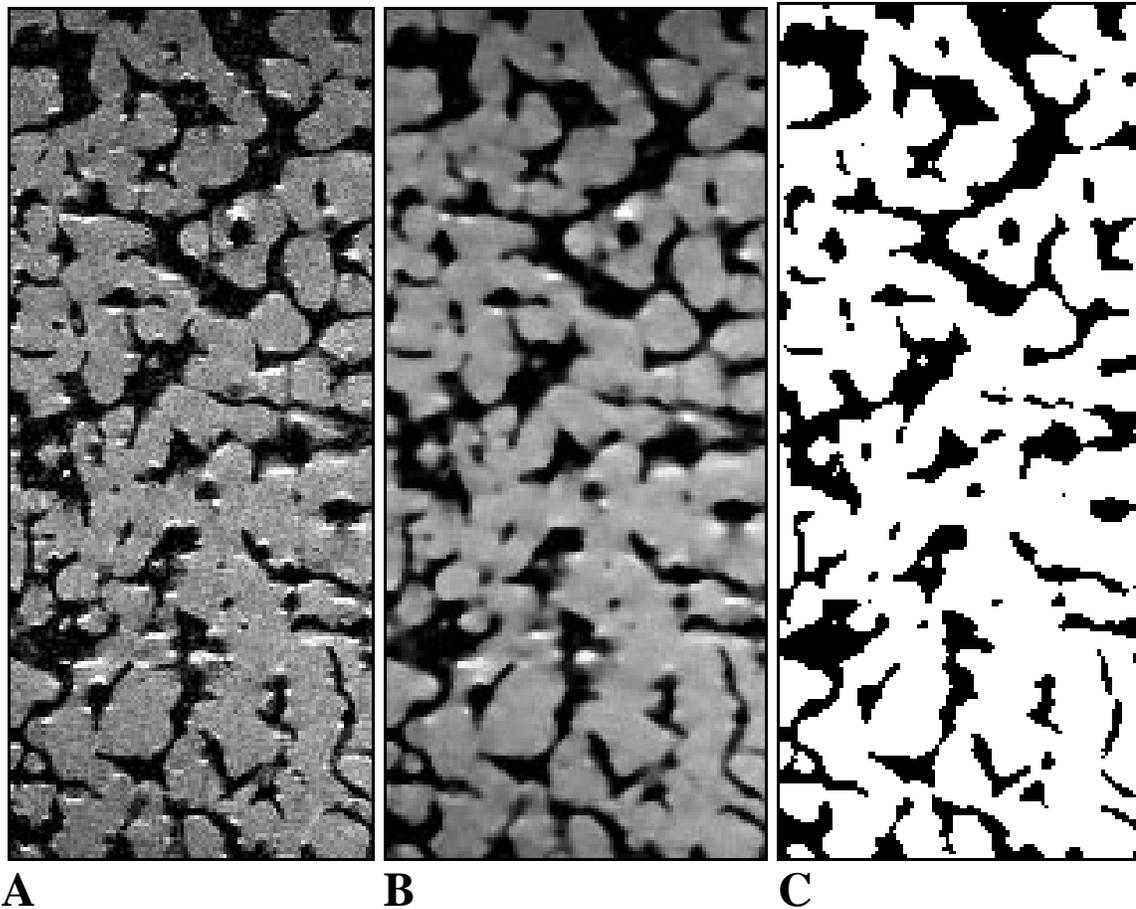


Figure 8-4. Transverse slice, $0.896 \times 2.12 \text{ cm}^2$, through a NMR microscopy image of human trabecular bone. The image was taken over an 11 hour and 10 minute acquisition time (TR = 600 ms, TE = 9.1 ms, spectral width: 123,457 Hz, 2 averages, matrix: $512 \times 256 \times 256$, field of view: $4.5 \times 2.25 \times 2.25 \text{ cm}^3$). The sample comes from the body of a L_3 lumbar vertebra. The image resolution is $88 \times 88 \times 88 \mu\text{m}^3$. A) Original NMR image. B) Image after application of a median filter. C) Image after binary segmentation. On both A) and B) susceptibility artifact is visible (the white spots). This artifact is not accounted for in the current study since it does not contribute to voxel effects. It will be investigated in future study.

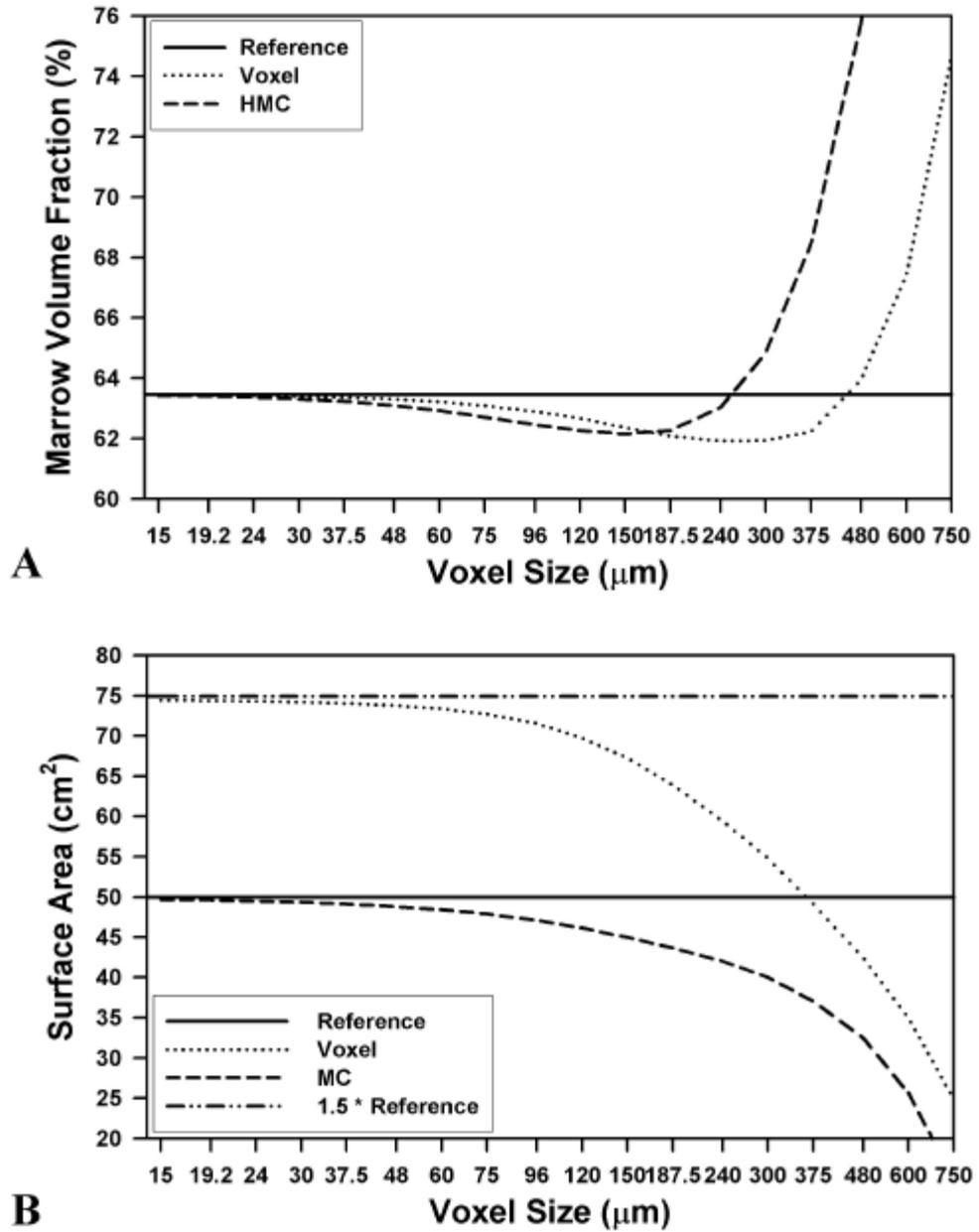


Figure 8-5. Geometrical parameters as a function of image resolution for the revised mathematical sample. A) Marrow volume fraction. B) Surface area of the bone-marrow interface. The horizontal solid lines indicate reference values.

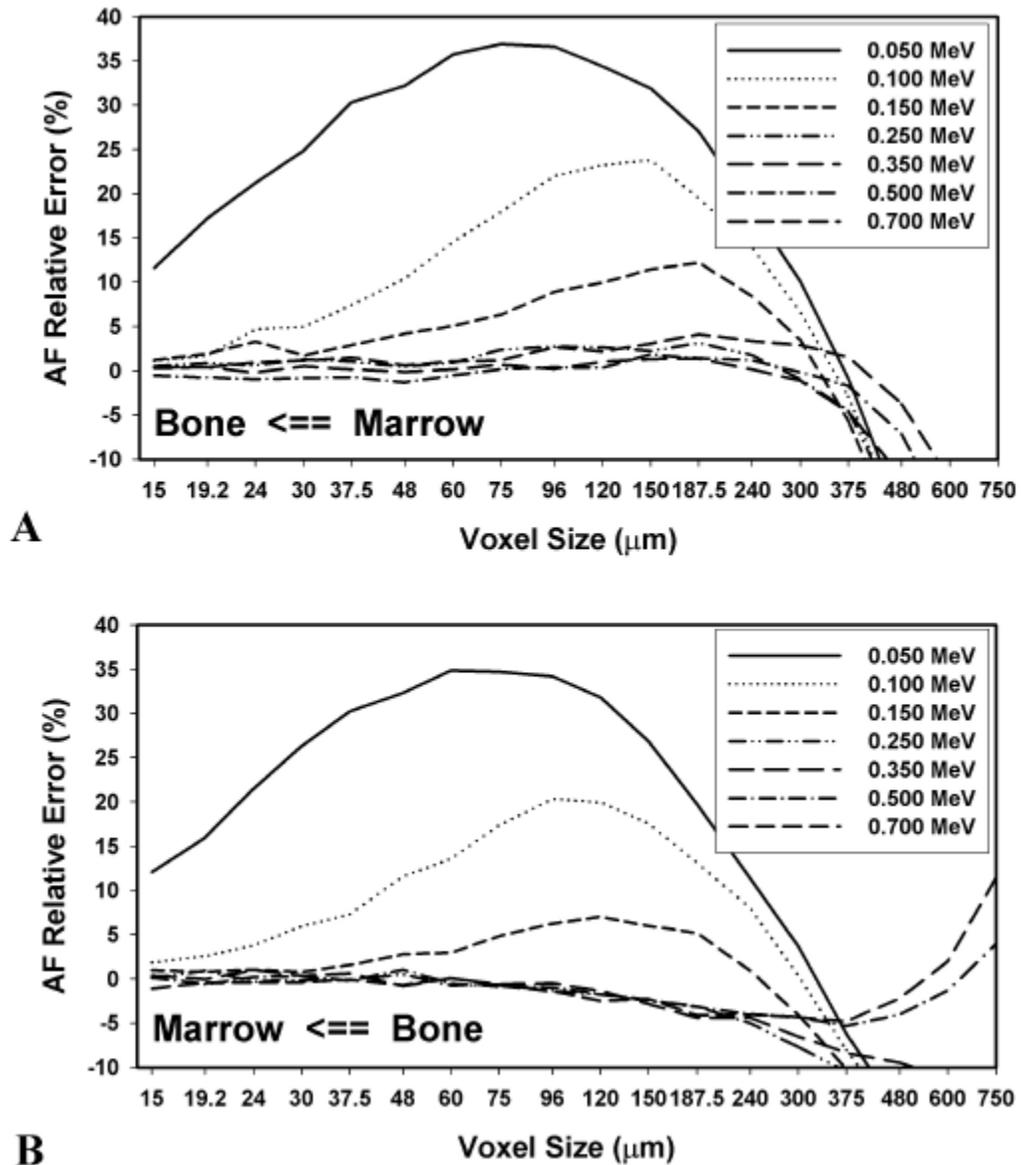


Figure 8-6. Relative error of the absorbed fractions calculated for simulated images of the mathematical sample as a function of the voxel size. The particles are transported across the voxel representation of the bone-marrow interface. A) AF in bone when irradiated by sources in marrow. B) AF in marrow when irradiated by sources in bone.

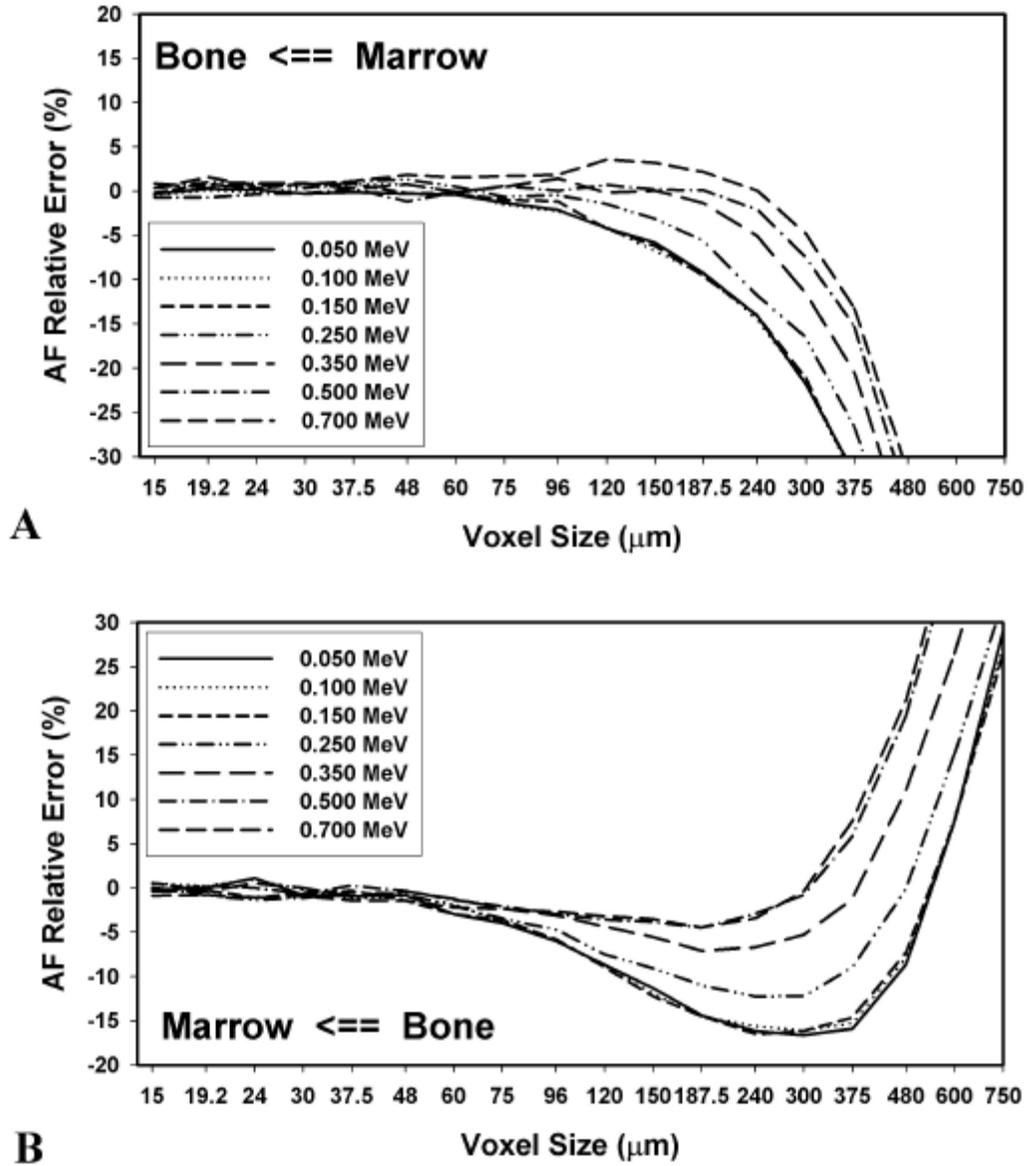


Figure 8-7. Relative error of the absorbed fractions calculated for simulated images of the mathematical sample as a function of the voxel size. The particles are transported across the HMC representation of the bone-marrow interface. A) AF in bone when irradiated by sources in marrow. B) AF in marrow when irradiated by sources in bone.

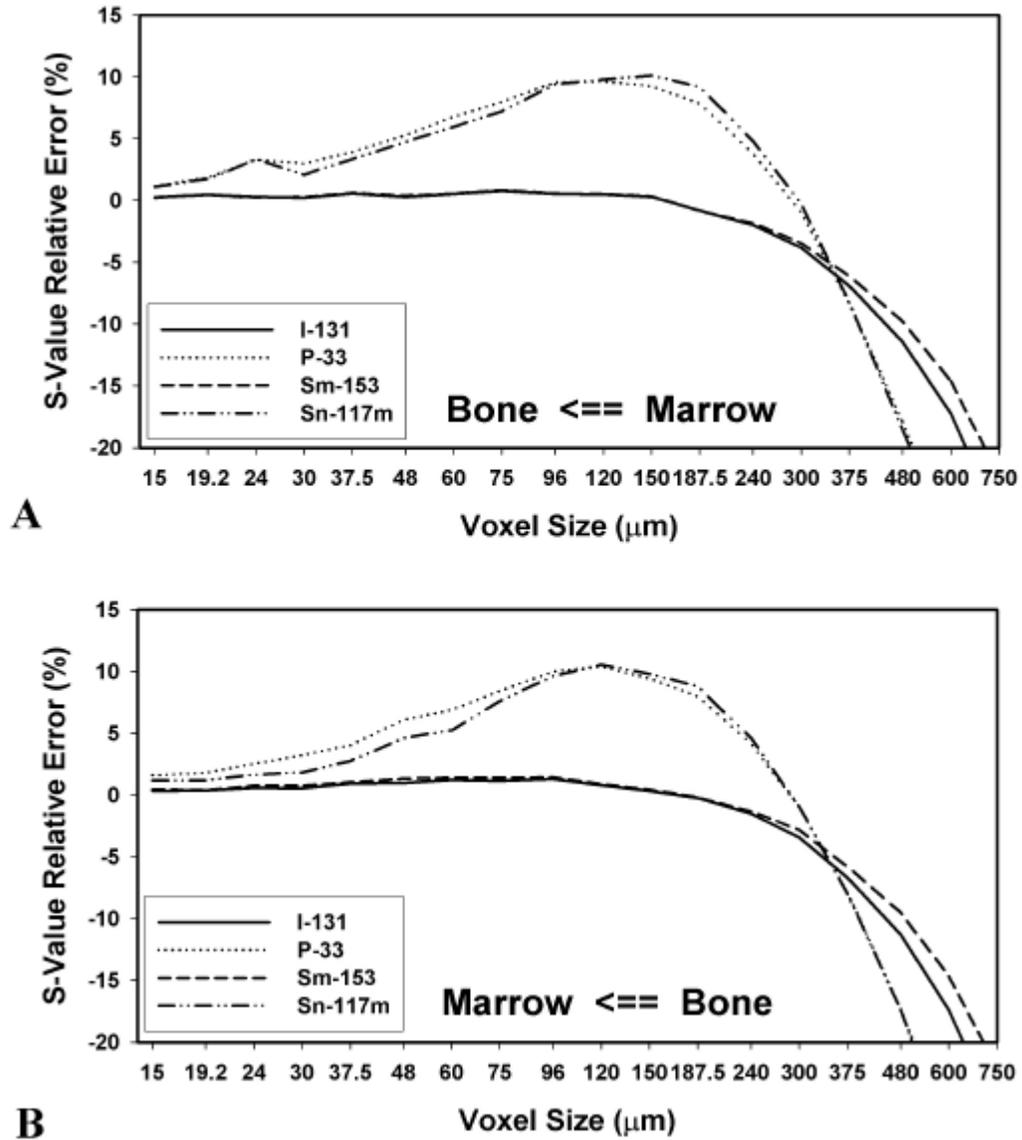


Figure 8-8. Relative error of the S values calculated for four radionuclides of interest in skeletal dosimetry as a function of the voxel size. The calculation uses AFs obtained within the voxel representation of the bone-marrow interface. A) S value in bone when irradiated by sources in marrow. B) S value in marrow when irradiated by sources in bone.

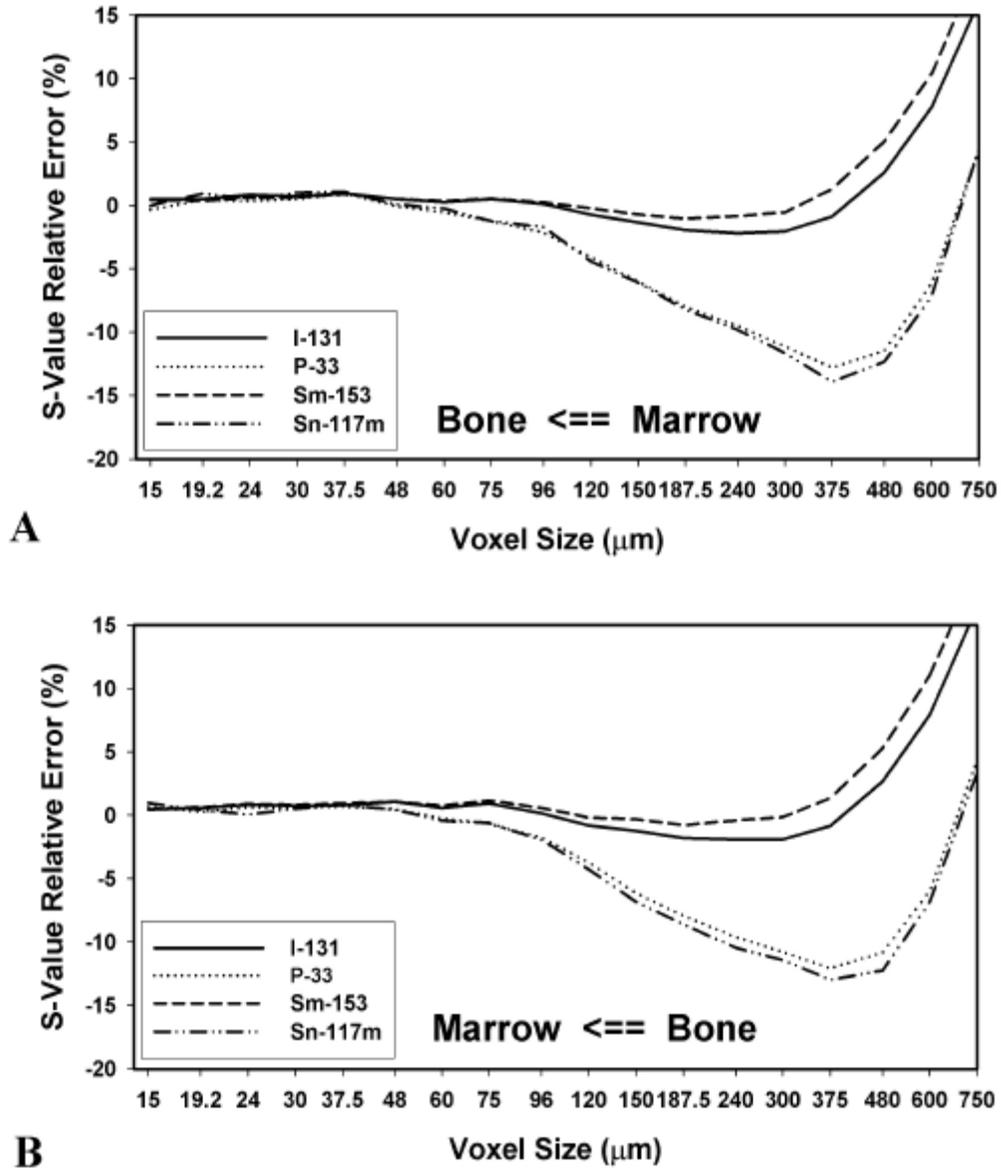


Figure 8-9. Relative error of the S values calculated for four radionuclides of interest for skeletal dosimetry as a function of the voxel size. The calculation is performed using AFs calculated within the image defined by the HMC representation of the bone-marrow interface. A) S value in bone when irradiated by sources in marrow. B) S value in marrow when irradiated by sources in bone.

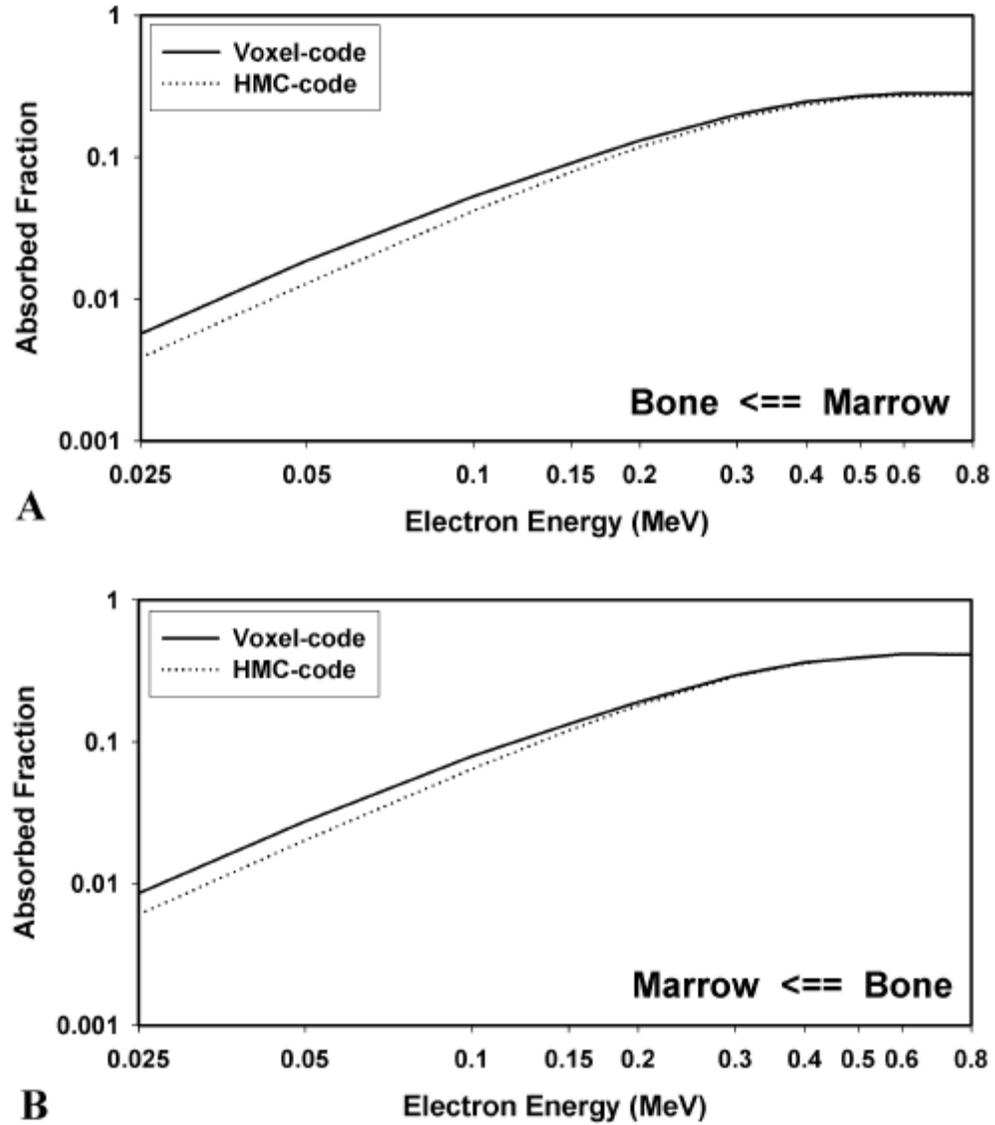


Figure 8-10. Absorbed fraction: voxel code compared to HMC code. The calculation is done within a NMR image of a real bone sample as a function of the electron energy. A) AF in bone when irradiated by marrow. B) AF in marrow when irradiated by bone.

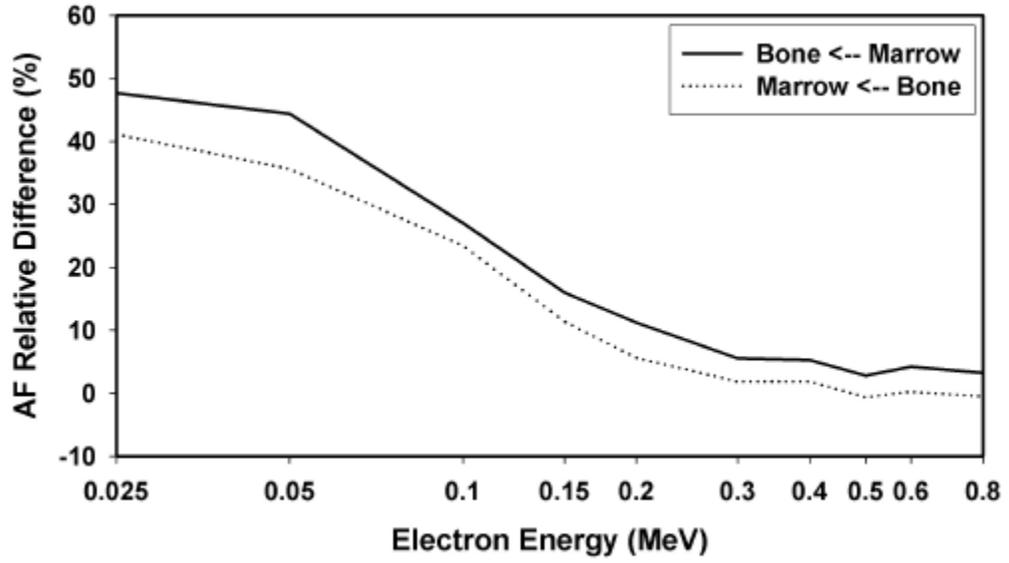


Figure 8-11. Relative error of the AF calculated with the voxel-code. The results from the HMC-code are assumed to be the reference values.

Table 8-1. Characteristics of the 19 segmented images of the mathematical bone sample. The voxel sizes range from 15 μm to 1000 μm . The marrow volume fraction was calculated within both the voxel image and the image defined using the HMC algorithm. The surface area was calculated within the voxel image and the image defined using the MC algorithm (triangulated surfaces). The isovalue used for both the voxel image and the MC algorithm was 127.5 on the 0-255 gray-level scale.

Image size (# of voxels)	Voxel size (μm)	Marrow volume fraction - Voxel (%)	Marrow volume fraction - HMC (%)	Surface area - Voxel (cm^2)	Surface area - MC (cm^2)
800	15.0	63.43	63.42	74.44	49.68
625	19.2	63.42	63.40	74.38	49.59
500	24.0	63.41	63.36	74.30	49.48
400	30.0	63.39	63.31	74.19	49.32
320	37.5	63.36	63.23	74.05	49.12
250	48.0	63.30	63.10	73.78	48.80
200	60.0	63.22	62.92	73.38	48.41
160	75.0	63.09	62.71	72.72	47.89
125	96.0	62.90	62.46	71.55	47.10
100	120.0	62.67	62.26	69.74	46.14
80	150.0	62.36	62.15	67.26	44.97
64	187.5	62.08	62.27	63.90	43.65
50	240.0	61.92	63.03	59.39	42.02
40	300.0	61.94	64.84	54.81	40.02
32	375.0	62.22	68.44	49.15	37.05
25	480.0	63.95	75.79	42.43	32.45
20	600.0	67.41	85.60	34.97	25.66
16	750.0	74.63	94.41	24.88	15.03
12	1000.0	91.20	99.72	7.70	1.54

Table 8-2. Energies used for AF calculations. For each energy, a different number of histories is run to achieve equivalent statistical precision. The last two columns give the AFs calculated within the mathematical sample (e.g., the reference values).

Electron energy (MeV)	Histories for source in marrow (thousands)	Histories for source in bone (thousands)	$\Phi_{(\text{bone} \leftarrow \text{marrow})}$	$\Phi_{(\text{marrow} \leftarrow \text{bone})}$
0.025	5800	4000	0.00681	0.00741
0.050	1500	1500	0.0231	0.0248
0.075	700	750	0.0461	0.0499
0.100	450	400	0.0746	0.0795
0.150	200	160	0.140	0.150
0.200	120	100	0.209	0.220
0.250	65	90	0.265	0.283
0.300	48	40	0.309	0.330
0.350	42	30	0.342	0.361
0.400	35	28	0.361	0.382
0.450	32	24	0.378	0.397
0.500	28	20	0.386	0.404
0.600	24	17	0.388	0.405
0.700	21	15	0.382	0.409
0.800	19	15	0.378	0.401

Table 8-3. Radiation characteristics of the radionuclides used for the S-value calculations. They represent the reference S values. They are sorted by mean beta energy.

	Mode of decay	Avg. Energy (MeV)	Max. Energy (MeV)	Half-life (days)	$S_{(\text{bone} \leftarrow \text{marrow})}$ (mGy/MBq-s)	$S_{(\text{marrow} \leftarrow \text{bone})}$ (mGy/MBq-s)
^{33}P	β^-	0.077	0.249	25.3	0.00108	0.00124
$^{117\text{m}}\text{Sn}$	I.T.	0.135	0.159	13.6	0.00249	0.00287
^{131}I	β^-	0.191	0.606	8.04	0.00694	0.00789
^{153}Sm	β^-	0.225	0.809	1.95	0.00937	0.01063

Table 8-4. S values calculated within the real bone sample for both the voxel and the HMC representations of the bone-marrow interface. Unit is mGy/MBq-s.

	$S_{(\text{bone} \leftarrow \text{marrow})}$ voxel-code	$S_{(\text{bone} \leftarrow \text{marrow})}$ HMC-code	Relative difference (%)	$S_{(\text{marrow} \leftarrow \text{bone})}$ voxel-code	$S_{(\text{marrow} \leftarrow \text{bone})}$ HMC-code	Relative difference (%)
^{33}P	0.00117	0.00099	18.2	0.00134	0.00118	13.6
$^{117\text{m}}\text{Sn}$	0.00267	0.00227	17.6	0.00308	0.00271	13.7
^{131}I	0.00756	0.00706	7.1	0.00861	0.00838	2.7
^{153}Sm	0.01040	0.00975	6.7	0.01183	0.01157	2.2

CHAPTER 9 CONCLUSIONS AND FUTURE WORK

Conclusions

NMR imaging of trabecular bone samples has been studied since 1995 at the University of Florida and it has become evident that it can provide an important tool for bone-marrow dosimetry assessments (Jokisch 1997; Jokisch 1999; Jokisch et al. 2001a; Jokisch et al. 1998; Jokisch et al. 2001b; Patton 1998; Patton 2000; Patton et al. 2002a; Patton et al. 2002b; Rajon 1999). These images can easily serve as geometrical input data for a radiation transport and other computer codes. Two examples of such codes that are of great interest in skeleton dosimetry are chord-length distribution measurements and ionizing-particle transport. In both examples, a Monte-Carlo technique is used within the geometry given by the image. For chord-length distribution measurements, millions of rays are fired all around the image and chord lengths are recorded as the rays penetrate in a straight line through the marrow cavities and the bone trabeculae. For particle transport, the principle is identical except that the particles follow complex paths because of their elastic and inelastic collisions within the bone and marrow tissue. Unfortunately, in both types of codes, the rectangular shape of the voxels that constitute the image was shown to introduce voxel effects in the simulation (Rajon 1999). These effects were responsible for large errors in the assessment of the dosimetric parameters.

Chapters 3 and 4 have explained these effects in the case of particle transport. The voxelization process when producing 3D images inevitably overestimates the surface area

of geometrical objects. This overestimation was shown to be independent of the voxel size. It was also demonstrated that the cross-absorbed-fraction calculation – when radiation is initiated within marrow and irradiates bone, or vice-versa – was also affected as being a direct consequence of the surface-area overestimation. At low electron energies, the error on the AF calculation can be as important as the error on the surface area. Only when the energy is higher than 300 keV does the effect becomes insignificant. For low-energy beta emitters, the consequence on the S value can be important and justify the need for a new representation of the bone-marrow interface.

For chord-length distribution measurements, [Chapters 5 and 6](#) explained how the voxel effects could artificially overestimate the frequency of the short chords. The direct consequence of this overestimation is a systematic shift of the mean chord length toward the short chords. Besides, the techniques developed by Jokisch et al. ([2001b](#)) to try to reduce these effects were shown to be voxel-size dependent and difficult to adapt to each specific image. This problem was another reason for the creation of a new representation of the bone-marrow interface.

The Marching-Cube algorithm was investigated in [Chapter 7](#). Its direct application provides a triangulated representation of the bone-marrow interface. Drawbacks of this technique are the large amount of memory required to store the triangles and the complexity of the programs that access the triangle list. For this reason, an adaptation of the MC algorithm was developed that allows using the initial image only. Using this new Hyperboloid Marching-Cube algorithm allowed removing the voxel effects when measuring a chord-length distribution within a 3D image. The mean chord length of the

distribution is preserved as well as the shape of the distribution. Furthermore, the overestimation of the surface area of the bone-marrow interface is eliminated.

Chapter 8 studied the impact of the new HMC algorithm on the particle transport. It showed that the AF overestimation, as well as the S-value overestimation, is reduced by the use of this new representation of the bone-marrow interface.

The new HMC algorithm clearly solves the voxel-effect problems by providing a smoother surface to delineate the bone regions from the marrow regions within 3D images of bone samples. Both the chord-length distribution measurement and the electron transport simulation have been successfully tested with this new method. Its utilization provides a good convergence of the dosimetric parameters as the image resolution is improved. At 60 μm , which is the resolution capability of current 3D NMR imaging systems, the error due to the digitization process is reduced to 1% for the surface-area measurement, the mean chord-length distribution within the marrow cavities, and the S-value calculation within marrow when irradiated by bone (or vice-versa). These 1% errors must be compared with the 50% overestimation of the surface area, the 30% underestimation of the mean marrow chord length, and a 10-25% overestimation of the S value (for ^{33}P or ^{117}mSn) when using the voxel representation of the bone-marrow interface to do the same simulations. These improvements fully justify the use of the new HMC technique, which is highly recommended whenever a computer program needs to be coupled with a digital image.

Future work

As a future utilization of the technique, new chord-length distributions should be measured using NMR images of bone samples from different skeleton sites. These

distributions can thus replace Spiers distributions that have been the basis of bone-marrow dosimetry models for almost 30 years. Even though the Spiers distributions clearly show voxel effects, they are still used to determine absorbed fractions to marrow in current clinical skeletal dose models ([Bouchet et al. 2000](#); [Eckerman and Stabin 2000](#)). Next, the particle transport within the NMR images should soon provide a new model to calculate these absorbed fractions without the need for chord-length distribution measurements. This technique will benefit from the use of the HMC representation when coupling the bone images with the Monte-Carlo transport code.

As mentioned in [Chapters 7 and 8](#), anthropomorphic models used in internal dosimetry are about to be changed into voxel models of the human body ([Chao et al. 2001a](#); [Chao et al. 2001b](#); [Xu et al. 2000](#); [Zankl and Wittmann 2001](#)). One reason for this change is the complexity of the current models. As new organs are added to the models or divided into more and more sub-organs, the computer codes become more and more complex and error prone. Besides, it has become a real challenge to fit a new organ among the already existing ones without overlap between regions. Furthermore, the geometrical shapes of the organs in stylized anthropomorphic models lack anatomical accuracy. A better match with the real organ shapes would be found with a geometry coming from a CT or NMR image of a real human body. As the models become more accurate and the regions smaller and smaller, electron dosimetry will be of great importance and these new anthropomorphic models will have to take into account the voxel effects – expected at energies as high as 700 keV for a 1 mm resolution – mentioned in this dissertation. The creators of these models will find the HMC representation of the organ boundaries to be a good opportunity to solve these voxel-effect problems.

Another skeletal tissue of interest in dosimetry is the endosteal layer located at the interface between bone and marrow. Jokisch et al. (2001a) used the NMR images to model a 10- μm layer of independent tissue on the internal surface of the marrow cavities. According to the model, this layer was located along the inner side of every marrow voxel face that is adjacent to a bone voxel. Using a radiation transport computer code, the absorbed fraction of energy deposited within the endosteum was recorded each time an electron crosses this layer. Assuming the layer thickness small compared to the voxel size, the endosteal region defined by this technique has a volume that is proportional to the surface area of the bone-marrow interface. As explained in Chapter 4, this surface area is overestimated by 50% and so is the endosteal volume. The consequence is an overestimation of the endosteal absorbed fraction. The surface-area problem is solved by the new HMC technique discussed in this dissertation. This suggests that the HMC algorithm be applied for endosteal tissue dosimetry. The hyperboloid surface is a complex mathematical representation and its adaptation to the endosteal problem may involve some even more complex problems to calculate the distance between a particle and the surface but its investigation should be considered. If the problem happens to be too complicated to be solved, the original MC algorithm, with its triangles and polygons, may also be used. The distance to a plane is a much easier equation to solve and this technique can be investigated with small images so that the memory size mentioned in Chapter 7 can be handled. In any case, the MC technique can be a solution for endosteal dosimetry.

Alpha particle dosimetry is another concern for radionuclide therapy. Because of their short range in tissue, alpha particle emitters are ideal subjects for irradiation of bone tumors. Bone-marrow doses are expected to be small but their accurate assessment is still

challenging researchers ([Hamacher and Sgouros 2001](#); [ICRP 1979](#); [McDevitt et al. 1998](#)). The use of 3D images of trabecular bone sample may become a solution to estimate the cross-absorbed fraction to bone marrow. The extremely short range of alpha particles (70 μm at 7 MeV) makes them highly exposed to voxel effects. As explain in [Chapter 4](#) and as concluded in [Chapter 8](#), the overestimation of the AF should be important (up to 50%) for voxel sizes larger than a third of the particle range. With an actual resolution on the order of 50 μm for both NMR and QCT, voxel effects are a concern for all known alpha-emitter radionuclides. This, again, suggests that the HMC algorithm be applied for alpha-particle dosimetry.

APPENDIX A LOOK-UP TABLES FOR MARCHING-CUBE ALGORITHM

This appendix contains the look-up tables used to generate the triangle list according to the Marching Cube algorithm. Three series are presented. The first (LOR) uses the patterns proposed initially by Lorensen. They are shown on [Figure 7-3](#) and were only used in this dissertation to illustrate the hole problem on [Figure 7-6a](#). The second (DIR) uses the patterns of [Figure 7-8](#) that implement the direct technique used to solve the hole problem. The third (REV) uses the patterns of [Figure 7-9](#) that implement the reverse technique used to solve the hole problem.

Each series contains three look-up tables. The pattern look-up table has one entry per pattern and tells how many surfaces (some patterns are made of several polygons) are used to design the pattern, how many triangles are used in each surface, and in how many designs can be organized the triangles within each surface.

The design look-up table was never mentioned in this dissertation because it does not bring a significant improvement and was completely ignored in [Chapter 7](#). For each surface, there are several ways to organize – or to design – the triangles within the surface. Each organization, since the surface is non-planar, can have a different surface area. The best design would be the one that gives the smallest surface since the purpose of the Marching Cube method is to smooth the surface. The program that generates the triangle list, presented in [Appendix D](#), calculates the surface area for each design and selects the design that produces the smallest surface area. Then it generates the triangles according to the selected design. Note that, for the LOR technique, only one design is considered: the one proposed by Lorensen and shown on [Figure 7-3](#). The design look-up table has one entry per pattern, per surface, and per design. For each entry it tells how to organize the triangles. For instance, using the DIR technique, pattern 6 has 2 surfaces, one has 2 triangles and 2 possible designs and the second has only one triangle and one design (this information is found in the pattern table). Thus, we find three entries for pattern 6 in the design table: two for the first surface and one for the second. If the four summits of the first surface are numbered 1, 2, 3, and 4, the two triangles will join the summits 1, 2, and 3 for the first one and 1, 3, and 4 for the second. Of course, a second design would be to join 1, 2, and 4 on one hand and 2, 3, and 4 on the other hand as indicates the second entry of the design table. For the second surface only one triangle has obviously one design joining the summits 1, 2, and 3. For the REV technique (see [Figure 7-9](#)), pattern 6 has only one surface but 5 triangles and the designs are far more complicated (16 possible designs).

The configuration look-up table has one entry per configuration. For each it tells the pattern number and the list of the summits on which are attached the surfaces (-1 means that the summit is not used). The numbers found in the summit list corresponds to the 12 edge numbers (from 0 to 11) as defined on [Figure 7-13](#). For instance, for pattern 6

Design Look-up Table (LOR)

Marching Cube Designs

Pat.	Surf.	Triangle list
1	1	1 2 3
2	1	1 2 3 1 3 4
3	1	1 2 3
3	2	1 2 3
4	1	1 2 3
4	2	1 2 3
5	1	1 2 3 1 3 4 1 4 5
6	1	1 2 3 1 3 4
6	2	1 2 3
7	1	1 2 3
7	2	1 2 3
7	3	1 2 3
8	1	1 2 3 1 3 4
9	1	1 2 3 1 3 4 1 4 6 4 5 6
10	1	1 2 3 1 3 4
10	2	1 2 3 1 3 4
11	1	1 2 3 1 3 5 3 4 5 1 5 6
12	1	1 2 3 1 3 4 1 4 5
12	2	1 2 3
13	1	1 2 3
13	2	1 2 3
13	3	1 2 3
13	4	1 2 3
14	1	1 2 3 1 3 5 3 4 5 1 5 6
15	1	1 2 3
15	2	1 2 3
15	3	1 2 3
16	1	1 2 3 1 3 4
16	2	1 2 3
17	1	1 2 3 1 3 4 1 4 5
18	1	1 2 3
18	2	1 2 3
19	1	1 2 3
19	2	1 2 3
20	1	1 2 3 1 3 4
21	1	1 2 3

Configuration Look-up Table (LOR)

Marching Cube Configurations

Conf.	Pat.	Summit	list
0	0	-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
1	1	0 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
2	1	1 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
4	1	2 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
8	1	3 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
16	1	7 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
32	1	4 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
64	1	5 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
128	1	6 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
3	2	1 9 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
6	2	2 10 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
12	2	3 11 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
9	2	0 8 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
48	2	7 8 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
96	2	4 9 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
192	2	5 10 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
144	2	6 11 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
17	2	3 0 4 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
34	2	0 1 5 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
68	2	1 2 6 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
136	2	2 3 7 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
5	3	0 8 3 2 10 1 -1 -1 -1 -1 -1 -1 -1 -1	
10	3	1 9 0 3 11 2 -1 -1 -1 -1 -1 -1 -1 -1	
80	3	7 8 4 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
160	3	4 9 5 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
129	3	0 8 3 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
24	3	3 11 2 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
66	3	1 9 0 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
36	3	2 10 1 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
33	3	0 8 3 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
18	3	1 9 0 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
72	3	3 11 2 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
132	3	2 10 1 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
65	4	0 8 3 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
130	4	1 9 0 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
20	4	2 10 1 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
40	4	3 11 2 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
7	5	10 9 8 3 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
14	5	11 10 9 0 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
13	5	8 11 10 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
11	5	9 8 11 2 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
112	5	8 9 10 6 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
224	5	9 10 11 7 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
208	5	10 11 8 4 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
176	5	11 8 9 5 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
137	5	6 2 0 8 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
152	5	4 6 2 3 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
145	5	0 4 6 11 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
25	5	2 0 4 7 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
70	5	0 2 6 5 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
100	5	2 6 4 9 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
98	5	6 4 0 1 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
38	5	4 0 2 10 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
35	5	3 1 5 4 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
50	5	1 5 7 8 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
49	5	5 7 3 0 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
19	5	7 3 1 9 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
76	5	5 1 3 11 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
196	5	7 5 1 2 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
200	5	3 7 5 10 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
140	5	1 3 7 6 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
67	6	1 9 8 3 5 10 6 -1 -1 -1 -1 -1 -1 -1	

131	6	1	9	8	3	6	11	7	-1	-1	-1	-1	-1
134	6	2	10	9	0	6	11	7	-1	-1	-1	-1	-1
22	6	2	10	9	0	7	8	4	-1	-1	-1	-1	-1
28	6	3	11	10	1	7	8	4	-1	-1	-1	-1	-1
44	6	3	11	10	1	4	9	5	-1	-1	-1	-1	-1
41	6	0	8	11	2	4	9	5	-1	-1	-1	-1	-1
73	6	0	8	11	2	5	10	6	-1	-1	-1	-1	-1
56	6	7	8	9	5	3	11	2	-1	-1	-1	-1	-1
52	6	7	8	9	5	2	10	1	-1	-1	-1	-1	-1
97	6	4	9	10	6	0	8	3	-1	-1	-1	-1	-1
104	6	4	9	10	6	3	11	2	-1	-1	-1	-1	-1
194	6	5	10	11	7	1	9	0	-1	-1	-1	-1	-1
193	6	5	10	11	7	0	8	3	-1	-1	-1	-1	-1
148	6	6	11	8	4	2	10	1	-1	-1	-1	-1	-1
146	6	6	11	8	4	1	9	0	-1	-1	-1	-1	-1
21	6	3	0	4	7	2	10	1	-1	-1	-1	-1	-1
81	6	3	0	4	7	5	10	6	-1	-1	-1	-1	-1
42	6	0	1	5	4	3	11	2	-1	-1	-1	-1	-1
162	6	0	1	5	4	6	11	7	-1	-1	-1	-1	-1
69	6	1	2	6	5	0	8	3	-1	-1	-1	-1	-1
84	6	1	2	6	5	7	8	4	-1	-1	-1	-1	-1
138	6	2	3	7	6	1	9	0	-1	-1	-1	-1	-1
168	6	2	3	7	6	4	9	5	-1	-1	-1	-1	-1
161	7	0	8	3	4	9	5	6	11	7	-1	-1	-1
82	7	1	9	0	5	10	6	7	8	4	-1	-1	-1
164	7	2	10	1	6	11	7	4	9	5	-1	-1	-1
88	7	3	11	2	7	8	4	5	10	6	-1	-1	-1
26	7	7	8	4	3	11	2	1	9	0	-1	-1	-1
37	7	4	9	5	0	8	3	2	10	1	-1	-1	-1
74	7	5	10	6	1	9	0	3	11	2	-1	-1	-1
133	7	6	11	7	2	10	1	0	8	3	-1	-1	-1
15	8	8	11	10	9	-1	-1	-1	-1	-1	-1	-1	-1
240	8	11	8	9	10	-1	-1	-1	-1	-1	-1	-1	-1
153	8	0	4	6	2	-1	-1	-1	-1	-1	-1	-1	-1
102	8	4	0	2	6	-1	-1	-1	-1	-1	-1	-1	-1
51	8	3	1	5	7	-1	-1	-1	-1	-1	-1	-1	-1
204	8	1	3	7	5	-1	-1	-1	-1	-1	-1	-1	-1
27	9	1	9	4	7	11	2	-1	-1	-1	-1	-1	-1
39	9	2	10	5	4	8	3	-1	-1	-1	-1	-1	-1
78	9	3	11	6	5	9	0	-1	-1	-1	-1	-1	-1
141	9	0	8	7	6	10	1	-1	-1	-1	-1	-1	-1
177	9	6	11	3	0	9	5	-1	-1	-1	-1	-1	-1
114	9	7	8	0	1	10	6	-1	-1	-1	-1	-1	-1
228	9	4	9	1	2	11	7	-1	-1	-1	-1	-1	-1
216	9	5	10	2	3	8	4	-1	-1	-1	-1	-1	-1
195	10	1	9	8	3	5	10	11	7	-1	-1	-1	-1
60	10	7	8	9	5	3	11	10	1	-1	-1	-1	-1
105	10	0	8	11	2	4	9	10	6	-1	-1	-1	-1
150	10	2	10	9	0	6	11	8	4	-1	-1	-1	-1
85	10	3	0	4	7	1	2	6	5	-1	-1	-1	-1
170	10	2	3	7	6	0	1	5	4	-1	-1	-1	-1
99	11	3	1	10	6	4	8	-1	-1	-1	-1	-1	-1
198	11	0	2	11	7	5	9	-1	-1	-1	-1	-1	-1
156	11	1	3	8	4	6	10	-1	-1	-1	-1	-1	-1
57	11	2	0	9	5	7	11	-1	-1	-1	-1	-1	-1
116	11	8	9	1	2	6	7	-1	-1	-1	-1	-1	-1
232	11	9	10	2	3	7	4	-1	-1	-1	-1	-1	-1
209	11	10	11	3	0	4	5	-1	-1	-1	-1	-1	-1
178	11	11	8	0	1	5	6	-1	-1	-1	-1	-1	-1
23	11	7	3	2	10	9	4	-1	-1	-1	-1	-1	-1
46	11	4	0	3	11	10	5	-1	-1	-1	-1	-1	-1
77	11	5	1	0	8	11	6	-1	-1	-1	-1	-1	-1
139	11	6	2	1	9	8	7	-1	-1	-1	-1	-1	-1
135	12	10	9	8	3	2	6	11	7	-1	-1	-1	-1
30	12	11	10	9	0	3	7	8	4	-1	-1	-1	-1
45	12	8	11	10	1	0	4	9	5	-1	-1	-1	-1
75	12	9	8	11	2	1	5	10	6	-1	-1	-1	-1

120	12	8	9	10	6	7	3	11	2	-1	-1	-1	-1
225	12	9	10	11	7	4	0	8	3	-1	-1	-1	-1
210	12	10	11	8	4	5	1	9	0	-1	-1	-1	-1
180	12	11	8	9	5	6	2	10	1	-1	-1	-1	-1
169	12	6	2	0	8	7	4	9	5	-1	-1	-1	-1
154	12	4	6	2	3	8	1	9	0	-1	-1	-1	-1
149	12	0	4	6	11	3	2	10	1	-1	-1	-1	-1
89	12	2	0	4	7	11	5	10	6	-1	-1	-1	-1
86	12	0	2	6	5	9	7	8	4	-1	-1	-1	-1
101	12	2	6	4	9	1	0	8	3	-1	-1	-1	-1
106	12	6	4	0	1	10	3	11	2	-1	-1	-1	-1
166	12	4	0	2	10	5	6	11	7	-1	-1	-1	-1
163	12	3	1	5	4	8	6	11	7	-1	-1	-1	-1
58	12	1	5	7	8	0	3	11	2	-1	-1	-1	-1
53	12	5	7	3	0	9	2	10	1	-1	-1	-1	-1
83	12	7	3	1	9	4	5	10	6	-1	-1	-1	-1
92	12	5	1	3	11	6	7	8	4	-1	-1	-1	-1
197	12	7	5	1	2	11	0	8	3	-1	-1	-1	-1
202	12	3	7	5	10	2	1	9	0	-1	-1	-1	-1
172	12	1	3	7	6	10	4	9	5	-1	-1	-1	-1
165	13	0	8	3	2	10	1	4	9	5	6	11	7
90	13	1	9	0	3	11	2	5	10	6	7	8	4
71	14	8	3	2	6	5	9	-1	-1	-1	-1	-1	-1
142	14	9	0	3	7	6	10	-1	-1	-1	-1	-1	-1
29	14	10	1	0	4	7	11	-1	-1	-1	-1	-1	-1
43	14	11	2	1	5	4	8	-1	-1	-1	-1	-1	-1
54	14	7	8	0	2	10	5	-1	-1	-1	-1	-1	-1
108	14	4	9	1	3	11	6	-1	-1	-1	-1	-1	-1
201	14	5	10	2	0	8	7	-1	-1	-1	-1	-1	-1
147	14	6	11	3	1	9	4	-1	-1	-1	-1	-1	-1
113	14	3	0	9	10	6	7	-1	-1	-1	-1	-1	-1
226	14	0	1	10	11	7	4	-1	-1	-1	-1	-1	-1
212	14	1	2	11	8	4	5	-1	-1	-1	-1	-1	-1
184	14	2	3	8	9	5	6	-1	-1	-1	-1	-1	-1
122	15	7	11	6	1	10	2	3	8	0	-1	-1	-1
181	15	6	10	5	0	9	1	2	11	3	-1	-1	-1
218	15	5	9	4	3	8	0	1	10	2	-1	-1	-1
229	15	4	8	7	2	11	3	0	9	1	-1	-1	-1
167	15	2	11	3	4	8	7	6	10	5	-1	-1	-1
91	15	1	10	2	7	11	6	5	9	4	-1	-1	-1
173	15	0	9	1	6	10	5	4	8	7	-1	-1	-1
94	15	3	8	0	5	9	4	7	11	6	-1	-1	-1
87	16	6	7	3	2	5	9	4	-1	-1	-1	-1	-1
117	16	6	7	3	2	0	9	1	-1	-1	-1	-1	-1
171	16	5	6	2	1	4	8	7	-1	-1	-1	-1	-1
186	16	5	6	2	1	3	8	0	-1	-1	-1	-1	-1
93	16	4	5	1	0	7	11	6	-1	-1	-1	-1	-1
213	16	4	5	1	0	2	11	3	-1	-1	-1	-1	-1
174	16	7	4	0	3	6	10	5	-1	-1	-1	-1	-1
234	16	7	4	0	3	1	10	2	-1	-1	-1	-1	-1
109	16	4	8	11	6	0	9	1	-1	-1	-1	-1	-1
107	16	4	8	11	6	1	10	2	-1	-1	-1	-1	-1
62	16	7	11	10	5	3	8	0	-1	-1	-1	-1	-1
61	16	7	11	10	5	0	9	1	-1	-1	-1	-1	-1
151	16	6	10	9	4	2	11	3	-1	-1	-1	-1	-1
158	16	6	10	9	4	3	8	0	-1	-1	-1	-1	-1
203	16	5	9	8	7	1	10	2	-1	-1	-1	-1	-1
199	16	5	9	8	7	2	11	3	-1	-1	-1	-1	-1
182	16	2	11	8	0	6	10	5	-1	-1	-1	-1	-1
214	16	2	11	8	0	5	9	4	-1	-1	-1	-1	-1
211	16	1	10	11	3	5	9	4	-1	-1	-1	-1	-1
227	16	1	10	11	3	4	8	7	-1	-1	-1	-1	-1
233	16	0	9	10	2	4	8	7	-1	-1	-1	-1	-1
121	16	0	9	10	2	7	11	6	-1	-1	-1	-1	-1
124	16	3	8	9	1	7	11	6	-1	-1	-1	-1	-1
188	16	3	8	9	1	6	10	5	-1	-1	-1	-1	-1
115	17	7	3	1	10	6	-1	-1	-1	-1	-1	-1	-1

Direct Technique (DIR)**Pattern Look-up Table (DIR)**

Marching Cube Patterns

Pat.	Surf.	Tr.	per Surf.	Des.	per Surf.
0	0				
1	1	1		1	
2	1	2		2	
3	2	1	1	1	1
4	2	1	1	1	1
5	1	3		5	
6	2	2	1	2	1
7	3	1	1	1	1
8	1	2		2	
9	1	4		14	
10	2	2	2	2	2
11	1	4		14	
12	2	3	1	5	1
13	4	1	1	1	1
14	1	4		14	
15	2	4	1	14	1
16	1	5		16	
17	1	3		5	
18	2	1	1	1	1
19	1	4		4	
20	1	2		2	
21	1	1		1	
22	0				

Design Look-up Table (DIR)

Marching Cube Designs

Pat.	Surf.	Triangle list
1	1	1 2 3
2	1	1 2 3 1 3 4
2	1	1 2 4 2 3 4
3	1	1 2 3
3	2	1 2 3
4	1	1 2 3
4	2	1 2 3
5	1	1 2 3 1 3 4 1 4 5
5	1	1 2 3 1 3 5 3 4 5
5	1	2 3 4 2 4 5 2 5 1
5	1	2 3 4 2 4 1 4 5 1
5	1	3 4 5 3 5 2 5 1 2
6	1	1 2 3 1 3 4
6	1	1 2 4 2 3 4
6	2	1 2 3
7	1	1 2 3
7	2	1 2 3
7	3	1 2 3
8	1	1 2 3 1 3 4
8	1	1 2 4 2 3 4
9	1	1 2 3 1 3 4 1 4 5 1 5 6
9	1	1 2 3 1 3 4 1 4 6 4 5 6
9	1	1 2 3 3 4 5 3 5 6 3 6 1
9	1	1 2 3 3 4 5 3 5 1 5 6 1
9	1	1 2 3 4 5 6 4 6 3 6 1 3
9	1	2 3 4 2 4 5 2 5 6 2 6 1
9	1	2 3 4 2 4 5 2 5 1 5 6 1
9	1	2 3 4 4 5 6 4 6 1 4 1 2
9	1	2 3 4 4 5 6 4 6 2 6 1 2
9	1	2 3 4 5 6 1 5 1 4 1 2 4
9	1	3 4 5 3 5 6 3 6 2 6 1 2
9	1	3 4 5 5 6 1 5 1 2 5 2 3
9	1	3 4 5 6 1 2 6 2 5 2 3 5
9	1	4 5 6 6 1 2 6 2 3 6 3 4
10	1	1 2 3 1 3 4
10	1	1 2 4 2 3 4
10	2	1 2 3 1 3 4
10	2	1 2 4 2 3 4
11	1	1 2 3 1 3 4 1 4 5 1 5 6
11	1	1 2 3 1 3 4 1 4 6 4 5 6
11	1	1 2 3 3 4 5 3 5 6 3 6 1
11	1	1 2 3 3 4 5 3 5 1 5 6 1
11	1	1 2 3 4 5 6 4 6 3 6 1 3
11	1	2 3 4 2 4 5 2 5 6 2 6 1
11	1	2 3 4 2 4 5 2 5 1 5 6 1
11	1	2 3 4 4 5 6 4 6 1 4 1 2
11	1	2 3 4 4 5 6 4 6 2 6 1 2
11	1	2 3 4 5 6 1 5 1 4 1 2 4
11	1	3 4 5 3 5 6 3 6 2 6 1 2
11	1	3 4 5 5 6 1 5 1 2 5 2 3
11	1	3 4 5 6 1 2 6 2 5 2 3 5
11	1	4 5 6 6 1 2 6 2 3 6 3 4
12	1	1 2 3 1 3 4 1 4 5
12	1	1 2 3 1 3 5 3 4 5
12	1	2 3 4 2 4 5 2 5 1
12	1	2 3 4 2 4 1 4 5 1
12	1	3 4 5 3 5 2 5 1 2
12	2	1 2 3
13	1	1 2 3
13	2	1 2 3
13	3	1 2 3
13	4	1 2 3

Configuration Look-up Table (DIR)

Marching Cube Configurations

Conf.	Pat.	Summit	list
0	0	-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
1	1	0 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
2	1	1 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
4	1	2 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
8	1	3 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
16	1	7 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
32	1	4 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
64	1	5 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
128	1	6 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
3	2	1 9 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
6	2	2 10 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
12	2	3 11 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
9	2	0 8 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
48	2	7 8 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
96	2	4 9 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
192	2	5 10 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
144	2	6 11 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
17	2	3 0 4 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
34	2	0 1 5 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
68	2	1 2 6 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
136	2	2 3 7 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
5	3	0 8 3 2 10 1 -1 -1 -1 -1 -1 -1 -1 -1	
10	3	1 9 0 3 11 2 -1 -1 -1 -1 -1 -1 -1 -1	
80	3	7 8 4 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
160	3	4 9 5 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
129	3	0 8 3 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
24	3	3 11 2 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
66	3	1 9 0 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
36	3	2 10 1 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
33	3	0 8 3 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
18	3	1 9 0 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
72	3	3 11 2 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
132	3	2 10 1 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
65	4	0 8 3 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
130	4	1 9 0 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
20	4	2 10 1 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
40	4	3 11 2 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
7	5	10 9 8 3 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
14	5	11 10 9 0 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
13	5	8 11 10 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
11	5	9 8 11 2 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
112	5	8 9 10 6 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
224	5	9 10 11 7 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
208	5	10 11 8 4 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
176	5	11 8 9 5 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
137	5	6 2 0 8 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
152	5	4 6 2 3 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
145	5	0 4 6 11 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
25	5	2 0 4 7 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
70	5	0 2 6 5 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
100	5	2 6 4 9 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
98	5	6 4 0 1 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
38	5	4 0 2 10 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
35	5	3 1 5 4 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
50	5	1 5 7 8 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
49	5	5 7 3 0 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
19	5	7 3 1 9 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
76	5	5 1 3 11 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
196	5	7 5 1 2 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
200	5	3 7 5 10 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
140	5	1 3 7 6 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
67	6	1 9 8 3 5 10 6 -1 -1 -1 -1 -1 -1 -1	

131	6	1	9	8	3	6	11	7	-1	-1	-1	-1	-1
134	6	2	10	9	0	6	11	7	-1	-1	-1	-1	-1
22	6	2	10	9	0	7	8	4	-1	-1	-1	-1	-1
28	6	3	11	10	1	7	8	4	-1	-1	-1	-1	-1
44	6	3	11	10	1	4	9	5	-1	-1	-1	-1	-1
41	6	0	8	11	2	4	9	5	-1	-1	-1	-1	-1
73	6	0	8	11	2	5	10	6	-1	-1	-1	-1	-1
56	6	7	8	9	5	3	11	2	-1	-1	-1	-1	-1
52	6	7	8	9	5	2	10	1	-1	-1	-1	-1	-1
97	6	4	9	10	6	0	8	3	-1	-1	-1	-1	-1
104	6	4	9	10	6	3	11	2	-1	-1	-1	-1	-1
194	6	5	10	11	7	1	9	0	-1	-1	-1	-1	-1
193	6	5	10	11	7	0	8	3	-1	-1	-1	-1	-1
148	6	6	11	8	4	2	10	1	-1	-1	-1	-1	-1
146	6	6	11	8	4	1	9	0	-1	-1	-1	-1	-1
21	6	3	0	4	7	2	10	1	-1	-1	-1	-1	-1
81	6	3	0	4	7	5	10	6	-1	-1	-1	-1	-1
42	6	0	1	5	4	3	11	2	-1	-1	-1	-1	-1
162	6	0	1	5	4	6	11	7	-1	-1	-1	-1	-1
69	6	1	2	6	5	0	8	3	-1	-1	-1	-1	-1
84	6	1	2	6	5	7	8	4	-1	-1	-1	-1	-1
138	6	2	3	7	6	1	9	0	-1	-1	-1	-1	-1
168	6	2	3	7	6	4	9	5	-1	-1	-1	-1	-1
161	7	0	8	3	4	9	5	6	11	7	-1	-1	-1
82	7	1	9	0	5	10	6	7	8	4	-1	-1	-1
164	7	2	10	1	6	11	7	4	9	5	-1	-1	-1
88	7	3	11	2	7	8	4	5	10	6	-1	-1	-1
26	7	7	8	4	3	11	2	1	9	0	-1	-1	-1
37	7	4	9	5	0	8	3	2	10	1	-1	-1	-1
74	7	5	10	6	1	9	0	3	11	2	-1	-1	-1
133	7	6	11	7	2	10	1	0	8	3	-1	-1	-1
15	8	8	11	10	9	-1	-1	-1	-1	-1	-1	-1	-1
240	8	11	8	9	10	-1	-1	-1	-1	-1	-1	-1	-1
153	8	0	4	6	2	-1	-1	-1	-1	-1	-1	-1	-1
102	8	4	0	2	6	-1	-1	-1	-1	-1	-1	-1	-1
51	8	3	1	5	7	-1	-1	-1	-1	-1	-1	-1	-1
204	8	1	3	7	5	-1	-1	-1	-1	-1	-1	-1	-1
27	9	1	9	4	7	11	2	-1	-1	-1	-1	-1	-1
39	9	2	10	5	4	8	3	-1	-1	-1	-1	-1	-1
78	9	3	11	6	5	9	0	-1	-1	-1	-1	-1	-1
141	9	0	8	7	6	10	1	-1	-1	-1	-1	-1	-1
177	9	6	11	3	0	9	5	-1	-1	-1	-1	-1	-1
114	9	7	8	0	1	10	6	-1	-1	-1	-1	-1	-1
228	9	4	9	1	2	11	7	-1	-1	-1	-1	-1	-1
216	9	5	10	2	3	8	4	-1	-1	-1	-1	-1	-1
195	10	1	9	8	3	5	10	11	7	-1	-1	-1	-1
60	10	7	8	9	5	3	11	10	1	-1	-1	-1	-1
105	10	0	8	11	2	4	9	10	6	-1	-1	-1	-1
150	10	2	10	9	0	6	11	8	4	-1	-1	-1	-1
85	10	3	0	4	7	1	2	6	5	-1	-1	-1	-1
170	10	2	3	7	6	0	1	5	4	-1	-1	-1	-1
99	11	3	1	10	6	4	8	-1	-1	-1	-1	-1	-1
198	11	0	2	11	7	5	9	-1	-1	-1	-1	-1	-1
156	11	1	3	8	4	6	10	-1	-1	-1	-1	-1	-1
57	11	2	0	9	5	7	11	-1	-1	-1	-1	-1	-1
116	11	8	9	1	2	6	7	-1	-1	-1	-1	-1	-1
232	11	9	10	2	3	7	4	-1	-1	-1	-1	-1	-1
209	11	10	11	3	0	4	5	-1	-1	-1	-1	-1	-1
178	11	11	8	0	1	5	6	-1	-1	-1	-1	-1	-1
23	11	7	3	2	10	9	4	-1	-1	-1	-1	-1	-1
46	11	4	0	3	11	10	5	-1	-1	-1	-1	-1	-1
77	11	5	1	0	8	11	6	-1	-1	-1	-1	-1	-1
139	11	6	2	1	9	8	7	-1	-1	-1	-1	-1	-1
135	12	10	9	8	3	2	6	11	7	-1	-1	-1	-1
30	12	11	10	9	0	3	7	8	4	-1	-1	-1	-1
45	12	8	11	10	1	0	4	9	5	-1	-1	-1	-1
75	12	9	8	11	2	1	5	10	6	-1	-1	-1	-1

120	12	8	9	10	6	7	3	11	2	-1	-1	-1	-1
225	12	9	10	11	7	4	0	8	3	-1	-1	-1	-1
210	12	10	11	8	4	5	1	9	0	-1	-1	-1	-1
180	12	11	8	9	5	6	2	10	1	-1	-1	-1	-1
169	12	6	2	0	8	7	4	9	5	-1	-1	-1	-1
154	12	4	6	2	3	8	1	9	0	-1	-1	-1	-1
149	12	0	4	6	11	3	2	10	1	-1	-1	-1	-1
89	12	2	0	4	7	11	5	10	6	-1	-1	-1	-1
86	12	0	2	6	5	9	7	8	4	-1	-1	-1	-1
101	12	2	6	4	9	1	0	8	3	-1	-1	-1	-1
106	12	6	4	0	1	10	3	11	2	-1	-1	-1	-1
166	12	4	0	2	10	5	6	11	7	-1	-1	-1	-1
163	12	3	1	5	4	8	6	11	7	-1	-1	-1	-1
58	12	1	5	7	8	0	3	11	2	-1	-1	-1	-1
53	12	5	7	3	0	9	2	10	1	-1	-1	-1	-1
83	12	7	3	1	9	4	5	10	6	-1	-1	-1	-1
92	12	5	1	3	11	6	7	8	4	-1	-1	-1	-1
197	12	7	5	1	2	11	0	8	3	-1	-1	-1	-1
202	12	3	7	5	10	2	1	9	0	-1	-1	-1	-1
172	12	1	3	7	6	10	4	9	5	-1	-1	-1	-1
165	13	0	8	3	2	10	1	4	9	5	6	11	7
90	13	1	9	0	3	11	2	5	10	6	7	8	4
71	14	8	3	2	6	5	9	-1	-1	-1	-1	-1	-1
142	14	9	0	3	7	6	10	-1	-1	-1	-1	-1	-1
29	14	10	1	0	4	7	11	-1	-1	-1	-1	-1	-1
43	14	11	2	1	5	4	8	-1	-1	-1	-1	-1	-1
54	14	7	8	0	2	10	5	-1	-1	-1	-1	-1	-1
108	14	4	9	1	3	11	6	-1	-1	-1	-1	-1	-1
201	14	5	10	2	0	8	7	-1	-1	-1	-1	-1	-1
147	14	6	11	3	1	9	4	-1	-1	-1	-1	-1	-1
113	14	3	0	9	10	6	7	-1	-1	-1	-1	-1	-1
226	14	0	1	10	11	7	4	-1	-1	-1	-1	-1	-1
212	14	1	2	11	8	4	5	-1	-1	-1	-1	-1	-1
184	14	2	3	8	9	5	6	-1	-1	-1	-1	-1	-1
122	15	7	8	0	1	10	6	3	11	2	-1	-1	-1
181	15	6	11	3	0	9	5	2	10	1	-1	-1	-1
218	15	5	10	2	3	8	4	1	9	0	-1	-1	-1
229	15	4	9	1	2	11	7	0	8	3	-1	-1	-1
167	15	2	10	5	4	8	3	6	11	7	-1	-1	-1
91	15	1	9	4	7	11	2	5	10	6	-1	-1	-1
173	15	0	8	7	6	10	1	4	9	5	-1	-1	-1
94	15	3	11	6	5	9	0	7	8	4	-1	-1	-1
87	16	9	4	7	3	2	6	5	-1	-1	-1	-1	-1
117	16	9	1	2	6	7	3	0	-1	-1	-1	-1	-1
171	16	8	7	6	2	1	5	4	-1	-1	-1	-1	-1
186	16	8	0	1	5	6	2	3	-1	-1	-1	-1	-1
93	16	11	6	5	1	0	4	7	-1	-1	-1	-1	-1
213	16	11	3	0	4	5	1	2	-1	-1	-1	-1	-1
174	16	10	5	4	0	3	7	6	-1	-1	-1	-1	-1
234	16	10	2	3	7	4	0	1	-1	-1	-1	-1	-1
109	16	1	0	8	11	6	4	9	-1	-1	-1	-1	-1
107	16	1	10	6	4	8	11	2	-1	-1	-1	-1	-1
62	16	0	3	11	10	5	7	8	-1	-1	-1	-1	-1
61	16	0	9	5	7	11	10	1	-1	-1	-1	-1	-1
151	16	3	2	10	9	4	6	11	-1	-1	-1	-1	-1
158	16	3	8	4	6	10	9	0	-1	-1	-1	-1	-1
203	16	2	1	9	8	7	5	10	-1	-1	-1	-1	-1
199	16	2	11	7	5	9	8	3	-1	-1	-1	-1	-1
182	16	5	6	11	8	0	2	10	-1	-1	-1	-1	-1
214	16	5	9	0	2	11	8	4	-1	-1	-1	-1	-1
211	16	4	5	10	11	3	1	9	-1	-1	-1	-1	-1
227	16	4	8	3	1	10	11	7	-1	-1	-1	-1	-1
233	16	7	4	9	10	2	0	8	-1	-1	-1	-1	-1
121	16	7	11	2	0	9	10	6	-1	-1	-1	-1	-1
124	16	6	7	8	9	1	3	11	-1	-1	-1	-1	-1
188	16	6	10	1	3	8	9	5	-1	-1	-1	-1	-1
115	17	7	3	1	10	6	-1	-1	-1	-1	-1	-1	-1

Reverse Technique (REV)**Pattern Look-up Table (REV)**

Marching Cube Patterns

Pat.	Surf.	Tr.	per	Surf.	Des.	per	Surf.
0	0						
1	1	1			1		
2	1	2			2		
3	1	4			4		
4	2	1	1		1	1	
5	1	3			5		
6	1	5			16		
7	2	4	1		14	1	
8	1	2			2		
9	1	4			14		
10	2	2	2		2	2	
11	1	4			14		
12	2	3	1		5	1	
13	4	1	1	1	1	1	1
14	1	4			14		
15	3	1	1	1	1	1	1
16	2	2	1		2	1	
17	1	3			5		
18	2	1	1		1	1	
19	2	1	1		1	1	
20	1	2			2		
21	1	1			1		
22	0						

Configuration Look-up Table (REV)

Marching Cube Configurations

Conf.	Pat.	Summit	list
0	0	-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
1	1	0 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
2	1	1 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
4	1	2 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
8	1	3 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
16	1	7 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
32	1	4 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
64	1	5 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
128	1	6 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
3	2	1 9 8 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
6	2	2 10 9 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
12	2	3 11 10 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
9	2	0 8 11 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
48	2	7 8 9 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
96	2	4 9 10 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
192	2	5 10 11 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
144	2	6 11 8 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
17	2	3 0 4 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
34	2	0 1 5 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
68	2	1 2 6 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
136	2	2 3 7 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
5	3	8 3 2 10 1 0 -1 -1 -1 -1 -1 -1 -1 -1	
10	3	9 0 3 11 2 1 -1 -1 -1 -1 -1 -1 -1 -1	
80	3	8 4 5 10 6 7 -1 -1 -1 -1 -1 -1 -1 -1	
160	3	9 5 6 11 7 4 -1 -1 -1 -1 -1 -1 -1 -1	
129	3	0 8 7 6 11 3 -1 -1 -1 -1 -1 -1 -1 -1	
24	3	2 3 8 4 7 11 -1 -1 -1 -1 -1 -1 -1 -1	
66	3	0 1 10 6 5 9 -1 -1 -1 -1 -1 -1 -1 -1	
36	3	2 10 5 4 9 1 -1 -1 -1 -1 -1 -1 -1 -1	
33	3	3 0 9 5 4 8 -1 -1 -1 -1 -1 -1 -1 -1	
18	3	7 8 0 1 9 4 -1 -1 -1 -1 -1 -1 -1 -1	
72	3	3 11 6 5 10 2 -1 -1 -1 -1 -1 -1 -1 -1	
132	3	7 6 10 1 2 11 -1 -1 -1 -1 -1 -1 -1 -1	
65	4	0 8 3 5 10 6 -1 -1 -1 -1 -1 -1 -1 -1	
130	4	1 9 0 6 11 7 -1 -1 -1 -1 -1 -1 -1 -1	
20	4	2 10 1 7 8 4 -1 -1 -1 -1 -1 -1 -1 -1	
40	4	3 11 2 4 9 5 -1 -1 -1 -1 -1 -1 -1 -1	
7	5	10 9 8 3 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
14	5	11 10 9 0 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
13	5	8 11 10 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
11	5	9 8 11 2 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
112	5	8 9 10 6 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
224	5	9 10 11 7 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
208	5	10 11 8 4 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
176	5	11 8 9 5 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
137	5	6 2 0 8 7 -1 -1 -1 -1 -1 -1 -1 -1 -1	
152	5	4 6 2 3 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
145	5	0 4 6 11 3 -1 -1 -1 -1 -1 -1 -1 -1 -1	
25	5	2 0 4 7 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
70	5	0 2 6 5 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
100	5	2 6 4 9 1 -1 -1 -1 -1 -1 -1 -1 -1 -1	
98	5	6 4 0 1 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
38	5	4 0 2 10 5 -1 -1 -1 -1 -1 -1 -1 -1 -1	
35	5	3 1 5 4 8 -1 -1 -1 -1 -1 -1 -1 -1 -1	
50	5	1 5 7 8 0 -1 -1 -1 -1 -1 -1 -1 -1 -1	
49	5	5 7 3 0 9 -1 -1 -1 -1 -1 -1 -1 -1 -1	
19	5	7 3 1 9 4 -1 -1 -1 -1 -1 -1 -1 -1 -1	
76	5	5 1 3 11 6 -1 -1 -1 -1 -1 -1 -1 -1 -1	
196	5	7 5 1 2 11 -1 -1 -1 -1 -1 -1 -1 -1 -1	
200	5	3 7 5 10 2 -1 -1 -1 -1 -1 -1 -1 -1 -1	
140	5	1 3 7 6 10 -1 -1 -1 -1 -1 -1 -1 -1 -1	
67	6	6 5 9 8 3 1 10 -1 -1 -1 -1 -1 -1 -1	

131	6	6	11	3	1	9	8	7	-1	-1	-1	-1	-1
134	6	7	6	10	9	0	2	11	-1	-1	-1	-1	-1
22	6	7	8	0	2	10	9	4	-1	-1	-1	-1	-1
28	6	4	7	11	10	1	3	8	-1	-1	-1	-1	-1
44	6	4	9	1	3	11	10	5	-1	-1	-1	-1	-1
41	6	5	4	8	11	2	0	9	-1	-1	-1	-1	-1
73	6	5	10	2	0	8	11	6	-1	-1	-1	-1	-1
56	6	2	3	8	9	5	7	11	-1	-1	-1	-1	-1
52	6	2	10	5	7	8	9	1	-1	-1	-1	-1	-1
97	6	3	0	9	10	6	4	8	-1	-1	-1	-1	-1
104	6	3	11	6	4	9	10	2	-1	-1	-1	-1	-1
194	6	0	1	10	11	7	5	9	-1	-1	-1	-1	-1
193	6	0	8	7	5	10	11	3	-1	-1	-1	-1	-1
148	6	1	2	11	8	4	6	10	-1	-1	-1	-1	-1
146	6	1	9	4	6	11	8	0	-1	-1	-1	-1	-1
21	6	10	1	0	4	7	3	2	-1	-1	-1	-1	-1
81	6	10	6	7	3	0	4	5	-1	-1	-1	-1	-1
42	6	11	2	1	5	4	0	3	-1	-1	-1	-1	-1
162	6	11	7	4	0	1	5	6	-1	-1	-1	-1	-1
69	6	8	3	2	6	5	1	0	-1	-1	-1	-1	-1
84	6	8	4	5	1	2	6	7	-1	-1	-1	-1	-1
138	6	9	0	3	7	6	2	1	-1	-1	-1	-1	-1
168	6	9	5	6	2	3	7	4	-1	-1	-1	-1	-1
161	7	6	11	3	0	9	5	4	8	7	-1	-1	-1
82	7	7	8	0	1	10	6	5	9	4	-1	-1	-1
164	7	4	9	1	2	11	7	6	10	5	-1	-1	-1
88	7	5	10	2	3	8	4	7	11	6	-1	-1	-1
26	7	1	9	4	7	11	2	3	8	0	-1	-1	-1
37	7	2	10	5	4	8	3	0	9	1	-1	-1	-1
74	7	3	11	6	5	9	0	1	10	2	-1	-1	-1
133	7	0	8	7	6	10	1	2	11	3	-1	-1	-1
15	8	8	11	10	9	-1	-1	-1	-1	-1	-1	-1	-1
240	8	11	8	9	10	-1	-1	-1	-1	-1	-1	-1	-1
153	8	0	4	6	2	-1	-1	-1	-1	-1	-1	-1	-1
102	8	4	0	2	6	-1	-1	-1	-1	-1	-1	-1	-1
51	8	3	1	5	7	-1	-1	-1	-1	-1	-1	-1	-1
204	8	1	3	7	5	-1	-1	-1	-1	-1	-1	-1	-1
27	9	1	9	4	7	11	2	-1	-1	-1	-1	-1	-1
39	9	2	10	5	4	8	3	-1	-1	-1	-1	-1	-1
78	9	3	11	6	5	9	0	-1	-1	-1	-1	-1	-1
141	9	0	8	7	6	10	1	-1	-1	-1	-1	-1	-1
177	9	6	11	3	0	9	5	-1	-1	-1	-1	-1	-1
114	9	7	8	0	1	10	6	-1	-1	-1	-1	-1	-1
228	9	4	9	1	2	11	7	-1	-1	-1	-1	-1	-1
216	9	5	10	2	3	8	4	-1	-1	-1	-1	-1	-1
195	10	1	10	11	3	5	9	8	7	-1	-1	-1	-1
60	10	7	11	10	5	3	8	9	1	-1	-1	-1	-1
105	10	0	9	10	2	4	8	11	6	-1	-1	-1	-1
150	10	2	11	8	0	6	10	9	4	-1	-1	-1	-1
85	10	3	2	6	7	1	0	4	5	-1	-1	-1	-1
170	10	2	1	5	6	0	3	7	4	-1	-1	-1	-1
99	11	3	1	10	6	4	8	-1	-1	-1	-1	-1	-1
198	11	0	2	11	7	5	9	-1	-1	-1	-1	-1	-1
156	11	1	3	8	4	6	10	-1	-1	-1	-1	-1	-1
57	11	2	0	9	5	7	11	-1	-1	-1	-1	-1	-1
116	11	8	9	1	2	6	7	-1	-1	-1	-1	-1	-1
232	11	9	10	2	3	7	4	-1	-1	-1	-1	-1	-1
209	11	10	11	3	0	4	5	-1	-1	-1	-1	-1	-1
178	11	11	8	0	1	5	6	-1	-1	-1	-1	-1	-1
23	11	7	3	2	10	9	4	-1	-1	-1	-1	-1	-1
46	11	4	0	3	11	10	5	-1	-1	-1	-1	-1	-1
77	11	5	1	0	8	11	6	-1	-1	-1	-1	-1	-1
139	11	6	2	1	9	8	7	-1	-1	-1	-1	-1	-1
135	12	10	9	8	7	6	2	11	3	-1	-1	-1	-1
30	12	11	10	9	4	7	3	8	0	-1	-1	-1	-1
45	12	8	11	10	5	4	0	9	1	-1	-1	-1	-1
75	12	9	8	11	6	5	1	10	2	-1	-1	-1	-1

120	12	8	9	10	2	3	7	11	6	-1	-1	-1	-1
225	12	9	10	11	3	0	4	8	7	-1	-1	-1	-1
210	12	10	11	8	0	1	5	9	4	-1	-1	-1	-1
180	12	11	8	9	1	2	6	10	5	-1	-1	-1	-1
169	12	6	2	0	9	5	4	8	7	-1	-1	-1	-1
154	12	4	6	2	1	9	3	8	0	-1	-1	-1	-1
149	12	0	4	6	10	1	2	11	3	-1	-1	-1	-1
89	12	2	0	4	5	10	7	11	6	-1	-1	-1	-1
86	12	0	2	6	7	8	5	9	4	-1	-1	-1	-1
101	12	2	6	4	8	3	0	9	1	-1	-1	-1	-1
106	12	6	4	0	3	11	1	10	2	-1	-1	-1	-1
166	12	4	0	2	11	7	6	10	5	-1	-1	-1	-1
163	12	3	1	5	6	11	4	8	7	-1	-1	-1	-1
58	12	1	5	7	11	2	3	8	0	-1	-1	-1	-1
53	12	5	7	3	2	10	0	9	1	-1	-1	-1	-1
83	12	7	3	1	10	6	5	9	4	-1	-1	-1	-1
92	12	5	1	3	8	4	7	11	6	-1	-1	-1	-1
197	12	7	5	1	0	8	2	11	3	-1	-1	-1	-1
202	12	3	7	5	9	0	1	10	2	-1	-1	-1	-1
172	12	1	3	7	4	9	6	10	5	-1	-1	-1	-1
165	13	0	9	1	2	11	3	6	10	5	4	8	7
90	13	3	8	0	1	10	2	5	9	4	7	11	6
71	14	8	3	2	6	5	9	-1	-1	-1	-1	-1	-1
142	14	9	0	3	7	6	10	-1	-1	-1	-1	-1	-1
29	14	10	1	0	4	7	11	-1	-1	-1	-1	-1	-1
43	14	11	2	1	5	4	8	-1	-1	-1	-1	-1	-1
54	14	7	8	0	2	10	5	-1	-1	-1	-1	-1	-1
108	14	4	9	1	3	11	6	-1	-1	-1	-1	-1	-1
201	14	5	10	2	0	8	7	-1	-1	-1	-1	-1	-1
147	14	6	11	3	1	9	4	-1	-1	-1	-1	-1	-1
113	14	3	0	9	10	6	7	-1	-1	-1	-1	-1	-1
226	14	0	1	10	11	7	4	-1	-1	-1	-1	-1	-1
212	14	1	2	11	8	4	5	-1	-1	-1	-1	-1	-1
184	14	2	3	8	9	5	6	-1	-1	-1	-1	-1	-1
122	15	7	11	6	1	10	2	3	8	0	-1	-1	-1
181	15	6	10	5	0	9	1	2	11	3	-1	-1	-1
218	15	5	9	4	3	8	0	1	10	2	-1	-1	-1
229	15	4	8	7	2	11	3	0	9	1	-1	-1	-1
167	15	2	11	3	4	8	7	6	10	5	-1	-1	-1
91	15	1	10	2	7	11	6	5	9	4	-1	-1	-1
173	15	0	9	1	6	10	5	4	8	7	-1	-1	-1
94	15	3	8	0	5	9	4	7	11	6	-1	-1	-1
87	16	3	2	6	7	5	9	4	-1	-1	-1	-1	-1
117	16	3	2	6	7	0	9	1	-1	-1	-1	-1	-1
171	16	2	1	5	6	4	8	7	-1	-1	-1	-1	-1
186	16	2	1	5	6	3	8	0	-1	-1	-1	-1	-1
93	16	1	0	4	5	7	11	6	-1	-1	-1	-1	-1
213	16	1	0	4	5	2	11	3	-1	-1	-1	-1	-1
174	16	0	3	7	4	6	10	5	-1	-1	-1	-1	-1
234	16	0	3	7	4	1	10	2	-1	-1	-1	-1	-1
109	16	11	6	4	8	0	9	1	-1	-1	-1	-1	-1
107	16	11	6	4	8	1	10	2	-1	-1	-1	-1	-1
62	16	10	5	7	11	3	8	0	-1	-1	-1	-1	-1
61	16	10	5	7	11	0	9	1	-1	-1	-1	-1	-1
151	16	9	4	6	10	2	11	3	-1	-1	-1	-1	-1
158	16	9	4	6	10	3	8	0	-1	-1	-1	-1	-1
203	16	8	7	5	9	1	10	2	-1	-1	-1	-1	-1
199	16	8	7	5	9	2	11	3	-1	-1	-1	-1	-1
182	16	8	0	2	11	6	10	5	-1	-1	-1	-1	-1
214	16	8	0	2	11	5	9	4	-1	-1	-1	-1	-1
211	16	11	3	1	10	5	9	4	-1	-1	-1	-1	-1
227	16	11	3	1	10	4	8	7	-1	-1	-1	-1	-1
233	16	10	2	0	9	4	8	7	-1	-1	-1	-1	-1
121	16	10	2	0	9	7	11	6	-1	-1	-1	-1	-1
124	16	9	1	3	8	7	11	6	-1	-1	-1	-1	-1
188	16	9	1	3	8	6	10	5	-1	-1	-1	-1	-1
115	17	7	3	1	10	6	-1	-1	-1	-1	-1	-1	-1

APPENDIX B GENERAL TOOLS (C++ PROGRAMS)

This appendix contains a few C++ tools developed for general purpose of this dissertation. They are

- FileStuff: functions used to get information about files.
- RandomNumber: to implement the Unix rand48 random number generator.
- Equations: to solve 1st, 2nd, and 3rd degree real equations.
- Histogram1D: to handle a data to store a one-dimension histogram.
- Histogram2D: to handle a data to store a two-dimension histogram.
- Image256GL: to handle a data to store a 3D gray-level image that contains one byte per voxel (256 gray levels).

Each tool contains a '.cpp' file that contains the code of the tool and a '.h' file that must be included (#include C pre-compiler directive) in any program that uses the tool.

FileStuff

FileStuff.h

```
/* FileStuff.h */
/* A few functions about files. */
unsigned int FileSize(char *FileName);
```

FileStuff.cpp

```
/* FileStuff.cpp */
/* A few functions about files. */
#include <stdio.h>
#include <stdlib.h>

/* to define the file-size location in the string returned by the Unix */
/* command 'ls -l' */
#define FILE_SIZE_LOCATION 33

/* unsigned int FileSize(char *FileName) */
/* Returns the file size (number of bytes) of a file which full name is */
/* 'FileName'.
```

```
/*                                                                 */
/*****                                                             */
unsigned int FileSize(char *FileName)
{
    FILE *ResultFile;
    char CmdString[500];
    char ResultString[500];
    unsigned int Size;

    sprintf(CmdString, "ls -l %s > toto 2> /dev/null", FileName);
    system(CmdString);
    ResultFile = fopen("toto","r");
    if ( fgets(ResultString, 500, ResultFile) == NULL ) {
        Size = 0;
    }
    else {
        Size = atoi(&ResultString[FILE_SIZE_LOCATION]);
    }
    fclose(ResultFile);
    system("rm toto");
    return(Size);
}
```

RandomNumber

RandomNumber.h

```

/*****
/*  RandomNumber.h
/*
/*  To use the Solaris rand48 random number generator.
/*
/*
/*****

void InitRandomGenerator(int SeedNumber);
double GetRandomNumber();

```

RandomNumber.cpp

```

/*****
/*  RandomNumber.cpp
/*
/*  To use the Solaris rand48 random number generator.
/*
/*
/*****
#include <stdio.h>
#include <stdlib.h>

/*****
/*
/*          void InitRandomGenerator(int SeedNumber)
/*
/*  To initialize the series.
/*
/*
/*****
void InitRandomGenerator(int SeedNumber)
{
    srand48(SeedNumber);
}

/*****
/*
/*          double GetRandomNumber()
/*
/*  To get the next number in the series.
/*
/*
/*****
double GetRandomNumber()
{
    double RandomNumber;

    RandomNumber = drand48();
    return(RandomNumber);
}

```



```

/*                                     double *Root)                               */
/*                                     */
/* Solves the second degree equation:  $axx + bx + c = 0$  where 'a', 'b', and */
/* 'c' are real. Returns the number of real roots in 'NbRoots' and the roots */
/* themselves in the array 'Root'. If 'a' is equals to zero, it will call */
/* 'FirstDegreeEqn' to solve  $bx + c = 0$ . */
/*                                     */
/*                                     */
/*****
void SecondDegreeEqn(double a, double b, double c,
                    unsigned int *NbRoots, double *Root)
{
    if (a == 0.0) { /* to solve  $bx + c = 0$  */
        FirstDegreeEqn(b, c, NbRoots, Root);
    }
    else {
        if (c == 0.0) {
            if (b == 0.0) { /* to solve  $axx = 0$  */
                Root[0] = 0.0;
                *NbRoots = 1;
            }
            else { /* to solve  $(ax + b)x = 0$  */
                Root[0] = -b/a;
                Root[1] = 0.0;
                *NbRoots = 2;
            }
        }
        else { /* to solve  $axx + bx + c = 0$  */
            double Dis = b*b - 4*a*c;
            if (Dis <= 0.0) { /* 1 or 0 solution */
                *NbRoots = 0;
                if (Dis == 0.0) { /* 1 solution */
                    Root[*NbRoots] = -b/(2.0*a);
                    *NbRoots += 1;
                }
            }
            else { /* 2 solutions */
                double q;
                if (b <= 0.0) { /* Note: if b = 0 then  $x_1 = -x_2$  */
                    q = -0.5*(b - sqrt(Dis));
                }
                else {
                    q = -0.5*(b + sqrt(Dis));
                }
                Root[0] = q / a;
                Root[1] = c / q;
                *NbRoots = 2;
            }
        }
    }
}

/*****
/*                                     */
/*                                     void ThirdDegreeEqn(double a, */
/*                                     double b, */
/*                                     double c, */
/*                                     double d, */
/*                                     unsigned int *NbRoots, */
/*                                     double *Root) */
/*                                     */
/* Solves the third degree equation:  $axxx + bxx + cx + d = 0$  where 'a', 'b', */
/* 'c', and 'd' are real. Returns the number of real roots in 'NbRoots' and */
/* the roots themselves in the array 'Root'. If 'a' is equals to zero, it */
/* will call 'SecondDegreeEqn' to solve  $bxx + cx + d = 0$ . If 'd' is equals */
/* to zero and none of the other coefficients are equal to zero, it will */
/* call 'SecondDegreeEqn' to solve  $axx + bx + c = 0$ . */
/*                                     */
/*                                     */
/*****
void ThirdDegreeEqn(double a, double b, double c, double d,
                  unsigned int *NbRoots, double *Root)
{
    if (a == 0.0) { /* to solve  $bxx + cx + d = 0$  */
        SecondDegreeEqn(b, c, d, NbRoots, Root);
    }
}

```

```

else {
  if (d == 0.0) {
    if (c == 0.0) {
      if (b == 0.0) { /* to solve axxx = 0 */
        Root[0] = 0.0;
        *NbRoots = 1;
      }
      else { /* to solve (ax + b)xx = 0 */
        Root[0] = -b/a;
        Root[1] = 0.0;
        *NbRoots = 2;
      }
    }
    else { /* to solve (axx + bx + c)x = 0 */
      SecondDegreeEqn(a, b, c, NbRoots, Root);
      Root[*NbRoots] = 0.0;
      *NbRoots += 1;
    }
  }
  else { /* to solve axxx + bxx + cx + d = 0 */
    double p = b/a;
    double q = c/a;
    double r = d/a;
    double Shift = p/3.0;
    double Q = (p*p - 3.0*q) / 9.0;
    double R = (2.0*p*p*p - 9.0*p*q + 27.0*r) / 54.0;
    double Dis = R*R - Q*Q*Q;
    if (Dis >= 0.0) { /* 1 or 2 solutions */
      double A, B;
      if (R >= 0.0) {
        A = - pow(( R + sqrt(Dis)), (1.0/3.0));
      }
      else {
        A = pow((-R + sqrt(Dis)), (1.0/3.0));
      }
      if (A == 0.0) {
        B = 0.0;
      }
      else {
        B = Q / A;
      }
      Root[0] = A + B - Shift;
      *NbRoots = 1;
      if (Dis == 0.0) { /* 2 solutions */
        Root[*NbRoots] = -0.5*(A + B) - Shift;
        *NbRoots += 1;
      }
    }
    else { /* 3 solutions */
      double Theta = acos(R/pow(Q, 1.5));
      double TwoSqrtQ = 2.0 * sqrt(Q);
      Root[0] = -TwoSqrtQ*cos(Theta/3.0) - Shift;
      Root[1] = -TwoSqrtQ*cos((Theta + 2.0*M_PI)/3.0) - Shift;
      Root[2] = -TwoSqrtQ*cos((Theta - 2.0*M_PI)/3.0) - Shift;
      *NbRoots = 3;
    }
  }
}
}
}
}

```

Histogram1D

Histogram1D.h

```

/*****
/*
/*   Histogram1D.h
/*
/*   This module is a tool to store a one dimension histogram. The bins
/*   range from 0 to BinNumber. Bin number BinNumber contains all values
/*   above or equal to BinNumber. Negatives values are ignored.
/*
/*
/*****

class Histogram1D {
private:
    int BinNumber;
    int *Bin;
public:
    Histogram1D(int Bins);
    ~Histogram1D(void);
    void AddValue(int Value);
    void StoreToDisk(char *FileName);
};

```

Histogram1D.cpp

```

/*****
/*
/*   Histogram1D.cpp
/*
/*   This module is a tool to store a one dimension histogram. The bins
/*   range from 0 to BinNumber. Bin number BinNumber contains all values
/*   above or equal to BinNumber. Negatives values are ignored.
/*
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include "Histogram1D.h"

/*****
/*
/*           Histogram1D::Histogram1D(int Bins)
/*
/*   Constructor that allocate the memory for an empty histogram of 'Bins'
/*   bins. Each bin is initialized with zero counts in it.
/*
/*
/*****

Histogram1D::Histogram1D(int Bins)
{
    int NumBin;

    /* to store the size of the histogram */
    Histogram1D::BinNumber = Bins;
    /* to allocate the memory */
    Histogram1D::Bin = new int[Bins + 1];
    if (Histogram1D::Bin == NULL) {
        printf("Error in Histogram1D::Histogram1D: ");
        printf("can't allocate memory.\n");
        exit(0);
    }
    /* to initialize the new histogram */
    for (NumBin=0; NumBin<=Bins; NumBin++) {
        Histogram1D::Bin[NumBin] = 0;
    }
}

```

```

}

/*****
/*
/*          Histogram1D::~Histogram1D(void)
/*
/* Destructeur that free the memory allocated to the histogram.
/*
/*
/*****
Histogram1D::~Histogram1D(void)
{
    /* to free the memory allocated by the constructor */
    delete Histogram1D::Bin;
}

/*****
/*
/*          void Histogram1D::AddValue(int Value)
/*
/* Add a new count in the bin corresponding to 'Value'.
/*
/*
/*****
void Histogram1D::AddValue(int Value)
{

    /* To add the new value in the corresponding bin */
    if (Value >= Histogram1D::BinNumber) {
        Histogram1D::Bin[Histogram1D::BinNumber] =
            Histogram1D::Bin[Histogram1D::BinNumber] + 1;
    }
    else {
        if (Value >= 0) {
            Histogram1D::Bin[Value] = Histogram1D::Bin[Value] + 1;
        }
    }
}

/*****
/*
/*          void Histogram1D::StoreToDisk(char *FileName)
/*
/* To store the histogram on the disk under the name 'FileName'.
/*
/*
/*****
void Histogram1D::StoreToDisk(char *FileName)
{
    int NumBin;
    FILE *OutputFile;
    int Counter;

    /* To open the file and test if ok */
    OutputFile = fopen(FileName, "w");
    if ( OutputFile == NULL ) {
        printf("Error in Histogram1D::StoreToDisk: ");
        printf("impossible to write output file '%s'.\n", FileName);
        exit(0);
    }
    /* to count the total cumul over the histogram */
    Counter = 0;
    for (NumBin=0; NumBin<=Histogram1D::BinNumber; NumBin++) {
        Counter = Counter + Histogram1D::Bin[NumBin];
    }
    /* To put the histogram in the file */
    /* header */
    fprintf(OutputFile, "One dimension histogram.\n");
    fprintf(OutputFile, "  Number of bins: %6d\n", Histogram1D::BinNumber);
    /* the histogram itself */
    fprintf(OutputFile, "  Bin #      Counts      Frequency\n\n");
    for (NumBin=0; NumBin<Histogram1D::BinNumber; NumBin++) {

```

```
fprintf(OutputFile, "%7d", NumBin);
fprintf(OutputFile, "%14d", Histogram1D::Bin[NumBin]);
fprintf(OutputFile, "%14.4f %%\n",
        100.0 * Histogram1D::Bin[NumBin] / Counter);
}
fprintf(OutputFile, " Above");
fprintf(OutputFile, "%14d", Histogram1D::Bin[Histogram1D::BinNumber]);
fprintf(OutputFile, "%14.4f %%\n\n",
        100.0 * Histogram1D::Bin[Histogram1D::BinNumber] / Counter);
fprintf(OutputFile, " Total");
fprintf(OutputFile, "%14d", Counter);
fprintf(OutputFile, "%14.4f %%\n", 100.0);
fclose(OutputFile);
}
```

Histogram2D

Histogram2D.h

```

/*****
/*
/*   Histogram2D.h
/*
/*   This module is a tool to store a two dimension histogram. The bins
/*   range from (0, 0) to (BinNumber1, BinNumber2). Bin number
/*   (BinNumber1, n) contains all values above or equal to BinNumber1 in
/*   the first dimension. Bin number (n, BinNumber2) contains all values
/*   above or equal to BinNumber2 in the second dimension. Negatives
/*   values are ignored.
/*
/*
/*****

class Histogram2D {
private:
    int BinNumber1;
    int BinNumber2;
    int **Bin;
public:
    Histogram2D(int Bins1, int Bins2);
    ~Histogram2D(void);
    void AddValue(int Value1, int Value2);
    void StoreToDisk(char *FileName);
};

```

Histogram2D.cpp

```

/*****
/*
/*   Histogram2D.cpp
/*
/*   This module is a tool to store a two dimension histogram. The bins
/*   range from (0, 0) to (BinNumber1, BinNumber2). Bin number
/*   (BinNumber1, n) contains all values above or equal to BinNumber1 in
/*   the first dimension. Bin number (n, BinNumber2) contains all values
/*   above or equal to BinNumber2 in the second dimension. Negatives
/*   values are ignored.
/*
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include "Histogram2D.h"

/*****
/*
/*           Histogram2D::Histogram2D(int Bins1,
/*                                   int Bins2)
/*
/*   Constructor that allocate the memory for an empty histogram of 'Bins1'
/*   time 'Bins2' bins. Each bin is initialized with zero counts in it.
/*
/*
/*****
Histogram2D::Histogram2D(int Bins1, int Bins2)
{
    int NumBin1, NumBin2;

    /* to store the size of the histogram */
    Histogram2D::BinNumber1 = Bins1;
    Histogram2D::BinNumber2 = Bins2;
    /* to allocate the memory */
    /* a) a pointer for each row */

```

```

Histogram2D::Bin = new int *[Bins1 + 1];
if (Histogram2D::Bin == NULL) {
    printf("Error in Histogram2D::Histogram2D: ");
    printf("can't allocate memory.\n");
    exit(0);
}
/* b) a vector for each row */
for (NumBin1=0; NumBin1<=Bins1; NumBin1++) {
    Histogram2D::Bin[NumBin1] = new int[Bins2 + 1];
    /* to initialize the new row */
    for (NumBin2=0; NumBin2<=Bins2; NumBin2++) {
        Histogram2D::Bin[NumBin1][NumBin2] = 0;
    }
}

/*****
/*
/*          Histogram2D::~Histogram2D(void)
/*
/*
/* Destructer that free the memory allocated to the histogram.
/*
/*
/*****
Histogram2D::~Histogram2D(void)
{
    unsigned int NumBin1;

    /* to free the memory allocated by the constructor */
    for (NumBin1=0; NumBin1<=Histogram2D::BinNumber1; NumBin1++) {
        delete Histogram2D::Bin[NumBin1];
    }
    delete Histogram2D::Bin;
}

/*****
/*
/*          void Histogram2D::AddValue(int Value1,
/*
/*          int Value2)
/*
/*
/* Add a new count in the bin corresponding to 'Value1' times 'Value2'.
/*
/*
/*****
void Histogram2D::AddValue(int Value1, int Value2)
{
    /* To add the new value in the corresponding bin */
    if (Value1 >= 0 && Value2 >=0) {
        if (Value1 >= Histogram2D::BinNumber1) {
            if (Value2 >= Histogram2D::BinNumber2) {
                Histogram2D::Bin[Histogram2D::BinNumber1]
                    [Histogram2D::BinNumber2] += 1;
            }
            else {
                Histogram2D::Bin[Histogram2D::BinNumber1][Value2] += 1;
            }
        }
        else {
            if (Value2 >= Histogram2D::BinNumber2) {
                Histogram2D::Bin[Value1][Histogram2D::BinNumber2] += 1;
            }
            else {
                Histogram2D::Bin[Value1][Value2] += 1;
            }
        }
    }
}

/*****
/*
/*          void Histogram2D::StoreToDisk(char *FileName)
/*

```

```

/*                                                                 */
/* To store the histogram on the disk under the name 'FileName'.  */
/*                                                                 */
/*****
void Histogram2D::StoreToDisk(char *FileName)
{
    int NumBin1, NumBin2;
    FILE *OutputFile;
    int Counter;

    /* To open the file and test if ok */
    OutputFile = fopen(FileName, "w");
    if ( OutputFile == NULL ) {
        printf("Error in Histogram2D::StoreToDisk: ");
        printf("impossible to write output file '%s'.\n", FileName);
        exit(0);
    }
    /* to count the total cumul over the histogram */
    Counter = 0;
    for (NumBin1=0; NumBin1<=Histogram2D::BinNumber1; NumBin1++) {
        for (NumBin2=0; NumBin2<=Histogram2D::BinNumber2; NumBin2++) {
            Counter = Counter + Histogram2D::Bin[NumBin1][NumBin2];
        }
    }
    /* To put the histogram in the file */
    /* header */
    fprintf(OutputFile, "Two dimension histogram.\n");
    fprintf(OutputFile, "  Number of bins: %d x %d\n",
        Histogram2D::BinNumber1, Histogram2D::BinNumber2);
    fprintf(OutputFile, "  Relative frequency per bin\n\n");
    /* the histogram itself */
    fprintf(OutputFile, "      ");
    for (NumBin2=0; NumBin2<Histogram2D::BinNumber2; NumBin2++) {
        fprintf(OutputFile, "%9d", NumBin2);
    }
    fprintf(OutputFile, "  Above\n");
    for (NumBin1=0; NumBin1<Histogram2D::BinNumber1; NumBin1++) {
        fprintf(OutputFile, "%7d", NumBin1);
        for (NumBin2=0; NumBin2<Histogram2D::BinNumber2; NumBin2++) {
            fprintf(OutputFile, "%9.2e",
                (float)Histogram2D::Bin[NumBin1]
                    [NumBin2] / Counter);
        }
        fprintf(OutputFile, "%9.2e\n",
            (float)Histogram2D::Bin[NumBin1]
                [Histogram2D::BinNumber2] / Counter);
    }
    fprintf(OutputFile, "  Above");
    for (NumBin2=0; NumBin2<Histogram2D::BinNumber2; NumBin2++) {
        fprintf(OutputFile, "%9.2e",
            (float)Histogram2D::Bin[Histogram2D::BinNumber1]
                [NumBin2] / Counter);
    }
    fprintf(OutputFile, "%9.2e\n",
        (float)Histogram2D::Bin[Histogram2D::BinNumber1]
            [Histogram2D::BinNumber2] / Counter);
    fprintf(OutputFile, "  Total");
    fprintf(OutputFile, "%14d", Counter);
    fprintf(OutputFile, "%14.4f %%\n", 100.0);
    fclose(OutputFile);
}

```

Image256GL

Image256GL.h

```

/*****
/*  Image256GL.h
/*
/*  Tool for management of an image file.  The image is a 256 gray level
/*  image and is stored with one byte per voxel.
/*  The image size is: Nz slices (indexed by K)
/*                      Ny rows per slice (indexed by J)
/*                      Nx voxels per row (indexed by I)
/*  The image is stored slice by slice, row by row within a slice, and
/*  voxel by voxel within a row.  The first voxel is (I,J,K)=(0,0,0), the
/*  second is (I,J,K)=(1,0,0), the third is (I,J,K)=(2,0,0), etc.  The
/*  three last voxels of the image are (Nx-3,Ny-1,Nz-1), (Nx-2,Ny-1,Nz-1),
/*  and (Nx-1,Ny-1,Nz-1).
/*
/*
/*****

class Image256GL {
private:
    unsigned char *Voxel;
public:
    Image256GL(int VoxelNumber, int RowNumberPerSlice, int SliceNumberPerRow);
    ~Image256GL(void);
    int Nx;
    int Ny;
    int Nz;
    void SetGrayLevel(int I, int J, int K, unsigned char GrayLevel);
    unsigned char GrayLevel(int I, int J, int K);
    void StoreToFile(char *ImageFileName);
    void LoadFromFile(char *ImageFileName);
};

```

Image256GL.cpp

```

/*****
/*  Image256GL.cpp
/*
/*  Tool for management of an image file.  The image is a 256 gray level
/*  image and is stored with one byte per voxel.
/*  The image size is: Nz slices (indexed by K)
/*                      Ny rows per slice (indexed by J)
/*                      Nx voxels per row (indexed by I)
/*  The image is stored slice by slice, row by row within a slice, and
/*  voxel by voxel within a row.  The first voxel is (I,J,K)=(0,0,0), the
/*  second is (I,J,K)=(1,0,0), the third is (I,J,K)=(2,0,0), etc.  The
/*  three last voxels of the image are (Nx-3,Ny-1,Nz-1), (Nx-2,Ny-1,Nz-1),
/*  and (Nx-1,Ny-1,Nz-1).
/*
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Image256GL.h"

/*****
/*
/*          Image256GL::Image256GL(int VoxelNumber,
/*                                int RowNumber,
/*                                int SliceNumber)
/*
/*  Constructor that allocate the memory for an empty 3D image that contains
/*  'SliceNumber' slices of 'RowNumber' rows of 'VoxelNumber' voxels.
/*

```

```

/*                                                                 */
/*****                                                             */
Image256GL::Image256GL(int VoxelNumber, int RowNumber, int SliceNumber)
{
    int NVoxels;
    int IVoxel;

    /* to save the size of the image */
    Image256GL::Nx = VoxelNumber;
    Image256GL::Ny = RowNumber;
    Image256GL::Nz = SliceNumber;
    /* allocate the memory to store the image */
    Image256GL::Voxel = new unsigned char[VoxelNumber*RowNumber*SliceNumber];
    if (Image256GL::Voxel == NULL) {
        printf(" Error in Image256GL::Image256GL: ");
        printf("can't allocate memory.\n");
        exit(0);
    }
    /* to initialize the memory with 0 */
    NVoxels = VoxelNumber*RowNumber*SliceNumber;
    for (IVoxel=0; IVoxel<NVoxels; IVoxel++) {
        Image256GL::Voxel[IVoxel] = 0;
    }
}

/*****                                                             */
/*                                                                 */
/*          Image256GL::~Image256GL(void)                          */
/*                                                                 */
/* Destructer that free the memory allocated to the image.        */
/*                                                                 */
/*****                                                             */
Image256GL::~Image256GL(void)
{
    /* to free the memory allocated by the constructor */
    delete Image256GL::Voxel;
}

/*****                                                             */
/*                                                                 */
/*          void Image256GL::SetGrayLevel(int I,                   */
/*                                     int J,                       */
/*                                     int K,                       */
/*                                     unsigned char GrayLevel)    */
/*                                                                 */
/* To set the gray level of the voxel identified by its location (I, J, K) */
/* with the value 'GrayLevel'.                                         */
/*                                                                 */
/*****                                                             */
void Image256GL::SetGrayLevel(int I, int J, int K, unsigned char GrayLevel)
{
    int IVoxel;

    /* to find the voxel location in the image */
    IVoxel = (K * Image256GL::Ny + J) * Image256GL::Nx + I;
    /* to set the gray level in the image */
    Image256GL::Voxel[IVoxel] = GrayLevel;
}

/*****                                                             */
/*                                                                 */
/*          unsigned char Image256GL::GrayLevel(int I,             */
/*                                     int J,                       */
/*                                     int K)                      */
/*                                                                 */
/* Returns the gray level of the voxel identified by its location (I, J, K). */
/*                                                                 */
/*****                                                             */
unsigned char Image256GL::GrayLevel(int I, int J, int K)

```

```

{
int IVoxel;
unsigned char GrayLevel;

/* to find the voxel location in the image */
IVoxel = (K * Image256GL::Ny + J) * Image256GL::Nx + I;
/* to get the gray level from the image */
GrayLevel = Image256GL::Voxel[IVoxel];
/* to return the value */
return(GrayLevel);
}

/*****
*/
/*          void Image256GL::StoreToFile(char *ImageFileName)          */
/*          */
/* To store the image into a file which name is contains in 'ImageFileName'. */
/*          */
/*****
void Image256GL::StoreToFile(char *ImageFileName)
{
FILE *ImageFileToWrite;
int NVoxels;
int IVoxel;

/* to open the file */
ImageFileToWrite = fopen(ImageFileName,"w");
if (ImageFileToWrite == NULL) {
printf("Error in Image256GL::StoreToFile: ");
printf("can't open file: '%s'\n", ImageFileName);
exit(0);
}

/* to write the file */
NVoxels = Image256GL::Nx * Image256GL::Ny * Image256GL::Nz;
for (IVoxel=0; IVoxel<NVoxels; IVoxel++) {
fputc( Image256GL::Voxel[IVoxel],ImageFileToWrite);
}
fclose(ImageFileToWrite);
}

/*****
*/
/*          void Image256GL::LoadFromFile(char *ImageFileName)          */
/*          */
/* To load the image in memory from the file which name is contains in */
/* 'ImageFileName'. */
/*          */
/*****
void Image256GL::LoadFromFile(char *ImageFileName)
{
FILE *ImageFileToRead;
int NVoxels;
int IVoxel;

/* to open the file */
ImageFileToRead = fopen(ImageFileName,"r");
if (ImageFileToRead == NULL) {
printf("Error in Image256GL::LoadFromFile: ");
printf("can't read file: '%s'\n", ImageFileName);
exit(0);
}

/* to read the file */
NVoxels = Image256GL::Nx * Image256GL::Ny * Image256GL::Nz;
for (IVoxel=0; IVoxel<NVoxels; IVoxel++) {
Image256GL::Voxel[IVoxel]=(unsigned char)fgetc(ImageFileToRead);
}
fclose(ImageFileToRead);
}

```

APPENDIX C LOOK-UP TABLE TOOLS (C++ PROGRAMS)

This appendix contains all the C++ classes and functions used to read and handle the look-up tables described in [Appendix A](#). They are

- **FileDefinition:** List of constant data that define the position of each column in the look-up tables. Only a '.h' file for this tool.
- **Design:** C++ class that stores one line of the design look-up table.
- **SurfaceP:** C++ class that stores the list of the designs associated to a surface of a pattern. It uses the tool Design to store each design.
- **Pattern:** C++ class that stores one line of the pattern look-up table and the list of the surfaces associated to the pattern. It uses the tool SurfaceP to store each surface.
- **SurfaceC:** C++ class that stores the list of the cube edges that define a surface of a configuration.
- **Configuration:** C++ class that stores one line of the configuration look-up table and the list of the surfaces associated to a configuration. It uses the tool SurfaceC to store each surface.
- **MCConfigurations:** C++ class that stores the entire database found in the three look-up tables. It stores the list of patterns and the list of configurations. It uses the tools Pattern and Configuration to store each pattern and each configuration respectively.

Each tool contains a '.cpp' file that contains the code of the tool and a '.h' file that must be included (#include C pre-compiler directive) in any program that uses the tool. Note that only the tool MCConfiguration should be used since the other one are just sub-tools that help to handle the complex structure. Any information concerning the patterns and the configurations is accessible from the MCConfiguration tool.

FileDefinition

FileDefinition.h

```
/* FileDefinition.h */
/* This include file contains all the constant values used to locate
   the different information in the input files that contain the Marching
   Cube configurations and patterns. */
/* Patterns look-up table. */
#define PAT_FILE_NAME "Patterns.prn"
#define PAT_LINE_HEADER 2
#define PAT_POS_PATTERN 0
#define PAT_POS_NB_SURFACES 8
#define PAT_POS_TRIANGLE_LIST 21
```

```
#define PAT_LG_PER_TRIANGLE      3
#define PAT_POS_DESIGN_LIST      38
#define PAT_LG_PER_DESIGN        3

/* Designs look-up table. */
#define DES_FILE_NAME            "Designs.prn"
#define DES_LINE_HEADER          2
#define DES_POS_PATTERN          0
#define DES_POS_SURFACE          8
#define DES_POS_SUMMIT_LIST      21
#define DES_LG_PER_SUMMIT        3

/* Configurations look-up table. */
#define CON_FILE_NAME            "Configurations.prn"
#define CON_LINE_HEADER          2
#define CON_POS_CONFIG           0
#define CON_POS_PATTERN          8
#define CON_POS_SUMMIT_LIST      21
#define CON_LG_PER_SUMMIT        3
```

Design

Design.h

```

/*****
/*   Design.h
/*
/*   This program is a tool to handle the triangle designs of a surface.
/*   For each triangle that constitutes the surface, it stores the summits
/*   of the surface used for designing the triangle. The order of the
/*   summits gives the positive orientation of the triangle. All triangles
/*   of the surface have the same positive orientation.
/*
/*
/*****

/* The class to store one design */
class Design {
private:
    int *SummitList; /* Pointer to list of summits. */
                    /* we don't know how many summits yet */
public:
    Design(int NbTr, char *StringSummitList);
    ~Design(void);
    int NbTriangles;
    int NbSummits;
    int SummitNoOfSurf(int TriangleNo, int SummitOfTr);
};

```

Design.cpp

```

/*****
/*   Design.cpp
/*
/*   This program is a tool to handle the triangle designs of a surface.
/*   For each triangle that constitutes the surface, it stores the summits
/*   of the surface used for designing the triangle. The order of the
/*   summits gives the positive orientation of the triangle. All triangles
/*   of the surface have the same positive orientation.
/*
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include "FileDefinition.h"
#include "Design.h"

#define NB_SUMMITS_IN_TRIANGLE          3

/*****
/*
/*           Design::Design(int NbTr,
/*                           char *StringSummitList)
/*
/*
/* Constructor that allocate the memory for the list of summits of the
/* 'NbTr' triangles. The summit list is read from the 'StringSummitList'
/* string. This string should come from the design look-up table.
/*
/*
/*****
Design::Design(int NbTr, char *StringSummitList)
{
    int TrNo;
    int SuNo;
    int LocationInString;
    int SummitNoInList;
    int SummitNoInSurf;
}

```

```

if (NbTr <= 0 ) {
    printf("Error in Design::constructor: ");
    printf("number of triangles %d cannot be negative.\n", NbTr);
    exit(0);
}
Design::NbTriangles = NbTr;
Design::NbSummits = NbTr + 2;
Design::SummitList = new int[NbTr*NB_SUMMITS_IN_TRIANGLE];
if (Design::SummitList == NULL) {
    printf("Error in Design::constructor: can't allocate memory.\n");
    exit(0);
}
for (TrNo=0; TrNo<NbTr; TrNo++) {
    for (SuNo=0; SuNo<NB_SUMMITS_IN_TRIANGLE; SuNo++) {
        LocationInString = (TrNo*NB_SUMMITS_IN_TRIANGLE + SuNo) *
            DES_LG_PER_SUMMIT;
        SummitNoInList = TrNo*NB_SUMMITS_IN_TRIANGLE + SuNo;
        SummitNoInSurf = atoi(&StringSummitList[LocationInString]);
        if (SummitNoInSurf <= 0 || SummitNoInSurf > Design::NbSummits) {
            printf("Error in Design::constructor: summit number ");
            printf("%d is not authorized for this surface of %d summits.\n",
                SummitNoInSurf, Design::NbSummits);
            exit(0);
        }
        Design::SummitList[SummitNoInList] = SummitNoInSurf;
    }
}
}

/*****
/*
/*          Design::~Design(void)
/*
/* Destructor that free the memory allocated to the design.
/*
/*
/*****
Design::~Design(void)
{
    /* to free the memory allocated by the constructor */
    delete Design::SummitList;
}

/*****
/*
/*          int Design::SummitNoOfSurf(int TriangleNo,
/*          int SummitOfTr)
/*
/*
/* Returns the summit number (of the surface) that corresponds to the
/* summit 'SummitOfTr' of the triangle 'TriangleNo'.
/*
/*
/*****
int Design::SummitNoOfSurf(int TriangleNo, int SummitOfTr)
{
    if (TriangleNo <= 0 || TriangleNo > Design::NbTriangles) {
        printf("Error in Design::SummitNoOfSurf: ");
        printf("triangle number %d is not authorized. ", TriangleNo);
        printf("Only %d triangles for this surface.\n",
            Design::NbTriangles);
        exit(0);
    }
    if (SummitOfTr <= 0 || SummitOfTr > NB_SUMMITS_IN_TRIANGLE) {
        printf("Error in Design::SummitNoOfSurf: ");
        printf("summit number %d is not authorized for a triangle.\n",
            SummitOfTr);
        exit(0);
    }
    return(Design::SummitList[(TriangleNo - 1)*3 + (SummitOfTr - 1)]);
}

```

SurfaceP

SurfaceP.h

```

/*****
/*   SurfaceP.h                                     */
/*                                             */
/*   This program is a tool to manage the surface designs of a pattern.   */
/*   It stores the different designs of a surface using the tool Design.   */
/*                                             */
/*****
#include "Design.h"

/* The class to store one surface */
class SurfaceP {
private:
    Design **DesignList; /* Pointer to array of pointers to Design */
                        /* We don't know how many pointers yet */
public:
    SurfaceP(int NbDes, int NbTr, char **StringDesignList);
    ~SurfaceP(void);
    int NbTriangles;
    int NbSummits;
    int NbDesigns;
    int SummitNoOfSurf(int DesignNo, int TriangleNo, int SummitOfTr);
};

```

SurfaceP.cpp

```

/*****
/*   SurfaceP.cpp                                     */
/*                                             */
/*   This program is a tool to handle teh designs of one surface of a     */
/*   pattern. It stores each designs of a surface using the tool Design.   */
/*                                             */
/*****
#include <stdio.h>
#include <stdlib.h>
#include "FileDefinition.h"
#include "SurfaceP.h"

/*****
/*                                             */
/*           SurfaceP::SurfaceP(int NbDes,      */
/*                               int NbTr,      */
/*                               char **StringDesignList)      */
/*                                             */
/* Constructor that allocate the memory for the list of 'NbDes' designs of */
/* one surface that contains 'NbTr' triangles and that belongs to a pattern */
/* The design list is read from the 'StringDesignList' string. This string  */
/* should come from the design look-up table. Note that the string contains */
/* several line of the design look-up table: one per possible design of the */
/* surface. */
/*                                             */
/*****
SurfaceP::SurfaceP(int NbDes, int NbTr, char **StringDesignList)
{
    int DesNo;

    if (NbDes <= 0 ) {
        printf("Error in SurfaceP::constructor: ");
        printf("number of designs %d cannot be negative.\n", NbDes);
        exit(0);
    }
    SurfaceP::NbTriangles = NbTr;

```


Pattern

Pattern.h

```

/*****
/*   Pattern.h
/*
/*   This program is a tool to handle a pattern of the Marching Cube
/*   algorithm.  It describes each surface of the pattern and access to
/*   the different designs of the surfaces.
/*
/*
/*****
#include "SurfaceP.h"

class Pattern {
private:
    SurfaceP **SurfaceList; /* Pointer to array of pointers to SurfaceP */
                          /* We don't know how many pointers yet */
public:
    Pattern(char *StringPattern, char **StringDesignList);
    ~Pattern(void);
    int NbSurfaces;
    int NbTriangles(int SurfaceNo);
    int NbSummits(int SurfaceNo);
    int NbDesigns(int SurfaceNo);
    int SummitNoOfSurf(int SurfaceNo, int DesignNo,
                       int TriangleNo, int SummitOfTr);
};

```

Pattern.cpp

```

/*****
/*   Pattern.h
/*
/*   This program is a tool to handle a pattern of the Marching Cube
/*   algorithm.  It describes each surface of the pattern and access to
/*   the different designs of the surfaces.
/*
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include "FileDefinition.h"
#include "Pattern.h"

/*****
/*
/*           Pattern::Pattern(char *StringPattern,
/*                               char **StringDesignList)
/*
/*
/* Constructor that allocate the memory for a pattern and its designs. The
/* entire data is read from the string 'StringPattern' and the list of
/* strings 'StringDesignList'. These strings come from the pattern look-up
/* table and the design look-up table respectively.
/*
/*
/*****
Pattern::Pattern(char *StringPattern, char **StringDesignList)
{
    int SurfNo;
    int NbTr;
    int NbDes;
    int NbTotalDes;

    Pattern::NbSurfaces = atoi(&StringPattern[PAT_POS_NB_SURFACES -
                                             PAT_POS_NB_SURFACES]);

    if ( Pattern::NbSurfaces < 0 ) {

```

```

printf("Error in Pattern::constructor: ");
printf("number of surfaces %d cannot be negative.\n",
      Pattern::NbSurfaces);
exit(0);
}
if (Pattern::NbSurfaces != 0) {
Pattern::SurfaceList = new SurfaceP *[Pattern::NbSurfaces];
if (Pattern::SurfaceList == NULL) {
printf("Error in Pattern::constructor: can't allocate memory.\n");
exit(0);
}
NbTotalDes = 0;
for (SurfNo=0; SurfNo<Pattern::NbSurfaces; SurfNo++) {
NbTr = atoi(&StringPattern[PAT_POS_TRIANGLE_LIST-PAT_POS_NB_SURFACES+
                          SurfNo*PAT_LG_PER_TRIANGLE]);
NbDes = atoi(&StringPattern[PAT_POS_DESIGN_LIST-PAT_POS_NB_SURFACES+
                          SurfNo*PAT_LG_PER_DESIGN]);
Pattern::SurfaceList[SurfNo] = new SurfaceP(NbDes, NbTr,
      &StringDesignList[NbTotalDes]);
if (Pattern::SurfaceList[SurfNo] == NULL) {
printf("Error in Pattern::constructor: can't allocate memory.\n");
exit(0);
}
NbTotalDes = NbTotalDes + NbDes;
}
}
}

/*****
/*
/*          Pattern::~Pattern(void)
/*
/* Destructor that free the memory allocated to the pattern.
/*
/*
/*****
Pattern::~Pattern(void)
{
int SurfNo;

/* to free the memory allocated by the constructor */
if (Pattern::NbSurfaces != 0) {
for (SurfNo=0; SurfNo<Pattern::NbSurfaces; SurfNo++) {
delete Pattern::SurfaceList[SurfNo];
}
delete Pattern::SurfaceList;
}
}

/*****
/*
/*          int Pattern::NbTriangles(int SurfaceNo)
/*
/* Returns the number of triangles to form the design of the surface
/* 'SurfaceNo' of a pattern.
/*
/*
/*****
int Pattern::NbTriangles(int SurfaceNo)
{
if (SurfaceNo <= 0 || SurfaceNo > Pattern::NbSurfaces) {
printf("Error in Pattern::NbTriangles: ");
printf("surface number %d is not authorized. ", SurfaceNo);
printf("Only %d surfaces for this pattern.\n", Pattern::NbSurfaces);
exit(0);
}
return(Pattern::SurfaceList[SurfaceNo - 1]->NbTriangles);
}

/*****
/*

```


SurfaceC

SurfaceC.h

```

/*****
/*   SurfaceC.h
/*
/*   This program is a tool to manage a surface of a configuration of the
/*   Marching Cube algorithm. it stores the edges of the cubes used for
/*   the summits of the surface. The order of the summits gives the
/*   positive orientation of the surface.
/*
/*****

/* The class to store one surface of a configuration */
class SurfaceC {
private:
    int *SummitList; /* Pointer to list of summits. */
                    /* we don't know how many summits yet */
public:
    SurfaceC(int NbSum, char *StringSummitList);
    ~SurfaceC(void);
    int NbSummits;
    int EdgeNoOfSummit(int SummitNo);
};

```

SurfaceC.cpp

```

/*****
/*   SurfaceC.cpp
/*
/*   This program is a tool to manage a surface of a configuration of the
/*   Marching Cube algorithm. it stores the edges of the cubes used for
/*   the summits of the surface. The order of the summits gives the
/*   positive orientation of the surface.
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include "FileDefinition.h"
#include "SurfaceC.h"

#define NB_EDGES_IN_CUBE      12
#define NB_SUMMITS_IN_TRIANGLE  3

/*****
/*
/*           SurfaceC::SurfaceC(int NbSum,
/*                               char *StringSummitList)
/*
/*
/* Constructor that allocate the memory for the list of 'NbSum' summits of
/* one surface of a configuration. The list of summits is read from the
/* .StringSummitList' string. This string should come from the
/* configuration look-up table. Note that the string is a part of a line of
/* the configuration look-up table: only one surface.
/*
/*****
SurfaceC::SurfaceC(int NbSum, char *StringSummitList)
{
    int SummitNo;
    int LocationInString;
    int EdgeNo;

    if (NbSum < NB_SUMMITS_IN_TRIANGLE || NbSum > NB_EDGES_IN_CUBE) {
        printf("Error in SurfaceC::constructor: number of summits ");
    }
}

```

```

    printf("%d cannot be less than %d or greater than %d.\n",
           NbSum, NB_SUMMITS_IN_TRIANGLE, NB_EDGES_IN_CUBE);
    exit(0);
}
SurfaceC::NbSummits = NbSum;
SurfaceC::SummitList = new int[NbSum];
if (SurfaceC::SummitList == NULL) {
    printf("Error in SurfaceC::constructor: can't allocate memory.\n");
    exit(0);
}
for (SummitNo=0; SummitNo<NbSum; SummitNo++) {
    LocationInString = SummitNo*CON_LG_PER_SUMMIT;
    EdgeNo = atoi(&StringSummitList[LocationInString]);
    if (EdgeNo < 0 || EdgeNo >= NB_EDGES_IN_CUBE) {
        printf("Error in SurfaceC::constructor: edge number ");
        printf("%d is not authorized for a cube.\n", EdgeNo);
        exit(0);
    }
    SurfaceC::SummitList[SummitNo] = EdgeNo;
}
}

/*****
/*
/*                               SurfaceC::~SurfaceC(void)                               */
/*
/*                               Destructor that free the memory allocated to the surface.   */
/*
/*****
SurfaceC::~SurfaceC(void)
{

    /* to free the memory allocated by the constructor */
    delete SurfaceC::SummitList;
}

/*****
/*
/*                               int SurfaceC::EdgeNoOfSummit(int SummitNo)                               */
/*
/*                               Returns the edge number (of the Marching cube) that corresponds to the */
/*                               summit 'SummitNo' of the surface of a configuration.           */
/*
/*****
int SurfaceC::EdgeNoOfSummit(int SummitNo)
{
    if (SummitNo <= 0 || SummitNo > SurfaceC::NbSummits) {
        printf("Error in SurfaceC::EdgeNoOfSummit: ");
        printf("summit number %d is not authorized. ", SummitNo);
        printf("Only %d summits for this surface.\n", SurfaceC::NbSummits);
        exit(0);
    }
    return(SurfaceC::SummitList[SummitNo - 1]);
}

```

Configuration

Configuration.h

```

/*****
/* Configuration.h */
/*
/* This program is a tool to handle the surfaces of a configuration. It */
/* stores the different surfaces of a configuration using the tool */
/* SurfaceC. */
/*
/*****
#include "SurfaceC.h"

/* The class to store one configuration */
class Configuration {
private:
    SurfaceC **SurfaceList; /* Pointer to array of pointers to SurfaceC */
                          /* We don't know how many pointers yet */
public:
    Configuration(int PatNo, int NbSurf, int *NbSumOfSurf,
                  char **StringSurfaceList);
    ~Configuration(void);
    int PatternNo;
    int NbSurfaces;
    int *NbSummitsOfSurface; /* Pointer to array of integers */
                          /* We don't know how many pointers yet */
    int EdgeNoOfSummit(int SurfNo, int SummitNo);
};

```

Configuration.cpp

```

/*****
/* Configuration.cpp */
/*
/* This program is a tool to handle the surfaces of a configuration. It */
/* stores the different surfaces of a configuration using the tool */
/* SurfaceC. */
/*
/*****
#include "Configuration.h"
#include <stdio.h>
#include <stdlib.h>
#include "FileDefinition.h"

/*****
/*
/* Configuration::Configuration(int PatNo,
/*                               int NbSurf,
/*                               int *NbSumOfSurf,
/*                               char **StringSurfaceList)
/*
/*
/* Constructor that allocate the memory for the list of 'NbSurf' surfaces of
/* one configuration that belongs to the pattern 'PatNo'. Each surface
/* contains 'NbSumOfSurf' summits. The summits are read from the list of
/* strings 'StringSurfaceList'. These strings should come from the
/* configuration look-up table. Note that each string contains one part of
/* the same line of the configuration look-up table: one per surface
/*
/*****
Configuration::Configuration(int PatNo, int NbSurf, int *NbSumOfSurf,
                             char **StringSurfaceList)
{
    int SurfNo;

```

```

Configuration::PatternNo = PatNo;
if (NbSurf < 0 ) {
    printf("Error in Configuration::constructor: ");
    printf("number of Surfaces %d cannot be negative.\n", NbSurf);
    exit(0);
}
Configuration::NbSurfaces = NbSurf;
if (Configuration::NbSurfaces != 0) {
    Configuration::NbSummitsOfSurface = new int[NbSurf];
    if (Configuration::NbSummitsOfSurface == NULL) {
        printf("Error in Configuration::constructor: can't allocate memory.\n");
        exit(0);
    }
    Configuration::SurfaceList = new SurfaceC *[NbSurf];
    if (Configuration::SurfaceList == NULL) {
        printf("Error in Configuration::constructor: can't allocate memory.\n");
        exit(0);
    }
    for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
        Configuration::NbSummitsOfSurface[SurfNo] = NbSumOfSurf[SurfNo];
        Configuration::SurfaceList[SurfNo] = new SurfaceC(NbSumOfSurf[SurfNo],
            StringSurfaceList[SurfNo]);
        if (Configuration::SurfaceList[SurfNo] == NULL) {
            printf("Error in Configuration::constructor: ");
            printf("can't allocate memory.\n");
            exit(0);
        }
    }
}
}

/*****
/*
/*          Configuration::~Configuration(void)          */
/*
/* Destructeur qui libere la memoire allouee a la configuration. */
/*
/*****
Configuration::~Configuration(void)
{
    int SurfNo;

    /* to free the memory allocated by the constructor */
    if (Configuration::NbSurfaces != 0) {
        for (SurfNo=0; SurfNo<Configuration::NbSurfaces; SurfNo++) {
            delete Configuration::SurfaceList[SurfNo];
        }
        delete Configuration::SurfaceList;
        delete Configuration::NbSummitsOfSurface;
    }
}

/*****
/*
/*          int Configuration::EdgeNoOfSummit(int SurfNo,          */
/*          int SummitNo)          */
/*
/* Returns the edge number (of the Marching cube) that corresponds to the */
/* summit 'SummitNo' of the surface 'SurfNo' of a configuration.          */
/*
/*****
int Configuration::EdgeNoOfSummit(int SurfNo, int SummitNo)
{
    if (SurfNo <= 0 || SurfNo > Configuration::NbSurfaces) {
        printf("Error in Configuration::EdgeNoOfSummit: ");
        printf("surface number %d is not authorized. ", SurfNo);
        printf("Only %d surfaces for this surface.\n", Configuration::NbSurfaces);
        exit(0);
    }
}

```

```
return(Configuration::SurfaceList[SurfNo - 1]->EdgeNoOfSummit(SummitNo));  
}
```

MCConfigurations

MCConfigurations.h

```

/*****
/*  MCConfigurations.h
/*
/*  This program is a tool to handle the 256 configurations of the
/*  Marching cube algorithm. The 256 configurations are regrouped into
/*  23 different patterns: the 15 proposed initially by Lorensen and 8
/*  new ones, symmetric of the patterns 0 to 7.
/*
/*
/*****
#include "Pattern.h"
#include "Configuration.h"

#define NB_PATTERNS          23
#define NB_CONFIGURATIONS   256

/* To define a class to store both the Patterns and the Configurations */
class MCConfigurations {
private:
    Pattern *PatternList[NB_PATTERNS];
    Configuration *ConfigList[NB_CONFIGURATIONS];
public:
    MCConfigurations(char *DirectoryName);
    ~MCConfigurations(void);
    int PatternNo(int Config);
    int NbSurfaces(int Config);
    int NbTriangles(int Config, int SurfaceNo);
    int NbSummits(int Config, int SurfaceNo);
    int NbDesigns(int Config, int SurfaceNo);
    int CubeEdgeNo(int Config, int SurfaceNo, int DesignNo,
                   int TriangleNo, int SummitNo);
};

```

MCConfigurations.cpp

```

/*****
/*  MCConfigurations.cpp
/*
/*  This program is a tool to handle the 256 configurations of the
/*  Marching cube algorithm. The 256 configurations are regrouped into
/*  23 different patterns: the 15 proposed initially by Lorensen and 8
/*  new ones, symmetric of the patterns 0 to 7.
/*
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "FileDefinition.h"
#include "MCConfigurations.h"

#define NB_SUMMITS_IN_TRIANGLE  3

/*****
/*
/*      MCConfigurations::MCConfigurations(char *DirectoryName)
/*
/*
/*  Constructor that allocate the memory for the entire list of patterns and
/*  configurations. The data are read from the three look-up tables that
/*  are in the directory 'DirectoryName'. The constructor read the tables
/*  and call the different tools to store the patterns, the configurations,
/*  the surfaces of the configurations and of the patterns and the designs.
/*
/*

```

```

/*****/
MCCConfigurations::MCCConfigurations(char *DirectoryName)
{
    FILE *PatternFile;
    FILE *DesignFile;
    FILE *ConfigFile;
    char PatternFileName[100];
    char DesignFileName[100];
    char ConfigFileName[100];
    int PatternLineNo;
    int DesignLineNo;
    int ConfigLineNo;
    char PatternLine[100];
    char DesignLine[100];
    char ConfigLine[100];
    int NbSurf, SurfNo;
    int NbDes, DesNo;
    int NbTr;
    char **StringDesignList;
    char **StringSurfaceList;
    int *NbSumOfSurf;
    int TotalNbDes, TotalDesNo;
    int TotalSumNo;
    int ConfNo;
    int PatNo;

    /* 1) To read the Pattern and Design files at the same time */
    /*****/
    /* a) To open the pattern file and skip the header */
    sprintf(PatternFileName, "%s/%s", DirectoryName, PAT_FILE_NAME);
    PatternFile = fopen(PatternFileName,"r");
    if (PatternFile == NULL) {
        printf("Error in MCCConfigurations::constructor: ");
        printf("can't open pattern file: '%s'\n", PatternFileName);
        exit(0);
    }
    for (PatternLineNo=0; PatternLineNo<PAT_LINE_HEADER; PatternLineNo++) {
        fgets(PatternLine,100,PatternFile);
    }
    /* b) To open the design file and skip the header */
    sprintf(DesignFileName, "%s/%s", DirectoryName, DES_FILE_NAME);
    DesignFile = fopen(DesignFileName,"r");
    if (DesignFile == NULL) {
        printf("Error in MCCConfigurations::constructor: ");
        printf("can't open design file: '%s'\n", DesignFileName);
        exit(0);
    }
    for (DesignLineNo=0; DesignLineNo<DES_LINE_HEADER; DesignLineNo++) {
        fgets(DesignLine,100,DesignFile);
    }
    /* c) Loop over the pattern file */
    for (PatternLineNo=0; PatternLineNo<NB_PATTERNS; PatternLineNo++) {
        fgets(PatternLine,100,PatternFile);
        PatNo = atoi(&PatternLine[PAT_POS_PATTERN]);
        if (PatNo<0 || PatNo>=NB_PATTERNS) {
            printf("Error in MCCConfigurations::constructor: ");
            printf("pattern number %d is not valid.\n", PatNo);
            exit(0);
        }
        /* To count how many design for this pattern */
        TotalNbDes = 0;
        NbSurf = atoi(&PatternLine[PAT_POS_NB_SURFACES]);
        for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
            NbDes = atoi(&PatternLine[PAT_POS_DESIGN_LIST +
                SurfNo*PAT_LG_PER_DESIGN]);
            TotalNbDes = TotalNbDes + NbDes;
        }
        /* To read the designs of this pattern from the design file */
        StringDesignList = new char *[TotalNbDes];

```

```

TotalDesNo = 0;
for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
    NbDes = atoi(&PatternLine[PAT_POS_DESIGN_LIST +
                             SurfNo*PAT_LG_PER_DESIGN]);
    NbTr = atoi(&PatternLine[PAT_POS_TRIANGLE_LIST +
                             SurfNo*PAT_LG_PER_TRIANGLE]);
    for (DesNo=0; DesNo<NbDes; DesNo++) {
        StringDesignList[TotalDesNo] = new char[NbTr *
                                                  NB_SUMMITS_IN_TRIANGLE * DES_LG_PER_SUMMIT + 1];
        fgets(DesignLine,100,DesignFile);
        strncpy(StringDesignList[TotalDesNo],
                &DesignLine[DES_POS_SUMMIT_LIST],
                NbTr*NB_SUMMITS_IN_TRIANGLE*DES_LG_PER_SUMMIT);
        StringDesignList[TotalDesNo][NbTr *
                                     NB_SUMMITS_IN_TRIANGLE * DES_LG_PER_SUMMIT] = 0;
        TotalDesNo = TotalDesNo + 1;
    }
}
/* To construct the pattern */
MCCConfigurations::PatternList[PatNo] =
    new Pattern(&PatternLine[PAT_POS_NB_SURFACES], StringDesignList);
/* to free the memory */
TotalDesNo = 0;
for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
    NbDes = atoi(&PatternLine[PAT_POS_DESIGN_LIST +
                             SurfNo*PAT_LG_PER_DESIGN]);
    for (DesNo=0; DesNo<NbDes; DesNo++) {
        delete StringDesignList[TotalDesNo];
        TotalDesNo = TotalDesNo + 1;
    }
}
delete StringDesignList;
}
/* d) To close the files */
fclose(PatternFile);
fclose(DesignFile);

/* 2) To read the Configuration file */
/*****
/* a) To open the design file and skip the header */
sprintf(ConfigFileName, "%s/%s", DirectoryName, CON_FILE_NAME);
ConfigFile = fopen(ConfigFileName,"r");
if (ConfigFile == NULL) {
    printf("Error in MCCConfigurations::constructor: ");
    printf("can't open configuration file: '%s'\n", ConfigFileName);
    exit(0);
}
for (ConfigLineNo=0; ConfigLineNo<CON_LINE_HEADER; ConfigLineNo++) {
    fgets(ConfigLine,100,ConfigFile);
}
/* b) Loop over the configuration file */
for (ConfigLineNo=0; ConfigLineNo<NB_CONFIGURATIONS; ConfigLineNo++) {
    fgets(ConfigLine,100,ConfigFile);
    ConfNo = atoi(&ConfigLine[CON_POS_CONFIG]);
    if (ConfNo<0 || ConfNo>=NB_CONFIGURATIONS) {
        printf("Error in MCCConfigurations::constructor: ");
        printf("configuration number %d is not valid.\n", ConfNo);
        exit(0);
    }
    PatNo = atoi(&ConfigLine[CON_POS_PATTERN]);
    if (PatNo<0 || PatNo>=NB_PATTERNS) {
        printf("Error in MCCConfigurations::constructor: ");
        printf("pattern number %d is not valid.\n", PatNo);
        exit(0);
    }
    NbSurf = PatternList[PatNo]->NbSurfaces;
    /* To read the surfaces of this pattern */
    StringSurfaceList = new char *[NbSurf];
    NbSumOfSurf = new int[NbSurf];

```

```

TotalSumNo = 0;
for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
    NbSumOfSurf[SurfNo] = PatternList[PatNo]->NbSummits(SurfNo+1);
    StringSurfaceList[SurfNo] = new char[NbSumOfSurf[SurfNo] *
        CON_LG_PER_SUMMIT + 1];

    strncpy(StringSurfaceList[SurfNo],
        &ConfigLine[CON_POS_SUMMIT_LIST + TotalSumNo*CON_LG_PER_SUMMIT],
        NbSumOfSurf[SurfNo] * CON_LG_PER_SUMMIT);
    StringSurfaceList[SurfNo][NbSumOfSurf[SurfNo]*CON_LG_PER_SUMMIT] = 0;
    TotalSumNo = TotalSumNo + NbSumOfSurf[SurfNo];
}
/* To construct the configuration */
MCCConfigurations::ConfigList[ConfNo] = new Configuration(PatNo, NbSurf,
    NbSumOfSurf, StringSurfaceList);

/* to free the memory */
for (SurfNo=0; SurfNo<NbSurf; SurfNo++) {
    delete StringSurfaceList[SurfNo];
}
delete StringSurfaceList;
delete NbSumOfSurf;
}
/* c) To close the file */
fclose(ConfigFile);
}

/*****
/*
/*          MCCConfigurations::~MCCConfigurations(void)          */
/*
/*  Destructor that free the memory allocated for the patterns and the  */
/*
/*
/*****
MCCConfigurations::~MCCConfigurations(void)
{
    int PatNo;
    int ConfNo;

    for (PatNo=0; PatNo<NB_PATTERNS; PatNo++) {
        delete MCCConfigurations::PatternList[PatNo];
    }
    for (ConfNo=0; ConfNo<NB_CONFIGURATIONS; ConfNo++) {
        delete MCCConfigurations::ConfigList[ConfNo];
    }
}

/*****
/*
/*          int MCCConfigurations::PatternNo(int Config)          */
/*
/*
/* Returns the pattern number of the configuration 'Config'.          */
/*
/*
/*****
int MCCConfigurations::PatternNo(int Config)
{
    if (Config < 0 || Config >= NB_CONFIGURATIONS) {
        printf("Error in MCCConfigurations::PatternNo: ");
        printf("configuration number %d is not valid.\n", Config);
        exit(0);
    }
    return(MCCConfigurations::ConfigList[Config]->PatternNo);
}

/*****
/*
/*          int MCCConfigurations::NbSurfaces(int Config)          */
/*
/*
/* Returns the number of surfaces of the configuration 'Config'.          */
/*
/*
/*****

```

```

int MConfigurations::NbSurfaces(int Config)
{
    int PatNo;

    PatNo = MConfigurations::PatternNo(Config);
    return(MConfigurations::PatternList[PatNo]->NbSurfaces);
}

/*****
/*
/*          int MConfigurations::NbTriangles(int Config,
/*          int SurfaceNo)
/*
/* Returns the number of triangles of the surface 'SurfaceNo of the
/* configuration 'Config'.
/*
/*
/*****
int MConfigurations::NbTriangles(int Config, int SurfaceNo)
{
    int PatNo;

    PatNo = MConfigurations::PatternNo(Config);
    return(MConfigurations::PatternList[PatNo]->NbTriangles(SurfaceNo));
}

/*****
/*
/*          int MConfigurations::NbSummits(int Config,
/*          int SurfaceNo)
/*
/* Returns the number of summits of the surface 'SurfaceNo of the
/* configuration 'Config'.
/*
/*
/*****
int MConfigurations::NbSummits(int Config, int SurfaceNo)
{
    int PatNo;

    PatNo = MConfigurations::PatternNo(Config);
    return(MConfigurations::PatternList[PatNo]->NbSummits(SurfaceNo));
}

/*****
/*
/*          int MConfigurations::NbDesigns(int Config,
/*          int SurfaceNo)
/*
/* Returns the number of designs of the surface 'SurfaceNo of the
/* configuration 'Config'.
/*
/*
/*****
int MConfigurations::NbDesigns(int Config, int SurfaceNo)
{
    int PatNo;

    PatNo = MConfigurations::PatternNo(Config);
    return(MConfigurations::PatternList[PatNo]->NbDesigns(SurfaceNo));
}

/*****
/*
/*          int MConfigurations::CubeEdgeNo(int Config,
/*          int SurfaceNo,
/*          int DesignNo,
/*          int TriangleNo,
/*          int SummitNo)
/*
/* Returns the cube edge number of the summit 'SummitNo' of the triangle
/* 'TriangleNo' of the surface 'SurfaceNo' of the configuration 'Config'
*/

```


APPENDIX D MARCHING-CUBE IMAGE TOOLS (C++ PROGRAMS)

This appendix contains all the C++ classes and functions used to create, read and handle the Marching-Cube image when stored in tables organized as explained on [Figure 7-12](#). They are:

- **TableFormat**: List of constant data that define the position of each column in the four tables. Only a '.h' file for this tool.
- **Cube0and255Table**: C++ class that stores the marching cubes that do not belong to the isosurface.
- **CubeOnIsoTable**: C++ class that stores the marching cubes that belong to the isosurface.
- **TriangleTable**: C++ class that stores the list of the triangles that constitutes the isosurface.
- **SummitTable**: C++ class that stores the list of the vertices of the triangle.

Each tool contains a '.cpp' file that contains the code of the tool and a '.h' file that must be included (#include C pre-compiler directive) in any program that uses the tool.

TableFormat

TableFormat.h

```

/*****
/*   TableFormat.h
/*
/*   This include file contains the constant values used for the storage
/*   of the Marching Cubes tables on disk.
/*
/*****

/* The possible formats to store the tables on the disk. The binary format */
/* is more compact but cannot be read directly with a text editor */
#define ASCII_FORMAT          'A'
#define BINARY_FORMAT        'B'

/* ASCII format of the summit table */
#define SUM_LINE_FORMAT      "%9.6f %9.6f %9.6f\n"
#define SUM_LG_LINE         30
#define SUM_POS_X            0
#define SUM_LG_X             9
#define SUM_POS_Y            10
#define SUM_LG_Y             9
#define SUM_POS_Z            20
#define SUM_LG_Z             9

/* ASCII format of the triangle table */
#define TRI_LINE_FORMAT      "%9d %9d %9d\n"
#define TRI_LG_LINE         30
#define TRI_POS_SUMMIT_1    0
#define TRI_LG_SUMMIT_1     9

```

```
#define TRI_POS_SUMMIT_2          10
#define TRI_LG_SUMMIT_2           9
#define TRI_POS_SUMMIT_3          20
#define TRI_LG_SUMMIT_3           9

/* ASCII format of the cube-0-and-255 table */
#define CU1_LINE_FORMAT           "%9d %3d\n"
#define CU1_LG_LINE               14
#define CU1_POS_CUBE_NO           0
#define CU1_LG_CUBE_NO            9
#define CU1_POS_CONFIG            10
#define CU1_LG_CONFIG              3

/* ASCII format of the cube-on-iso-surface table */
#define CU2_LINE_FORMAT           "%9d %3d %1d %9d\n"
#define CU2_LG_LINE               26
#define CU2_POS_CUBE_NO           0
#define CU2_LG_CUBE_NO            9
#define CU2_POS_CONFIG            10
#define CU2_LG_CONFIG              3
#define CU2_POS_NB_TRIANGLES      14
#define CU2_LG_NB_TRIANGLES        1
#define CU2_POS_TRIANGLE_LIST     16
#define CU2_LG_TRIANGLE_LIST       9
```

Cube0and255Table

Cube0and255Table.h

```

/*****
/*   Cube0and255Table.h                                     */
/*                                                                 */
/*   This program is a tool to handle the cubes that are pure bone or pure  */
/*   marrow with the Marching Cube algorithm.                            */
/*                                                                 */
/*****

/* The class to store the entire table */
class Cube0and255Table {
private:
    unsigned int TableSize;
    unsigned int *CubeNo;          /* Pointer to list of cube numbers. */
    unsigned char *Config;        /* Pointer to list of configurations. */
                                   /* we don't know how many cubes yet */
    unsigned int CurrentIndex;
    unsigned int WhichRankInTable(unsigned int Cube);
public:
    Cube0and255Table(unsigned int NbCubes);
    Cube0and255Table(char *FileName, char TableFormat);
    ~Cube0and255Table(void);
    void AddToTable(unsigned int Cube, unsigned int CubeConfig);
    unsigned int WhichConfig(unsigned int Cube);
    void SaveToDisk(char *FileName, char TableFormat);
    unsigned int NbCubesInTable(void);
    unsigned int MemoryPerCube(void);
    unsigned int DiskSpacePerCube(char TableFormat);
};

```

Cube0and255Table.cpp

```

/*****
/*   Cube0and255Table.cpp                                     */
/*                                                                 */
/*   This program is a tool to handle the cubes that are pure bone or pure  */
/*   marrow with the Marching Cube algorithm.                            */
/*                                                                 */
/*****

#include <stdio.h>
#include <stdlib.h>
#include "../CppTools/FileStuff.h"
#include "TableFormat.h"
#include "Cube0and255Table.h"

/*****
/*                                                                 */
/*   Cube0and255Table::Cube0and255Table(unsigned int NbCubes)          */
/*                                                                 */
/*   Constructor that allocate the memory for an empty table of 'NbCubes' */
/*   cubes.                                                            */
/*                                                                 */
/*****
Cube0and255Table::Cube0and255Table(unsigned int NbCubes)
{
    if (NbCubes == 0 ) {
        printf("Error in Cube0and255Table::constructor: ");
        printf("number of cubes cannot be 0.\n");
        exit(0);
    }
    Cube0and255Table::CubeNo = new unsigned int[NbCubes];
    if (Cube0and255Table::CubeNo == NULL) {

```

```

    printf("Error in Cube0and255Table::constructor: can't allocate memory.\n");
    exit(0);
}
Cube0and255Table::Config = new unsigned char[NbCubes];
if (Cube0and255Table::Config == NULL) {
    printf("Error in Cube0and255Table::constructor: can't allocate memory.\n");
    exit(0);
}
Cube0and255Table::TableSize = NbCubes;
Cube0and255Table::CurrentIndex = 0;
}

/*****
/*
/*          Cube0and255Table::Cube0and255Table(char *FileName,
/*          char TableFormat)
/*
/*
/* Constructor that allocate the memory and load a table from a file which
/* name is in the parameter 'FileName'. The number of cubes is determined
/* by the file size. 'TableFormat' is the format used when the table was
/* saved on disk (BINARY or ASCII).
/*
/*
/*****
Cube0and255Table::Cube0and255Table(char *FileName, char TableFormat)
{
    FILE *FileToRead;
    char StringToRead[CU1_LG_LINE+1];
    unsigned int SizeOfFile;
    unsigned int NbCubes;
    unsigned int RankInTable;

    /* to get the size of the table file */
    SizeOfFile = FileSize(FileName);
    if (SizeOfFile == 0) {
        printf("Error in Cube0and255Table::Cube0and255Table: ");
        printf("table file %s is empty or does not exist.\n", FileName);
        exit(0);
    }

    /* To get the number of elements in the table */
    if (TableFormat != ASCII_FORMAT && TableFormat != BINARY_FORMAT) {
        printf("Error in Cube0and255Table::Cube0and255Table: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
    NbCubes = SizeOfFile / Cube0and255Table::DiskSpacePerCube(TableFormat);

    /* To allocate the memory for the table */
    Cube0and255Table::CubeNo = new unsigned int[NbCubes];
    if (Cube0and255Table::CubeNo == NULL) {
        printf("Error in Cube0and255Table::constructor: can't allocate memory.\n");
        exit(0);
    }
    Cube0and255Table::Config = new unsigned char[NbCubes];
    if (Cube0and255Table::Config == NULL) {
        printf("Error in Cube0and255Table::constructor: can't allocate memory.\n");
        exit(0);
    }
    Cube0and255Table::TableSize = NbCubes;
    Cube0and255Table::CurrentIndex = Cube0and255Table::TableSize;

    /* to read the file and store it in the memory structure */
    FileToRead = fopen(FileName,"r");
    for (RankInTable=1;RankInTable<=Cube0and255Table::TableSize;RankInTable++) {
        switch (TableFormat) {
            case ASCII_FORMAT:
                fgets(StringToRead, CU1_LG_LINE+1, FileToRead);
                Cube0and255Table::CubeNo[RankInTable - 1] =
                    atoi(&StringToRead[CU1_POS_CUBE_NO]);

```

```

        Cube0and255Table::Config[RankInTable - 1] =
            atoi(&StringToRead[CU1_POS_CONFIG]);
        break;
    case BINARY_FORMAT:
        fread(&Cube0and255Table::CubeNo[RankInTable - 1],
            sizeof(*Cube0and255Table::CubeNo), 1, FileToRead);
        fread(&Cube0and255Table::Config[RankInTable - 1],
            sizeof(*Cube0and255Table::Config), 1, FileToRead);
        break;
    }
}
fclose(FileToRead);
}

/*****
/*
/*          Cube0and255Table::~Cube0and255Table(void)          */
/*
/* Destructor of a table that frees the memory allocated for the table. */
/*
/*
*****/
Cube0and255Table::~Cube0and255Table(void)
{
    /* to free the memory allocated by the constructor */
    delete Cube0and255Table::CubeNo;
    delete Cube0and255Table::Config;
}

/*****
/*
/*          void Cube0and255Table::AddToTable(unsigned int Cube,
/*          unsigned int CubeConfig)          */
/*
/* To add a new cube to the table. 'Cube' is the identification number
/* of the cube. It is used to retrieve the summit in the table. The
/* table must be created sorted for a fast access to the summits by using
/* a binary search on the table. 'CubeConfig' is the configuration of the
/* cube.
/*
/*
*****/
void Cube0and255Table::AddToTable(unsigned int Cube, unsigned int CubeConfig)
{
    if (Cube0and255Table::CurrentIndex >= Cube0and255Table::TableSize ) {
        printf("Error in Cube0and255Table::AddToTable: ");
        printf("cannot add another cube in table of size %d.\n",
            Cube0and255Table::TableSize);
        exit(0);
    }
    Cube0and255Table::CubeNo[Cube0and255Table::CurrentIndex] = Cube;
    Cube0and255Table::Config[Cube0and255Table::CurrentIndex] = CubeConfig;
    Cube0and255Table::CurrentIndex++;
}

/*****
/*
/*          unsigned int Cube0and255Table::WhichRankInTable(unsigned int Cube)
/*
/* Returns the rank (from 1 to NbCubes) of a cube in the table. Uses a
/* binary search on the 'Cube' number to retrieve the cube in the table.
/* If the cube is not found, then the function returns 0.
/*
/*
*****/
unsigned int Cube0and255Table::WhichRankInTable(unsigned int Cube)
{
    bool CubeFound;
    unsigned int inf, sup, center;

    /* binary search in a sorted array */
    inf = 1;

```

```

sup = Cube0and255Table::TableSize;
CubeFound = false;
while ( (CubeFound == false) && (inf <= sup) ) {
    center = (inf + sup) / 2;
    if (Cube0and255Table::CubeNo[center - 1] == Cube) {
        CubeFound = true;
    }
    else {
        if (Cube0and255Table::CubeNo[center - 1] > Cube) {
            sup = center - 1;
        }
        else {
            inf = center + 1;
        }
    }
}
if (CubeFound == true) {
    return(center);
}
else {
    if (Cube0and255Table::CubeNo[center - 1] < Cube) {
        return(center); /* same config as this one */
    }
    else {
        if (center == 1) {
            return(0); /* 0 means not found */
        }
        else {
            return(center-1);
        }
    }
}
}

/*****
/*
/*      unsigned int Cube0and255Table::WhichConfig(unsigned int Cube)      */
/*
/* Returns the configuration number of a cube given by its identification */
/* number: 'Cube'.
/*
/*
/*****
unsigned int Cube0and255Table::WhichConfig(unsigned int Cube)
{
    unsigned int RankInTable;

    RankInTable = Cube0and255Table::WhichRankInTable(Cube);
    if (RankInTable == 0) {
        printf("Error in Cube0and255Table::WhichConfig: ");
        printf("cube number %d below smallest value in table.\n", Cube);
        exit(0);
    }
    return(Cube0and255Table::Config[RankInTable - 1]);
}

/*****
/*
/*      void Cube0and255Table::SaveToDisk(char *FileName,
/*
/*      char TableFormat)
/*
/* Saves a table on the disk. 'TableFormat' specify the format to save
/* the table (BINARY or ASCII). If the file already exists, it is replaced
/* by the new one.
/*
/*
/*****
void Cube0and255Table::SaveToDisk(char *FileName, char TableFormat)
{
    FILE *FileToWrite;
    unsigned int RankInTable;

```

```

char StringToWrite[CU1_LG_LINE+1];

/* to open the file */
FileToWrite = fopen(FileName, "w");
if (FileToWrite == NULL) {
    printf("Error in Cube0and255Table::SaveToDisk: ");
    printf("can't open file: '%s'\n", FileName);
    exit(0);
}
/* to write the file */
for (RankInTable=1; RankInTable<=Cube0and255Table::TableSize; RankInTable++) {
    switch (TableFormat) {
        case ASCII_FORMAT:
            sprintf(StringToWrite, CU1_LINE_FORMAT,
                    Cube0and255Table::CubeNo[RankInTable - 1],
                    Cube0and255Table::Config[RankInTable - 1]);
            fputs(StringToWrite, FileToWrite);
            break;
        case BINARY_FORMAT:
            fwrite(&Cube0and255Table::CubeNo[RankInTable - 1],
                  sizeof(*Cube0and255Table::CubeNo), 1, FileToWrite);
            fwrite(&Cube0and255Table::Config[RankInTable - 1],
                  sizeof(*Cube0and255Table::Config), 1, FileToWrite);
            break;
        default:
            printf("Error in Cube0and255Table::SaveToDisk: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
    }
}
fclose(FileToWrite);
}

/*****
/*
/*          unsigned int Cube0and255Table::NbCubesInTable(void)          */
/*
/* Returns the current number of cubes in the table.                    */
/*
/*
/*****
unsigned int Cube0and255Table::NbCubesInTable(void)
{
    return(Cube0and255Table::CurrentIndex);
}

/*****
/*
/*          unsigned int Cube0and255Table::MemoryPerCube(void)          */
/*
/* Returns the memory requirement (in bytes) to store one cube.        */
/* It is useful for large images to estimate the memory that will be used */
/* by the program that handle the table.                                */
/*
/*
/*****
unsigned int Cube0and255Table::MemoryPerCube(void)
{
    return( sizeof(*Cube0and255Table::CubeNo) +
            sizeof(*Cube0and255Table::Config) );
}

/*****
/*
/*          unsigned int Cube0and255Table::DiskSpacePerCube(char TableFormat) */
/*
/* Returns the space requirement (in bytes) to store one cube on the disk. */
/* It is useful for large images to estimate the space that will be used */
/* to store the image on the disk.                                         */
/* TableFormat must specify the format that is to be used to store the table */

```

```
/* (BINARY or ASCII). */
/*
/*****
unsigned int Cube0and255Table::DiskSpacePerCube(char TableFormat)
{
    unsigned int DiskSpace;

    switch (TableFormat) {
        case ASCII_FORMAT:
            DiskSpace = CU1_LG_LINE;
            break;
        case BINARY_FORMAT:
            DiskSpace = sizeof(*Cube0and255Table::CubeNo) +
                sizeof(*Cube0and255Table::Config);
            break;
        default:
            printf("Error in Cube0and255Table::DiskSpacePerCube: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
    }
    return(DiskSpace);
}
```

CubeOnIsoTable

CubeOnIsoTable.h

```

/*****
/*   CubeOnIsoTable.h                                     */
/*                                                     */
/*   This program is a tool to handle the cubes that are on the isosurface */
/*   with the Marching Cube algorithm.                                     */
/*                                                     */
/*****

/* The class to store the entire table */
class CubeOnIsoTable {
private:
    unsigned int TableSize;
    unsigned int *CubeNo;          /* Pointer to list of cube numbers. */
    unsigned char *CubeConfig;    /* Pointer to list of cube config. */
    unsigned char *NbTriangles;  /* Pointer to list of numbers of tr. */
    unsigned int *TriangleList;  /* Pointer to list of triangles. */
                                /* we don't know how many cubes yet */

    unsigned int CurrentIndex;
    unsigned int WhichRankInTable(unsigned int Cube);
public:
    CubeOnIsoTable(unsigned int NbCubes);
    CubeOnIsoTable(char *FileName, char TableFormat);
    ~CubeOnIsoTable(void);
    void AddToTable(unsigned int Cube, unsigned char Config,
                   unsigned char NbTr, unsigned int TrList);
    void GetCube(unsigned int Cube, unsigned char *Config,
                unsigned char *NbTr, unsigned int *FirstTr);
    void SaveToDisk(char *FileName, char TableFormat);
    unsigned int NbCubesInTable(void);
    unsigned int MemoryPerCube(void);
    unsigned int DiskSpacePerCube(char TableFormat);
};

```

CubeOnIsoTable.cpp

```

/*****
/*   CubeOnIsoTable.h                                     */
/*                                                     */
/*   This program is a tool to handle the cubes that are on the isosurface */
/*   with the Marching Cube algorithm.                                     */
/*                                                     */
/*****

#include <stdio.h>
#include <stdlib.h>
#include "../CppTools/FileStuff.h"
#include "TableFormat.h"
#include "CubeOnIsoTable.h"

/*****
/*
/*           CubeOnIsoTable::CubeOnIsoTable(unsigned int NbCubes)
/*
/*   Constructor that allocate the memory for an empty table of 'NbCubes'
/*   cubes.
/*
/*****
CubeOnIsoTable::CubeOnIsoTable(unsigned int NbCubes)
{
    if (NbCubes == 0 ) {
        printf("Error in CubeOnIsoTable::constructor: ");
        printf("number of cubes cannot be 0.\n");
    }
}

```

```

        exit(0);
    }
    CubeOnIsoTable::CubeNo = new unsigned int[NbCubes];
    if (CubeOnIsoTable::CubeNo == NULL) {
        printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    CubeOnIsoTable::CubeConfig = new unsigned char[NbCubes];
    if (CubeOnIsoTable::CubeConfig == NULL) {
        printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    CubeOnIsoTable::NbTriangles = new unsigned char[NbCubes];
    if (CubeOnIsoTable::NbTriangles == NULL) {
        printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    CubeOnIsoTable::TriangleList = new unsigned int[NbCubes];
    if (CubeOnIsoTable::TriangleList == NULL) {
        printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    CubeOnIsoTable::TableSize = NbCubes;
    CubeOnIsoTable::CurrentIndex = 0;
}

/*****
/*
/*          CubeOnIsoTable::CubeOnIsoTable(char *FileName,
/*          char TableFormat)
/*
/*
/* Constructor that allocate the memory and load a table from a file which
/* name is in the parameter 'FileName'. The number of cubes is determined
/* by the file size. 'TableFormat' is the format used when the table was
/* saved on disk (BINARY or ASCII).
/*
/*
/*****
CubeOnIsoTable::CubeOnIsoTable(char *FileName, char TableFormat)
{
    FILE *FileToRead;
    char StringToRead[CU2_LG_LINE+1];
    unsigned int SizeOfFile;
    unsigned int NbCubes;
    unsigned int RankInTable;

    /* to get the size of the table file */
    SizeOfFile = FileSize(FileName);
    if (SizeOfFile == 0) {
        printf("Error in CubeOnIsoTable::CubeOnIsoTable: ");
        printf("table file %s is empty or does not exist.\n", FileName);
        exit(0);
    }

    /* To get the number of elements in the table */
    if (TableFormat != ASCII_FORMAT && TableFormat != BINARY_FORMAT) {
        printf("Error in CubeOnIsoTable::CubeOnIsoTable: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
}
NbCubes = SizeOfFile / CubeOnIsoTable::DiskSpacePerCube(TableFormat);

/* To allocate the memory for the table */
CubeOnIsoTable::CubeNo = new unsigned int[NbCubes];
if (CubeOnIsoTable::CubeNo == NULL) {
    printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
    exit(0);
}
CubeOnIsoTable::CubeConfig = new unsigned char[NbCubes];
if (CubeOnIsoTable::CubeConfig == NULL) {

```

```

    printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
    exit(0);
}
CubeOnIsoTable::NbTriangles = new unsigned char[NbCubes];
if (CubeOnIsoTable::NbTriangles == NULL) {
    printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
    exit(0);
}
CubeOnIsoTable::TriangleList = new unsigned int[NbCubes];
if (CubeOnIsoTable::TriangleList == NULL) {
    printf("Error in CubeOnIsoTable::constructor: can't allocate memory.\n");
    exit(0);
}
CubeOnIsoTable::TableSize = NbCubes;
CubeOnIsoTable::CurrentIndex = CubeOnIsoTable::TableSize;

/* to read the file and store it in the memory structure */
FileToRead = fopen(FileName,"r");
for (RankInTable=1; RankInTable<=CubeOnIsoTable::TableSize; RankInTable++) {
    switch (TableFormat) {
        case ASCII_FORMAT:
            fgets(StringToRead, CU2_LG_LINE+1, FileToRead);
            CubeOnIsoTable::CubeNo[RankInTable - 1] =
                atoi(&StringToRead[CU2_POS_CUBE_NO]);
            CubeOnIsoTable::CubeConfig[RankInTable - 1] =
                atoi(&StringToRead[CU2_POS_CONFIG]);
            CubeOnIsoTable::NbTriangles[RankInTable - 1] =
                atoi(&StringToRead[CU2_POS_NB_TRIANGLES]);
            CubeOnIsoTable::TriangleList[RankInTable - 1] =
                atoi(&StringToRead[CU2_POS_TRIANGLE_LIST]);
            break;
        case BINARY_FORMAT:
            fread(&CubeOnIsoTable::CubeNo[RankInTable - 1],
                sizeof(*CubeOnIsoTable::CubeNo), 1, FileToRead);
            fread(&CubeOnIsoTable::CubeConfig[RankInTable - 1],
                sizeof(*CubeOnIsoTable::CubeConfig), 1, FileToRead);
            fread(&CubeOnIsoTable::NbTriangles[RankInTable - 1],
                sizeof(*CubeOnIsoTable::NbTriangles), 1, FileToRead);
            fread(&CubeOnIsoTable::TriangleList[RankInTable - 1],
                sizeof(*CubeOnIsoTable::TriangleList), 1, FileToRead);
            break;
    }
}
fclose(FileToRead);
}

/*****
/*
/*          CubeOnIsoTable::~CubeOnIsoTable(void)
/*
/* Destructeur of a table that frees the memory allocated for the table.
/*
/*
/*****
CubeOnIsoTable::~CubeOnIsoTable(void)
{
/* to free the memory allocated by the constructor */
delete CubeOnIsoTable::CubeNo;
delete CubeOnIsoTable::CubeConfig;
delete CubeOnIsoTable::NbTriangles;
delete CubeOnIsoTable::TriangleList;
}

/*****
/*
/*          void CubeOnIsoTable::AddToTable(unsigned int Cube,
/*
/*          unsigned char Config,
/*
/*          unsigned char NbTr,
/*
/*          unsigned int TrList)
/*
/*
/*****/

```



```

/*          unsigned char *NbTr,          */
/*          unsigned int *FirstTr)       */
/*          */
/* Returns the configuration number, the number of triangles and the pointer */
/* to the first triangle in the table list of a cube given by its          */
/* identification number: 'Cube'. The information is returned into          */
/* 'Config', 'NbTr', and 'FirstTr' respectively.                          */
/*          */
/*          */
/*****
void CubeOnIsoTable::GetCube(unsigned int Cube, unsigned char *Config,
                            unsigned char *NbTr, unsigned int *FirstTr)
{
    unsigned int RankInTable;

    RankInTable = CubeOnIsoTable::WhichRankInTable(Cube);
    if (RankInTable == 0) {
        *Config = 0;
        *NbTr = 0;
    }
    else {
        *Config = CubeOnIsoTable::CubeConfig[RankInTable - 1];
        *NbTr = CubeOnIsoTable::NbTriangles[RankInTable - 1];
        *FirstTr = CubeOnIsoTable::TriangleList[RankInTable - 1];
    }
}

/*****
/*          */
/*          void CubeOnIsoTable::SaveToDisk(char *FileName,          */
/*          char TableFormat)          */
/*          */
/* Saves a table on the disk. 'TableFormat' specify the format to save     */
/* the table (BINARY or ASCII). If the file already exists, it is replaced  */
/* by the new one.                                                         */
/*          */
/*          */
/*****
void CubeOnIsoTable::SaveToDisk(char *FileName, char TableFormat)
{
    FILE *FileToWrite;
    unsigned int RankInTable;
    char StringToWrite[CU2_LG_LINE+1];

    /* to open the file */
    FileToWrite = fopen(FileName, "w");
    if (FileToWrite == NULL) {
        printf("Error in CubeOnIsoTable::SaveToDisk: ");
        printf("can't open file: '%s'\n", FileName);
        exit(0);
    }
    /* to write the file */
    for (RankInTable=1; RankInTable<=CubeOnIsoTable::TableSize; RankInTable++) {
        switch (TableFormat) {
            case ASCII_FORMAT:
                sprintf(StringToWrite, CU2_LINE_FORMAT,
                        CubeOnIsoTable::CubeNo[RankInTable - 1],
                        CubeOnIsoTable::CubeConfig[RankInTable - 1],
                        CubeOnIsoTable::NbTriangles[RankInTable - 1],
                        CubeOnIsoTable::TriangleList[RankInTable - 1]);
                fputs(StringToWrite, FileToWrite);
                break;
            case BINARY_FORMAT:
                fwrite(&CubeOnIsoTable::CubeNo[RankInTable - 1],
                    sizeof(*CubeOnIsoTable::CubeNo), 1, FileToWrite);
                fwrite(&CubeOnIsoTable::CubeConfig[RankInTable - 1],
                    sizeof(*CubeOnIsoTable::CubeConfig), 1, FileToWrite);
                fwrite(&CubeOnIsoTable::NbTriangles[RankInTable - 1],
                    sizeof(*CubeOnIsoTable::NbTriangles), 1, FileToWrite);
                fwrite(&CubeOnIsoTable::TriangleList[RankInTable - 1],
                    sizeof(*CubeOnIsoTable::TriangleList), 1, FileToWrite);

```

```

        break;
    default:
        printf("Error in CubeOnIsoTable::SaveToDisk: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
}
fclose(FileToWrite);
}

/*****
/*
/*          unsigned int CubeOnIsoTable::NbCubesInTable(void)          */
/*
/* Returns the current number of cubes in the table.                    */
/*
/*
*****/
unsigned int CubeOnIsoTable::NbCubesInTable(void)
{
    return(CubeOnIsoTable::CurrentIndex);
}

/*****
/*
/*          unsigned int CubeOnIsoTable::MemoryPerCube(void)          */
/*
/* Returns the memory requirement (in bytes) to store one cube.        */
/* It is useful for large images to estimate the memory that will be used */
/* by the program that handle the table.                                */
/*
*****/
unsigned int CubeOnIsoTable::MemoryPerCube(void)
{
    return( sizeof(*CubeOnIsoTable::CubeNo) +
            sizeof(*CubeOnIsoTable::CubeConfig) +
            sizeof(*CubeOnIsoTable::NbTriangles) +
            sizeof(*CubeOnIsoTable::TriangleList) );
}

/*****
/*
/*          unsigned int CubeOnIsoTable::DiskSpacePerCube(char TableFormat) */
/*
/* Returns the space requirement (in bytes) to store one cube on the disk. */
/* It is useful for large images to estimate the space that will be used    */
/* to store the image on the disk.                                          */
/* TableFormat must specify the format that is to be used to store the table */
/* (BINARY or ASCII).                                                       */
/*
*****/
unsigned int CubeOnIsoTable::DiskSpacePerCube(char TableFormat)
{
    unsigned int DiskSpace;

    switch (TableFormat) {
        case ASCII_FORMAT:
            DiskSpace = CU2_LG_LINE;
            break;
        case BINARY_FORMAT:
            DiskSpace = sizeof(*CubeOnIsoTable::CubeNo) +
                        sizeof(*CubeOnIsoTable::CubeConfig) +
                        sizeof(*CubeOnIsoTable::NbTriangles) +
                        sizeof(*CubeOnIsoTable::TriangleList);
            break;
        default:
            printf("Error in CubeOnIsoTable::DiskSpacePerCube: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
    }
}

```

```
return(DiskSpace);  
}
```

TriangleTable

TriangleTable.h

```

/*****
/*   TriangleTable.h                                     */
/*                                                     */
/*   This program is a tool to handle the triangle table of an image */
/*   created with the Marching Cube algorithm.           */
/*                                                     */
/*****

/* The class to store the entire table */
class TriangleTable {
private:
    unsigned int TableSize;
    unsigned int *Summit1; /* Pointer to list of 1st summit of triangle. */
    unsigned int *Summit2; /* Pointer to list of 2nd summit of triangle. */
    unsigned int *Summit3; /* Pointer to list of 3rd summit of triangle. */
                        /* we don't know how many summits yet */
    int CurrentIndex;
public:
    TriangleTable(unsigned int NbTriangles);
    TriangleTable(char *FileName, char TableFormat);
    ~TriangleTable(void);
    void AddToTable(unsigned int Sum1, unsigned int Sum2, unsigned int Sum3);
    void GetTriangle(unsigned int TriangleNo,
                    unsigned int *Sum1, unsigned int *Sum2, unsigned int *Sum3);
    void SaveToDisk(char *FileName, char TableFormat);
    unsigned int NbTrianglesInTable(void);
    unsigned int MemoryPerTriangle(void);
    unsigned int DiskSpacePerTriangle(char TableFormat);
};

```

TriangleTable.cpp

```

/*****
/*   TriangleTable.cpp                                 */
/*                                                     */
/*   This program is a tool to handle the triangle table of an image */
/*   created with the Marching Cube algorithm.           */
/*                                                     */
/*****

#include <stdio.h>
#include <stdlib.h>
#include "../CppTools/FileStuff.h"
#include "TableFormat.h"
#include "TriangleTable.h"

/*****
/*
/*           TriangleTable::TriangleTable(unsigned int NbTriangles)
/*
/* Constructor that allocate the memory for an empty table of 'NbTriangles'
/* triangles.
/*
/*****
TriangleTable::TriangleTable(unsigned int NbTriangles)
{
    if (NbTriangles == 0 ) {
        printf("Error in TriangleTable::constructor: ");
        printf("number of triangles cannot be 0.\n");
        exit(0);
    }
    TriangleTable::Summit1 = new unsigned int[NbTriangles];

```

```

if (TriangleTable::Summit1 == NULL) {
    printf("Error in TriangleTable::constructor: can't allocate memory.\n");
    exit(0);
}
TriangleTable::Summit2 = new unsigned int[NbTriangles];
if (TriangleTable::Summit2 == NULL) {
    printf("Error in TriangleTable::constructor: can't allocate memory.\n");
    exit(0);
}
TriangleTable::Summit3 = new unsigned int[NbTriangles];
if (TriangleTable::Summit3 == NULL) {
    printf("Error in TriangleTable::constructor: can't allocate memory.\n");
    exit(0);
}
TriangleTable::TableSize = NbTriangles;
TriangleTable::CurrentIndex = 0;
}

/*****
/*
/*      TriangleTable::TriangleTable(char *FileName,
/*                                     char TableFormat)
/*
/*
/* Constructor that allocate the memory and load a table from a file which
/* name is in the parameter 'FileName'. The number of triangles is
/* determined by the file size. 'TableFormat' is the format used when the
/* table was saved on disk (BINARY or ASCII).
/*
/*
*****/
TriangleTable::TriangleTable(char *FileName, char TableFormat)
{
    FILE *FileToRead;
    char StringToRead[TRI_LG_LINE+1];
    unsigned int SizeOfFile;
    unsigned int NbTriangles;
    unsigned int RankInTable;

    /* to get the size of the table file */
    SizeOfFile = FileSize(FileName);
    if (SizeOfFile == 0) {
        printf("Error in TriangleTable::TriangleTable: ");
        printf("table file %s is empty or does not exist.\n", FileName);
        exit(0);
    }

    /* To get the number of elements in the table */
    if (TableFormat != ASCII_FORMAT && TableFormat != BINARY_FORMAT) {
        printf("Error in TriangleTable::TriangleTable: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
    NbTriangles = SizeOfFile / TriangleTable::DiskSpacePerTriangle(TableFormat);

    /* To allocate the memory for the table */
    TriangleTable::Summit1 = new unsigned int[NbTriangles];
    if (TriangleTable::Summit1 == NULL) {
        printf("Error in TriangleTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    TriangleTable::Summit2 = new unsigned int[NbTriangles];
    if (TriangleTable::Summit2 == NULL) {
        printf("Error in TriangleTable::constructor: can't allocate memory.\n");
        exit(0);
    }
    TriangleTable::Summit3 = new unsigned int[NbTriangles];
    if (TriangleTable::Summit3 == NULL) {
        printf("Error in TriangleTable::constructor: can't allocate memory.\n");
        exit(0);
    }
}

```

```

TriangleTable::TableSize = NbTriangles;
TriangleTable::CurrentIndex = TriangleTable::TableSize;

/* to read the file and store it in the memory structure */
FileToRead = fopen(fileName,"r");
for (RankInTable=1; RankInTable<=TriangleTable::TableSize; RankInTable++) {
    switch (TableFormat) {
        case ASCII_FORMAT:
            fgets(StringToRead, TRI_LG_LINE+1, FileToRead);
            TriangleTable::Summit1[RankInTable - 1] =
                atoi(&StringToRead[TRI_POS_SUMMIT_1]);
            TriangleTable::Summit2[RankInTable - 1] =
                atoi(&StringToRead[TRI_POS_SUMMIT_2]);
            TriangleTable::Summit3[RankInTable - 1] =
                atoi(&StringToRead[TRI_POS_SUMMIT_3]);
            break;
        case BINARY_FORMAT:
            fread(&TriangleTable::Summit1[RankInTable - 1],
                sizeof(*TriangleTable::Summit1), 1, FileToRead);
            fread(&TriangleTable::Summit2[RankInTable - 1],
                sizeof(*TriangleTable::Summit2), 1, FileToRead);
            fread(&TriangleTable::Summit3[RankInTable - 1],
                sizeof(*TriangleTable::Summit3), 1, FileToRead);
            break;
    }
}
fclose(FileToRead);
}

/*****
/*
/*          TriangleTable::~TriangleTable(void)          */
/*
/*  Destructor of a table that frees the memory allocated for the table.  */
/*
/*****
TriangleTable::~TriangleTable(void)
{
    /* to free the memory allocated by the constructor */
    delete TriangleTable::Summit1;
    delete TriangleTable::Summit2;
    delete TriangleTable::Summit3;
}

/*****
/*
/*          void TriangleTable::AddToTable(unsigned int Sum1,
/*                                     unsigned int Sum2,
/*                                     unsigned int Sum3)
/*
/*  To add a new triangle to the table.
/*  'Sum1', 'Sum2', and 'Sum3' are the pointers to the three summits of the
/*  triangle.
/*
/*****
void TriangleTable::AddToTable(unsigned int Sum1, unsigned int Sum2,
                               unsigned int Sum3)
{
    if (TriangleTable::CurrentIndex >= TriangleTable::TableSize ) {
        printf("Error in TriangleTable::AddToTable: ");
        printf("cannot add another triangle in table of size %d.\n",
            TriangleTable::TableSize);
        exit(0);
    }
    TriangleTable::Summit1[TriangleTable::CurrentIndex] = Sum1;
    TriangleTable::Summit2[TriangleTable::CurrentIndex] = Sum2;
    TriangleTable::Summit3[TriangleTable::CurrentIndex] = Sum3;
    TriangleTable::CurrentIndex++;
}

```

```

/*****
*/
/*      void TriangleTable::GetTriangle(unsigned int TriangleNo,
/*                                     unsigned int *Sum1,
/*                                     unsigned int *Sum2,
/*                                     unsigned int *Sum3)
*/
/*
/* Set 'Sum1', 'Sum2', and 'Sum3' with the pointers to the three summit of
/* the triangle 'TriangleNo'.
*/
*/
/*****
void TriangleTable::GetTriangle(unsigned int TriangleNo,
    unsigned int *Sum1, unsigned int *Sum2, unsigned int *Sum3)
{
    if (TriangleNo == 0) {
        printf("Error in TriangleTable::GetTriangle: ");
        printf("triangle number cannot be 0.\n");
        exit(0);
    }
    if (TriangleNo > TriangleTable::TableSize) {
        printf("Error in TriangleTable::GetTriangle: ");
        printf("cannot find triangle %d in table of size %d.\n", TriangleNo,
            TriangleTable::TableSize);
        exit(0);
    }
    *Sum1 = TriangleTable::Summit1[TriangleNo - 1];
    *Sum2 = TriangleTable::Summit2[TriangleNo - 1];
    *Sum3 = TriangleTable::Summit3[TriangleNo - 1];
}

/*****
*/
/*      void TriangleTable::SaveToDisk(char *FileName,
/*                                     char TableFormat)
*/
/*
/* Saves a table on the disk. 'TableFormat' specify the format to save
/* the table (BINARY or ASCII). If the file already exists, it is replaced
/* by the new one.
*/
*/
/*****
void TriangleTable::SaveToDisk(char *FileName, char TableFormat)
{
    FILE *FileToWrite;
    unsigned int RankInTable;
    char StringToWrite[TRI_LG_LINE+1];

    /* to open the file */
    FileToWrite = fopen(FileName,"w");
    if (FileToWrite == NULL) {
        printf("Error in TriangleTable::SaveToDisk: ");
        printf("can't open file: '%s'\n", FileName);
        exit(0);
    }
    /* to write the file */
    for (RankInTable=1; RankInTable<=TriangleTable::TableSize; RankInTable++) {
        switch (TableFormat) {
            case ASCII_FORMAT:
                sprintf(StringToWrite, TRI_LINE_FORMAT,
                    TriangleTable::Summit1[RankInTable - 1],
                    TriangleTable::Summit2[RankInTable - 1],
                    TriangleTable::Summit3[RankInTable - 1]);
                fputs(StringToWrite, FileToWrite);
                break;
            case BINARY_FORMAT:
                fwrite(&TriangleTable::Summit1[RankInTable - 1],
                    sizeof(*TriangleTable::Summit1), 1, FileToWrite);
                fwrite(&TriangleTable::Summit2[RankInTable - 1],
                    sizeof(*TriangleTable::Summit2), 1, FileToWrite);

```

```

        fwrite(&TriangleTable::Summit3[RankInTable - 1],
              sizeof(*TriangleTable::Summit3), 1, FileToWrite);
        break;
    default:
        printf("Error in TriangleTable::SaveToDisk: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
}
fclose(FileToWrite);
}

/*****
/*
/*      unsigned int TriangleTable::NbTrianglesInTable(void)      */
/*
/* Returns the current number of triangles in the table.          */
/*
/*
*****/
unsigned int TriangleTable::NbTrianglesInTable(void)
{
    return(TriangleTable::CurrentIndex);
}

/*****
/*
/*      unsigned int TriangleTable::MemoryPerTriangle(void)      */
/*
/* Returns the memory requirement (in bytes) to store one triangle. */
/* It is useful for large images to estimate the memory that will be used */
/* by the program that handle the table.
/*
/*
*****/
unsigned int TriangleTable::MemoryPerTriangle(void)
{
    return( sizeof(*TriangleTable::Summit1) +
           sizeof(*TriangleTable::Summit2) +
           sizeof(*TriangleTable::Summit3) );
}

/*****
/*
/*      unsigned int TriangleTable::DiskSpacePerTriangle(char TableFormat) */
/*
/* Returns the space requirement (in bytes) to store one triangle on the */
/* disk. It is useful for large images to estimate the space that will be */
/* used to store the image on the disk.
/*
/* TableFormat must specify the format that is to be used to store the table */
/* (BINARY or ASCII).
/*
/*
*****/
unsigned int TriangleTable::DiskSpacePerTriangle(char TableFormat)
{
    unsigned int DiskSpace;

    switch (TableFormat) {
        case ASCII_FORMAT:
            DiskSpace = TRI_LG_LINE;
            break;
        case BINARY_FORMAT:
            DiskSpace = sizeof(*TriangleTable::Summit1) +
                       sizeof(*TriangleTable::Summit2) +
                       sizeof(*TriangleTable::Summit3);
            break;
        default:
            printf("Error in TriangleTable::DiskSpacePerTriangle: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
    }
}

```

```
return(DiskSpace);  
}
```

SummitTable

SummitTable.h

```

/*****
/*  SummitTable.h
/*
/*  This program is a tool (C++ class plus functions) to handle the summit
/*  table of an image created with the Marching Cube algorithm.
/*
/*
/*****

/* The class to store the entire table */
class SummitTable {
private:
    unsigned int TableSize; /* Number of summits in the table */
    unsigned int *EdgeNo; /* Pointer to list of edge numbers. */
    float *X; /* Pointer to list of X abscissa. */
    float *Y; /* Pointer to list of Y abscissa. */
    float *Z; /* Pointer to list of Z abscissa. */
                /* we don't know how many summits yet. It */
                /* be set by the constructor of the class */

    unsigned int CurrentIndex;
public:
    SummitTable(unsigned int NbSummits);
    SummitTable(char *FileName, char TableFormat);
    ~SummitTable(void);
    void AddToTable(unsigned int Edge,
                    float XSummit, float YSummit, float ZSummit);
    unsigned int WhichRankInTable(unsigned int Edge);
    void GetSummit(unsigned int SummitNo,
                    float *XOfSummit, float *YOfSummit, float *ZOfSummit);
    void SaveToDisk(char *FileName, char TableFormat);
    unsigned int NbSummitsInTable(void);
    unsigned int MemoryPerSummit(void);
    unsigned int DiskSpacePerSummit(char TableFormat);
};

```

SummitTable.cpp

```

/*****
/*  SummitTable.cpp
/*
/*  This program is a tool (C++ class plus functions) to handle the summit
/*  table of an image created with the Marching Cube algorithm.
/*
/*
/*****

#include <stdio.h>
#include <stdlib.h>
#include "../CppTools/FileStuff.h"
#include "TableFormat.h"
#include "SummitTable.h"

/*****
/*
/*          SummitTable::SummitTable(unsigned int NbSummits)
/*
/*
/*  Constructor that allocate the memory for an empty table of 'NbSummits'
/*  summits.
/*
/*
/*****
SummitTable::SummitTable(unsigned int NbSummits)
{
    if (NbSummits == 0 ) {
        printf("Error in SummitTable::constructor: ");

```

```

    printf("number of summits cannot be 0.\n");
    exit(0);
}
SummitTable::EdgeNo = new unsigned int[NbSummits];
if (SummitTable::EdgeNo == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::X = new float[NbSummits];
if (SummitTable::X == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::Y = new float[NbSummits];
if (SummitTable::Y == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::Z = new float[NbSummits];
if (SummitTable::Z == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::TableSize = NbSummits;
SummitTable::CurrentIndex = 0;
}

/*****
/*
/*          SummitTable::SummitTable(char *FileName,
/*          char TableFormat)
/*
/* Constructor that allocate the memory and load a table from a file which
/* name is in the parameter 'FileName'. The number of summits is determined
/* by the file size. 'TableFormat' is the format used when the table was
/* saved on disk (BINARY or ASCII).
/*
/*
/*****
SummitTable::SummitTable(char *FileName, char TableFormat)
{
    FILE *FileToRead;
    char StringToRead[SUM_LG_LINE+1];
    unsigned int SizeOfFile;
    unsigned int NbSummits;
    unsigned int RankInTable;

    /* to get the size of the table file */
    SizeOfFile = FileSize(FileName);
    if (SizeOfFile == 0) {
        printf("Error in SummitTable::SummitTable: ");
        printf("table file %s is empty or does not exist.\n", FileName);
        exit(0);
    }

    /* To get the number of elements in the table */
    if (TableFormat != ASCII_FORMAT && TableFormat != BINARY_FORMAT) {
        printf("Error in SummitTable::SummitTable: ");
        printf("unknown table format: '%c'\n", TableFormat);
        exit(0);
    }
    NbSummits = SizeOfFile / SummitTable::DiskSpacePerSummit(TableFormat);

    /* To allocate the memory for the table */
    SummitTable::EdgeNo = NULL;
    SummitTable::X = new float[NbSummits];
    if (SummitTable::X == NULL) {
        printf("Error in SummitTable::constructor: can't allocate memory.\n");
        exit(0);
    }
}

```

```

SummitTable::Y = new float[NbSummits];
if (SummitTable::Y == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::Z = new float[NbSummits];
if (SummitTable::Z == NULL) {
    printf("Error in SummitTable::constructor: can't allocate memory.\n");
    exit(0);
}
SummitTable::TableSize = NbSummits;
SummitTable::CurrentIndex = SummitTable::TableSize;

/* to read the file and store it in the memory structure */
FileToRead = fopen(FileName,"r");
for (RankInTable=1; RankInTable<=SummitTable::TableSize; RankInTable++) {
    switch (TableFormat) {
        case ASCII_FORMAT:
            fgets(StringToRead, SUM_LG_LINE+1, FileToRead);
            SummitTable::X[RankInTable - 1] = atof(&StringToRead[SUM_POS_X]);
            SummitTable::Y[RankInTable - 1] = atof(&StringToRead[SUM_POS_Y]);
            SummitTable::Z[RankInTable - 1] = atof(&StringToRead[SUM_POS_Z]);
            break;
        case BINARY_FORMAT:
            fread(&SummitTable::X[RankInTable - 1],
                sizeof(*SummitTable::X), 1, FileToRead);
            fread(&SummitTable::Y[RankInTable - 1],
                sizeof(*SummitTable::Y), 1, FileToRead);
            fread(&SummitTable::Z[RankInTable - 1],
                sizeof(*SummitTable::Z), 1, FileToRead);
            break;
    }
}
fclose(FileToRead);
}

/*****
/*
/*                               SummitTable::~SummitTable(void)                               */
/*
/*                               Destructor of a table that frees the memory allocated for the table.   */
/*
/*                               *****/
SummitTable::~SummitTable(void)
{
    /* to free the memory allocated by the constructor */
    if (EdgeNo != NULL) {
        delete SummitTable::EdgeNo;
    }
    delete SummitTable::X;
    delete SummitTable::Y;
    delete SummitTable::Z;
}

/*****
/*
/*                               void SummitTable::AddToTable(unsigned int Edge,                               */
/*                               float XSummit,                               */
/*                               float YSummit,                               */
/*                               float ZSummit)                               */
/*
/*                               To add a new summit to the table. 'Edge' is the identification number   */
/*                               of the edge that hold the summit. It is unique for each edge over the   */
/*                               entire image. It is used to retrieve the summit in the table. The       */
/*                               table must be created sorted for a fast access to the summits by using   */
/*                               a binary search on the table.                                     */
/*                               'XSummit', 'YSummit', and 'ZSummit' are the coordinate of the summit in   */
/*                               the absolute referential system.                                     */
/*
/*

```

```

/*****
void SummitTable::AddToTable(unsigned int Edge,
                             float XSummit, float YSummit, float ZSummit)
{
    if (SummitTable::CurrentIndex >= SummitTable::TableSize ) {
        printf("Error in SummitTable::AddToTable: ");
        printf("cannot add another summit in table of size %d.\n",
               SummitTable::TableSize);
        exit(0);
    }
    if (EdgeNo != NULL) {
        SummitTable::EdgeNo[SummitTable::CurrentIndex] = Edge;
    }
    SummitTable::X[SummitTable::CurrentIndex] = XSummit;
    SummitTable::Y[SummitTable::CurrentIndex] = YSummit;
    SummitTable::Z[SummitTable::CurrentIndex] = ZSummit;
    SummitTable::CurrentIndex++;
}

/*****
/*
/*      unsigned int SummitTable::WhichRankInTable(unsigned int Edge)      */
/*
/* Returns the rank (from 1 to NbSummits) of a summit in the table. Uses  */
/* a binary search on the 'Edge' number to retrieve the summit in the table. */
/* If the edge is not found, then the function returns 0.                  */
/*
/*
/*****
unsigned int SummitTable::WhichRankInTable(unsigned int Edge)
{
    bool SummitFound;
    unsigned int inf, sup, center;

    /* Check if edge list is allocated */
    if (EdgeNo == NULL) {
        printf("Error in SummitTable::WhichRankInTable: ");
        printf("edge number table has not been allocated.\n");
        exit(0);
    }

    /* binary search in a sorted array */
    inf = 1;
    sup = SummitTable::TableSize;
    SummitFound = false;
    while ( (SummitFound == false) && (inf <= sup) ) {
        center = (inf + sup) / 2;
        if (SummitTable::EdgeNo[center - 1] == Edge) {
            SummitFound = true;
        }
        else {
            if (SummitTable::EdgeNo[center - 1] > Edge) {
                sup = center - 1;
            }
            else {
                inf = center + 1;
            }
        }
    }
    if (SummitFound == true) {
        return(center);
    }
    else {
        return(0); /* 0 means not found */
    }
}

/*****
/*
/*      void SummitTable::GetSummit(unsigned int SummitNo,                */
/*

```

```

/*                                     float *XOfSummit,          */
/*                                     float *YOfSummit,          */
/*                                     float *ZOfSummit)         */
/*                                     */
/* Set 'XOfSummit', 'YOfSummit', and 'ZOfSummit' with the coordinates of the */
/* summit which rank in the table is 'SummitNo'. If the rank is unknown,    */
/* it can be found by calling the previous function 'WhichRankInTable'.     */
/*                                     */
/*                                     */
/*****
void SummitTable::GetSummit(unsigned int SummitNo,
                           float *XOfSummit, float *YOfSummit, float *ZOfSummit)
{
    if (SummitNo == 0) {
        printf("Error in SummitTable::GetSummit: ");
        printf("summit number cannot be 0.\n");
        exit(0);
    }
    if (SummitNo > SummitTable::TableSize) {
        printf("Error in SummitTable::GetSummit: ");
        printf("cannot find summit %d in table of size %d.\n", SummitNo,
              SummitTable::TableSize);
        exit(0);
    }
    *XOfSummit = SummitTable::X[SummitNo - 1];
    *YOfSummit = SummitTable::Y[SummitNo - 1];
    *ZOfSummit = SummitTable::Z[SummitNo - 1];
}

/*****
/*                                     */
/* void SummitTable::SaveToDisk(char *FileName,          */
/*                               char TableFormat)      */
/*                                     */
/* Saves a table on the disk. 'TableFormat' specify the format to save     */
/* the table (BINARY or ASCII). If the file already exists, it is replaced */
/* by the new one. */
/*                                     */
/*****
void SummitTable::SaveToDisk(char *FileName, char TableFormat)
{
    FILE *FileToWrite;
    unsigned int RankInTable;
    char StringToWrite[SUM_LG_LINE+1];

    /* to open the file */
    FileToWrite = fopen(FileName, "w");
    if (FileToWrite == NULL) {
        printf("Error in SummitTable::SaveToDisk: ");
        printf("can't open file: '%s'\n", FileName);
        exit(0);
    }
    /* to write the file */
    for (RankInTable=1; RankInTable<=SummitTable::TableSize; RankInTable++) {
        switch (TableFormat) {
            case ASCII_FORMAT:
                sprintf(StringToWrite, SUM_LINE_FORMAT,
                      SummitTable::X[RankInTable - 1],
                      SummitTable::Y[RankInTable - 1],
                      SummitTable::Z[RankInTable - 1]);
                fputs(StringToWrite, FileToWrite);
                break;
            case BINARY_FORMAT:
                fwrite(&SummitTable::X[RankInTable - 1],
                      sizeof(*SummitTable::X), 1, FileToWrite);
                fwrite(&SummitTable::Y[RankInTable - 1],
                      sizeof(*SummitTable::Y), 1, FileToWrite);
                fwrite(&SummitTable::Z[RankInTable - 1],
                      sizeof(*SummitTable::Z), 1, FileToWrite);
                break;

```

```

        default:
            printf("Error in SummitTable::SaveToDisk: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
        }
    }
    fclose(FileToWrite);
}

/*****
/*
/*          unsigned int SummitTable::NbSummitsInTable(void)          */
/*
/* Returns the current number of summits in the table.                */
/*
/*
/*****
unsigned int SummitTable::NbSummitsInTable(void)
{
    return(SummitTable::CurrentIndex);
}

/*****
/*
/*          unsigned int SummitTable::MemoryPerSummit(void)          */
/*
/* Returns the memory requirement (in bytes) to store one summit.     */
/* It is useful for large images to estimate the memory that will be used */
/* by the program that handle the table.                               */
/*
/*
/*****
unsigned int SummitTable::MemoryPerSummit(void)
{
    return( sizeof(*SummitTable::EdgeNo) +
            sizeof(*SummitTable::X) +
            sizeof(*SummitTable::Y) +
            sizeof(*SummitTable::Z) );
}

/*****
/*
/*          unsigned int SummitTable::DiskSpacePerSummit(char TableFormat) */
/*
/* Returns the space requirement (in bytes) to store one summit on the disk. */
/* It is useful for large images to estimate the space that will be used */
/* to store the image on the disk.                                         */
/* TableFormat must specify the format that is to be used to store the table */
/* (BINARY or ASCII).                                                       */
/*
/*
/*****
unsigned int SummitTable::DiskSpacePerSummit(char TableFormat)
{
    unsigned int DiskSpace;

    switch (TableFormat) {
        case ASCII_FORMAT:
            DiskSpace = SUM_LG_LINE;
            break;
        case BINARY_FORMAT:
            DiskSpace = sizeof(*SummitTable::X) +
                        sizeof(*SummitTable::Y) +
                        sizeof(*SummitTable::Z);
            break;
        default:
            printf("Error in SummitTable::DiskSpacePerSummit: ");
            printf("unknown table format: '%c'\n", TableFormat);
            exit(0);
    }
    return(DiskSpace);
}

```

APPENDIX E

MC TRIANGLE GENERATION (C++ PROGRAMS)

This appendix contains the C++ program that generates the triangle list by mean of the Marching Cube algorithm. The program also calculates the surface area of the entire triangulated surface. The program uses some tools of the previous appendices:

- [Appendix B](#):
 - Image256GL
 - Histogram1D
 - Histogram2D
- [Appendix C](#):
 - MCConfigurations
- [Appendix D](#):
 - TableFormat
 - SummitTable
 - TriangleTable
 - Cube0and255Table
 - CubeOnIsoTable

```

/*****
/*  MCImageGeneration.cpp
/*
/*  This program transforms a gray level image into the isosurface that
/*  follows a given gray level of the image. The program uses the
/*  improved Marching Cube algorithm.
/*  Input:
/*    - The input image file: ImageName.IGL.
/*    - The dimensions of the image: Nx, Ny, Nz in number of voxels.
/*    - The dimensions of each voxel: Vx, Vy, Vz in centimeters.
/*    - The threshold value to determine the isosurface.
/*    - The type of methode to use: LOR, DIR, REV, DIR+, or REV+.
/*    - yes or no if one wants to build the isosurface or only calculate
/*      the statistical data.
/*    - The format for the output files: A for ASCII or B for Binary
/*  Output:
/*    - 4 tables that totaly define the isosurface (if asked):
/*      - 1) the list of the Marching cubes of the image (two tables:
/*        one for the pure bone or pure marrow cubes and one for
/*        the cubes on the isosurface). These lists allow to
/*        find the triangles that belong to each cube.
/*      - 2) the list of the summits that support the triangles.
/*      - 3) the list of the triangles that build the isosurface.
/*    - Some statistical data concerning the image:
/*      - the gray level histogram in file 'ImageName.GLH'
/*      - the Marching Cube configuration histogram in file
/*        'ImageName.COH'
/*      - the Marching Cube pattern histogram in file 'ImageName.PAH'
/*      - the edge histogram (where, along the edge, is the
/*        isosurface intersecting the edge) in file 'ImageName.EDG'
/*      - the gray level histogram of the ambiguous faces in file
/*        'ImageName.FAH'
*****/

```

```

/*          - the total number of Marching Cubes,          */
/*          - the number of Marching Cubes containing the isosurface */
/*          - the total number of Marching Edges,          */
/*          - the number of Marching Edges intersecting the isosurface */
/*          - the number of Marching Triangles within the entire image */
/*          - the total area of the isosurface             */
/*          */
/*****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "../CppTools/Image256GL.h"
#include "../CppTools/Histogram1D.h"
#include "../CppTools/Histogram2D.h"
#include "../MCConfigurations/MCConfigurations.h"
#include "../MCTables/TableFormat.h"
#include "../MCTables/SummitTable.h"
#include "../MCTables/TriangleTable.h"
#include "../MCTables/Cube0and255Table.h"
#include "../MCTables/CubeOnIsoTable.h"

/* The directories where to find the Marching Cubes description */
#define MC_DIR_LOR          "../MCConfigurations/DataLorensen"
#define MC_DIR_DIR         "../MCConfigurations/DataDirect"
#define MC_DIR_REV         "../MCConfigurations/DataReverse"
#define MC_DIR_DIR_PLUS   "../MCConfigurations/DataDirectPlus"
#define MC_DIR_REV_PLUS   "../MCConfigurations/DataReversePlus"

/* These are the parameters of the program. */
/* Every function needs to access them */
unsigned int Nx, Ny, Nz;
double Vx, Vy, Vz;
double Threshold;
bool DoWeCreateTables;

void GrayLevelHistogram(Image256GL *Image, char *FileName);
void ConfPatHistograms(Image256GL *Image, MCConfigurations *MCConf,
    char *COHistFileName, char *PAHistFileName,
    unsigned int *NbCubes, unsigned int *NbBoneCubes,
    unsigned int *NbMarrowCubes, unsigned int *NbTriangles,
    unsigned int *CubeTableSize);
void EdgeHistogram(Image256GL *Image, char *FileName, unsigned int *NbEdges);
void AmbiguousFaceHistogram(Image256GL *Image, char *FileName,
    unsigned int *NbAmbiguousFaces, unsigned int *NbFacesAbove);
void CreateCubeTables(Image256GL *Image, MCConfigurations *MCConf,
    Cube0and255Table *MCCube0and255Table, CubeOnIsoTable *MCCubeOnIsoTable);
void CreateSummitTable(Image256GL *Image, SummitTable *MCSummitTable);
void CreateTriangleTable(Image256GL *Image, MCConfigurations *MCConf,
    SummitTable *MCSummitTable, TriangleTable *MCTriangleTable,
    double *TotalArea);
void SurfaceCrossesAt(unsigned int I, unsigned int J, unsigned int K,
    unsigned int EdgeOfCube, Image256GL *Image,
    double *XSummit, double *YSummit, double *ZSummit);
unsigned int GetEdgeNo(unsigned int I, unsigned int J, unsigned int K,
    unsigned int EdgeOfCube);
double InterpolatedIntersection(double Threshold, unsigned int GLStart,
    unsigned int GLEnd);

main( int argc,
    char *argv[] )
{
    char MCDirectory[100];
    char ImageFileName[100];
    unsigned int TotalNbCubes;
    unsigned int NbCubesOnIsosurface;
    unsigned int NbPureBoneCubes;
    unsigned int NbPureMarrowCubes;
    unsigned int NbTrianglesOnIsosurface;

```

```

unsigned int TotalNbEdges;
unsigned int TotalNbFaces;
unsigned int NbEdgesOnIsosurface;
unsigned int NbAmbiguousFaces;
unsigned int NbFacesAbove;
unsigned int SizeOfCube0and255Table;
double TotalArea;
char TableFormat;

/* 1) To get the parameters and read the image file */
/*****
/* to check the command line */
if ( argc != 12 ) {
    printf("Error: bad command-line. Try again with command-line: ");
    printf("MCImageGeneration ImageFileName Nx Ny Nz Vx Vy Vz ");
    printf("Threshold MCMMethod CreateTable TableFormat\n");
    printf(" - ImageFileName: the full path name (without the ");
    printf("extension) of the\n    image file from the current ");
    printf("directory. The image must have been\n    stored ");
    printf("with the Image256GL tool and have the extension IGL.\n");
    printf(" - Nx: the number of voxels per row.\n");
    printf(" - Ny: the number of rows per slice.\n");
    printf(" - Nz: the number of slices in the image.\n");
    printf(" - Vx, Vy, and Vz: the voxel size (in cm).\n");
    printf(" - Threshold: the threshold used to separate bone from ");
    printf("marrow.\n");
    printf(" - MCMMethod: a code for the Marching Cube method to use:\n");
    printf("     LOR: initial Lorensen method,\n");
    printf("     DIR: direct method (with the 8 new patterns),\n");
    printf("     REV: reverse method (with the 8 new patterns),\n");
    printf("     DIR+: improved direct method (smoothest design),\n");
    printf("     REV+: improved reverse method (smoothest design),\n");
    printf(" - CreateTable: 'y' if you want the tables to be created ");
    printf("\n    or 'n' if you just want the statistical data.\n");
    printf(" - TableFormat: 'A' for tables in ASCII format ");
    printf("\n    or 'B' for tables in binary format.\n");
    printf("ex: MCImageGeneration ../../Database/MyImageName ");
    printf("200 200 200 0.0088 0.0088 0.0088 127.5 DIR y A\n");
    exit(0);
}

/* to get the parameters */
sprintf(ImageFileName, "%s.IGL", argv[1]);
Nx = atoi(argv[2]);
Ny = atoi(argv[3]);
Nz = atoi(argv[4]);
Vx = atof(argv[5]);
Vy = atof(argv[6]);
Vz = atof(argv[7]);
Threshold = atof(argv[8]);
if (strcmp(argv[9], "LOR") == 0) {
    strcpy(MCDirectory, MC_DIR_LOR);
}
else {
    if (strcmp(argv[9], "DIR") == 0) {
        strcpy(MCDirectory, MC_DIR_DIR);
    }
    else {
        if (strcmp(argv[9], "REV") == 0) {
            strcpy(MCDirectory, MC_DIR_REV);
        }
        else {
            if (strcmp(argv[9], "DIR+") == 0) {
                strcpy(MCDirectory, MC_DIR_DIR_PLUS);
            }
            else {
                if (strcmp(argv[9], "REV+") == 0) {
                    strcpy(MCDirectory, MC_DIR_REV_PLUS);
                }
                else {

```

```

        printf("Error: unknown Marching Cube method: %s.\n",
               argv[9]);
        exit(0);
    }
}
}
}
}
if (argv[10][0] == 'y') {
    DoWeCreateTables = true;
}
else {
    DoWeCreateTables = false;
}
if (argv[11][0] == ASCII_FORMAT || argv[11][0] == BINARY_FORMAT) {
    TableFormat = argv[11][0];
}
else {
    printf("Error: unknown table format: %c.\n", argv[11][0]);
    exit(0);
}
printf("\nMarching Cube image generation:\n");
printf("    Image file: %s.\n", ImageFileName);
printf("    Image size: %3u x %3u x %3u => %u voxels.\n", Nx, Ny, Nz, Nx*Ny*Nz);
printf("    Voxel size: %8.6f x %8.6f x %8.6f cm3.\n", Vx, Vy, Vz);
printf("    Threshold: %7.3f.\n", Threshold);
if (strcmp(argv[9], "LOR") == 0) {
    printf("    Marching Cube method: Initial Lorensen.\n");
}
else {
    if (strcmp(argv[9], "DIR") == 0) {
        printf("    Marching Cube method: Initial Direct.\n");
    }
    else {
        if (strcmp(argv[9], "REV") == 0) {
            printf("    Marching Cube method: Initial Reverse.\n");
        }
        else {
            if (strcmp(argv[9], "DIR+") == 0) {
                printf("    Marching Cube method: Improved Direct.\n");
            }
            else {
                if (strcmp(argv[9], "REV+") == 0) {
                    printf("    Marching Cube method: Improved Reverse.\n");
                }
            }
        }
    }
}
}
if (DoWeCreateTables == true) {
    printf("    Statistical data and isosurface building.\n");
}
else {
    printf("    Statistical data only.\n");
}
if (TableFormat == ASCII_FORMAT) {
    printf("    Tables are in ASCII format.\n");
}
else {
    printf("    Tables are in binary format.\n");
}
/* to read the image */
printf("\n1) Reading the image file %s.\n", ImageFileName);
Image256GL Image(Nx, Ny, Nz);
Image.LoadFromFile(ImageFileName);

/* 2) To get the Marching Cubes configuration definition */
/*****
printf("\n2) Reading the Marching Cube definition in the directory %s.\n",

```

```

MCDirectory);
MCConfigurations MCConfig(MCDirectory);

/* 3) To calculate the gray level histogram of the image */
/*****
printf("\n3) Calculating the gray level histogram of the image.\n");
char GLHistFileName[100];
sprintf(GLHistFileName, "%s.GLH", argv[1]);
GrayLevelHistogram(&Image, GLHistFileName);

/* 4) To calculate the configuration and pattern histograms of the */
/*   Marching Cube image */
/*****
printf("\n4) Calculating the configuration and pattern histograms ");
printf("of the Marching Cube image.\n");
char COHistFileName[100];
sprintf(COHistFileName, "%s.COH", argv[1]);
char PAHistFileName[100];
sprintf(PAHistFileName, "%s.PAH", argv[1]);
ConfPatHistograms(&Image, &MCConfig, COHistFileName, PAHistFileName,
                  &NbCubesOnIsosurface, &NbPureBoneCubes, &NbPureMarrowCubes,
                  &NbTrianglesOnIsosurface, &SizeOfCube0and255Table);

/* 5) To calculate the edge histogram of the */
/*   Marching Cube image */
/*****
printf("\n5) Calculating the edge histogram of the Marching Cube image.\n");
char EDHistFileName[100];
sprintf(EDHistFileName, "%s.EDG", argv[1]);
EdgeHistogram(&Image, EDHistFileName, &NbEdgesOnIsosurface);

/* 6) To calculate the gray level histogram of the ambiguous faces */
/*****
printf("\n6) Calculating the GL histogram of the ambiguous faces.\n");
char FAHistFileName[100];
sprintf(FAHistFileName, "%s.FAH", argv[1]);
AmbiguousFaceHistogram(&Image, FAHistFileName,
                       &NbAmbiguousFaces, &NbFacesAbove);

/* 7) To show the statistical data */
/*****
printf("\n7) Statistical data for Marching Cube image:\n");
TotalNbEdges = (Nx-1)*Ny*Nz + Nx*(Ny-1)*Nz + Nx*Ny*(Nz-1);
TotalNbFaces = Nx*(Ny-1)*(Nz-1) + (Nx-1)*Ny*(Nz-1) + (Nx-1)*(Ny-1)*Nz;
TotalNbCubes = (Nx-1)*(Ny-1)*(Nz-1);
printf("  Total number of Marching Cubes in image:      %11u\n",
       TotalNbCubes);
printf("  Number of cubes intersecting the isosurface: %11u => %5.2f %%\n",
       NbCubesOnIsosurface, 100.0*NbCubesOnIsosurface / TotalNbCubes);
printf("  Number of pure bone cubes:                    %11u => %5.2f %%\n",
       NbPureBoneCubes, 100.0*NbPureBoneCubes / TotalNbCubes);
printf("  Number of pure marrow cubes:                   %11u => %5.2f %%\n",
       NbPureMarrowCubes, 100.0*NbPureMarrowCubes / TotalNbCubes);
printf("  Number of triangles on isosurface:              %11u\n",
       NbTrianglesOnIsosurface);
printf("  Total number of Marching Edges in image:        %11u\n",
       TotalNbEdges);
printf("  Number of edges intersecting the isosurface: %11u => %5.2f %%\n",
       NbEdgesOnIsosurface, 100.0*NbEdgesOnIsosurface / TotalNbEdges);
printf("  Total number of Marching Faces in image:        %11u\n",
       TotalNbFaces);
printf("  Number of ambiguous faces within image:         %11u => %5.2f %%\n",
       NbAmbiguousFaces, 100.0*NbAmbiguousFaces / TotalNbFaces);
printf("  Above threshold:                               %11u => %5.2f %%\n",
       NbFacesAbove, 100.0*NbFacesAbove / NbAmbiguousFaces);
printf("  Bellow threshold:                              %11u => %5.2f %%\n",
       NbAmbiguousFaces - NbFacesAbove,
       100.0*(NbAmbiguousFaces - NbFacesAbove) / NbAmbiguousFaces);

```

```

/* 8) Memory space required for the Marching Cube image */
/*****/
/* need to define fake tables to get to the size of each element */
Cube0and255Table *MCCube0and255Table;
MCCube0and255Table = new Cube0and255Table(1);
CubeOnIsoTable *MCCubeOnIsoTable;
MCCubeOnIsoTable = new CubeOnIsoTable(1);
SummitTable *MCSummitTable;
MCSummitTable = new SummitTable(1);
TriangleTable *MCTriangleTable;
MCTriangleTable = new TriangleTable(1);
printf("\n8) Memory required for the Marching Cube image:\n");
printf("   Cube 0 and 255 table:      %10u lines => %10u bytes\n",
       SizeOfCube0and255Table,
       SizeOfCube0and255Table*MCCube0and255Table->MemoryPerCube());
printf("   Cube on isosurface table: %10u lines => %10u bytes\n",
       NbCubesOnIsosurface,
       NbCubesOnIsosurface*MCCubeOnIsoTable->MemoryPerCube());
printf("   Summit table:                %10u lines => %10u bytes\n",
       NbEdgesOnIsosurface,
       NbEdgesOnIsosurface*MCSummitTable->MemoryPerSummit());
printf("   Triangle table:              %10u lines => %10u bytes\n",
       NbTrianglesOnIsosurface,
       NbTrianglesOnIsosurface*MCTriangleTable->MemoryPerTriangle());
printf("                               Total memory: %10u bytes\n",
       NbEdgesOnIsosurface*MCSummitTable->MemoryPerSummit() +
       NbTrianglesOnIsosurface*MCTriangleTable->MemoryPerTriangle() +
       SizeOfCube0and255Table*MCCube0and255Table->MemoryPerCube() +
       NbCubesOnIsosurface*MCCubeOnIsoTable->MemoryPerCube());

/* 9) Disk space space required for the Marching Cube image */
/*****/
printf("\n9) Disk space required for the Marching Cube image:\n");
printf("   Cube 0 and 255 table:      %10u lines => %10u bytes\n",
       SizeOfCube0and255Table, SizeOfCube0and255Table *
       MCCube0and255Table->DiskSpacePerCube(TableFormat));
printf("   Cube on isosurface table: %10u lines => %10u bytes\n",
       NbCubesOnIsosurface, NbCubesOnIsosurface *
       MCCubeOnIsoTable->DiskSpacePerCube(TableFormat));
printf("   Summit table:                %10u lines => %10u bytes\n",
       NbEdgesOnIsosurface, NbEdgesOnIsosurface *
       MCSummitTable->DiskSpacePerSummit(TableFormat));
printf("   Triangle table:              %10u lines => %10u bytes\n",
       NbTrianglesOnIsosurface, NbTrianglesOnIsosurface *
       MCTriangleTable->DiskSpacePerTriangle(TableFormat));
printf("                               Total disk space: %10u bytes\n",
       NbEdgesOnIsosurface *
       MCSummitTable->DiskSpacePerSummit(TableFormat) +
       NbTrianglesOnIsosurface *
       MCTriangleTable->DiskSpacePerTriangle(TableFormat) +
       SizeOfCube0and255Table *
       MCCube0and255Table->DiskSpacePerCube(TableFormat) +
       NbCubesOnIsosurface *
       MCCubeOnIsoTable->DiskSpacePerCube(TableFormat));
/* now we can delete the fake tables */
delete MCCube0and255Table;
delete MCCubeOnIsoTable;
delete MCSummitTable;
delete MCTriangleTable;
if (NbCubesOnIsosurface <= 0) {
    printf("Error: image has no isosurface.");
    exit(0);
}

/* 10) To create the cube tables */
/*****/
if (DoWeCreateTables == true) {
    printf("\n10) Creating the cube tables.\n");
    if (SizeOfCube0and255Table > 0) {

```

```

        MCCube0and255Table = new Cube0and255Table(SizeOfCube0and255Table);
    }
    else {
        MCCube0and255Table = new Cube0and255Table(1);
        MCCube0and255Table->AddToTable(0, 0);
    }
    MCCubeOnIsoTable = new CubeOnIsoTable(NbCubesOnIsosurface);
    CreateCubeTables(&Image,&MCCConfig,MCCube0and255Table,MCCubeOnIsoTable);
    char Cube0and255FileName[100];
    sprintf(Cube0and255FileName, "%s.CU1", argv[1]);
    printf("    Saving file %s on disk\n", Cube0and255FileName);
    MCCube0and255Table->SaveToDisk(Cube0and255FileName, TableFormat);
    char CubeOnIsoFileName[100];
    sprintf(CubeOnIsoFileName, "%s.CU2", argv[1]);
    printf("    Saving file %s on disk\n", CubeOnIsoFileName);
    MCCubeOnIsoTable->SaveToDisk(CubeOnIsoFileName, TableFormat);
    delete MCCube0and255Table;
    delete MCCubeOnIsoTable;
}
else {
    printf("\n10) No creation of cube tables.\n");
}

/* 11) To create the summit table */
/*****
if (DoWeCreateTables == true) {
    printf("\n11) Creating the summit table.\n");
    MCSummitTable = new SummitTable(NbEdgesOnIsosurface);
    CreateSummitTable(&Image, MCSummitTable);
    char SummitFileName[100];
    sprintf(SummitFileName, "%s.SUM", argv[1]);
    printf("    Saving file %s on disk\n", SummitFileName);
    MCSummitTable->SaveToDisk(SummitFileName, TableFormat);
}
else {
    printf("\n11) No creation of summit table.\n");
}

/* 12) To create the triangle table */
/*****
if (DoWeCreateTables == true) {
    printf("\n12) Creating the triangle table and ");
    printf("calculating the isosurface area.\n");
    MCTriangleTable = new TriangleTable(NbTrianglesOnIsosurface);
    CreateTriangleTable(&Image, &MCCConfig, MCSummitTable, MCTriangleTable,
        &TotalArea);
    char TriangleFileName[100];
    sprintf(TriangleFileName, "%s.TRI", argv[1]);
    printf("    Saving file %s on disk\n", TriangleFileName);
    MCTriangleTable->SaveToDisk(TriangleFileName, TableFormat);
    delete MCTriangleTable;
    delete MCSummitTable;
}
else {
    printf("\n12) Calculating the isosurface area; ");
    printf("no creation of triangle table.\n");
    CreateTriangleTable(&Image, &MCCConfig, MCSummitTable, MCTriangleTable,
        &TotalArea);
}
printf("    Total area of the isosurface: %20.15f cm2.\n", TotalArea);
printf("\n");
}

/*****
/* Local functions */
/*****

void GrayLevelHistogram(Image256GL *Image, char *FileName)
{

```

```

Histogram1D *GrayLevelHist;
unsigned int I, J, K;
unsigned char Voxel;

GrayLevelHist = new Histogram1D(256);
for ( K=0; K<Nz; K++ ) {
    for ( J=0; J<Ny; J++ ) {
        for ( I=0; I<Nx; I++ ) {
            Voxel = Image->GrayLevel(I, J, K);
            GrayLevelHist->AddValue(Voxel);
        }
    }
}
/* to store the histogram on the disk */
printf(" Saving file %s on disk\n", FileName);
GrayLevelHist->StoreToDisk(FileName);
/* do free the memory */
delete GrayLevelHist;
}

void ConfPatHistograms(Image256GL *Image, MCCConfigurations *MCCConf,
    char *COHistFileName, char *PAHistFileName,
    unsigned int *NbCubes, unsigned int *NbBoneCubes,
    unsigned int *NbMarrowCubes, unsigned int *NbTriangles,
    unsigned int *CubeTableSize)
{
Histogram1D *ConfigHist;
Histogram1D *PatternHist;
unsigned int I, J, K;
unsigned int CubeConfig;
unsigned int CubePattern;
unsigned int NbSurf, SurfNo;
unsigned int PreviousConfig;

ConfigHist = new Histogram1D(NB_CONFIGURATIONS);
PatternHist = new Histogram1D(NB_PATTERNS);
*NbCubes = 0;
*NbBoneCubes = 0;
*NbMarrowCubes = 0;
*NbTriangles = 0;
*CubeTableSize = 0;
PreviousConfig = -1;
for ( K=1; K<Nz; K++ ) {
    for ( J=1; J<Ny; J++ ) {
        for ( I=1; I<Nx; I++ ) {
            /* 1) the configurations */
            CubeConfig = 0;
            if (Image->GrayLevel(I-1, J-1, K-1) > Threshold) {
                CubeConfig = CubeConfig + 1; /* Vertex 0 */
            }
            if (Image->GrayLevel(I, J-1, K-1) > Threshold) {
                CubeConfig = CubeConfig + 2; /* Vertex 1 */
            }
            if (Image->GrayLevel(I, J, K-1) > Threshold) {
                CubeConfig = CubeConfig + 4; /* Vertex 2 */
            }
            if (Image->GrayLevel(I-1, J, K-1) > Threshold) {
                CubeConfig = CubeConfig + 8; /* Vertex 3 */
            }
            if (Image->GrayLevel(I-1, J-1, K) > Threshold) {
                CubeConfig = CubeConfig + 16; /* Vertex 4 */
            }
            if (Image->GrayLevel(I, J-1, K) > Threshold) {
                CubeConfig = CubeConfig + 32; /* Vertex 5 */
            }
            if (Image->GrayLevel(I, J, K) > Threshold) {
                CubeConfig = CubeConfig + 64; /* Vertex 6 */
            }
            if (Image->GrayLevel(I-1, J, K) > Threshold) {

```

```

        CubeConfig = CubeConfig + 128; /* Vertex 7 */
    }
    ConfigHist->AddValue(CubeConfig);
    /* 2) the patterns */
    CubePattern = MConf->PatternNo(CubeConfig);
    PatternHist->AddValue(CubePattern);
    if (CubeConfig == 0) {
        *NbBoneCubes = *NbBoneCubes + 1;
    }
    else {
        if (CubeConfig == (NB_CONFIGURATIONS - 1)) {
            *NbMarrowCubes = *NbMarrowCubes + 1;
        }
        else {
            *NbCubes = *NbCubes + 1;
            NbSurf = MConf->NbSurfaces(CubeConfig);
            for (SurfNo=1; SurfNo<=NbSurf; SurfNo++) {
                *NbTriangles = *NbTriangles
                    + MConf->NbTriangles(CubeConfig, SurfNo);
            }
        }
    }
    /* to count another element in the table if new configuration */
    if (CubeConfig == 0 || CubeConfig == 255) {
        if ( CubeConfig != PreviousConfig ) {
            *CubeTableSize = *CubeTableSize + 1;
            PreviousConfig = CubeConfig;
        }
    }
}
}

/* to store the histograms on the disk */
printf(" Saving file %s on disk\n", COHistFileName);
ConfigHist->StoreToDisk(COHistFileName);
printf(" Saving file %s on disk\n", PAHistFileName);
PatternHist->StoreToDisk(PAHistFileName);
/* do free the memory */
delete ConfigHist;
delete PatternHist;
}

#define EDGE_HISTO_SIZE 16

void EdgeHistogram(Image256GL *Image, char *FileName, unsigned int *NbEdges)
{
    Histogram2D *EdgeHist;
    unsigned int I, J, K;
    unsigned int NbEdgesForDimension;
    unsigned int TotalNbEdges;
    unsigned int GLStartEdge, GLEndEdge;

    EdgeHist = new Histogram2D(EDGE_HISTO_SIZE, EDGE_HISTO_SIZE);
    *NbEdges = 0;
    /* 1) edges parallel to (0,x) */
    TotalNbEdges = (Nx - 1)*Ny*Nz;
    NbEdgesForDimension = 0;
    for ( K=0; K<Nz; K++ ) {
        for ( J=0; J<Ny; J++ ) {
            for ( I=0; I<(Nx-1); I++ ) {
                GLStartEdge = Image->GrayLevel(I, J, K);
                GLEndEdge = Image->GrayLevel(I+1, J, K);
                if ( ( GLStartEdge > Threshold ) && ( GLEndEdge < Threshold ) ||
                    ( GLStartEdge < Threshold ) && ( GLEndEdge > Threshold ) ) {
                    /* to calculate the location of the summit */
                    EdgeHist->AddValue(GLStartEdge/(256/EDGE_HISTO_SIZE),
                                        GLEndEdge/(256/EDGE_HISTO_SIZE));
                    /* to count the edges */
                    NbEdgesForDimension = NbEdgesForDimension + 1;
                }
            }
        }
    }
}

```

```

    }
  }
}
printf("  Number of (O,x) edges crossing the surface: %llu => %5.2f %%\n",
       NbEdgesForDimension, 100.0*NbEdgesForDimension / TotalNbEdges);
*NbEdges = *NbEdges + NbEdgesForDimension;
/* 2) edges parallel to (O,y) */
TotalNbEdges = Nx*(Ny - 1)*Nz;
NbEdgesForDimension = 0;
for ( K=0; K<Nz; K++ ) {
  for ( J=0; J<(Ny-1); J++ ) {
    for ( I=0; I<Nx; I++ ) {
      GLStartEdge = Image->GrayLevel(I, J, K);
      GLEndEdge = Image->GrayLevel(I, J+1, K);
      if ( ( (GLStartEdge > Threshold) && (GLEndEdge < Threshold) ) ||
          ( (GLStartEdge < Threshold) && (GLEndEdge > Threshold) ) ) {
        /* to calculate the location of the summit */
        EdgeHist->AddValue(GLStartEdge/(256/EDGE_HISTO_SIZE),
                          GLEndEdge/(256/EDGE_HISTO_SIZE));
        /* to count the edges */
        NbEdgesForDimension = NbEdgesForDimension + 1;
      }
    }
  }
}
printf("  Number of (O,y) edges crossing the surface: %llu => %5.2f %%\n",
       NbEdgesForDimension, 100.0*NbEdgesForDimension / TotalNbEdges);
*NbEdges = *NbEdges + NbEdgesForDimension;
/* 3) edges parallel to (O,z) */
TotalNbEdges = Nx*Ny*(Nz-1);
NbEdgesForDimension = 0;
for ( K=0; K<(Nz-1); K++ ) {
  for ( J=0; J<Ny; J++ ) {
    for ( I=0; I<Nx; I++ ) {
      GLStartEdge = Image->GrayLevel(I, J, K);
      GLEndEdge = Image->GrayLevel(I, J, K+1);
      if ( ( (GLStartEdge > Threshold) && (GLEndEdge < Threshold) ) ||
          ( (GLStartEdge < Threshold) && (GLEndEdge > Threshold) ) ) {
        /* to calculate the location of the summit */
        EdgeHist->AddValue(GLStartEdge/(256/EDGE_HISTO_SIZE),
                          GLEndEdge/(256/EDGE_HISTO_SIZE));
        /* to count the edges */
        NbEdgesForDimension = NbEdgesForDimension + 1;
      }
    }
  }
}
printf("  Number of (O,z) edges crossing the surface: %llu => %5.2f %%\n",
       NbEdgesForDimension, 100.0*NbEdgesForDimension / TotalNbEdges);
*NbEdges = *NbEdges + NbEdgesForDimension;
/* to store the histogram on the disk */
printf("  Saving file %s on disk\n", FileName);
EdgeHist->StoreToDisk(FileName);
/* do free the memory */
delete EdgeHist;
}

void AmbiguousFaceHistogram(Image256GL *Image, char *FileName,
                           unsigned int *NbAmbiguousFaces, unsigned int *NbFacesAbove)
{
  Histogram1D *FaceHist;
  unsigned int I, J, K;
  unsigned int FaceGrayLevel;
  unsigned char GL00, GL10, GL01, GL11;
  double AsymptoticDecider;

  FaceHist = new Histogram1D(256);
  *NbAmbiguousFaces = 0;

```



```

                (double)GL10 - (double)GL01);
        if (AsymptoticDecider > Threshold) {
            *NbFacesAbove += 1;
        }
        FaceGrayLevel = AsymptoticDecider;
        FaceHist->AddValue(FaceGrayLevel);
    }
}

/* to store the histogram on the disk */
printf(" Saving file %s on disk\n", FileName);
FaceHist->StoreToDisk(FileName);
/* do free the memory */
delete FaceHist;
}

void CreateCubeTables(Image256GL *Image, MCCConfigurations *MCCConf,
    Cube0and255Table *MCCCube0and255Table, CubeOnIsoTable *MCCCubeOnIsoTable)
{
    unsigned int I, J, K;
    unsigned int CubeConfig;
    unsigned int CubeNo;
    unsigned int PreviousConfig;
    unsigned int NbSurf, SurfNo;
    unsigned int NbTr, FirstTr;

    FirstTr = 1;
    CubeNo = 0;
    PreviousConfig = -1;
    for ( K=1; K<Nz; K++ ) {
        for ( J=1; J<Ny; J++ ) {
            for ( I=1; I<Nx; I++ ) {
                /* to calculate the cube config */
                CubeConfig = 0;
                if (Image->GrayLevel(I-1, J-1, K-1) > Threshold) {
                    CubeConfig = CubeConfig + 1; /* Vertex 0 */
                }
                if (Image->GrayLevel(I, J-1, K-1) > Threshold) {
                    CubeConfig = CubeConfig + 2; /* Vertex 1 */
                }
                if (Image->GrayLevel(I, J, K-1) > Threshold) {
                    CubeConfig = CubeConfig + 4; /* Vertex 2 */
                }
                if (Image->GrayLevel(I-1, J, K-1) > Threshold) {
                    CubeConfig = CubeConfig + 8; /* Vertex 3 */
                }
                if (Image->GrayLevel(I-1, J-1, K) > Threshold) {
                    CubeConfig = CubeConfig + 16; /* Vertex 4 */
                }
                if (Image->GrayLevel(I, J-1, K) > Threshold) {
                    CubeConfig = CubeConfig + 32; /* Vertex 5 */
                }
                if (Image->GrayLevel(I, J, K) > Threshold) {
                    CubeConfig = CubeConfig + 64; /* Vertex 6 */
                }
                if (Image->GrayLevel(I-1, J, K) > Threshold) {
                    CubeConfig = CubeConfig + 128; /* Vertex 7 */
                }
                /* to add in the correct table */
                if (CubeConfig == 0 || CubeConfig == 255) {
                    if ( CubeConfig != PreviousConfig ) {
                        MCCCube0and255Table->AddToTable(CubeNo, CubeConfig);
                        PreviousConfig = CubeConfig;
                    }
                }
                else {
                    NbSurf = MCCConf->NbSurfaces(CubeConfig);
                    NbTr = 0;
                }
            }
        }
    }
}

```

```

        for (SurfNo=1; SurfNo<=NbSurf; SurfNo++) {
            NbTr = NbTr + MCConf->NbTriangles(CubeConfig, SurfNo);
        }
        MCCubeOnIsoTable->AddToTable(CubeNo, CubeConfig, NbTr, FirstTr);
        FirstTr = FirstTr + NbTr;
    }
    CubeNo = CubeNo + 1;
}
}
}

void CreateSummitTable(Image256GL *Image, SummitTable *MCSummitTable)
{
    unsigned int I, J, K;
    unsigned int EdgeNo;
    unsigned int GLStartEdge, GLEndEdge;
    double Intersection;
    double SummitX, SummitY, SummitZ;

    EdgeNo = 0;
    /* a) edges parallel to (0,x) */
    /*******/
    for ( K=0; K<Nz; K++ ) {
        for ( J=0; J<Ny; J++ ) {
            for ( I=0; I<(Nx-1); I++ ) {
                GLStartEdge = Image->GrayLevel(I, J, K);
                GLEndEdge = Image->GrayLevel(I+1, J, K);
                if ( ( (GLStartEdge > Threshold) && (GLEndEdge < Threshold) ) ||
                    ( (GLStartEdge < Threshold) && (GLEndEdge > Threshold) ) ) {
                    /* to calculate the location of the summit */
                    Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                                GLEndEdge);

                    SummitX = (I + 0.5 + Intersection) * Vx;
                    SummitY = (J + 0.5) * Vy;
                    SummitZ = (K + 0.5) * Vz;
                    /* add edge in table */
                    MCSummitTable->AddToTable(EdgeNo, SummitX, SummitY, SummitZ);
                }
                EdgeNo = EdgeNo + 1;
            }
        }
    }
    /* b) edges parallel to (0,y) */
    /*******/
    for ( K=0; K<Nz; K++ ) {
        for ( J=0; J<(Ny-1); J++ ) {
            for ( I=0; I<Nx; I++ ) {
                GLStartEdge = Image->GrayLevel(I, J, K);
                GLEndEdge = Image->GrayLevel(I, J+1, K);
                if ( ( (GLStartEdge > Threshold) && (GLEndEdge < Threshold) ) ||
                    ( (GLStartEdge < Threshold) && (GLEndEdge > Threshold) ) ) {
                    Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                                GLEndEdge);

                    SummitX = (I + 0.5) * Vx;
                    SummitY = (J + 0.5 + Intersection) * Vy;
                    SummitZ = (K + 0.5) * Vz;
                    /* add edge in table */
                    MCSummitTable->AddToTable(EdgeNo, SummitX, SummitY, SummitZ);
                }
                EdgeNo = EdgeNo + 1;
            }
        }
    }
    /* c) edges parallel to (0,z) */
    /*******/
    for ( K=0; K<(Nz-1); K++ ) {
        for ( J=0; J<Ny; J++ ) {
            for ( I=0; I<Nx; I++ ) {

```

```

GLStartEdge = Image->GrayLevel(I, J, K);
GLEndEdge = Image->GrayLevel(I, J, K+1);
if ( ( GLStartEdge > Threshold) ||
      ( GLEndEdge < Threshold) ) ||
    ( (GLStartEdge < Threshold) && (GLEndEdge > Threshold) ) ) {
    Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                           GLEndEdge);

    SummitX = (I + 0.5) * Vx;
    SummitY = (J + 0.5) * Vy;
    SummitZ = (K + 0.5 + Intersection) * Vz;
    /* add edge in table */
    MCSummitTable->AddToTable(EdgeNo, SummitX, SummitY, SummitZ);
}
EdgeNo = EdgeNo + 1;
}
}
}

void CreateTriangleTable(Image256GL *Image, MConfigurations *MCCConf,
                        SummitTable *MCSummitTable, TriangleTable *MCTriangleTable,
                        double *TotalArea)
{
    unsigned int I, J, K;
    unsigned int CubeConfig;
    unsigned int NbSurf, SurfNo;
    unsigned int NbDes, DesNo;
    unsigned int NbTr, TrNo;
    double TriangleArea;
    double DesignArea;
    double SurfaceArea;
    double MaxArea;
    double XSummit1, YSummit1, ZSummit1;
    double XSummit2, YSummit2, ZSummit2;
    double XSummit3, YSummit3, ZSummit3;
    unsigned int EdgeSummit1, EdgeSummit2, EdgeSummit3;
    unsigned int EdgeNoSum1, EdgeNoSum2, EdgeNoSum3;
    unsigned int SumNoSummit1, SumNoSummit2, SumNoSummit3;
    double XVectProd, YVectProd, ZVectProd;
    unsigned int BestDesign;

    *TotalArea = 0.0;
    MaxArea = 2*(Vx*Vy + Vy*Vz + Vz*Vx); /* outside area of the cube */
    for ( K=0; K<(Nz-1); K++ ) {
        for ( J=0; J<(Ny-1); J++ ) {
            for ( I=0; I<(Nx-1); I++ ) {
                /* to calculate the configuration number */
                CubeConfig = 0;
                if (Image->GrayLevel(I, J, K) > Threshold) {
                    CubeConfig = CubeConfig + 1; /* Vertex 0 */
                }
                if (Image->GrayLevel(I+1, J, K) > Threshold) {
                    CubeConfig = CubeConfig + 2; /* Vertex 1 */
                }
                if (Image->GrayLevel(I+1, J+1, K) > Threshold) {
                    CubeConfig = CubeConfig + 4; /* Vertex 2 */
                }
                if (Image->GrayLevel(I, J+1, K) > Threshold) {
                    CubeConfig = CubeConfig + 8; /* Vertex 3 */
                }
                if (Image->GrayLevel(I, J, K+1) > Threshold) {
                    CubeConfig = CubeConfig + 16; /* Vertex 4 */
                }
                if (Image->GrayLevel(I+1, J, K+1) > Threshold) {
                    CubeConfig = CubeConfig + 32; /* Vertex 5 */
                }
                if (Image->GrayLevel(I+1, J+1, K+1) > Threshold) {
                    CubeConfig = CubeConfig + 64; /* Vertex 6 */
                }
                if (Image->GrayLevel(I, J+1, K+1) > Threshold) {

```

```

CubeConfig = CubeConfig + 128; /* Vertex 7 */
}
/* to get the pattern numbers and the number of triangles */
NbSurf = MConf->NbSurfaces(CubeConfig);
if (NbSurf != 0) {
  for (SurfNo=1; SurfNo<=NbSurf; SurfNo++) {
    NbTr = MConf->NbTriangles(CubeConfig, SurfNo);
    NbDes = MConf->NbDesigns(CubeConfig, SurfNo);
    /* to check which design gives the smallest area */
    SurfaceArea = MaxArea;
    for (DesNo=1; DesNo<=NbDes; DesNo++) {
      DesignArea = 0.0;
      for (TrNo=1; TrNo<=NbTr; TrNo++) {
        /* to get the edge numbers within the cube */
        EdgeSummit1 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                         DesNo, TrNo, 1);
        EdgeSummit2 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                         DesNo, TrNo, 2);
        EdgeSummit3 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                         DesNo, TrNo, 3);
        /* to get the coordinates of the summits */
        SurfaceCrossesAt(I, J, K, EdgeSummit1, Image,
                        &XSummit1, &YSummit1, &ZSummit1);
        SurfaceCrossesAt(I, J, K, EdgeSummit2, Image,
                        &XSummit2, &YSummit2, &ZSummit2);
        SurfaceCrossesAt(I, J, K, EdgeSummit3, Image,
                        &XSummit3, &YSummit3, &ZSummit3);
        /* the area of the triangle is the vectorial product */
        /* of two vector sides divided by 2 */
        XVectProd = (YSummit2-YSummit1)*(ZSummit3-ZSummit1) -
                    (YSummit3-YSummit1)*(ZSummit2-ZSummit1);
        YVectProd = (XSummit3-XSummit1)*(ZSummit2-ZSummit1) -
                    (XSummit2-XSummit1)*(ZSummit3-ZSummit1);
        ZVectProd = (XSummit2-XSummit1)*(YSummit3-YSummit1) -
                    (XSummit3-XSummit1)*(YSummit2-YSummit1);
        TriangleArea = sqrt(XVectProd*XVectProd +
                            YVectProd*YVectProd +
                            ZVectProd*ZVectProd) / 2.0;
        DesignArea = DesignArea + TriangleArea;
      }
      if (DesignArea < SurfaceArea) {
        SurfaceArea = DesignArea;
        BestDesign = DesNo;
      }
    }
  }
  /* to cumulate the total area of the isosurface */
  *TotalArea = *TotalArea + SurfaceArea;
  /* to store the best design (smallest area) in the table */
  if (DoWeCreateTables == true) {
    for (TrNo=1; TrNo<=NbTr; TrNo++) {
      /* to get the edge numbers within the cube */
      EdgeSummit1 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                       BestDesign, TrNo, 1);
      EdgeSummit2 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                       BestDesign, TrNo, 2);
      EdgeSummit3 = MConf->CubeEdgeNo(CubeConfig, SurfNo,
                                       BestDesign, TrNo, 3);
      /* to get the edge numbers within the entire image */
      EdgeNoSum1 = GetEdgeNo(I, J, K, EdgeSummit1);
      EdgeNoSum2 = GetEdgeNo(I, J, K, EdgeSummit2);
      EdgeNoSum3 = GetEdgeNo(I, J, K, EdgeSummit3);
      /* to get the summit number in the summit table */
      SumNoSummit1 = MCSummitTable->WhichRankInTable(
                    EdgeNoSum1);
      SumNoSummit2 = MCSummitTable->WhichRankInTable(
                    EdgeNoSum2);
      SumNoSummit3 = MCSummitTable->WhichRankInTable(
                    EdgeNoSum3);
      /* to add the triangle */
    }
  }
}

```

```

MCTriangleTable->AddToTable(SumNoSummit1, SumNoSummit2,
                             SumNoSummit3);
    }
    }
    }
    }
    }
}

void SurfaceCrossesAt(unsigned int I, unsigned int J, unsigned int K,
                     unsigned int EdgeOfCube, Image256GL *Image,
                     double *XSummit, double *YSummit, double *ZSummit)
{
    unsigned int GLStartEdge, GLEndEdge;
    double Intersection;

    switch (EdgeOfCube) {
        case 0: GLStartEdge = Image->GrayLevel(I , J , K );
                GLEndEdge   = Image->GrayLevel(I+1, J , K );
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 0.5 + Intersection) * Vx;
                *YSummit = ((double)J + 0.5) * Vy;
                *ZSummit = ((double)K + 0.5) * Vz;
                break;

        case 1: GLStartEdge = Image->GrayLevel(I+1, J , K );
                GLEndEdge   = Image->GrayLevel(I+1, J+1, K );
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 1.5) * Vx;
                *YSummit = ((double)J + 0.5 + Intersection) * Vy;
                *ZSummit = ((double)K + 0.5) * Vz;
                break;

        case 2: GLStartEdge = Image->GrayLevel(I , J+1, K );
                GLEndEdge   = Image->GrayLevel(I+1, J+1, K );
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 0.5 + Intersection) * Vx;
                *YSummit = ((double)J + 1.5) * Vy;
                *ZSummit = ((double)K + 0.5) * Vz;
                break;

        case 3: GLStartEdge = Image->GrayLevel(I , J , K );
                GLEndEdge   = Image->GrayLevel(I , J+1, K );
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 0.5) * Vx;
                *YSummit = ((double)J + 0.5 + Intersection) * Vy;
                *ZSummit = ((double)K + 0.5) * Vz;
                break;

        case 4: GLStartEdge = Image->GrayLevel(I , J , K+1);
                GLEndEdge   = Image->GrayLevel(I+1, J , K+1);
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 0.5 + Intersection) * Vx;
                *YSummit = ((double)J + 0.5) * Vy;
                *ZSummit = ((double)K + 1.5) * Vz;
                break;

        case 5: GLStartEdge = Image->GrayLevel(I+1, J , K+1);
                GLEndEdge   = Image->GrayLevel(I+1, J+1, K+1);
                Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                         GLEndEdge);

                *XSummit = ((double)I + 1.5) * Vx;
                *YSummit = ((double)J + 0.5 + Intersection) * Vy;
                *ZSummit = ((double)K + 1.5) * Vz;
                break;

        case 6: GLStartEdge = Image->GrayLevel(I , J+1, K+1);
                GLEndEdge   = Image->GrayLevel(I+1, J+1, K+1);
    }
}

```

```

        Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                GLEndEdge);
        *XSummit = ((double)I + 0.5 + Intersection) * Vx;
        *YSummit = ((double)J + 1.5) * Vy;
        *ZSummit = ((double)K + 1.5) * Vz;
        break;
    case 7: GLStartEdge = Image->GrayLevel(I , J , K+1);
           GLEndEdge   = Image->GrayLevel(I , J+1, K+1);
           Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                   GLEndEdge);

           *XSummit = ((double)I + 0.5) * Vx;
           *YSummit = ((double)J + 0.5 + Intersection) * Vy;
           *ZSummit = ((double)K + 1.5) * Vz;
           break;
    case 8: GLStartEdge = Image->GrayLevel(I , J , K );
           GLEndEdge   = Image->GrayLevel(I , J , K+1);
           Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                   GLEndEdge);

           *XSummit = ((double)I + 0.5) * Vx;
           *YSummit = ((double)J + 0.5) * Vy;
           *ZSummit = ((double)K + 0.5 + Intersection) * Vz;
           break;
    case 9: GLStartEdge = Image->GrayLevel(I+1, J , K );
           GLEndEdge   = Image->GrayLevel(I+1, J , K+1);
           Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                   GLEndEdge);

           *XSummit = ((double)I + 1.5) * Vx;
           *YSummit = ((double)J + 0.5) * Vy;
           *ZSummit = ((double)K + 0.5 + Intersection) * Vz;
           break;
    case 10: GLStartEdge = Image->GrayLevel(I+1, J+1, K );
            GLEndEdge   = Image->GrayLevel(I+1, J+1, K+1);
            Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                    GLEndEdge);

            *XSummit = ((double)I + 1.5) * Vx;
            *YSummit = ((double)J + 1.5) * Vy;
            *ZSummit = ((double)K + 0.5 + Intersection) * Vz;
            break;
    case 11: GLStartEdge = Image->GrayLevel(I , J+1, K );
            GLEndEdge   = Image->GrayLevel(I , J+1, K+1);
            Intersection = InterpolatedIntersection(Threshold, GLStartEdge,
                                                    GLEndEdge);

            *XSummit = ((double)I + 0.5) * Vx;
            *YSummit = ((double)J + 1.5) * Vy;
            *ZSummit = ((double)K + 0.5 + Intersection) * Vz;
            break;
    }
}

```

```

unsigned int GetEdgeNo(unsigned int I, unsigned int J, unsigned int K,
                      unsigned int EdgeOfCube)
{
    unsigned int EdgeNo;

    if (EdgeOfCube == 0) {
        EdgeNo = I + (Nx-1)*(J + Ny*K);
    }
    if (EdgeOfCube == 2) {
        EdgeNo = I + (Nx-1)*(J + Ny*K + 1);
    }
    if (EdgeOfCube == 4) {
        EdgeNo = I + (Nx-1)*(J + Ny*(K + 1));
    }
    if (EdgeOfCube == 6) {
        EdgeNo = I + (Nx-1)*(J + Ny*(K + 1) + 1);
    }
    if (EdgeOfCube == 3) {

```

```

    EdgeNo = (Nx-1)*Ny*Nz + I + Nx*(J + (Ny-1)*K);
}
if (EdgeOfCube == 1) {
    EdgeNo = (Nx-1)*Ny*Nz + I + Nx*(J + (Ny-1)*K) + 1;
}
if (EdgeOfCube == 7) {
    EdgeNo = (Nx-1)*Ny*Nz + I + Nx*(J + (Ny-1)*(K+1));
}
if (EdgeOfCube == 5) {
    EdgeNo = (Nx-1)*Ny*Nz + I + Nx*(J + (Ny-1)*(K+1)) + 1;
}
if (EdgeOfCube == 8) {
    EdgeNo = (Nx-1)*Ny*Nz + Nx*(Ny-1)*Nz + I + Nx*(J + Ny*K);
}
if (EdgeOfCube == 9) {
    EdgeNo = (Nx-1)*Ny*Nz + Nx*(Ny-1)*Nz + I + Nx*(J + Ny*K) + 1;
}
if (EdgeOfCube == 11) {
    EdgeNo = (Nx-1)*Ny*Nz + Nx*(Ny-1)*Nz + I + Nx*(J + Ny*K + 1);
}
if (EdgeOfCube == 10) {
    EdgeNo = (Nx-1)*Ny*Nz + Nx*(Ny-1)*Nz + I + Nx*(J + Ny*K + 1) + 1;
}
return(EdgeNo);
}

double InterpolatedIntersection(double Threshold, unsigned int GLStart,
                               unsigned int GLEnd)
{
    double Intersection;

    if ( (GLStart < Threshold && GLEnd < Threshold) ||
         (GLStart > Threshold && GLEnd > Threshold) ) {
        printf("Error: interpolation is impossible for gray levels ");
        printf("%d and %d with threshold %.3f.\n", GLStart, GLEnd, Threshold);
        exit(0);
    }
    Intersection = (Threshold - (double)GLStart) /
        ((double)GLEnd - (double)GLStart);
    return(Intersection);
}

```

APPENDIX F MC CHORD-LENGTH DISTRIBUTION (C++ PROGRAMS)

This appendix contains the C++ program that calculates the chord-length distribution within the Marching Cube representation of the bone-marrow surface that uses the triangles stored in the four tables generated by the program of [Appendix E](#). The program uses some tools of the previous appendices:

- [Appendix B](#):
 - RandomNumber
 - Histogram1D
- [Appendix D](#):
 - TableFormat
 - SummitTable
 - TriangleTable
 - Cube0and255Table
 - CubeOnIsoTable

```

/*****
/*   MCChordLength.cpp
/*
/*   This program calculates the chord length distribution of both the
/*   inside and the outside of the isosurface calculated by the Marching
/*   Cube algorithm.
/*   Input:
/*     - The input image file: ImageName (no extension).
/*     The 4 tables must have been created:
/*       - ImageName.CU1 contains the 0 and 255 cube list
/*       - ImageName.CU2 contains the cubes on the isosurface
/*       - ImageName.TRI contains the triangles of the isosurface
/*       - ImageName.SUM contains the summits of the isosurface
/*     - The dimensions of the image: Nx, Ny, Nz in number of voxels.
/*     - The dimensions of each voxel: Vx, Vy, Vz in centimeters.
/*     - The format for the input files: A for ASCII or B for Binary
/*     - The number of rays to fire
/*     - The radius of the sphere that surrounds the image
/*     - The number of histogram bins
/*     - The step per bin (in cm)
/*   Output:
/*     - The inside chord length distribution in "ImageName_DIR.CLM"
/*     - The outside chord length distribution in "ImageName_DIR.CLB"
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "../CppTools/RandomNumber.h"
#include "../CppTools/Histogram1D.h"
#include "../MCTables/TableFormat.h"
#include "../MCTables/Cube0and255Table.h"
#include "../MCTables/CubeOnIsoTable.h"
#include "../MCTables/TriangleTable.h"

```

```

#include "../MCTables/SummitTable.h"

#define VERY_BIG          99999999.99
#define VERY_SMALL       1.0E-10
#define ABOVE_THRESHOLD   1
#define BELOW_THRESHOLD   0

/* prototypes of local functions */
void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W,
                    bool *RayStartsInImage, bool *RayCrossesImage,
                    Histogram1D *AboveHist, Histogram1D *BelowHist);
double DistanceToEntrancePoint(double X, double Y, double Z,
                              double U, double V, double W,
                              double *EntranceX, double *EntranceY, double *EntranceZ);
void GoToNextCube(double X, double Y, double Z,
                 double U, double V, double W,
                 double *ExitX, double *ExitY, double *ExitZ);
int WhatSideOfSurface(double X, double Y, double Z);
void GoToOtherSide(double X, double Y, double Z,
                  double U, double V, double W,
                  unsigned int CubeNo,
                  double *ExitX, double *ExitY, double *ExitZ);

/* These are the parameters of the program. */
/* Every function needs to access them */
unsigned int Nx, Ny, Nz;
double Vx, Vy, Vz;
unsigned int NbRays;
double SphereRadius;
unsigned int NbBins;
double WidthOfBin;
double ImageMinX, ImageMaxX;
double ImageMinY, ImageMaxY;
double ImageMinZ, ImageMaxZ;
double CubeMinX, CubeMaxX;
double CubeMinY, CubeMaxY;
double CubeMinZ, CubeMaxZ;
/* The image is global so that every function can access it */
Cube0and255Table *MCCube0and255Table;
CubeOnIsoTable *MCCubeOnIsoTable;
SummitTable *MCSummitTable;
TriangleTable *MCTriangleTable;

main( int argc,
      char *argv[] )
{
    char ImageFileName[100];
    char TableFormat;

    /* 1) To get the parameters and read the image file */
    /******
    /* to check the command line */
    if ( argc != 13 ) {
        printf("Error: bad command-line. Try again with command-line: ");
        printf("MCChordLength ImageFileName Nx Ny Nz Vx Vy Vz TableFormat ");
        printf("NbRays SphereRadius Nbins WidthOfBin\n");
        printf(" - ImageFileName: the full path name (without the ");
        printf("extension) of the\n image file from the current ");
        printf("directory. The tables have been\n stored ");
        printf("with the same name as the initial image (IGLimage).\n");
        printf(" - Nx: the number of voxels per row.\n");
        printf(" - Ny: the number of rows per slice.\n");
        printf(" - Nz: the number of slices in the image.\n");
        printf(" - Vx, Vy, and Vz: the voxel size (in cm).\n");
        printf(" - TableFormat: 'A' for tables in ASCII format ");
        printf("or 'B' for tables in binary format.\n");
        printf(" - NbRays: the total number of rays to fire.\n");
        printf(" - SphereRadius: the radius of the sphere around the image.\n");
    }
}

```

```

printf(" - NbBins: the number of bins for the histogram.\n");
printf(" - WidthOfBin: the width of each bin (in cm).\n");
printf("ex: MCChordLength ../../Database/MyImageName ");
printf("200 200 200 0.0088 0.0088 0.0088 A 1000000 1.525 800 0.001\n");
exit(0);
}
/* to get the parameters */
sprintf(ImageFileName, "%s.IGL",argv[1]);
Nx = atoi(argv[2]);
Ny = atoi(argv[3]);
Nz = atoi(argv[4]);
Vx = atof(argv[5]);
Vy = atof(argv[6]);
Vz = atof(argv[7]);
if (argv[8][0] == ASCII_FORMAT || argv[8][0] == BINARY_FORMAT) {
    TableFormat = argv[8][0];
}
else {
    printf("Error: unknown table format: %c.\n", argv[8][0]);
    exit(0);
}
NbRays = atoi(argv[9]);
SphereRadius = atof(argv[10]);
NbBins = atoi(argv[11]);
WidthOfBin = atof(argv[12]);

printf("\nMarching Cube: chord length distribution calculation.\n");
printf("\n\1 Parameters entered:\n");
printf(" Image file: %s.\n",ImageFileName);
printf(" Image size: %3u x %3u x %3u => %u voxels.\n",Nx,Ny,Nz,Nx*Ny*Nz);
printf(" Voxel size: %8.6f x %8.6f x %8.6f cm3.\n",Vx,Vy,Vz);
if (TableFormat == ASCII_FORMAT) {
    printf(" Tables are in ASCII format.\n");
}
else {
    printf(" Tables are in binary format.\n");
}
printf(" Rays fired: %u.\n",NbRays);
printf(" Sphere radius: %8.6f cm.\n",SphereRadius);
printf(" Histogram: %u bins of size %f cm => max length = %f cm.\n",
NbBins, WidthOfBin, NbBins*WidthOfBin);

/* 2) To read the image */
/*****/
printf("\n2) Reading the image.\n");
char Cube0and255FileName[100];
sprintf(Cube0and255FileName, "%s.CU1", argv[1]);
printf(" Reading 0 & 255 cube table from %s. ", Cube0and255FileName);
MCCube0and255Table = new Cube0and255Table(Cube0and255FileName, TableFormat);
printf(" %9u elements read.\n", MCCube0and255Table->NbCubesInTable());
char CubeOnIsoFileName[100];
sprintf(CubeOnIsoFileName, "%s.CU2", argv[1]);
printf(" Reading cube on isosurface table from %s.", CubeOnIsoFileName);
MCCubeOnIsoTable = new CubeOnIsoTable(CubeOnIsoFileName, TableFormat);
printf(" %9u elements read.\n", MCCubeOnIsoTable->NbCubesInTable());
char TriangleFileName[100];
sprintf(TriangleFileName, "%s.TRI", argv[1]);
printf(" Reading triangle table from %s. ", TriangleFileName);
MCTriangleTable = new TriangleTable(TriangleFileName, TableFormat);
printf(" %9u elements read.\n", MCTriangleTable->NbTrianglesInTable());
char SummitFileName[100];
sprintf(SummitFileName, "%s.SUM", argv[1]);
printf(" Reading summit table from %s. ", SummitFileName);
MCSummitTable = new SummitTable(SummitFileName, TableFormat);
printf(" %9u elements read.\n", MCSummitTable->NbSummitsInTable());

/* 3) To create the histograms */
/*****/
printf("\n3) Creating the chord length distribution histograms.\n");

```

```

/* a) to init the histograms */
printf("  Initializing histograms in memory.\n");
Histogram1D *AboveHist;
AboveHist = new Histogram1D(NbBins);
Histogram1D *BelowHist;
BelowHist = new Histogram1D(NbBins);
/* b) to initialize the image size */
double ShiftX, ShiftY, ShiftZ;
ShiftX = Vx * Nx / 2.0;
ShiftY = Vy * Ny / 2.0;
ShiftZ = Vz * Nz / 2.0;
ImageMinX = Vx / 2.0;
ImageMaxX = (Nx-0.5)*Vx;
ImageMinY = Vy / 2.0;
ImageMaxY = (Ny-0.5)*Vy;
ImageMinZ = Vz / 2.0;
ImageMaxZ = (Nz-0.5)*Vz;
/* c) to loop over the number of rays */
printf("  Starting Chord Length measurement.\n");
int RayNumber;
double Rn1, Rn2, Rn3, Rn4;
double Theta, Phi;
double RhoP, PhiP;
double XPPrime, YPPrime, ZPPrime;
double UPPrime, VPPrime, WPPrime;
bool RayStartsInImage;
bool RayCrossesImage;
unsigned int NbRaysStartInImage = 0;
unsigned int NbRaysDoNotIntersect = 0;
/* initialize the random number generator */
InitRandomGenerator(0);
/* for each rays do */
for ( RayNumber=0; RayNumber<NbRays; RayNumber++ ) {
  /* to show the evolution of the process */
  if ((double)(10*RayNumber/NbRays) ==
      10.0*(double)RayNumber/(double)NbRays) {
    printf("      %5.1f %% done.\n", 100.0*RayNumber/NbRays);
    fflush(stdout);
  }
  /* get 4 random numbers Rn1, Rn2, Rn3, and Rn4 */
  Rn1 = GetRandomNumber();
  Rn2 = GetRandomNumber();
  Rn3 = GetRandomNumber();
  Rn4 = GetRandomNumber();
  /* calculate Theta and Phi using Eq. (2) */
  Theta = acos(1.0 - 2.0*Rn1);
  Phi = 2.0 * M_PI * Rn2;
  /* calculate RhoP and PhiP using Eq. (8) */
  RhoP = SphereRadius * sqrt(Rn3);
  PhiP = 2.0 * M_PI * Rn4;
  /* calculate the starting position of the particle using Eq. (11) */
  XPPrime = SphereRadius*sin(Theta)*cos(Phi)
    + RhoP*(sin(Phi)*cos(PhiP) + cos(Theta)*cos(Phi)*sin(PhiP));
  YPPrime = SphereRadius*sin(Theta)*sin(Phi)
    - RhoP*(cos(Phi)*cos(PhiP) - cos(Theta)*sin(Phi)*sin(PhiP));
  ZPPrime = SphereRadius*cos(Theta)
    - RhoP*sin(Theta)*sin(PhiP);
  /* calculate the direction of the particle using Eq. (14) */
  UPPrime = -sin(Theta) * cos(Phi);
  VPPrime = -sin(Theta) * sin(Phi);
  WPPrime = -cos(Theta);
  /* to apply the shift to use the same referential as for the image */
  /* since the image is within the positive part of the space, whereas */
  /* the isotropic field is centered at (0, 0, 0). */
  XPPrime = XPPrime + ShiftX;
  YPPrime = YPPrime + ShiftY;
  ZPPrime = ZPPrime + ShiftZ;
  /* treat the interaction of the particle with the object */
  TreatInteraction(XPPrime, YPPrime, ZPPrime, UPPrime, VPPrime, WPPrime,

```

```

        &RayStartsInImage, &RayCrossesImage,
        AboveHist, BelowHist);
    if (RayStartsInImage == true) {
        NbRaysStartInImage += 1;
    }
    else {
        if (RayCrossesImage == false) {
            NbRaysDoNotIntersect += 1;
        }
    }
}
printf("          %5.1f %% done.\n", 100.0);
printf(" Nb rays started within the image:          %7d => %5.2f %%\n",
        NbRaysStartInImage, 100.0 * NbRaysStartInImage / NbRays);
printf("          (if not 0 increase the radius of the sphere)\n");
printf(" Nb rays that do not intersect the image: %7d => %5.2f %%\n",
        NbRaysDoNotIntersect, 100.0 * NbRaysDoNotIntersect / NbRays);
printf("          (if too big, decrease the radius of the sphere)\n");

/* d) to store the histogram on the disk */
char AboveHistFileName[100];
sprintf(AboveHistFileName, "%s_DIR.CLM", argv[1]);
printf(" Saving file %s on disk\n", AboveHistFileName);
AboveHist->StoreToDisk(AboveHistFileName);
char BelowHistFileName[100];
sprintf(BelowHistFileName, "%s_DIR.CLB", argv[1]);
printf(" Saving file %s on disk\n", BelowHistFileName);
BelowHist->StoreToDisk(BelowHistFileName);

/* e) to free the memory */
delete MCCube0and255Table;
delete MCCubeOnIsoTable;
delete MCSummitTable;
delete MCTriangleTable;
delete AboveHist;
delete BelowHist;
}

void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W,
                    bool *RayStartsInImage, bool *RayCrossesImage,
                    Histogram1D *AboveHist, Histogram1D *BelowHist)
{
    double EntranceDistance;
    double EntranceX, EntranceY, EntranceZ;
    double CurrentX, CurrentY, CurrentZ;
    double ChordLength;
    int CurrentSide;
    double StartChordX, StartChordY, StartChordZ;
    bool FirstChord;

    /* test if starting point inside the image */
    if ( (X > ImageMinX) && (X < ImageMaxX) &&
        (Y > ImageMinY) && (Y < ImageMaxY) &&
        (Z > ImageMinZ) && (Z < ImageMaxZ) ) {
        *RayStartsInImage = true;
    }
    else {
        *RayStartsInImage = false;
        /* to get the entrance point of the ray */
        EntranceDistance = DistanceToEntrancePoint(X, Y, Z, U, V, W,
            &EntranceX, &EntranceY, &EntranceZ);
        if (EntranceDistance < 0.0) { /* we missed the image */
            *RayCrossesImage = false;
        }
        else {
            *RayCrossesImage = true;
            /* to know on which side we are at the begining. after, we will */
            /* switch between inside and outside every time we cross the */

```

```

/* isosurface. */
CurrentSide = WhatSideOfSurface(EntranceX, EntranceY, EntranceZ);
StartChordX=EntranceX; StartChordY=EntranceY; StartChordZ=EntranceZ;
FirstChord = true; /* we do not want to record the first chord */
/* we start from the entrance point, and marche cube after cube */
/* until we leave the image. */
CurrentX=EntranceX; CurrentY=EntranceY; CurrentZ=EntranceZ;
do {
    /* we need to know the limits of the cube to check when we leave */
    int I, J, K;
    I = (CurrentX / Vx) - 0.5;
    CubeMinX = Vx*(I + 0.5);
    CubeMaxX = CubeMinX + Vx;
    J = (CurrentY / Vy) - 0.5;
    CubeMinY = Vy*(J + 0.5);
    CubeMaxY = CubeMinY + Vy;
    K = (CurrentZ / Vz) - 0.5;
    CubeMinZ = Vz*(K + 0.5);
    CubeMaxZ = CubeMinZ + Vz;
    unsigned int CubeNo = (K*(Ny-1) + J)*(Nx-1) + I;

    /* to test if the cube is on the isosurface */
    unsigned char Config;
    unsigned char NbTr;
    unsigned int FirstTr;
    MCCubeOnIsoTable->GetCube(CubeNo, &Config, &NbTr, &FirstTr);
    if (NbTr != 0) { /* we are on isosurface */
        /* We need to explore this cube. Since we can cross several */
        /* time the surface within the same cube, we need to loop */
        /* as long as we remain in the same cube. */

        /* loop as long as within the limit of the current cube */
        do {
            GoToOtherSide(CurrentX, CurrentY, CurrentZ, U, V, W,
                CubeNo, &CurrentX, &CurrentY, &CurrentZ);
            /* we change the medium and record the chord length */
            /* only if the new position is still in the same cube */
            if ( (CurrentX > CubeMinX) && (CurrentX < CubeMaxX) &&
                (CurrentY > CubeMinY) && (CurrentY < CubeMaxY) &&
                (CurrentZ > CubeMinZ) && (CurrentZ < CubeMaxZ) ) {
                if (FirstChord == true) {
                    /* we don't record the first chord since it starts */
                    /* outside the image */
                    FirstChord = false;
                }
                else {
                    ChordLength = sqrt((StartChordX - CurrentX)*
                        (StartChordX - CurrentX) +
                        (StartChordY - CurrentY)*
                        (StartChordY - CurrentY) +
                        (StartChordZ - CurrentZ)*
                        (StartChordZ - CurrentZ));
                    if (CurrentSide == ABOVE_THRESHOLD) {
                        AboveHist->AddValue(ChordLength / WidthOfBin);
                    }
                    else {
                        BelowHist->AddValue(ChordLength / WidthOfBin);
                    }
                }
            }
            /* a new chord starts */
            StartChordX = CurrentX;
            StartChordY = CurrentY;
            StartChordZ = CurrentZ;
            /* we cross the surface => we switch to the other medium */
            if (CurrentSide == ABOVE_THRESHOLD) {
                CurrentSide = BELOW_THRESHOLD;
            }
            else {
                CurrentSide = ABOVE_THRESHOLD;
            }
        }
    }
}

```

```

    }
    }
    while ( (CurrentX > CubeMinX) && (CurrentX < CubeMaxX) &&
            (CurrentY > CubeMinY) && (CurrentY < CubeMaxY) &&
            (CurrentZ > CubeMinZ) && (CurrentZ < CubeMaxZ) );
    }
    else {
        /* the cube is not on the surface, thus we just */
        /* need to go across the current cube. */
        GoToNextCube(CurrentX, CurrentY, CurrentZ, U, V, W,
                    &CurrentX, &CurrentY, &CurrentZ);
    }
}
while ( (CurrentX > ImageMinX) && (CurrentX < ImageMaxX) &&
        (CurrentY > ImageMinY) && (CurrentY < ImageMaxY) &&
        (CurrentZ > ImageMinZ) && (CurrentZ < ImageMaxZ) );
}
}
}

double DistanceToEntrancePoint(double X, double Y, double Z,
                               double U, double V, double W,
                               double *EntranceX, double *EntranceY, double *EntranceZ)
{
    double Distance, ShortestDistance;

    ShortestDistance = VERY_BIG;
    Distance = (ImageMinX - X) / U;
    if (Distance > 0.0) {
        *EntranceY = Y + V*Distance;
        *EntranceZ = Z + W*Distance;
        if ( *EntranceY > ImageMinY && *EntranceY < ImageMaxY &&
             *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ ) {
            if (Distance < ShortestDistance) {
                ShortestDistance = Distance;
            }
        }
    }
    Distance = (ImageMaxX - X) / U;
    if (Distance > 0.0) {
        *EntranceY = Y + V*Distance;
        *EntranceZ = Z + W*Distance;
        if ( *EntranceY > ImageMinY && *EntranceY < ImageMaxY &&
             *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ ) {
            if (Distance < ShortestDistance) {
                ShortestDistance = Distance;
            }
        }
    }
    Distance = (ImageMinY - Y) / V;
    if (Distance > 0.0) {
        *EntranceZ = Z + W*Distance;
        *EntranceX = X + U*Distance;
        if ( *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ &&
             *EntranceX > ImageMinX && *EntranceX < ImageMaxX ) {
            if (Distance < ShortestDistance) {
                ShortestDistance = Distance;
            }
        }
    }
    Distance = (ImageMaxY - Y) / V;
    if (Distance > 0.0) {
        *EntranceZ = Z + W*Distance;
        *EntranceX = X + U*Distance;
        if ( *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ &&
             *EntranceX > ImageMinX && *EntranceX < ImageMaxX ) {
            if (Distance < ShortestDistance) {
                ShortestDistance = Distance;
            }
        }
    }
}

```

```

    }
}
Distance = (ImageMinZ - Z) / W;
if (Distance > 0.0) {
    *EntranceX = X + U*Distance;
    *EntranceY = Y + V*Distance;
    if ( *EntranceX > ImageMinX && *EntranceX < ImageMaxX &&
        *EntranceY > ImageMinY && *EntranceY < ImageMaxY ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
Distance = (ImageMaxZ - Z) / W;
if (Distance > 0.0) {
    *EntranceX = X + U*Distance;
    *EntranceY = Y + V*Distance;
    if ( *EntranceX > ImageMinX && *EntranceX < ImageMaxX &&
        *EntranceY > ImageMinY && *EntranceY < ImageMaxY ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
ShortestDistance += VERY_SMALL;
/* to check if an intersection point has been found */
if (ShortestDistance > (VERY_BIG / 2.0)) {
    return(-1.0); /* means that we missed the image */
}
else {
    *EntranceX = X + U*ShortestDistance;
    *EntranceY = Y + V*ShortestDistance;
    *EntranceZ = Z + W*ShortestDistance;
    /* in case the += VERY_SMALL put the entrance point outside the image. */
    /* This can happens at the corners */
    if ( (*EntranceX < ImageMinX) || (*EntranceX > ImageMaxX) ||
        (*EntranceY < ImageMinY) || (*EntranceY > ImageMaxY) ||
        (*EntranceZ < ImageMinZ) || (*EntranceZ > ImageMaxZ) ) {
        return(-1.0); /* means that we are to close to a corner, and */
        /*assume that we miss the image */
    }
    else {
        return(ShortestDistance);
    }
}
}

void GoToNextCube(double X, double Y, double Z,
                 double U, double V, double W,
                 double *ExitX, double *ExitY, double *ExitZ)
{
    double Distance, ShortestDistance;

    /* get the shortest distance to go out */
    ShortestDistance = VERY_BIG;
    Distance = (CubeMinX - X) / U;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxX - X) / U;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMinY - Y) / V;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxY - Y) / V;

```

```

    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMinZ - Z) / W;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxZ - Z) / W;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    ShortestDistance += VERY_SMALL;
    /* to get the exit point */
    *ExitX = X + U*ShortestDistance;
    *ExitY = Y + V*ShortestDistance;
    *ExitZ = Z + W*ShortestDistance;
}

int WhatSideOfSurface(double X, double Y, double Z)
{
    unsigned int I, J, K;
    unsigned int CubeNo;
    unsigned char Config;
    unsigned char NbTr;
    unsigned int FirstTr;

    /* to get the cube number */
    I = (X / Vx) + 0.5;
    J = (Y / Vy) + 0.5;
    K = (Z / Vz) + 0.5;
    CubeNo = ((K-1)*(Ny-1) + (J-1))*(Nx-1) + (I-1);
    /* to get the cube configuration */
    MCCubeOnIsoTable->GetCube(CubeNo, &Config, &NbTr, &FirstTr);
    if (NbTr == 0) { /* not on isosurface */
        Config = MCCube0and255Table->WhichConfig(CubeNo);
        if (Config == 0) {
            return(BELOW_THRESHOLD);
        }
        else {
            return(ABOVE_THRESHOLD);
        }
    }
    else { /* on isosurface, need to check on which side */
        /* A point is on the same side of the isosurface than the 0,0,0 corner */
        /* of the cube if one crosses an even number of triangles on the way */
        /* from the point to the 0,0,0 corner of the cube. */
        /* to define the data we need for that */
        unsigned int TrNo;
        unsigned int Summit1, Summit2, Summit3; /* summits of the triangles */
        double XSummit1, YSummit1, ZSummit1;
        double XSummit2, YSummit2, ZSummit2;
        double XSummit3, YSummit3, ZSummit3;
        double XNormal, YNormal, ZNormal; /* normal vectors of the triangles */
        double TriangleArea; /* the surface area of the triangles */
        float XSummit, YSummit, ZSummit;
        double XCorner, YCorner, ZCorner;
        /* to get the coordinates of the 0,0,0 corner of the Marching Cube */
        XCorner = (I - 0.5) * Vx;
        YCorner = (J - 0.5) * Vy;
        ZCorner = (K - 0.5) * Vz;
        /* to get the direction from the point to the 0,0,0 corner */
        double DistanceToCorner;
        DistanceToCorner = sqrt((XCorner-X)*(XCorner-X) +
                               (YCorner-Y)*(YCorner-Y) +
                               (ZCorner-Z)*(ZCorner-Z));

        double U, V, W;
        U = (XCorner-X) / DistanceToCorner;
        V = (YCorner-Y) / DistanceToCorner;
        W = (ZCorner-Z) / DistanceToCorner;
    }
}

```

```

/* to count how many triangles are crossed from the point to */
/* the 0,0,0 corner */
unsigned int NbTrianglesCrossed = 0;
for (TrNo=0; TrNo<NbTr; TrNo++) {
    /* to get the info of the triangle */
    MCTriangleTable->GetTriangle(FirstTr + TrNo,
                                &Summit1, &Summit2, &Summit3);

    /* The summit coordinates */
    MCSummitTable->GetSummit(Summit1, &XSummit, &YSummit, &ZSummit);
    XSummit1 = XSummit; YSummit1 = YSummit; ZSummit1 = ZSummit;
    MCSummitTable->GetSummit(Summit2, &XSummit, &YSummit, &ZSummit);
    XSummit2 = XSummit; YSummit2 = YSummit; ZSummit2 = ZSummit;
    MCSummitTable->GetSummit(Summit3, &XSummit, &YSummit, &ZSummit);
    XSummit3 = XSummit; YSummit3 = YSummit; ZSummit3 = ZSummit;
    /* The components of the normal vectors (vector product) */
    XNormal = (YSummit2 - YSummit1) * (ZSummit3 - ZSummit1) -
              (ZSummit2 - ZSummit1) * (YSummit3 - YSummit1);
    YNormal = (ZSummit2 - ZSummit1) * (XSummit3 - XSummit1) -
              (XSummit2 - XSummit1) * (ZSummit3 - ZSummit1);
    ZNormal = (XSummit2 - XSummit1) * (YSummit3 - YSummit1) -
              (YSummit2 - YSummit1) * (XSummit3 - XSummit1);
    /* surface area of the triangle (half the module of vector product) */
    TriangleArea = 0.5 * sqrt(XNormal*XNormal + YNormal*YNormal +
                              ZNormal*ZNormal);

    /* to get the distance to the plan */
    if (U*XNormal + V*YNormal + W*ZNormal == 0.0) {
        /* The direction is parallel to the plan => no intersection */
    }
    else {
        double DistanceToPlan;
        /* it is given by putting the parametric equation of the */
        /* trajectory into the equation of the plane */
        /* plane: Nx*(Xp - X1) + Ny*(Yp - Y1) + Nz*(Zp - Z1) = 0 */
        /* and Xp = X + U*Dis; Yp = Y + V*Dis; Zp = Z + W*Dis */
        /* it comes: */
        DistanceToPlan = ((XSummit1 - X)*XNormal + (YSummit1 - Y)*YNormal +
                        (ZSummit1 - Z)*ZNormal) / (U*XNormal + V*YNormal + W*ZNormal);
        if (DistanceToPlan < 0.0) {
            /* the direction is fleeing the plan => no intersection */
        }
        else {
            /* to get the coordinates of the intersection point */
            double XIntersect, YIntersect, ZIntersect;
            XIntersect = X + U*DistanceToPlan;
            YIntersect = Y + V*DistanceToPlan;
            ZIntersect = Z + W*DistanceToPlan;
            /* to check if the intersection is within the triangle */
            /* It is within the triangle if the sum of the surface */
            /* areas of any 2 of the 3 subtriangles it generates is */
            /* less than the surface area of the entire triangle. */
            double XCrossProduct, YCrossProduct, ZCrossProduct;
            double SubTr1Area, SubTr2Area, SubTr3Area;
            /* first subtriangle */
            XCrossProduct = (YSummit1-YIntersect)*(ZSummit2-ZIntersect) -
                          (ZSummit1-ZIntersect)*(YSummit2-YIntersect);
            YCrossProduct = (ZSummit1-ZIntersect)*(XSummit2-XIntersect) -
                          (XSummit1-XIntersect)*(ZSummit2-ZIntersect);
            ZCrossProduct = (XSummit1-XIntersect)*(YSummit2-YIntersect) -
                          (YSummit1-YIntersect)*(XSummit2-XIntersect);
            SubTr1Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
                                   YCrossProduct*YCrossProduct +
                                   ZCrossProduct*ZCrossProduct);

            /* second subtriangle */
            XCrossProduct = (YSummit1-YIntersect)*(ZSummit3-ZIntersect) -
                          (ZSummit1-ZIntersect)*(YSummit3-YIntersect);
            YCrossProduct = (ZSummit1-ZIntersect)*(XSummit3-XIntersect) -
                          (XSummit1-XIntersect)*(ZSummit3-ZIntersect);
            ZCrossProduct = (XSummit1-XIntersect)*(YSummit3-YIntersect) -

```

```

        (YSummit1-YIntersect)*(XSummit3-XIntersect);
SubTr2Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
        YCrossProduct*YCrossProduct +
        ZCrossProduct*ZCrossProduct);
/* third subtriangle */
XCrossProduct = (YSummit2-YIntersect)*(ZSummit3-ZIntersect) -
        (ZSummit2-ZIntersect)*(YSummit3-YIntersect);
YCrossProduct = (ZSummit2-ZIntersect)*(XSummit3-XIntersect) -
        (XSummit2-XIntersect)*(ZSummit3-ZIntersect);
ZCrossProduct = (XSummit2-XIntersect)*(YSummit3-YIntersect) -
        (YSummit2-YIntersect)*(XSummit3-XIntersect);
SubTr3Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
        YCrossProduct*YCrossProduct +
        ZCrossProduct*ZCrossProduct);
if ((SubTr1Area + SubTr2Area) < TriangleArea &&
    (SubTr2Area + SubTr3Area) < TriangleArea &&
    (SubTr3Area + SubTr1Area) < TriangleArea) {
    /* the straight line really intersect the plan inside */
    /* the triangle */
    NbTrianglesCrossed += 1;
}
}
}
}
/* to compare the number of triangles crossed with the medium at */
/* the 0,0,0 corner of the Marching Cube */
if ((NbTrianglesCrossed % 2) == (Config % 2)) {
    return(BELOW_THRESHOLD);
}
else {
    return(ABOVE_THRESHOLD);
}
}
}

void GoToOtherSide(double X, double Y, double Z,
        double U, double V, double W,
        unsigned int CubeNo,
        double *ExitX, double *ExitY, double *ExitZ)
{
    unsigned char Config;
    unsigned char NbTr;
    unsigned int FirstTr;

    /* to get the cube configuration */
    MCCubeOnIsoTable->GetCube(CubeNo, &Config, &NbTr, &FirstTr);
    /* Need to get the shortest distance from the triangles */
    /* to define the data we need for that */
    unsigned int TrNo;
    unsigned int Summit1, Summit2, Summit3; /* summits of the triangles */
    double XSummit1, YSummit1, ZSummit1;
    double XSummit2, YSummit2, ZSummit2;
    double XSummit3, YSummit3, ZSummit3;
    double XNormal, YNormal, ZNormal; /* normal vectors of the triangles */
    double TriangleArea; /* the surface area of the triangles */
    float XSummit, YSummit, ZSummit;
    /* to check with each triangle */
    double Distance, ShortestDistance;
    ShortestDistance = VERY_BIG;
    for (TrNo=0; TrNo<NbTr; TrNo++) {
        /* to get the info of the triangle */
        MCTriangleTable->GetTriangle(FirstTr + TrNo,
                &Summit1, &Summit2, &Summit3);
        /* The summit coordinates */
        MCSummitTable->GetSummit(Summit1, &XSummit, &YSummit, &ZSummit);
        XSummit1 = XSummit; YSummit1 = YSummit; ZSummit1 = ZSummit;
        MCSummitTable->GetSummit(Summit2, &XSummit, &YSummit, &ZSummit);
        XSummit2 = XSummit; YSummit2 = YSummit; ZSummit2 = ZSummit;
        MCSummitTable->GetSummit(Summit3, &XSummit, &YSummit, &ZSummit);
    }
}

```

```

XSummit3 = XSummit; YSummit3 = YSummit; ZSummit3 = ZSummit;
/* The components of the normal vectors (vector product) */
XNormal = (YSummit2 - YSummit1) * (ZSummit3 - ZSummit1) -
          (ZSummit2 - ZSummit1) * (YSummit3 - YSummit1);
YNormal = (ZSummit2 - ZSummit1) * (XSummit3 - XSummit1) -
          (XSummit2 - XSummit1) * (ZSummit3 - ZSummit1);
ZNormal = (XSummit2 - XSummit1) * (YSummit3 - YSummit1) -
          (YSummit2 - YSummit1) * (XSummit3 - XSummit1);
/* surface area of the triangle (half the module of vector product) */
TriangleArea = 0.5 * sqrt(XNormal*XNormal + YNormal*YNormal +
                        ZNormal*ZNormal);
/* to get the distance to the plan */
if (U*XNormal + V*YNormal + W*ZNormal == 0.0) {
    /* The direction is parallel to the plan => no intersection */
}
else {
    /* it is given by putting the parametric equation of the */
    /* trajectory into the equation of the plane */
    /* plane: Nx*(Xp - X1) + Ny*(Yp - Y1) + Nz*(Zp - Z1) = 0 */
    /* and Xp = X + U*Dis; Yp = Y + V*Dis; Zp = Z + W*Dis */
    /* it comes: */
    Distance = ((XSummit1 - X)*XNormal + (YSummit1 - Y)*YNormal +
               (ZSummit1 - Z)*ZNormal) / (U*XNormal + V*YNormal + W*ZNormal);
    if (Distance < 0.0) {
        /* the direction is fleeing the plan => no intersection */
    }
    else {
        /* to get the coordinates of the intersection point */
        double XIntersect, YIntersect, ZIntersect;
        XIntersect = X + U*Distance;
        YIntersect = Y + V*Distance;
        ZIntersect = Z + W*Distance;
        /* to check if the intersection is within the triangle */
        /* It is within the triangle if the sum of the surface */
        /* areas of any 2 of the 3 subtriangles it generates is */
        /* less than the surface area of the entire triangle. */
        double XCrossProduct, YCrossProduct, ZCrossProduct;
        double SubTr1Area, SubTr2Area, SubTr3Area;
        /* first subtriangle */
        XCrossProduct = (YSummit1-YIntersect)*(ZSummit2-ZIntersect) -
                       (ZSummit1-ZIntersect)*(YSummit2-YIntersect);
        YCrossProduct = (ZSummit1-ZIntersect)*(XSummit2-XIntersect) -
                       (XSummit1-XIntersect)*(ZSummit2-ZIntersect);
        ZCrossProduct = (XSummit1-XIntersect)*(YSummit2-YIntersect) -
                       (YSummit1-YIntersect)*(XSummit2-XIntersect);
        SubTr1Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
                               YCrossProduct*YCrossProduct +
                               ZCrossProduct*ZCrossProduct);
        /* second subtriangle */
        XCrossProduct = (YSummit1-YIntersect)*(ZSummit3-ZIntersect) -
                       (ZSummit1-ZIntersect)*(YSummit3-YIntersect);
        YCrossProduct = (ZSummit1-ZIntersect)*(XSummit3-XIntersect) -
                       (XSummit1-XIntersect)*(ZSummit3-ZIntersect);
        ZCrossProduct = (XSummit1-XIntersect)*(YSummit3-YIntersect) -
                       (YSummit1-YIntersect)*(XSummit3-XIntersect);
        SubTr2Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
                               YCrossProduct*YCrossProduct +
                               ZCrossProduct*ZCrossProduct);
        /* third subtriangle */
        XCrossProduct = (YSummit2-YIntersect)*(ZSummit3-ZIntersect) -
                       (ZSummit2-ZIntersect)*(YSummit3-YIntersect);
        YCrossProduct = (ZSummit2-ZIntersect)*(XSummit3-XIntersect) -
                       (XSummit2-XIntersect)*(ZSummit3-ZIntersect);
        ZCrossProduct = (XSummit2-XIntersect)*(YSummit3-YIntersect) -
                       (YSummit2-YIntersect)*(XSummit3-XIntersect);
        SubTr3Area = 0.5 * sqrt(XCrossProduct*XCrossProduct +
                               YCrossProduct*YCrossProduct +
                               ZCrossProduct*ZCrossProduct);
        if ((SubTr1Area + SubTr2Area) < TriangleArea &&

```

```

        (SubTr2Area + SubTr3Area) < TriangleArea &&
        (SubTr3Area + SubTr1Area) < TriangleArea) {
    /* The intersection is within the limit of the triangle */
    /* one can check this one. */
    if (Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
}
}
}
}
ShortestDistance += VERY_SMALL;
if (ShortestDistance > (VERY_BIG / 2.0)) {
    /* no triangle is crossed, need to go to the next cube */
    GoToNextCube(X, Y, Z, U, V, W, ExitX, ExitY, ExitZ);
}
else {
    /* get the exit point */
    *ExitX = X + U*ShortestDistance;
    *ExitY = Y + V*ShortestDistance;
    *ExitZ = Z + W*ShortestDistance;
}
}
}
}

```

APPENDIX G HMC CHORD-LENGTH DISTRIBUTION (C++ PROGRAMS)

This appendix contains the C++ program that calculates the chord-length distribution within the Hyperboloid Marching Cube representation of the bone-marrow surface. The program uses some tools of the previous appendices:

- [Appendix B](#):
 - RandomNumber
 - Histogram1D
 - Image256GL
 - Equations

```

/*****
/*   TLChordLength.cpp
/*
/*   This program calculates the chord length distribution of both the
/*   'above the threshold' and the 'below the threshold' of a 3D object
/*   using the trilinear interpolation adaptation of the Marching Cubes
/*   algorithm.
/*   Input:
/*     - The input image file: ImageName (no extension).
/*     - The dimensions of the image: Nx, Ny, Nz in number of voxels.
/*     - The dimensions of each voxel: Vx, Vy, Vz in centimeters.
/*     - The threshold to separate the two media.
/*     - The number of rays to fire
/*     - The radius of the sphere that surrounds the image
/*     - The number of histogram bins
/*     - The step per bin (in cm)
/*   Output:
/*     - The above chord length distribution in "ImageName_TL.CLM"
/*     - The below chord length distribution in "ImageName_TL.CLB"
/*
*****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "../CppTools/RandomNumber.h"
#include "../CppTools/Histogram1D.h"
#include "../CppTools/Image256GL.h"
#include "../CppTools/Equations.h"

#define VERY_BIG      99999999.99
#define VERY_SMALL    1.0E-08
#define BELOW_THRESHOLD  0
#define ABOVE_THRESHOLD  1

/* prototypes of local functions */
void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W,
                    bool *RayStartsInImage, bool *RayCrossesImage,
                    Histogram1D *BelowHist, Histogram1D *AboveHist);
double DistanceToEntrancePoint(double X, double Y, double Z,
                              double U, double V, double W,
                              double *EntranceX, double *EntranceY, double *EntranceZ);

```

```

int WhatSideOfSurface(double X, double Y, double Z);
bool CubeOnIsosurface(unsigned int I, unsigned int J, unsigned int K);
void GoToNextCube(double X, double Y, double Z,
                 double U, double V, double W,
                 double *ExitX, double *ExitY, double *ExitZ);
void GoToOtherSide(double X, double Y, double Z,
                  double U, double V, double W,
                  unsigned int I, unsigned int J, unsigned int K,
                  int CurrentSide,
                  double *ExitX, double *ExitY, double *ExitZ);

/* These are the parameters of the program. */
/* Every function needs to access them */
unsigned int Nx, Ny, Nz;
double Vx, Vy, Vz;
double Threshold;
unsigned int NbRays;
double SphereRadius;
unsigned int NbBins;
double WidthOfBin;
double ImageMinX, ImageMaxX;
double ImageMinY, ImageMaxY;
double ImageMinZ, ImageMaxZ;
double CubeMinX, CubeMaxX;
double CubeMinY, CubeMaxY;
double CubeMinZ, CubeMaxZ;
/* The image is global so that every function can access it */
Image256GL *Image;

main( int argc,
      char *argv[] )
{
    char ImageFileName[100];

    /* 1) To get the parameters and read the image file */
    /******
    /* to check the command line */
    if ( argc != 13 ) {
        printf("Error: bad command-line. Try again with command-line: ");
        printf("TLChordLength ImageFileName Nx Ny Nz Vx Vy Vz Threshold ");
        printf("NbRays SphereRadius NbBins WidthOfBin\n");
        printf(" - ImageFileName: the full path name (without the ");
        printf("extension) of the image file from the current ");
        printf("directory.\n");
        printf(" - Nx: the number of voxels per row.\n");
        printf(" - Ny: the number of rows per slice.\n");
        printf(" - Nz: the number of slices in the image.\n");
        printf(" - Vx, Vy, and Vz: the voxel size (in cm).\n");
        printf(" - Threshold: the threshold value that separates ");
        printf("the inside from the outside.\n");
        printf(" - NbRays: the total number of rays to fire.\n");
        printf(" - SphereRadius: the radius of the sphere around the image.\n");
        printf(" - NbBins: the number of bins for the histogram.\n");
        printf(" - WidthOfBin: the width of each bin (in cm).\n");
        printf("ex: TLChordLength ../../Database/MyImageName ");
        printf("200 200 200 0.0088 0.0088 0.0088 127.5 10000 1.525 800 0.001\n");
        exit(0);
    }

    /* to get the parameters */
    sprintf(ImageFileName, "%s.IGL", argv[1]);
    Nx = atoi(argv[2]);
    Ny = atoi(argv[3]);
    Nz = atoi(argv[4]);
    Vx = atof(argv[5]);
    Vy = atof(argv[6]);
    Vz = atof(argv[7]);
    Threshold = atof(argv[8]);
    NbRays = atoi(argv[9]);
    SphereRadius = atof(argv[10]);

```

```

NbBins = atoi(argv[11]);
WidthOfBin = atof(argv[12]);

printf("\nTrilinear interpolation: chord length dist. calculation.\n");
printf("\n\l) Parameters entered:\n");
printf("  Image file: %s.\n", ImageFileName);
printf("  Image size: %3u x %3u x %3u => %u voxels.\n", Nx, Ny, Nz, Nx*Ny*Nz);
printf("  Voxel size: %8.6f x %8.6f x %8.6f cm3.\n", Vx, Vy, Vz);
printf("  Threshold: %8.4f.\n", Threshold);
printf("  Rays fired: %u.\n", NbRays);
printf("  Sphere radius: %8.6f cm.\n", SphereRadius);
printf("  Histogram: %u bins of size %f cm => max length = %f cm.\n",
       NbBins, WidthOfBin, NbBins*WidthOfBin);

/* 2) To read the image */
/*****/
printf("\n2) Reading the image.\n");
char GLImageFileName[100];
sprintf(GLImageFileName, "%s.IGL", argv[1]);
printf("  Reading the image file from %s.\n", GLImageFileName);
Image = new Image256GL(Nx, Ny, Nz);
Image->LoadFromFile(GLImageFileName);

/* 3) To create the histograms */
/*****/
printf("\n3) Creating the chord length distribution histograms.\n");
/* a) to init the histograms */
printf("  Initializing histograms in memory.\n");
char BelowHistFileName[100];
sprintf(BelowHistFileName, "%s_TL.CLB", argv[1]);
Histogram1D *BelowHist;
BelowHist = new Histogram1D(NbBins);
char AboveHistFileName[100];
sprintf(AboveHistFileName, "%s_TL.CLM", argv[1]);
Histogram1D *AboveHist;
AboveHist = new Histogram1D(NbBins);
/* b) to initialize the image size */
double ShiftX, ShiftY, ShiftZ;
ShiftX = Vx * Nx / 2.0;
ShiftY = Vy * Ny / 2.0;
ShiftZ = Vz * Nz / 2.0;
ImageMinX = Vx / 2.0;
ImageMaxX = (Nx-0.5)*Vx;
ImageMinY = Vy / 2.0;
ImageMaxY = (Ny-0.5)*Vy;
ImageMinZ = Vz / 2.0;
ImageMaxZ = (Nz-0.5)*Vz;
/* c) to loop over the number of rays */
printf("  Starting Chord Length measurement.\n");
int RayNumber;
double Rn1, Rn2, Rn3, Rn4;
double Theta, Phi;
double RhoP, PhiP;
double XPPRime, YPPRime, ZPPRime;
double UPPRime, VPPRime, WPPRime;
bool RayStartsInImage;
bool RayCrossesImage;
unsigned int NbRaysStartInImage = 0;
unsigned int NbRaysDoNotIntersect = 0;
/* initialize the random number generator */
InitRandomGenerator(0);
/* for each rays do */
for ( RayNumber=0; RayNumber<NbRays; RayNumber++ ) {
  /* to show the evolution of the process */
  if ( ((double)(10*RayNumber/NbRays) ==
        10.0*(double)RayNumber/(double)NbRays) ) {
    printf("          %5.1f %% done.\n", 100.0*RayNumber/NbRays);
    fflush(stdout);
  }
}

```

```

/* get 4 random numbers Rn1, Rn2, Rn3, and Rn4 */
Rn1 = GetRandomNumber();
Rn2 = GetRandomNumber();
Rn3 = GetRandomNumber();
Rn4 = GetRandomNumber();
/* calculate Theta and Phi using Eq. (2) */
Theta = acos(1.0 - 2.0*Rn1);
Phi = 2.0 * M_PI * Rn2;
/* calculate RhoP and PhiP using Eq. (8) */
RhoP = SphereRadius * sqrt(Rn3);
PhiP = 2.0 * M_PI * Rn4;
/* calculate the starting position of the particle using Eq. (11) */
XPPRime = SphereRadius*sin(Theta)*cos(Phi)
          + RhoP*(sin(Phi)*cos(PhiP) + cos(Theta)*cos(Phi)*sin(PhiP));
YPPRime = SphereRadius*sin(Theta)*sin(Phi)
          - RhoP*(cos(Phi)*cos(PhiP) - cos(Theta)*sin(Phi)*sin(PhiP));
ZPPRime = SphereRadius*cos(Theta)
          - RhoP*sin(Theta)*sin(PhiP);
/* calculate the direction of the particle using Eq. (14) */
UPPRime = -sin(Theta) * cos(Phi);
VPPRime = -sin(Theta) * sin(Phi);
WPPRime = -cos(Theta);
/* to apply the shift to use the same referential as for the image */
/* since the image is within the positive part of the space, whereas */
/* the isotropic field is centered at (0, 0, 0). */
XPPRime = XPPRime + ShiftX;
YPPRime = YPPRime + ShiftY;
ZPPRime = ZPPRime + ShiftZ;
/* treat the interaction of the particle with the object */
TreatInteraction(XPPRime, YPPRime, ZPPRime, UPPRime, VPPRime, WPPRime,
                &RayStartsInImage, &RayCrossesImage,
                BelowHist, AboveHist);
if (RayStartsInImage == true) {
    NbRaysStartInImage += 1;
}
else {
    if (RayCrossesImage == false) {
        NbRaysDoNotIntersect += 1;
    }
}
}
printf("          %5.1f %% done.\n", 100.0);
printf(" Nb rays started within the image:          %7d => %5.2f %%\n",
       NbRaysStartInImage, 100.0 * NbRaysStartInImage / NbRays);
printf("          (if not 0 increase the radius of the sphere)\n");
printf(" Nb rays that do not intersect the image: %7d => %5.2f %%\n",
       NbRaysDoNotIntersect, 100.0 * NbRaysDoNotIntersect / NbRays);
printf("          (if too big, decrease the radius of the sphere)\n");
/* d) to store the histogram on the disk */
printf(" Saving file %s on disk\n", BelowHistFileName);
BelowHist->StoreToDisk(BelowHistFileName);
printf(" Saving file %s on disk\n", AboveHistFileName);
AboveHist->StoreToDisk(AboveHistFileName);
/* e) to free the memory */
delete Image;
delete BelowHist;
delete AboveHist;
}

void TreatInteraction(double X, double Y, double Z,
                    double U, double V, double W,
                    bool *RayStartsInImage, bool *RayCrossesImage,
                    Histogram1D *BelowHist, Histogram1D *AboveHist)
{
/* test if starting point inside the image */
if ( (X > ImageMinX) && (X < ImageMaxX) &&
     (Y > ImageMinY) && (Y < ImageMaxY) &&
     (Z > ImageMinZ) && (Z < ImageMaxZ) ) {
    *RayStartsInImage = true;
}
}

```

```

}
else {
    *RayStartsInImage = false;
    /* to get the entrance point of the ray */
    double EntranceX, EntranceY, EntranceZ;
    double EntranceDistance = DistanceToEntrancePoint(X, Y, Z, U, V, W,
        &EntranceX, &EntranceY, &EntranceZ);
    if (EntranceDistance < 0.0) { /* we missed the image */
        *RayCrossesImage = false;
    }
    else {
        *RayCrossesImage = true;
        /* to know on which side we are at the begining. after, we will */
        /* switch between inside and outside every time we cross the */
        /* isosurface. */
        int CurrentSide = WhatSideOfSurface(EntranceX, EntranceY, EntranceZ);
        double StartChordX=EntranceX;
        double StartChordY=EntranceY;
        double StartChordZ=EntranceZ;
        bool FirstChord = true; /* we do not want to record the first chord */
        /* we start from the entrance point, and marche cube after cube */
        /* until we leave the image. */
        double CurrentX=EntranceX;
        double CurrentY=EntranceY;
        double CurrentZ=EntranceZ;
        do {
            /* we need to know the limits of the cube to check when we leave */
            int I = (CurrentX / Vx) - 0.5;
            CubeMinX = Vx*(I + 0.5);
            CubeMaxX = CubeMinX + Vx;
            int J = (CurrentY / Vy) - 0.5;
            CubeMinY = Vy*(J + 0.5);
            CubeMaxY = CubeMinY + Vy;
            int K = (CurrentZ / Vz) - 0.5;
            CubeMinZ = Vz*(K + 0.5);
            CubeMaxZ = CubeMinZ + Vz;
            /* to check if the cube is on the isosurface */
            if (CubeOnIsosurface(I, J, K) == true) {
                /* We need to explore this cube. Since we can cross several */
                /* times the surface within the same cube, we need to loop */
                /* as long as we remain in the same cube. */
                do {
                    GoToOtherSide(CurrentX, CurrentY, CurrentZ, U, V, W, I, J, K,
                        CurrentSide, &CurrentX, &CurrentY, &CurrentZ);
                    /* we change the medium and record the chord length */
                    /* only if the new position is still in the same cube. */
                    /* If it is not, it means that we have crossed the cube */
                    /* without crossing any surface */
                    if ( (CurrentX > CubeMinX) && (CurrentX < CubeMaxX) &&
                        (CurrentY > CubeMinY) && (CurrentY < CubeMaxY) &&
                        (CurrentZ > CubeMinZ) && (CurrentZ < CubeMaxZ) ) {
                        /* at this point, we must have crossed the isosurface. */
                        /* check if the new point is on the other side. */
                        /* to test if we are on the right side of the isovalue. */
                        int Side = WhatSideOfSurface(CurrentX, CurrentY, CurrentZ);
                        if (Side == CurrentSide) {
                            printf("Error: Point XYZ = %18.15f %18.15f %18.15f",
                                CurrentX, CurrentY, CurrentZ);
                            if (Side == ABOVE_THRESHOLD) {
                                printf(" must be BELOW the threshold\n");
                            }
                            else {
                                printf(" must be ABOVE the threshold\n");
                            }
                        }
                        /* We cannot go any further. The function must end. */
                        CurrentX = VERY_BIG;
                    }
                }
                else {
                    if (FirstChord == true) {

```

```

        /* we don't record the first chord since it starts */
        /* outside the image */
        FirstChord = false;
    }
    else {
        double ChordLength = sqrt((StartChordX - CurrentX)*
                                   (StartChordX - CurrentX) +
                                   (StartChordY - CurrentY)*
                                   (StartChordY - CurrentY) +
                                   (StartChordZ - CurrentZ)*
                                   (StartChordZ - CurrentZ));
        if (CurrentSide == BELOW_THRESHOLD) {
            BelowHist->AddValue(ChordLength / WidthOfBin);
        }
        else {
            AboveHist->AddValue(ChordLength / WidthOfBin);
        }
    }
    /* a new chord starts */
    StartChordX = CurrentX;
    StartChordY = CurrentY;
    StartChordZ = CurrentZ;
    /* we cross the surf. => we switch to the other med. */
    if (CurrentSide == BELOW_THRESHOLD) {
        CurrentSide = ABOVE_THRESHOLD;
    }
    else {
        CurrentSide = BELOW_THRESHOLD;
    }
}
}
while ( (CurrentX > CubeMinX) && (CurrentX < CubeMaxX) &&
        (CurrentY > CubeMinY) && (CurrentY < CubeMaxY) &&
        (CurrentZ > CubeMinZ) && (CurrentZ < CubeMaxZ) );
}
else {
    /* the cube is not on the surface, thus we just */
    /* need to go across the current cube. */
    GoToNextCube(CurrentX, CurrentY, CurrentZ, U, V, W,
                 &CurrentX, &CurrentY, &CurrentZ);
}
}
while ( (CurrentX > ImageMinX) && (CurrentX < ImageMaxX) &&
        (CurrentY > ImageMinY) && (CurrentY < ImageMaxY) &&
        (CurrentZ > ImageMinZ) && (CurrentZ < ImageMaxZ) );
}
}
}

double DistanceToEntrancePoint(double X, double Y, double Z,
                               double U, double V, double W,
                               double *EntranceX, double *EntranceY, double *EntranceZ)
{
    double ShortestDistance = VERY_BIG;
    double Distance = (ImageMinX - X) / U;
    if (Distance > 0.0) {
        *EntranceY = Y + V*Distance;
        *EntranceZ = Z + W*Distance;
        if ( *EntranceY > ImageMinY && *EntranceY < ImageMaxY &&
             *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ ) {
            if (Distance < ShortestDistance) {
                ShortestDistance = Distance;
            }
        }
    }
}
Distance = (ImageMaxX - X) / U;
if (Distance > 0.0) {
    *EntranceY = Y + V*Distance;

```

```

*EntranceZ = Z + W*Distance;
if ( *EntranceY > ImageMinY && *EntranceY < ImageMaxY &&
    *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ ) {
    if (Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
}
}
Distance = (ImageMinY - Y) / V;
if (Distance > 0.0) {
    *EntranceZ = Z + W*Distance;
    *EntranceX = X + U*Distance;
    if ( *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ &&
        *EntranceX > ImageMinX && *EntranceX < ImageMaxX ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
Distance = (ImageMaxY - Y) / V;
if (Distance > 0.0) {
    *EntranceZ = Z + W*Distance;
    *EntranceX = X + U*Distance;
    if ( *EntranceZ > ImageMinZ && *EntranceZ < ImageMaxZ &&
        *EntranceX > ImageMinX && *EntranceX < ImageMaxX ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
Distance = (ImageMinZ - Z) / W;
if (Distance > 0.0) {
    *EntranceX = X + U*Distance;
    *EntranceY = Y + V*Distance;
    if ( *EntranceX > ImageMinX && *EntranceX < ImageMaxX &&
        *EntranceY > ImageMinY && *EntranceY < ImageMaxY ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
Distance = (ImageMaxZ - Z) / W;
if (Distance > 0.0) {
    *EntranceX = X + U*Distance;
    *EntranceY = Y + V*Distance;
    if ( *EntranceX > ImageMinX && *EntranceX < ImageMaxX &&
        *EntranceY > ImageMinY && *EntranceY < ImageMaxY ) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
ShortestDistance += VERY_SMALL;
/* to check if an intersection point has been found */
if (ShortestDistance > (VERY_BIG / 2.0)) {
    return(-1.0); /* means that we missed the image */
}
else {
    *EntranceX = X + U*ShortestDistance;
    *EntranceY = Y + V*ShortestDistance;
    *EntranceZ = Z + W*ShortestDistance;
    /* in case the += VERY_SMALL keep the point outside the image. */
    /* This can happens at the corners, or if the direction is */
    /* almost parallel to on face of the image */
    if ( (*EntranceX < ImageMinX) || (*EntranceX > ImageMaxX) ||
        (*EntranceY < ImageMinY) || (*EntranceY > ImageMaxY) ||
        (*EntranceZ < ImageMinZ) || (*EntranceZ > ImageMaxZ) ) {
        printf("Error: Entrance point %18.15f %18.15f %18.15f is ",
            EntranceX, EntranceY, EntranceZ);
    }
}

```

```

        printf("not inside the image\n");
        return(-1.0); /* means that we are too close to a corner, and */
                       /* we assume that we miss the image */
    }
    else {
        return(ShortestDistance);
    }
}

int WhatSideOfSurface(double X, double Y, double Z)
{
    /* to get the cube number */
    unsigned int I = (X / Vx) - 0.5;
    unsigned int J = (Y / Vy) - 0.5;
    unsigned int K = (Z / Vz) - 0.5;
    /* to get the gray levels at the 8 vertices */
    unsigned char B0 = Image->GrayLevel(I, J, K);
    unsigned char B1 = Image->GrayLevel(I+1, J, K);
    unsigned char B2 = Image->GrayLevel(I+1, J+1, K);
    unsigned char B3 = Image->GrayLevel(I, J+1, K);
    unsigned char B4 = Image->GrayLevel(I, J, K+1);
    unsigned char B5 = Image->GrayLevel(I+1, J, K+1);
    unsigned char B6 = Image->GrayLevel(I+1, J+1, K+1);
    unsigned char B7 = Image->GrayLevel(I, J+1, K+1);
    /* to check if on isosurface */
    if ( B0 > Threshold && B1 > Threshold &&
        B2 > Threshold && B3 > Threshold &&
        B4 > Threshold && B5 > Threshold &&
        B6 > Threshold && B7 > Threshold ) {
        return(ABOVE_THRESHOLD);
    }
    else {
        if ( B0 < Threshold && B1 < Threshold &&
            B2 < Threshold && B3 < Threshold &&
            B4 < Threshold && B5 < Threshold &&
            B6 < Threshold && B7 < Threshold ) {
            return(BELOW_THRESHOLD);
        }
        else {
            /* to calculate the trilinear interpolated gray level at */
            /* (X, Y, Z) relatively to the cube */
            double LocalX = (X - (I + 0.5)*Vx) / Vx;
            double LocalY = (Y - (J + 0.5)*Vy) / Vy;
            double LocalZ = (Z - (K + 0.5)*Vz) / Vz;
            double Bxyz = B0*(1.0 - LocalX)*(1.0 - LocalY)*(1.0 - LocalZ) +
                B1*      LocalX *(1.0 - LocalY)*(1.0 - LocalZ) +
                B2*      LocalX *      LocalY *(1.0 - LocalZ) +
                B3*(1.0 - LocalX)*      LocalY *(1.0 - LocalZ) +
                B4*(1.0 - LocalX)*(1.0 - LocalY)*      LocalZ +
                B5*      LocalX *(1.0 - LocalY)*      LocalZ +
                B6*      LocalX *      LocalY *      LocalZ +
                B7*(1.0 - LocalX)*      LocalY *      LocalZ;
            /* check if above or below the threshold */
            if (Bxyz > Threshold) {
                return(ABOVE_THRESHOLD);
            }
            else {
                return(BELOW_THRESHOLD);
            }
        }
    }
}

bool CubeOnIsosurface(unsigned int I, unsigned int J, unsigned int K)
{
    unsigned char B0 = Image->GrayLevel(I, J, K);
    unsigned char B1 = Image->GrayLevel(I+1, J, K);
    unsigned char B2 = Image->GrayLevel(I+1, J+1, K);

```

```

unsigned char B3 = Image->GrayLevel(I , J+1, K );
unsigned char B4 = Image->GrayLevel(I , J , K+1);
unsigned char B5 = Image->GrayLevel(I+1, J , K+1);
unsigned char B7 = Image->GrayLevel(I , J+1, K+1);
unsigned char B6 = Image->GrayLevel(I+1, J+1, K+1);
if ( ( B0 > Threshold && B1 > Threshold &&
      B2 > Threshold && B3 > Threshold &&
      B4 > Threshold && B5 > Threshold &&
      B6 > Threshold && B7 > Threshold ) ||
      ( B0 < Threshold && B1 < Threshold &&
      B2 < Threshold && B3 < Threshold &&
      B4 < Threshold && B5 < Threshold &&
      B6 < Threshold && B7 < Threshold ) ) {
    return(false);
}
else {
    return(true);
}
}

void GoToNextCube(double X, double Y, double Z,
                 double U, double V, double W,
                 double *ExitX, double *ExitY, double *ExitZ)
{
    /* get the shortest distance to go out */
    double ShortestDistance = VERY_BIG;
    double Distance = (CubeMinX - X) / U;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxX - X) / U;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMinY - Y) / V;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxY - Y) / V;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMinZ - Z) / W;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    Distance = (CubeMaxZ - Z) / W;
    if (Distance > 0.0 && Distance < ShortestDistance) {
        ShortestDistance = Distance;
    }
    /* to force the particle to cross the boundary of the cube */
    ShortestDistance += VERY_SMALL;
    /* to return the exit point */
    *ExitX = X + U*ShortestDistance;
    *ExitY = Y + V*ShortestDistance;
    *ExitZ = Z + W*ShortestDistance;
}

void GoToOtherSide(double X, double Y, double Z,
                  double U, double V, double W,
                  unsigned int I, unsigned int J, unsigned int K,
                  int CurrentSide,
                  double *ExitX, double *ExitY, double *ExitZ)
{
    /* to get the gray levels of the vertices of the cube */
    unsigned char B0 = Image->GrayLevel(I , J , K );
    unsigned char B1 = Image->GrayLevel(I+1, J , K );
    unsigned char B2 = Image->GrayLevel(I+1, J+1, K );
    unsigned char B3 = Image->GrayLevel(I , J+1, K );

```

```

unsigned char B4 = Image->GrayLevel(I , J , K+1);
unsigned char B5 = Image->GrayLevel(I+1, J , K+1);
unsigned char B6 = Image->GrayLevel(I+1, J+1, K+1);
unsigned char B7 = Image->GrayLevel(I , J+1, K+1);
/* to calculate the location (X0, Y0, Z0) relatively to the cube */
double X0 = X - (I + 0.5)*Vx;
double Y0 = Y - (J + 0.5)*Vy;
double Z0 = Z - (K + 0.5)*Vz;
/* to calculate the intersection of the isosurface with the direction */
/* It's a third degree equation. The coefficients a, b, c, and d are */
/* (check Maple file on PC to see the calculation) */
double DelB = B0 + B2 + B5 + B7 -(B1 + B3 + B4 + B6);
double DelX = B0 + B7 -(B3 + B4);
double DelY = B0 + B5 -(B1 + B4);
double DelZ = B0 + B2 -(B1 + B3);
double DelXY = B0 - B4;
double DelYZ = B0 - B1;
double DelZX = B0 - B3;
double a = U*V*W*DelB;
double b = U*V*(DelB*Z0 - DelZ*Vz)
          + V*W*(DelB*X0 - DelX*Vx)
          + W*U*(DelB*Y0 - DelY*Vy);
double c = U*(DelB*Y0*Z0 - DelY*Z0*Vy - DelZ*Y0*Vz + DelYZ*Vy*Vz)
          + V*(DelB*Z0*X0 - DelZ*X0*Vz - DelX*Z0*Vx + DelZX*Vz*Vx)
          + W*(DelB*X0*Y0 - DelX*Y0*Vx - DelY*X0*Vy + DelXY*Vx*Vy);
double d = (Threshold - B0)*Vx*Vy*Vz + DelB*X0*Y0*Z0
          + DelXY*Z0*Vx*Vy + DelYZ*X0*Vy*Vz + DelZX*Y0*Vz*Vx
          -(DelX*Y0*Z0*Vx + DelY*Z0*X0*Vy + DelZ*X0*Y0*Vz);
/* to solve the third degree equation */
double Root[3];
unsigned int NbRoots;
ThirdDegreeEqn(a, b, c, d, &NbRoots, Root);
/* to get the shortest distance to the surface */
double ShortestDistance = VERY_BIG;
double Distance;
for (unsigned int RootNo=0; RootNo < NbRoots; RootNo++) {
    Distance = Root[RootNo];
    if (Distance >= 0.0) {
        if (Distance < ShortestDistance) {
            ShortestDistance = Distance;
        }
    }
}
/* to check if the distance takes the particle outside the cube */
/* 1) with x = 0.0 */
Distance = -X0 / U;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
/* 2) with x = Vx */
Distance = (Vx - X0) / U;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
/* 3) with y = 0.0 */
Distance = -Y0 / V;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
/* 4) with y = Vy */
Distance = (Vy - Y0) / V;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
/* 5) with z = 0.0 */
Distance = -Z0 / W;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
}

```

```
/* 6) with z = Vz */
Distance = (Vz - Z0) / W;
if (Distance > 0.0 && Distance < ShortestDistance) {
    ShortestDistance = Distance;
}
/* to force the particle to cross the isosurface */
ShortestDistance += VERY_SMALL;
/* to return the exit point */
*ExitX = X0 + U*ShortestDistance;
*ExitY = Y0 + V*ShortestDistance;
*ExitZ = Z0 + W*ShortestDistance;
/* to get the exit point in the absolute coordinate system */
*ExitX = (*ExitX + (I + 0.5)*Vx);
*ExitY = (*ExitY + (J + 0.5)*Vy);
*ExitZ = (*ExitZ + (K + 0.5)*Vz);
}
```

APPENDIX H HMC VOLUME FRACTION (C++ PROGRAMS)

This appendix contains the C++ program that calculates the volume fraction of marrow that is separated from the bone by the hyperboloid surface generated by the Hyperboloid Marching Cube algorithm. The program uses some tools of the previous appendices:

- [Appendix B](#):
 - RandomNumber
 - Equations

```

/*****
/*   TLVolumeFraction.cpp
/*
/*   This program is used to calculate the volume fraction within the
/*   isosurface given by a threshold value. It compare the gray level of
/*   each voxel with the threshold, and assign the entire voxel above or
/*   below the isosurface
/*   It also account for the part of the volume that will be outside
/*   the Marching Cube image.
/*
/*
/*****
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "../CppTools/Image256GL.h"
#include "../CppTools/RandomNumber.h"

#define NB_POINTS      1000000

main( int argc,
      char *argv[] )
{
    /* 1) To get the parameters and read the image file */
    /*****
    /* to check the command line */
    if ( argc != 9 ) {
        printf("Error: bad command-line. Try again with command-line: ");
        printf("TLVolumeFraction ImageFileName Nx Ny Nz Vx Vy Vz Threshold\n");
        printf(" - ImageFileName: the full path name (without the ");
        printf("extension) of the\n image file from the current ");
        printf("directory. The image must have been\n stored ");
        printf("with the Image256GL tool and have the extension IGL.\n");
        printf(" - Nx: the number of voxels per row.\n");
        printf(" - Ny: the number of rows per slice.\n");
        printf(" - Nz: the number of slices in the image.\n");
        printf(" - Vx, Vy, and Vz: the voxel size (in cm).\n");
        printf(" - Threshold: the threshold used to separate bone from ");
        printf("marrow.\n");
        printf("ex: TLVolumeFraction ../../Database/MyImageName ");
        printf("200 200 200 0.0088 0.0088 0.0088 127.5\n");
        exit(0);
    }
    /* to get the parameters */

```

```

char ImageFileName[100];
sprintf(ImageFileName, "%s.IGL", argv[1]);
int Nx = atoi(argv[2]);
int Ny = atoi(argv[3]);
int Nz = atoi(argv[4]);
double Vx = atof(argv[5]);
double Vy = atof(argv[6]);
double Vz = atof(argv[7]);
double Threshold = atof(argv[8]);
printf("\nGray level image volume fraction calculation:\n");
printf("  Image file: %s.\n", ImageFileName);
printf("  Image size: %4u x %4u x %4u => %u voxels.\n", Nx, Ny, Nz, Nx*Ny*Nz);
printf("  Voxel size: %8.6f x %8.6f x %8.6f cm3.\n", Vx, Vy, Vz);
printf("  Threshold: %7.3f.\n", Threshold);
/* to read the image */
printf("\n1) Reading the image file %s.\n", ImageFileName);
Image256GL Image(Nx, Ny, Nz);
Image.LoadFromFile(ImageFileName);

/* 2) To calculate the volume fraction above the threshold */
/*****
printf("\n2) Calculating the volume fraction above the threshold.\n");
unsigned int PointsAbove = 0;
for ( unsigned int PointNo=0; PointNo<NB_POINTS; PointNo++ ) {
  /* get a random point */
  double X = ((Nx - 1)*GetRandomNumber() + 0.5) * Vx;
  double Y = ((Ny - 1)*GetRandomNumber() + 0.5) * Vy;
  double Z = ((Nz - 1)*GetRandomNumber() + 0.5) * Vz;
  /* to calculate the cube number */
  int I = (X / Vx) - 0.5;
  int J = (Y / Vy) - 0.5;
  int K = (Z / Vz) - 0.5;
  /* to get the gray levels of the vertices of the cube */
  unsigned char B0 = Image.GrayLevel(I , J , K );
  unsigned char B1 = Image.GrayLevel(I+1, J , K );
  unsigned char B2 = Image.GrayLevel(I+1, J+1, K );
  unsigned char B3 = Image.GrayLevel(I , J+1, K );
  unsigned char B4 = Image.GrayLevel(I , J , K+1);
  unsigned char B5 = Image.GrayLevel(I+1, J , K+1);
  unsigned char B6 = Image.GrayLevel(I+1, J+1, K+1);
  unsigned char B7 = Image.GrayLevel(I , J+1, K+1);
  /* to check if the cube is a full marrow cube */
  if ( B0 > Threshold && B1 > Threshold &&
        B2 > Threshold && B3 > Threshold &&
        B4 > Threshold && B5 > Threshold &&
        B6 > Threshold && B7 > Threshold ) {
    PointsAbove = PointsAbove + 1;
  }
  else {
    /* to check if the cube is a full bone cube */
    if ( B0 < Threshold && B1 < Threshold &&
          B2 < Threshold && B3 < Threshold &&
          B4 < Threshold && B5 < Threshold &&
          B6 < Threshold && B7 < Threshold ) {
    }
    else {
      /* to calculate the trilinear interpolated gray level at */
      /* (X, Y, Z) relatively to the cube */
      double LocalX = (X - (I + 0.5)*Vx) / Vx;
      double LocalY = (Y - (J + 0.5)*Vy) / Vy;
      double LocalZ = (Z - (K + 0.5)*Vz) / Vz;
      double Bxyz = B0*(1.0 - LocalX)*(1.0 - LocalY)*(1.0 - LocalZ) +
                    B1*      LocalX *(1.0 - LocalY)*(1.0 - LocalZ) +
                    B2*      LocalX *      LocalY *(1.0 - LocalZ) +
                    B3*(1.0 - LocalX)*      LocalY *(1.0 - LocalZ) +
                    B4*(1.0 - LocalX)*(1.0 - LocalY)*      LocalZ +
                    B5*      LocalX *(1.0 - LocalY)*      LocalZ +
                    B6*      LocalX *      LocalY *      LocalZ +
                    B7*(1.0 - LocalX)*      LocalY *      LocalZ;
    }
  }
}
*****/

```

```
        /* check if above or below the threshold */
        if (Bxyz > Threshold) {
            PointsAbove = PointsAbove + 1;
        }
    }
}

/* 3) To show the results */
/*****/
printf("\n3) Volume fraction above the threshold:\n");
printf("    Number of points above the threshold: %10u (%6.2f%%)\n",
        PointsAbove, 100.0*(double)PointsAbove/NB_POINTS);
double VolumeAbove = (double)PointsAbove * Vx * Vy * Vz *
        (Nx-1) * (Ny-1) * (Nz-1) / NB_POINTS;
printf("    Volume above threshold: %14.10f cm3\n",VolumeAbove);
}
```

APPENDIX I HMC TRANSPORT CODE (EGSNRC USER CODE)

This appendix contains the MORTRAN user code of the EGSnrc radiation code that has been developed to transport electrons within the hyperboloid representation of the bone-marrow interface. It uses the Hyperboloid Marching Cube algorithm to convert the gray-level input image into the hyperboloid isosurface. This appendix also shows examples of the following files used with this code:

- Configuration file
- Input file
- Output file

EGSnrc MORTRAN User Code

```

!INDENT M 4;      "INDENT EACH MORTRAN NESTING LEVEL BY 4"
!INDENT F 2;      "INDENT EACH FORTRAN NESTING LEVEL BY 2"
"This line is 80 characters long, use it to set up the screen width"
"23456789|123456789|123456789|123456789|123456789|123456789|123456789|123456789"
"*****"
"
"
"          *****
"          *
"          * GraySphereModelHyperb.mortran *
"          *
"          *****
"
" This program calculates the absorbed fraction of energy within the bone
" trabeculae and the marrow cavities of a trabecular bone sample. The geometry
" is defined by a 3D gray-level image and a threshold that represents the
" limit between the bone voxels and the marrow voxels. The surface between
" bone and marrow is determined by the trilinear interpolation adaptation of
" the Marching Cube algorithm. The center of 8 adjacent voxels are the
" vertices of a cube within which the surface is given by the equation of a
" 3D hyperboloid. The gray level field within the cube is found by the
" the trilinear interpolation from the 8 vertices. The equation of the surface
" is given by the isosurface of this field that corresponds to the threshold.
" Several configurations can be simulated by the program. The parameters of
" each configuration are read from the input file: Input.dat. This file
" contains one line per configuration. For each configuration it provides:
"
"     - the type of particle: -1 for electrons, 0 for photons
"     - the initial energy of the particles
"     - the number of histories per configuration.
" The results are in the file Output.dat. Each result is shown with a 95%
" confidence interval.
"
"*****"
"-----"
" Step 1:  To override the EGSnrc macros
"-----"
" 1) so that all real variables are in double precision "
REPLACE {$REAL} WITH {DOUBLE PRECISION}
" 2) the size of the arrays used by EGSnrc. "
REPLACE {$MXMED} WITH {2}           "2 medium in the problem (default 10)"
REPLACE {$MXREG} WITH {4}           "4 geometric regions (default 2000)"

```

```

REPLACE {SMXSTACK} WITH {100}          "less than 100 particles on stack at once"
REPLACE {SMXMDSH} WITH {100}          "max. nb of shells per medium for "
                                        "incoherent scattering"

" 3) for compatibility with the old EGS4. "
REPLACE {$CALL-HOWNEAR(#);} WITH {CALL HOWNEAR({P1},X(NP),Y(NP),Z(NP),IRL);}
"-----"
" Step 1.a. To define user constant values "
"-----"
REPLACE {$REG_BONE} WITH {1}          " region within the bone trabeculae "
REPLACE {$REG_MARROW} WITH {2}        " region within the marrow cavities "
REPLACE {$REG_OUTSIDE} WITH {3}       " region outside the study "
REPLACE {$REG_LOST} WITH {4}          " region for lost particles "
REPLACE {$IMAGE_FILE} WITH {23}       " file to read the image "
REPLACE {$INPUT_FILE} WITH {24}       " file to get the parameters "
REPLACE {$OUTPUT_FILE} WITH {25}      " file to record the results "
REPLACE {$N_RUN} WITH {100}           " number of run for each configuration"
REPLACE {$INFINITY} WITH {1.0D+99}    " to simulate infinity long distance "
REPLACE {$PI} WITH {3.1415926535897932D+00} " need Pi in Source "
" this is to solve the boundary crossing problem. The particle is "
" transported a little farther than the exact boundary "
REPLACE {$BOUNDARY_THICKNESS} WITH {1.0D-09} " that's 0.10 Angstrom "
" for the geometrical model "
REPLACE {$VOXEL_SIZE_X} WITH {0.008789D+00} " in cm "
REPLACE {$VOXEL_SIZE_Y} WITH {0.008789D+00} " in cm "
REPLACE {$VOXEL_SIZE_Z} WITH {0.008789D+00} " in cm "
REPLACE {$IMAGE_NX} WITH {102}        " nb of voxels along (O,x) "
REPLACE {$IMAGE_NY} WITH {241}       " nb of voxels along (O,y) "
REPLACE {$IMAGE_NZ} WITH {79}        " nb of voxels along (O,z) "
REPLACE {$THRESHOLD} WITH {13.5}     " threshold between bone and marrow "
"-----"
" Step 1.b. To define the user common variables "
"-----"
" a) for scoring the results "
REPLACE {COMIN/SCOR/;} WITH
    {COMMON/SCOR/ CumulEnergyBone,CumulEnergyMarrow,
      CumulEnergyOutside,CumulEnergyLost;
      $REAL CumulEnergyBone; $REAL CumulEnergyMarrow;
      $REAL CumulEnergyOutside; $REAL CumulEnergyLost;}
" b) for the geometry "
REPLACE {COMIN/GEOM/;} WITH
    {COMMON/GEOM/BoneImage;
      BYTE BoneImage($IMAGE_NZ * $IMAGE_NY * $IMAGE_NX);}
"-----"
" Step 1.c. To define the variables of the main program "
"-----"
$IMPLICIT-NONE; " to make sure that all variables are declared "
" 1) all the common that you need in the main programm "
COMMON/BOUNDS,MEDIA,MISC,USEFUL,RANDOM,GEOM,SCOR/;
" The above expands into COMMON statements "
" BOUNDS contains ECUT and PCUT "
" MEDIA contains NMED and the array concerning media "
" MISC contains the medium per region and Rayleigh parameters "
" USEFUL contains electron rest mass "
" RANDOM contains the RANMAR parameters "
" GEOM passes info to HOWFAR and HOWNEAR routines "
" SCOR passes info to AUSGAB routine "

" 2) local variables of the main program "
$REAL XIN, YIN, ZIN; " particle location (to give to SHOWER) "
$REAL UIN, VIN, WIN; " particle direction (to give to SHOWER) "
$REAL EIN; " particle energy (to give to SHOWER) "
$REAL WTIN; " particle weight (to give to SHOWER) "
$INTEGER IQIN; " particle type (to give to SHOWER) "
$INTEGER IRIN; " particle region (to give to SHOWER) "
$INTEGER PartNo; " particle # to loop for each particle "
$INTEGER RunNo; " run number to loop for each run "
$INTEGER ConfigNo; " configuration number to loop for each one "
LOGICAL NoMoreConfig; " to test the end of the input file "
$INTEGER ParticleType; " particle type got from the input file "
$REAL KineticEnergy; " kinetic energy got from the input file "

```

```

$INTEGER NumberOfHistories;    " number of histories got from the input file "
$INTEGER ParticlePerRun;        " number of particles per run "
" for statistical results: mean, standard deviation, standard deviation "
" of the mean, 95% confidence interval, and 95% confidence error "
$REAL AFBone;
$REAL MeanAFBone;
$REAL StdDevAFBone;
$REAL StdDevOfMeanAFBone;
$REAL ConfIntOfMeanAFBone;
$REAL ConfErrOfMeanAFBone;
$REAL AFMarrow;
$REAL MeanAFMarrow;
$REAL StdDevAFMarrow;
$REAL StdDevOfMeanAFMarrow;
$REAL ConfIntOfMeanAFMarrow;
$REAL ConfErrOfMeanAFMarrow;
$REAL AFOutside;
$REAL MeanAFOutside;
$REAL StdDevAFOutside;
$REAL StdDevOfMeanAFOutside;
$REAL ConfIntOfMeanAFOutside;
$REAL ConfErrOfMeanAFOutside;
$REAL AFLost;
$REAL MeanAFLost;
$REAL StdDevAFLost;
$REAL StdDevOfMeanAFLost;
$REAL ConfIntOfMeanAFLost;
$REAL ConfErrOfMeanAFLost;
" 3) system functions invoked in the main program "
$REAL DSQRT;
INTRINSIC DSQRT;
"-----"
" Step 2.  To initialize the EGSnrc data "
"-----"
" 1) to place medium names in an array. "
"   $$ is a MORTRAN macro to expand strings "
CHARACTER*4 MEDARR(24,$MXMED);
$INTEGER I, J;
DATA MEDARR /$$'CorticalBone',12*' ', $$'Marrow',18*' ';/;
NMED = $MXMED;          "Set number of media."
DO J = 1,$MXMED [
  DO I=1,24 [
    MEDIA(I,J) = MEDARR(I,J);
    ] " this is to avoid a DATA STATEMENT for a variable in COMMON"
    " NMED and DUNIT default to 1, i.e. one medium and we work in cm "
  ]
]
" 2) to initialize the medium in each region "
MED($REG_BONE)      = 1;  "cortical bone in the bone trabeculae"
MED($REG_MARROW)    = 2;  "bone marrow in the marrow cavities"
MED($REG_OUTSIDE)   = 0;  "vacuum outside the study region "
MED($REG_LOST)      = 0;  "vacuum if particles are lost (does not matter)"
" 3) to initialize the cutoff energy for both electrons and "
"   photons in each region "
ECUT($REG_BONE)     = 0.001D+00 + PRM;  " 1 keV + rest mass for electrons "
PCUT($REG_BONE)     = 0.001D+00;        " 1 keV for photons "
ECUT($REG_MARROW)   = 0.001D+00 + PRM;
PCUT($REG_MARROW)   = 0.001D+00;
ECUT($REG_OUTSIDE) = 0.001D+00 + PRM;
PCUT($REG_OUTSIDE) = 0.001D+00;
ECUT($REG_LOST)    = 0.001D+00 + PRM;
PCUT($REG_LOST)    = 0.001D+00;
" 4) to ask EGSnrc to treat the Rayleigh scattering in each region "
IRAYLR($REG_BONE)   = 1;
IRAYLR($REG_MARROW) = 1;
IRAYLR($REG_OUTSIDE) = 1;
IRAYLR($REG_LOST)  = 1;
" 5) to initialize the random number generator "
IXX = 1; JXX = 1; " seed # to initialize the random number series "
$RNG=INITIALIZATION;

```

```

"-----"
" Step 3. To pick up the cross sections precalculated by pegs4 "
"-----"
CALL HATCH;          " data file must be assigned to unit 12 "
PRINT *, 'End of HATCH';
"-----"
" Step 3.a. To initialize the output file "
"-----"
OPEN ( UNIT=$OUTPUT_FILE, FILE='../Output.dat', STATUS='unknown' );
WRITE($OUTPUT_FILE, '(A,A)') 'Absorbed fractions for irradiation ',
      'from bone trabeculae.';
"-----"
" Step 3.b. To open and read the image file "
"-----"
OPEN($IMAGE_FILE,
     FILE='../.../BoneProject/Database/M408_L4_A1/M408_L4_A1_RAW_FIL_ROI.IGL',
     ACCESS='DIRECT',
     FORM='UNFORMATTED',
     RECL=$IMAGE_NZ*$IMAGE_NY*$IMAGE_NX);
PRINT *, 'ok opening image file';
READ($IMAGE_FILE, REC=1) BoneImage;
CLOSE ( $IMAGE_FILE );
PRINT *, 'ok reading image file';
"-----"
" Step 3.c. For each configuration in the input file "
"-----"
" One execution is performed for each line of the input file "
OPEN ( UNIT=$INPUT_FILE, FILE='../Input.dat', STATUS='old' );
READ ( $INPUT_FILE, * ); " to skip the first line "
NoMoreConfig = .FALSE.;
ConfigNo = 0;
LOOP [" until no more line in the file "
     "-----"
     " Step 3.d. To read a new line in the input file "
     "-----"
     READ ( $INPUT_FILE, *, END = :EndInput: )
           ParticleType, KineticEnergy, NumberOfHistories;
           GO TO :NextInput:;
           :EndInput: NoMoreConfig = .TRUE.;
           :NextInput: CONTINUE;
     "-----"
     " Step 3.e. If a new line exists, initialize the data for this config. "
     "-----"
     IF (~NoMoreConfig) [
       " 1) to display the new configuration "
       ConfigNo = ConfigNo + 1;
       PRINT *, 'Configuration no:', ConfigNo;
       " 2) how many particles per run? "
       ParticlePerRun = NumberOfHistories / $N_RUN;
       " 3) to output the parameters of the configuration "
       WRITE($OUTPUT_FILE, '(A)') ' ';
       WRITE($OUTPUT_FILE, '(A,I3)') 'Configuration No:', ConfigNo;
       WRITE($OUTPUT_FILE, '(A)') 'The calculation is performed for: ';
       WRITE($OUTPUT_FILE, '(A,I5,A)') ' ', $N_RUN, ' runs';
       IF (ParticleType = 0) [
         WRITE($OUTPUT_FILE, '(A,I6,A)') ' ', ParticlePerRun,
           ' photons per run';
       ]
       ELSE [
         WRITE($OUTPUT_FILE, '(A,I6,A)') ' ', ParticlePerRun,
           ' electrons per run';
       ]
       WRITE($OUTPUT_FILE, '(A,I8,A)') ' Total: ',
         ParticlePerRun*$N_RUN, ' histories.';
       WRITE($OUTPUT_FILE, '(A,F7.3,A)') ' Initial kinetic energy: ',
         KineticEnergy, ' MeV.';
       " 4) to initialize the statistical data "
       MeanAFBone = 0.0;
       MeanAFMarrow = 0.0;
     ]

```

```

MeanAFOutside = 0.0;
MeanAFLost = 0.0;
StdDevAFBone = 0.0;
StdDevAFMarrow = 0.0;
StdDevAFOutside = 0.0;
StdDevAFLost = 0.0;
"-----"
" Step 3.f. For each run "
"-----"
DO RunNo=1,$N_RUN [
  PRINT *, ' Run no:', RunNo;
  "-----"
  " Step 4. To initialize the geometry for HOWFAR and HOWNEAR "
  "-----"
  " done when reading the input file "
  "-----"
  " Step 5. To initialize the scoring variables for AUSGAB "
  "-----"
  CumulEnergyBone = 0.0;
  CumulEnergyMarrow = 0.0;
  CumulEnergyOutside = 0.0;
  CumulEnergyLost = 0.0;
  "-----"
  " Step 5.a. For each particle "
  "-----"
  DO PartNo=1, ParticlePerRun [
    " to have a display of the progression of the code "
    IF (MOD(PartNo,100) = 0) [
      PRINT *, ' Particle: ', PartNo;
    ]
    "-----"
    " Step 6. To define the particle parameters "
    "-----"
    IF (ParticleType = 0) [
      EIN = KineticEnergy; " initial kinetic energy"
    ]
    ELSE [
      EIN = KineticEnergy + PRM; " initial kinetic + rest mass energy"
    ]
    IQIN=ParticleType;
    WTIN=1.0D+00; " weight = 1 since no variance reduction used"
    " to get the initial location and direction of the particle. "
    CALL SourceBone(XIN,YIN,ZIN,UIN,VIN,WIN,IRIN);
    "-----"
    " Step 7. To transport the particle "
    "-----"
    CALL SHOWER(IQIN,EIN,XIN,YIN,ZIN,UIN,VIN,WIN,IRIN,WTIN);
  ]
  "-----"
  " Step 7.a. To calculate and display the result for this run "
  "-----"
  AFBone = CumulEnergyBone / (ParticlePerRun * KineticEnergy);
  AFMarrow = CumulEnergyMarrow / (ParticlePerRun * KineticEnergy);
  AFOutside = CumulEnergyOutside / (ParticlePerRun * KineticEnergy);
  AFLost = CumulEnergyLost / (ParticlePerRun * KineticEnergy);
  PRINT *, ' Data for this run: ';
  PRINT *, ' AF in bone: ', AFBone;
  PRINT *, ' AF in marrow: ', AFMarrow;
  PRINT *, ' AF outside: ', AFOutside;
  PRINT *, ' AF lost: ', AFLost;
  PRINT *, ' Total AF: ', AFBone+AFMarrow+ AFOutside+AFLost;
  "-----"
  " Step 7.b. To cumulate the statistical data "
  "-----"
  MeanAFBone = MeanAFBone + AFBone;
  MeanAFMarrow = MeanAFMarrow + AFMarrow;
  MeanAFOutside = MeanAFOutside + AFOutside;
  MeanAFLost = MeanAFLost + AFLost;
  StdDevAFBone = StdDevAFBone + AFBone*AFBone;

```

```

StdDevAFMarrow = StdDevAFMarrow + AFMarrow*AFMarrow;
StdDevAFOutside = StdDevAFOutside + AFOutside*AFOutside;
StdDevAFLost = StdDevAFLost + AFLost*AFLost;
] " End of this run "
"-----"
" Step 7.c. To calculate the statistical data "
"-----"
" a) the mean "
MeanAFBone = MeanAFBone / $N_RUN;
MeanAFMarrow = MeanAFMarrow / $N_RUN;
MeanAFOutside = MeanAFOutside / $N_RUN;
MeanAFLost = MeanAFLost / $N_RUN;
" b) the standard deviation of the sample "
StdDevAFBone = StdDevAFBone - $N_RUN*MeanAFBone*MeanAFBone;
StdDevAFMarrow = StdDevAFMarrow - $N_RUN*MeanAFMarrow*MeanAFMarrow;
StdDevAFOutside = StdDevAFOutside - $N_RUN*MeanAFOutside*MeanAFOutside;
StdDevAFLost = StdDevAFLost - $N_RUN*MeanAFLost*MeanAFLost;
StdDevAFBone = StdDevAFBone / ($N_RUN - 1);
StdDevAFMarrow = StdDevAFMarrow / ($N_RUN - 1);
StdDevAFOutside = StdDevAFOutside / ($N_RUN - 1);
StdDevAFLost = StdDevAFLost / ($N_RUN - 1);
StdDevAFBone = DSQRT(StdDevAFBone);
StdDevAFMarrow = DSQRT(StdDevAFMarrow);
StdDevAFOutside = DSQRT(StdDevAFOutside);
StdDevAFLost = DSQRT(StdDevAFLost);
" c) the standard deviation of the mean */
StdDevOfMeanAFBone = StdDevAFBone / DSQRT(DBLE($N_RUN));
StdDevOfMeanAFMarrow = StdDevAFMarrow / DSQRT(DBLE($N_RUN));
StdDevOfMeanAFOutside = StdDevAFOutside / DSQRT(DBLE($N_RUN));
StdDevOfMeanAFLost = StdDevAFLost / DSQRT(DBLE($N_RUN));
" d) the 95% confidence interval of the mean */
ConfIntOfMeanAFBone = 1.96*StdDevOfMeanAFBone;
ConfIntOfMeanAFMarrow = 1.96*StdDevOfMeanAFMarrow;
ConfIntOfMeanAFOutside = 1.96*StdDevOfMeanAFOutside;
ConfIntOfMeanAFLost = 1.96*StdDevOfMeanAFLost;
" e) the 95% confidence error of the mean */
ConfErrOfMeanAFBone = 1.0D+02 * ConfIntOfMeanAFBone / MeanAFBone;
ConfErrOfMeanAFMarrow = 1.0D+02 * ConfIntOfMeanAFMarrow / MeanAFMarrow;
ConfErrOfMeanAFOutside = 1.0D+02 * ConfIntOfMeanAFOutside / MeanAFOutside;
ConfErrOfMeanAFLost = 1.0D+02 * ConfIntOfMeanAFLost / MeanAFLost;
"-----"
" Step 8. To print out the results to the output file "
"-----"
WRITE($OUTPUT_FILE, '(A,A)') ' Absorbed fractions with 95%',
' confidence intervals: ';
WRITE($OUTPUT_FILE, '(A,F16.14,A,F16.14,A,F6.2,A)')
' AF in bone: ', MeanAFBone,
' +/- ', ConfIntOfMeanAFBone, ' (', ConfErrOfMeanAFBone, '%)';
WRITE($OUTPUT_FILE, '(A,F16.14,A,F16.14,A,F6.2,A)')
' AF in marrow: ', MeanAFMarrow,
' +/- ', ConfIntOfMeanAFMarrow, ' (', ConfErrOfMeanAFMarrow, '%)';
WRITE($OUTPUT_FILE, '(A,F16.14,A,F16.14,A,F6.2,A)')
' AF outside: ', MeanAFOutside,
' +/- ', ConfIntOfMeanAFOutside, ' (', ConfErrOfMeanAFOutside, '%)';
WRITE($OUTPUT_FILE, '(A,F16.14,A,F16.14,A,F6.2,A)')
' AF lost: ', MeanAFLost,
' +/- ', ConfIntOfMeanAFLost, ' (', ConfErrOfMeanAFLost, '%)';
WRITE($OUTPUT_FILE, '(A,F16.14)') ' Total AF: ',
MeanAFBone+MeanAFMarrow+MeanAFOutside+MeanAFLost;
]
] " End of this configuration "
UNTIL (NoMoreConfig);
"-----"
" Step 8.a. Don't forget to close the files "
"-----"
CLOSE($INPUT_FILE);
CLOSE($OUTPUT_FILE);
END; " End of main program "
"*****"

```

```

"                                     HOWFAR                                     "
"*****"
"
" The HOWFAR subroutine measures the distance between the location of the "
" particle (Xp, Yp, Zp) and the next boundary crossed by the particle when "
" traveling to the direction (Up, Vp, Wp). "
" The returned values are: "
"     IDISC is set to 1 if we need to discard the particle "
"     USTEP is shortened if the boundary is reached by the particle "
"     IRNEW is set with the region number that lies beyond the boundary "
"*****"
SUBROUTINE HOWFAR;
  $IMPLICIT-NONE; " to make sure that all variables are declared "
  " COMMON variables "
  COMIN/STACK,EPCONT/;
  " The above expands into COMMON statements "
  "   STACK contains IR(NP), X,Y,Z(NP), and U,V,W(NP) "
  "   EPCONT contains USTEP: the distance EGSnrc is to transport the part. "
  " local variables "
  $REAL Xp, Yp, Zp; " the position of the particle "
  $REAL Up, Vp, Wp; " the direction of the particle "
  $INTEGER IReg; " the region number"
  $REAL Distance; " the distance to the boundary "
  $INTEGER RegNew; " the region beyond the boundary "
  " user functions invoked in the subroutine "
  LOGICAL InsideBone;
  LOGICAL InsideMarrow;
  $REAL DistanceToExitLocalCell;

  "-----"
  " 1) To get the data from EGSnrc "
  "-----"
  Xp = X(NP); Yp = Y(NP); Zp = Z(NP);
  Up = U(NP); Vp = V(NP); Wp = W(NP);
  IReg = IR(NP);
  "-----"
  " 2) To check the data returned by EGSnrc "
  "-----"
  " if a mismatch is detected, the particle is discarded imediatly (IDISC=1) "
  " IR(NP) is set to the region $REG_LOST so that AUSGAB can detect the "
  " problem (IRNEW is not used by EGS since it does not transport the "
  " particle before it calls AUSGAB) "
  " a) to check the region numbers "
  IF ( (IReg ~= $REG_BONE) & (IReg ~= $REG_MARROW) &
      (IReg ~= $REG_OUTSIDE) ) [
    PRINT *, 'Error in HOWFAR: wrong region number: ', IReg;
    IDISC = 1;
    IR(NP) = $REG_LOST;
    RETURN;
  ]
  " b) to check if the region number matches the location "
  IF (IReg = $REG_BONE) [
    IF (~InsideBone(Xp, Yp, Zp)) [
      PRINT *, 'Error in HOWFAR: particle is not in bone.';
      IDISC = 1;
      IR(NP) = $REG_LOST;
      RETURN;
    ]
  ]
  ELSEIF (IReg = $REG_MARROW) [
    IF (~InsideMarrow(Xp, Yp, Zp)) [
      PRINT *, 'Error in HOWFAR: particle is not in marrow.';
      IDISC = 1;
      IR(NP) = $REG_LOST;
      RETURN;
    ]
  ]
  ]
  ELSE [

```

```

    IF ( InsideBone(Xp, Yp, Zp) |
          InsideMarrow(Xp, Yp, Zp) ) [
        PRINT *, 'Error in HOWFAR: particle is not outside.';
        IDISC = 1;
        IR(NP) = $REG_LOST;
        RETURN;
    ]
]
"-----"
" 3) To discard the particle if it goes outside the study region "
"-----"
IF ( IReg = $REG_OUTSIDE ) [
    IDISC = 1;
]
ELSE [
    "-----"
    " 4) To calculate the distance to the boundary "
    "-----"

    RegNew = IReg;
    Distance = DistanceToExitLocalCell(Xp, Yp, Zp, Up, Vp, Wp, RegNew);
    "-----"
    " 5) To make sure the particle jumps on the other side of the boundary "
    "-----"

    Distance = Distance + $BOUNDARY_THICKNESS;
    "-----"
    " 6) To check if the distance is shorter than USTEP "
    " 7) To calculate the region beyond the boundary "
    "-----"

    IF ( Distance < USTEP ) [
        USTEP = Distance;
        IRNEW = RegNew;
    ]
]
END; " End of subroutine HOWFAR "
"-----"
"                                     HOWNEAR "
"-----"
" The HOWNEAR subroutine measures the shortest distance between the location "
" of the particle (Xp, Yp, Zp) and the boundary of the actual region IReg. "
" The returned values are: "
"     TPerp is the shortest distance from the particle location to "
"     the boundary of the region IReg "
"-----"
SUBROUTINE HOWNEAR(TPerp, Xp, Yp, Zp, IReg);
  $IMPLICIT-NONE; " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL TPerp; " the shortest distance to return "
  $REAL Xp, Yp, Zp; " the current location of the particle "
  $INTEGER IReg; " the current region of the particle "
  " user functions invoked in the subroutine "
  $REAL ShortestDistanceToLocalCellBoundary;

  "-----"
  " 1) To check if the particle has become out of study "
  "-----"
  IF ( IReg = $REG_OUTSIDE ) [
    TPerp = 0.0; " so that HOWFAR is called and discard the particle "
  ]
  ELSE [
    "-----"
    " 2) To calculate the shortest distance "
    "-----"
    TPerp = ShortestDistanceToLocalCellBoundary(Xp, Yp, Zp);
    "-----"
    " 3) To make sure the particle will not be too close to the boundary "
    "-----"
    TPerp = TPerp - $BOUNDARY_THICKNESS;
  ]
]

```

```

        IF (TPerp < 0.0) [
            TPerp = 0.0;
        ]
    ]
END; " End of subroutine HOWNEAR "
"*****"
"                                     AUSGAB                                     "
"*****"
"
" The AUSGAB subroutine cumulates the energy deposited within the regions. "
" The energy is stored in the 'CumulEnergy' variables.                       "
"
" Input:
"   . IARG : A flag (see EGSnrc documentation) which is set to 3 if the "
"             particle is discarded by the HOWFAR subroutine, in our "
"             situation, that means that the particle is going outside "
"             the study region or that it has been lost.                   "
"*****"
SUBROUTINE AUSGAB(IARG);
    $IMPLICIT-NONE; " to make sure that all variables are declared "
    " parameters of the routine "
    $INTEGER IARG;
    " COMMON variables "
    COMIN/STACK,EPCONT,SCOR/;
    " The above expands into COMMON statements "
    "   STACK contains IR(NP) "
    "   EPCONT contains EDEP: the energy deposited now "
    "   SCOR contains the variables to cumulate the energy deposited "
    " local variables "
    $INTEGER IReg; " to store the region number"

    "-----"
    " 1) To get the data from EGSnrc "
    "-----"
    IReg = IR(NP);
    "-----"
    " 2) To test if the particle has been discarded by HOWFAR "
    "-----"
    IF (IARG = 3) [
        " test why it has been discarded "
        IF (IReg = $REG_OUTSIDE) [
            CumulEnergyOutside = CumulEnergyOutside + EDEP;
        ]
        ELSE [
            CumulEnergyLost = CumulEnergyLost + EDEP;
        ]
    ]
    ELSE [
        "-----"
        " 3) To cumulate the energy in the right region "
        "-----"
        IF (IReg = $REG_BONE) [
            CumulEnergyBone = CumulEnergyBone + EDEP;
        ]
        ELSE [
            CumulEnergyMarrow = CumulEnergyMarrow + EDEP;
        ]
    ]
END; " End of subroutine AUSGAB "
"*****"
"                                     SourceBone                                     "
"*****"
"
" The SourceBone subroutine returns a particle starting within the bone "
" regions of the image. The source is isotropic and uniform within the bone "
" region.
" The direction is equiprobable, that means:
"   - Phi is equiprobable within the [0, 2Pi] interval,

```

```

"           - Theta is not equiprobable within [0, Pi], but cos(Theta) is "
"           equiprobable within the [-1, 1] interval. "
" Hence, the Phi and Theta values are (if Rn1 and Rn2 are two random numbers) "
"           Phi = 2*Pi*Rn1 "
"           Theta = arcos(1 - 2*Pi) "
" "
"*****"
SUBROUTINE SourceBone(XSrc,YSrc,ZSrc,USrc,VSrc,WSrc,RegSrc);
$IMPLICIT-NONE; " to make sure that all variables are declared "
" parameters of the routine "
$REAL XSrc; $REAL YSrc; $REAL ZSrc;
$REAL USrc; $REAL VSrc; $REAL WSrc;
$INTEGER RegSrc;
" COMMON variables "
COMIN/RANDOM,GEOM/;
" The above expands into COMMON statements "
" RANDOM contains the RANMAR parameters "
" GEOM contains the image "
" local variables "
$REAL Random1, Random2, Random3;
$REAL Theta, Phi;
" system functions invoked in subroutine "
$REAL DACOS, DCOS, DSIN;
INTRINSIC DACOS, DCOS, DSIN;
" user functions invoked in the subroutine "
LOGICAL InsideBone;

"-----"
" 1) to return the starting position "
"-----"
" The three coordinates are first chosen within the image "
" Then a test checks if it is located within a bone voxel. "
LOOP [ " until the position is inside bone "
$RANDOMSET Random1;
$RANDOMSET Random2;
$RANDOMSET Random3;
XSrc = $VOXEL_SIZE_X * (($IMAGE_NX - 1) * Random1 + 0.5D+00);
YSrc = $VOXEL_SIZE_Y * (($IMAGE_NY - 1) * Random2 + 0.5D+00);
ZSrc = $VOXEL_SIZE_Z * (($IMAGE_NZ - 1) * Random3 + 0.5D+00);
]
UNTIL ( InsideBone(XSrc, YSrc, ZSrc) );
"-----"
" 2) to return the direction "
"-----"
" To choose a random direction. In the spherical coordinate frame: "
" - Phi is equiprobable within the [0, 2Pi] interval "
" - cos(Theta) is equiprobable within the [-1, +1] interval "
$RANDOMSET Random1;
$RANDOMSET Random2;
Theta = DACOS(1 - 2.0D+00*Random1);
Phi = 2.0D+00 * $PI * Random2;
USrc = DSIN(Theta) * DCOS(Phi);
VSrc = DSIN(Theta) * DSIN(Phi);
WSrc = DCOS(Theta);
"-----"
" 3) to return the region number "
"-----"
RegSrc = $REG_BONE;
END; " End of subroutine SourceBone "
"*****"
" SourceMarrow "
"*****"
" The SourceMarrow subroutine returns a particle starting within the marrow "
" regions of the image. The source is isotropic and uniform within the "
" regions. "
" The direction is equiprobable, that means: "
" - Phi is equiprobable within the [0, 2Pi] interval, "
" - Theta is not equiprobable within [0, Pi], but cos(Theta) is "

```

```

"           equiprobable within the [-1, 1] interval. "
" Hence, the Phi and Theta values are (if Rn1 and Rn2 are two random numbers) "
"           Phi = 2*Pi*Rn1 "
"           Theta = arcos(1 - 2*Pi) "
" "
"*****"
SUBROUTINE SourceMarrow(XSrc,YSrc,ZSrc,USrc,VSrc,WSrc,RegSrc);
$IMPLICIT-NONE; " to make sure that all variables are declared "
" parameters of the routine "
$REAL XSrc; $REAL YSrc; $REAL ZSrc;
$REAL USrc; $REAL VSrc; $REAL WSrc;
$INTEGER RegSrc;
" COMMON variables "
COMIN/RANDOM,GEOM/;
" The above expands into COMMON statements "
" RANDOM contains the RANMAR parameters "
" GEOM contains the image "
" local variables "
$REAL Random1, Random2, Random3;
$REAL Theta, Phi;
" system functions invoked in subroutine "
$REAL DACOS, DCOS, DSIN;
INTRINSIC DACOS, DCOS, DSIN;
" user functions invoked in the subroutine "
LOGICAL InsideMarrow;

"-----"
" 1) to return the starting position "
"-----"
" The three coordinates are first chosen within the image "
" Then a test checks if it is located within a marrow voxel. "
LOOP [ " until the position is inside marrow "
$RANDOMSET Random1;
$RANDOMSET Random2;
$RANDOMSET Random3;
XSrc = $VOXEL_SIZE_X * (($IMAGE_NX - 1) * Random1 + 0.5D+00);
YSrc = $VOXEL_SIZE_Y * (($IMAGE_NY - 1) * Random2 + 0.5D+00);
ZSrc = $VOXEL_SIZE_Z * (($IMAGE_NZ - 1) * Random3 + 0.5D+00);
]
UNTIL ( InsideMarrow(XSrc, YSrc, ZSrc) );
"-----"
" 2) to return the direction "
"-----"
" To choose a random direction. In the spherical coordinate frame: "
" - Phi is equiprobable within the [0, 2Pi] interval "
" - cos(Theta) is equiprobable within the [-1, +1] interval "
$RANDOMSET Random1;
$RANDOMSET Random2;
Theta = DACOS(1 - 2.0D+00*Random1);
Phi = 2.0D+00 * $PI * Random2;
USrc = DSIN(Theta) * DCOS(Phi);
VSrc = DSIN(Theta) * DSIN(Phi);
WSrc = DCOS(Theta);
"-----"
" 3) to return the region number "
"-----"
RegSrc = $REG_MARROW;
END; " End of subroutine SourceMarrow "
"*****"
"           Function InsideBone "
"*****"
" Test if a given position (X, Y, Z) is on the bone side of the isosurface. "
" "
" Input: "
" . X, Y, Z: the position to be tested. "
" "
" Return: "
" .TRUE. if the position is inside the region. "

```

```

"      .FALSE. if the position is not inside the region.      "
"
"*****"
LOGICAL FUNCTION InsideBone(X, Y, Z);
  $IMPLICIT-NONE;      " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL X, Y, Z;
  " local variables "
  $INTEGER I, J, K;      " to store the position of the cube "
  $INTEGER B0, B1, B2, B3, B4, B5, B6, B7; " vertex gray levels "
  $REAL LocalX, LocalY, LocalZ; " in the coordinate system of the cube "
  $REAL Bxyz;      " Gray level value interpolated at (X,Y,Z) "
  " user functions invoked in the subroutine "
  $INTEGER GrayLevelOf;
  LOGICAL InsideROI;

  "-----"
  " 1) to check if (X, Y, Z) is inside the image "
  "-----"
  IF (~InsideROI(X, Y, Z)) [
    InsideBone = .FALSE.;
  ]
  ELSE [
    "-----"
    " 2) to calculate the cube number in the image "
    "-----"
    I = (X / $VOXEL_SIZE_X) + 0.5D+00;
    J = (Y / $VOXEL_SIZE_Y) + 0.5D+00;
    K = (Z / $VOXEL_SIZE_Z) + 0.5D+00;
    "-----"
    " 3) to get the gray levels of each vertex "
    "-----"
    B0 = GrayLevelOf(I , J , K );
    B1 = GrayLevelOf(I+1, J , K );
    B2 = GrayLevelOf(I+1, J+1, K );
    B3 = GrayLevelOf(I , J+1, K );
    B4 = GrayLevelOf(I , J , K+1);
    B5 = GrayLevelOf(I+1, J , K+1);
    B6 = GrayLevelOf(I+1, J+1, K+1);
    B7 = GrayLevelOf(I , J+1, K+1);
    "-----"
    " 4) to check if pure bone or pure marrow within the cube "
    "-----"
    IF ( B0 > $THRESHOLD & B1 > $THRESHOLD &
        B2 > $THRESHOLD & B3 > $THRESHOLD &
        B4 > $THRESHOLD & B5 > $THRESHOLD &
        B6 > $THRESHOLD & B7 > $THRESHOLD ) [
      InsideBone = .FALSE.;
    ]
    ELSEIF ( B0 < $THRESHOLD & B1 < $THRESHOLD &
            B2 < $THRESHOLD & B3 < $THRESHOLD &
            B4 < $THRESHOLD & B5 < $THRESHOLD &
            B6 < $THRESHOLD & B7 < $THRESHOLD ) [
      InsideBone = .TRUE.;
    ]
    ELSE [
      "-----"
      " 5) cube on isosurface, check on which side "
      "-----"
      " a) to calculate the coordinates relative to the cube "
      " In this new coordinate system the cube has unity size. "
      LocalX = (X - (I - 0.5D+00)*$VOXEL_SIZE_X) / $VOXEL_SIZE_X;
      LocalY = (Y - (J - 0.5D+00)*$VOXEL_SIZE_Y) / $VOXEL_SIZE_Y;
      LocalZ = (Z - (K - 0.5D+00)*$VOXEL_SIZE_Z) / $VOXEL_SIZE_Z;
      " b) get the gray level at (X,Y,Z) by trilinear interpolation "
      Bxyz = B0*(1.0D+00 - LocalX)*(1.0D+00 - LocalY)*(1.0D+00 - LocalZ) +
            B1*      LocalX *(1.0D+00 - LocalY)*(1.0D+00 - LocalZ) +
            B2*      LocalX *      LocalY *(1.0D+00 - LocalZ) +
            B3*(1.0D+00 - LocalX)*      LocalY *(1.0D+00 - LocalZ) +

```

```

        B4*(1.0D+00 - LocalX)*(1.0D+00 - LocalY)*      LocalZ +
        B5*          LocalX *(1.0D+00 - LocalY)*      LocalZ +
        B6*          LocalX *   LocalY *              LocalZ +
        B7*(1.0D+00 - LocalX)*          LocalY *      LocalZ;
    " c) check on which side of the isosurface "
    IF (Bxyz > $THRESHOLD) [
        InsideBone = .FALSE.;
    ]
    ELSE [
        InsideBone = .TRUE.;
    ]
]
END; " End of function InsideBone "
*****
"                               Function InsideMarrow                               "
*****
"
" Test if a given position (X, Y, Z) is on the marrow side of the isosurface. "
"
" Input:
"   . X, Y, Z: the position to be tested.
"
" Return:
"   .TRUE. if the position is inside the region.
"   .FALSE. if the position is not inside the region.
"
*****
LOGICAL FUNCTION InsideMarrow(X, Y, Z);
    $IMPLICIT-NONE; " to make sure that all variables are declared "
    " parameters of the routine "
    $REAL X, Y, Z;
    " local variables "
    $INTEGER I, J, K; " to store the position of the cube "
    $INTEGER B0, B1, B2, B3, B4, B5, B6, B7; " vertex gray levels "
    $REAL LocalX, LocalY, LocalZ; " in the coordinate system of the cube "
    $REAL Bxyz; " Gray level value interpolated at (X,Y,Z) "
    " user functions invoked in the subroutine "
    $INTEGER GrayLevelOf;
    LOGICAL InsideROI;

    "-----"
    " 1) to check if (X, Y, Z) is inside the image "
    "-----"
    IF (~InsideROI(X, Y, Z)) [
        InsideMarrow = .FALSE.;
    ]
    ELSE [
        "-----"
        " 2) to calculate the cube number in the image "
        "-----"
        I = (X / $VOXEL_SIZE_X) + 0.5D+00;
        J = (Y / $VOXEL_SIZE_Y) + 0.5D+00;
        K = (Z / $VOXEL_SIZE_Z) + 0.5D+00;
        "-----"
        " 3) to get the gray levels of each vertex "
        "-----"
        B0 = GrayLevelOf(I , J , K );
        B1 = GrayLevelOf(I+1, J , K );
        B2 = GrayLevelOf(I+1, J+1, K );
        B3 = GrayLevelOf(I , J+1, K );
        B4 = GrayLevelOf(I , J , K+1);
        B5 = GrayLevelOf(I+1, J , K+1);
        B6 = GrayLevelOf(I+1, J+1, K+1);
        B7 = GrayLevelOf(I , J+1, K+1);
        "-----"
        " 4) to check if pure bone or pure marrow within the cube "
        "-----"
        IF ( B0 > $THRESHOLD & B1 > $THRESHOLD &

```

```

        B2 > $THRESHOLD & B3 > $THRESHOLD &
        B4 > $THRESHOLD & B5 > $THRESHOLD &
        B6 > $THRESHOLD & B7 > $THRESHOLD ) [
    InsideMarrow = .TRUE.;
]
ELSEIF ( B0 < $THRESHOLD & B1 < $THRESHOLD &
        B2 < $THRESHOLD & B3 < $THRESHOLD &
        B4 < $THRESHOLD & B5 < $THRESHOLD &
        B6 < $THRESHOLD & B7 < $THRESHOLD ) [
    InsideMarrow = .FALSE.;
]
ELSE [
    "-----"
    " 5) cube on isosurface, check on which side "
    "-----"
    " a) to calculate the coordinates relative to the cube "
    "   In this new coordinate system the cube has unity size. "
    LocalX = (X - (I - 0.5D+00)*$VOXEL_SIZE_X) / $VOXEL_SIZE_X;
    LocalY = (Y - (J - 0.5D+00)*$VOXEL_SIZE_Y) / $VOXEL_SIZE_Y;
    LocalZ = (Z - (K - 0.5D+00)*$VOXEL_SIZE_Z) / $VOXEL_SIZE_Z;
    " b) get the gray level at (X,Y,Z) by trilinear interpolation "
    Bxyz = B0*(1.0D+00 - LocalX)*(1.0D+00 - LocalY)*(1.0D+00 - LocalZ) +
           B1*      LocalX *(1.0D+00 - LocalY)*(1.0D+00 - LocalZ) +
           B2*      LocalX *      LocalY *(1.0D+00 - LocalZ) +
           B3*(1.0D+00 - LocalX)*      LocalY *(1.0D+00 - LocalZ) +
           B4*(1.0D+00 - LocalX)*(1.0D+00 - LocalY)*      LocalZ +
           B5*      LocalX *(1.0D+00 - LocalY)*      LocalZ +
           B6*      LocalX *      LocalY *      LocalZ +
           B7*(1.0D+00 - LocalX)*      LocalY *      LocalZ;
    " c) check on which side of the isosurface "
    IF (Bxyz > $THRESHOLD) [
        InsideMarrow = .TRUE.;
    ]
    ELSE [
        InsideMarrow = .FALSE.;
    ]
]
]
END; " End of function InsideMarrow "
*****
"                               Function InsiderOI                               "
*****
"
" Test if a given position (X, Y, Z) is inside the limits of the Region Of
" Interest. In the trilinear interpolation technique, the cubes vertices are
" the centers of the voxels. Therefore, the study has to exclude the most
" external half voxel.
"
" Input:
"   . X, Y, Z: the position to be tested.
"
" Return:
"   .TRUE. if the position is inside the image.
"   .FALSE. if the position is not inside the image.
"
*****
LOGICAL FUNCTION InsiderOI(X, Y, Z);
    $IMPLICIT-NONE; " to make sure that all variables are declared "
    " parameters of the routine "
    $REAL X, Y, Z;

    " The limits extend to the limits of the Marching Cubes. Therefore, "
    " a half voxel is excluded from the ROI. "
    IF ( (X >= 0.5D+00*$VOXEL_SIZE_X) &
        (X < ($IMAGE_NX - 0.5D+00)*$VOXEL_SIZE_X) &
        (Y >= 0.5D+00*$VOXEL_SIZE_Y) &
        (Y < ($IMAGE_NY - 0.5D+00)*$VOXEL_SIZE_Y) &
        (Z >= 0.5D+00*$VOXEL_SIZE_Z) &
        (Z < ($IMAGE_NZ - 0.5D+00)*$VOXEL_SIZE_Z) ) [

```

```

        InsideROI = .TRUE.;
    ]
ELSE [
    InsideROI = .FALSE.;
]
END; " End of function InsideROI "
*****
"                               Function DistanceToExitLocalCell                               "
*****
"
" Returns the distance from the position (X, Y, Z) to the nearest boundary "
" of the region when following the direction (U, V, W). The boundary can be "
" either the cube limit or the isosurface, whichever is reached first. "
"
" Input: "
"   . X, Y, Z: the position to be tested. "
"   . U, V, W: the direction to follow. "
"
" Return: "
"   . the distance to the boundary. "
"   . the region on the other side of the boundary. "
"
*****
$REAL FUNCTION DistanceToExitLocalCell(X, Y, Z, U, V, W, RegNew);
  $IMPLICIT-NONE; " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL X, Y, Z;
  $REAL U, V, W;
  $INTEGER RegNew;
  " local variables "
  $INTEGER I, J, K; " to store the position of the cube "
  $INTEGER B0, B1, B2, B3, B4, B5, B6, B7; " vertex gray levels "
  $REAL Distance;
  $REAL ShortestDistance;
  $REAL XMin, YMin, ZMin; " for the boundary of the cube "
  $REAL XMax, YMax, ZMax; " for the boundary of the cube "
  $INTEGER INew, JNew, KNew; " the position of the voxel beyond the boundary "
  $REAL LocalX, LocalY, LocalZ; " in the coordinate system of the cube "
  $REAL a, b, c, d; " coef. of the 3rd degree eqn. "
  $REAL DeltaB; " temp. variable to calculate eqn. "
  $REAL DeltaX, DeltaY, DeltaZ; " temp. variable to calculate eqn. "
  $REAL DeltaXY, DeltaYZ, DeltaZX; " temp. variable to calculate eqn. "
  $INTEGER NRoot;
  $REAL Root(3);
  $INTEGER RootNo;
  $INTEGER RegBeyond;
  " user functions invoked in the subroutine "
  $INTEGER GrayLevelOf;

  "-----"
  " 1) to calculate the boundary of the current cube "
  "-----"
  I = (X / $VOXEL_SIZE_X) + 0.5D+00;
  J = (Y / $VOXEL_SIZE_Y) + 0.5D+00;
  K = (Z / $VOXEL_SIZE_Z) + 0.5D+00;
  XMin = (I - 0.5D+00) * $VOXEL_SIZE_X;
  XMax = XMin + $VOXEL_SIZE_X;
  YMin = (J - 0.5D+00) * $VOXEL_SIZE_Y;
  YMax = YMin + $VOXEL_SIZE_Y;
  ZMin = (K - 0.5D+00) * $VOXEL_SIZE_Z;
  ZMax = ZMin + $VOXEL_SIZE_Z;
  "-----"
  " 2) to calculate the distance to the boundary of the cube "
  "-----"
  ShortestDistance = $INFINITY;
  " a) along the X axis "
  IF ( U > 0.0 ) [
    Distance = (XMax - X) / U;
    IF (Distance < ShortestDistance) [

```

```

        ShortestDistance = Distance;
        INew = I + 1;
        JNew = J;
        KNew = K;
    ]
]
ELSEIF ( U < 0.0 ) [
    Distance = (XMin - X) / U;
    IF (Distance < ShortestDistance) [
        ShortestDistance = Distance;
        INew = I - 1;
        JNew = J;
        KNew = K;
    ]
]
ELSE [ " particle goes perpendicular to (0,x), nothing to do "
]
" b) along the Y axis "
IF ( V > 0.0 ) [
    Distance = (YMax - Y) / V;
    IF (Distance < ShortestDistance) [
        ShortestDistance = Distance;
        INew = I;
        JNew = J + 1;
        KNew = K;
    ]
]
ELSEIF ( V < 0.0 ) [
    Distance = (YMin - Y) / V;
    IF (Distance < ShortestDistance) [
        ShortestDistance = Distance;
        INew = I;
        JNew = J - 1;
        KNew = K;
    ]
]
ELSE [ " particle goes perpendicular to (0,y), nothing to do "
]
" c) along the Z axis "
IF ( W > 0.0 ) [
    Distance = (ZMax - Z) / W;
    IF (Distance < ShortestDistance) [
        ShortestDistance = Distance;
        INew = I;
        JNew = J;
        KNew = K + 1;
    ]
]
ELSEIF ( W < 0.0 ) [
    Distance = (ZMin - Z) / W;
    IF (Distance < ShortestDistance) [
        ShortestDistance = Distance;
        INew = I;
        JNew = J;
        KNew = K - 1;
    ]
]
ELSE [ " particle goes perpendicular to (0,z), nothing to do "
]
"-----"
" 3) to set teh region beyond the cube limit "
"-----"
" if there is no isosurface to cross, the other side is in the "
" same region as the current side, unless it is outside "
RegBeyond = RegNew;
IF ( INew <= 0 | INew >= $IMAGE_NX |
    JNew <= 0 | JNew >= $IMAGE_NY |
    KNew <= 0 | KNew >= $IMAGE_NZ ) [
    RegBeyond = $REG_OUTSIDE;
]

```

```

]
"-----"
" 4) to check if the cube is on the isosurface "
"-----"
" a) to get the gray levels of each vertex "
B0 = GrayLevelOf(I , J , K );
B1 = GrayLevelOf(I+1, J , K );
B2 = GrayLevelOf(I+1, J+1, K );
B3 = GrayLevelOf(I , J+1, K );
B4 = GrayLevelOf(I , J , K+1);
B5 = GrayLevelOf(I+1, J , K+1);
B6 = GrayLevelOf(I+1, J+1, K+1);
B7 = GrayLevelOf(I , J+1, K+1);
" b) to check if the cube is on the isosurface "
IF ( ( B0 > $THRESHOLD & B1 > $THRESHOLD &
      B2 > $THRESHOLD & B3 > $THRESHOLD &
      B4 > $THRESHOLD & B5 > $THRESHOLD &
      B6 > $THRESHOLD & B7 > $THRESHOLD ) |
      ( B0 < $THRESHOLD & B1 < $THRESHOLD &
      B2 < $THRESHOLD & B3 < $THRESHOLD &
      B4 < $THRESHOLD & B5 < $THRESHOLD &
      B6 < $THRESHOLD & B7 < $THRESHOLD ) ) [
" if it is not, the distance is the one from the cube limits "
" and the region changes only if the particle goes outside. "
" It is already done. "
]
ELSE [
"-----"
" 5) to calculate the distance to the trilinear interpolated isosurface "
"-----"
" a) to calculate the location (X, Y, Z) relatively to the cube "
" In this new coordinate system the cube has not changed its size. "
" Just a shift is applied so that the new origin is the vertex 0 "
" of the cube. The size change cannot be applied here, because "
" the resizing would not preserve the distances. "
LocalX = X - (I - 0.5D+00)*$VOXEL_SIZE_X;
LocalY = Y - (J - 0.5D+00)*$VOXEL_SIZE_Y;
LocalZ = Z - (K - 0.5D+00)*$VOXEL_SIZE_Z;
" b) to calculate the intersection of the isosurface with the direction "
" It's a third degree equation. The coefficients a, b, c, and d are "
" defined as (check Maple file on PC to see the calculation): "
DeltaB = B0 + B2 + B5 + B7 - B1 - B3 - B4 - B6;
DeltaX = B0 + B7 - B3 - B4;
DeltaY = B0 + B5 - B1 - B4;
DeltaZ = B0 + B2 - B1 - B3;
DeltaXY = B0 - B4;
DeltaYZ = B0 - B1;
DeltaZX = B0 - B3;
a = U*V*W*DeltaB;
b = U*V*(DeltaB*LocalZ - DeltaZ*$VOXEL_SIZE_Z)
+ V*W*(DeltaB*LocalX - DeltaX*$VOXEL_SIZE_X)
+ W*U*(DeltaB*LocalY - DeltaY*$VOXEL_SIZE_Y);
c = U*(DeltaB*LocalY*LocalZ - DeltaY*LocalZ*$VOXEL_SIZE_Y
- DeltaZ*LocalY*$VOXEL_SIZE_Z + DeltaYZ*$VOXEL_SIZE_Y*$VOXEL_SIZE_Z)
+ V*(DeltaB*LocalZ*LocalX - DeltaZ*LocalX*$VOXEL_SIZE_Z
- DeltaX*LocalZ*$VOXEL_SIZE_X + DeltaZX*$VOXEL_SIZE_Z*$VOXEL_SIZE_X)
+ W*(DeltaB*LocalX*LocalY - DeltaX*LocalY*$VOXEL_SIZE_X
- DeltaY*LocalX*$VOXEL_SIZE_Y + DeltaXY*$VOXEL_SIZE_X*$VOXEL_SIZE_Y);
d = ($THRESHOLD - B0)*$VOXEL_SIZE_X*$VOXEL_SIZE_Y*$VOXEL_SIZE_Z
+ DeltaB*LocalX*LocalY*LocalZ
+ DeltaXY*LocalZ*$VOXEL_SIZE_X*$VOXEL_SIZE_Y
- DeltaX*LocalY*LocalZ*$VOXEL_SIZE_X
+ DeltaYZ*LocalX*$VOXEL_SIZE_Y*$VOXEL_SIZE_Z
- DeltaY*LocalZ*LocalX*$VOXEL_SIZE_Y
+ DeltaZX*LocalY*$VOXEL_SIZE_Z*$VOXEL_SIZE_X
- DeltaZ*LocalX*LocalY*$VOXEL_SIZE_Z;
" c) to solve the third degree equation "
CALL ThirdDegreeEqn(a, b, c, d, NRoot, Root);
" d) to get the shortest distance to the isosurface if it is shorter "

```

```

"    than the distance to the cube limits found previously "
DO RootNo=1, NRoot [
  Distance = Root(RootNo);
  IF (Distance >= 0.0) [
    IF (Distance < ShortestDistance) [
      ShortestDistance = Distance;
      " we also need to switch the medium "
      IF ( RegNew = $REG_BONE ) [
        RegBeyond = $REG_MARROW;
      ]
      ELSE [
        RegBeyond = $REG_BONE;
      ]
    ]
  ]
]
]

"-----"
" 6) to return the distance and the new region number "
"-----"

RegNew = RegBeyond;
DistanceToExitLocalCell = ShortestDistance;
END; " End of function DistanceToExitLocalCell "
"*****"
"          Function ShortestDistanceToLocalCellBoundary          "
"*****"
" Returns the shortest distance from the position (X, Y, Z) to the nearest "
" boundary of the cube "
" "
" Input: "
"   . X, Y, Z: the position to be tested. "
" "
" Return: "
"   . the shortest distance to the boundary. "
" "
"*****"
$REAL FUNCTION ShortestDistanceToLocalCellBoundary(X, Y, Z);
  $IMPLICIT-NONE; " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL X, Y, Z;
  " local variables "
  $INTEGER I, J, K; " to store the position of the cube "
  $INTEGER B0, B1, B2, B3, B4, B5, B6, B7; " vertex gray levels "
  $REAL Distance;
  $REAL ShortestDistance;
  $REAL XMin, YMin, ZMin; " for the boundary of the cube "
  $REAL XMax, YMax, ZMax; " for the boundary of the cube "
  " user functions invoked in the subroutine "
  $INTEGER GrayLevelOf;

"-----"
" 1) to calculate the cube number in the image "
"-----"
I = (X / $VOXEL_SIZE_X) + 0.5D+00;
J = (Y / $VOXEL_SIZE_Y) + 0.5D+00;
K = (Z / $VOXEL_SIZE_Z) + 0.5D+00;
"-----"
" 2) to get the gray levels of each vertex "
"-----"
B0 = GrayLevelOf(I , J , K );
B1 = GrayLevelOf(I+1, J , K );
B2 = GrayLevelOf(I+1, J+1, K );
B3 = GrayLevelOf(I , J+1, K );
B4 = GrayLevelOf(I , J , K+1);
B5 = GrayLevelOf(I+1, J , K+1);
B6 = GrayLevelOf(I+1, J+1, K+1);
B7 = GrayLevelOf(I , J+1, K+1);
"-----"

```

```

" 3) to check if pure bone or pure marrow "
"-----"
IF ( ( B0 > $THRESHOLD & B1 > $THRESHOLD &
      B2 > $THRESHOLD & B3 > $THRESHOLD &
      B4 > $THRESHOLD & B5 > $THRESHOLD &
      B6 > $THRESHOLD & B7 > $THRESHOLD ) |
      ( B0 < $THRESHOLD & B1 < $THRESHOLD &
      B2 < $THRESHOLD & B3 < $THRESHOLD &
      B4 < $THRESHOLD & B5 < $THRESHOLD &
      B6 < $THRESHOLD & B7 < $THRESHOLD ) ) [
"-----"
" 4) cube is not on isosurface; return the distance to the cube "
"-----"
" a) to calculate the limits of the current cube "
XMin = ( I - 0.5D+00 ) * $VOXEL_SIZE_X;
XMax = XMin + $VOXEL_SIZE_X;
YMin = ( J - 0.5D+00 ) * $VOXEL_SIZE_Y;
YMax = YMin + $VOXEL_SIZE_Y;
ZMin = ( K - 0.5D+00 ) * $VOXEL_SIZE_Z;
ZMax = ZMin + $VOXEL_SIZE_Z;
" b) to measure the distance to the limit of the cube "
ShortestDistance = $INFINITY;
Distance = X - XMin;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
Distance = XMax - X;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
Distance = Y - YMin;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
Distance = YMax - Y;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
Distance = Z - ZMin;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
Distance = ZMax - Z;
IF ( Distance < ShortestDistance ) [ ShortestDistance = Distance; ]
" c) to return the distance "
ShortestDistanceToLocalCellBoundary = ShortestDistance;
]
ELSE [
"-----"
" 5) cube is on isosurface; return the distance to the isosurface "
"-----"
" the calcul is too complex. Therefore, returns 0.0 "
ShortestDistanceToLocalCellBoundary = 0.0;
]
END; " End of function ShortestDistanceToLocalCellBoundary "
*****
"                               Function GrayLevelOf                               "
*****
" Returns the gray level of the voxel (I, J, K). "
" "
" Input: "
"   . I, J, K: the position of the voxel in the image. "
" "
" Return: "
"   The gray level of the voxel "
"   -1 if it is outside the limit of the image "
" "
*****
$INTEGER FUNCTION GrayLevelOf(I, J, K);
$IMPLICIT-NONE; " to make sure that all variables are declared "
" parameters of the routine "
$INTEGER I, J, K;
" COMMON variables "
COMIN/GEOM/;
" The above expands into COMMON statements "
" GEOM contains the image "
" local variables "
$INTEGER VoxelNo; " to store the position of the voxel within the file "

```

```

$INTEGER VoxelValue; " the gray level of the voxel resd "

IF ( I <= 0 | I > $IMAGE_NX |
    J <= 0 | J > $IMAGE_NY |
    K <= 0 | K > $IMAGE_NZ ) [
    GrayLevelOf = -1;
]
ELSE [
    VoxelNo = ((K-1)*$IMAGE_NY + (J-1))*$IMAGE_NX + (I-1) + 1;
    VoxelValue = BoneImage(VoxelNo);
    IF (VoxelValue < 0) [
        GrayLevelOf = 256 + VoxelValue;
    ]
    ELSE [
        GrayLevelOf = VoxelValue;
    ]
]
END; " End of function GrayLevelOf "
*****
"                                     ThirdDegreeEqn                                     "
*****
" Solves a third degree equation. The technique used here is given in: "
" W. H. Press et al. Numerical recipes in C. 2nd Ed. Cambridge University "
" Press; 1992. p. 183. "
" "
" Input: "
" . a, b, c, d: the coefficients of the equation axxx + bxx + cx + d = 0 "
" "
" Return: "
" . NRoot: the number of solutions. "
" . Root: an array that contains the list of solutions (0, 1, 2, or 3) "
*****
SUBROUTINE ThirdDegreeEqn(a, b, c, d, NRoot, Root);
$IMPLICIT-NONE; " to make sure that all variables are declared "
" parameters of the routine "
$REAL a, b, c, d;
$INTEGER NRoot;
$REAL Root(1);
" local variables "
$REAL p, q, r;
$REAL BigQ, BigR;
$REAL Delta;
$REAL BigA, BigB;
$REAL Theta, TwoSqrtQ;
" system functions invoked in the subroutine "
$REAL DSQRT, DACOS, DCOS;
INTRINSIC DSQRT, DACOS, DCOS;

"-----"
" 1) to test if it is a third degree equation "
"-----"
IF (a = 0.0) [ " to solve bxx + cx + d = 0 "
    CALL SecondDegreeEqn(b, c, d, NRoot, Root);
]
ELSE [
    IF (d = 0.0) [
        IF (c = 0.0) [
            IF (b = 0.0) [ " to solve axxx = 0 "
                Root(1) = 0.0;
                NRoot = 1;
            ]
            ELSE [ " to solve (ax + b)xx = 0 "
                Root(1) = -b / a;
                Root(2) = 0.0;
                NRoot = 2;
            ]
        ]
    ]
]
]

```

```

ELSE [ " to solve (axx + bx + c)x = 0 "
      CALL SecondDegreeEqn(a, b, c, NRoot, Root);
      Root(NRoot + 1) = 0.0;
      NRoot = NRoot + 1;
    ]
]
ELSE [
  "-----"
  " 2) to calculate the solutions of axxx + bxx + cx + d = 0 "
  "-----"
  " a) to transform the equation into xxx + pxx + qx + r = 0 "
  " so that we can use the same formulae as in W. H. Press'book "
  " with a becoming p, b becoming q, and c becoming r "
  p = b / a;
  q = c / a;
  r = d / a;
  " b) to follow W. H. Press'method (p<=>a, q<=>b, and r<=>c "
  BigQ = (p*p - 3.0D+00*q) / 9.0D+00;
  BigR = (2.0D+00*p*p*p - 9.0D+00*p*q + 27.0D+00*r) / 54.0D+00;
  Delta = BigR*BigR - BigQ*BigQ*BigQ;
  IF (Delta < 0.0) [ " 3 real solutions "
    Theta = DACOS(BigR / (BigQ**1.5D+00));
    TwoSqrtQ = 2.0D+00 * DSQRT(BigQ);
    Root(1) = - TwoSqrtQ*DCOS(Theta / 3.0D+00)
              - (p / 3.0D+00);
    Root(2) = - TwoSqrtQ*DCOS((Theta + 2.0D+00*$PI) / 3.0D+00)
              - (p / 3.0D+00);
    Root(3) = - TwoSqrtQ*DCOS((Theta - 2.0D+00*$PI) / 3.0D+00)
              - (p / 3.0D+00);
    NRoot = 3;
  ]
  ELSE [ " 1 or 2 real solutions "
    IF (BigR >= 0.0) [
      BigA = - ( BigR + DSQRT(Delta))**(1.0D+00/3.0D+00);
    ]
    ELSE [
      BigA = (-BigR + DSQRT(Delta))**(1.0D+00/3.0D+00);
    ]
    IF (BigA = 0.0) [
      BigB = 0.0;
    ]
    ELSE [
      BigB = BigQ / BigA;
    ]
    Root(1) = BigA + BigB - (p / 3.0D+00);
    NRoot = 1;
    IF (Delta = 0.0) [ " 2 real solutions "
      Root(NRoot + 1) = -0.5D+00*(BigA + BigB) - (p / 3.0D+00);
      NRoot = NRoot + 1;
    ]
  ]
]
]
END; " End of subroutine ThirdDegreeEqn "
*****
"                               SecondDegreeEqn                               "
*****
"
" Solves a second degree equation. The technique used here is given in:
" W. H. Press et al. Numerical recipes in C. 2nd Ed. Cambridge University
" Press; 1992. p. 183.
"
" Input:
"   . a, b, c: the coefficients of the equation axx + bx + c = 0
"
" Return:
"   . NRoot: the number of solutions.
"   . Root: an array that contains the list of solutions (0, 1, or 2)
"

```

```

*****
SUBROUTINE SecondDegreeEqn(a, b, c, NRoot, Root);
  $IMPLICIT-NONE;    " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL a, b, c;
  $INTEGER NRoot;
  $REAL Root(1);
  " local variables "
  $REAL Delta, q;
  " system functions invoked in the subroutine "
  $REAL DSQRT;
  INTRINSIC DSQRT;

  "-----"
  " 1) to test if it is a second degree equation "
  "-----"
  IF (a = 0.0) [ " to solve bx + c = 0 "
    CALL FirstDegreeEqn(b, c, NRoot, Root);
  ]
  ELSE [
    IF (c = 0.0) [
      IF (b = 0.0) [ " to solve axx = 0 "
        Root(1) = 0.0;
        NRoot = 1;
      ]
      ELSE [ " to solve (ax+b)x = 0 "
        Root(1) = -b / a;
        Root(2) = 0.0;
        NRoot = 2;
      ]
    ]
    ELSE [
      "-----"
      " 2) to calculate the solutions of axx + bx + c = 0 "
      "-----"
      Delta = b*b - 4.0D+00*a*c;
      IF (Delta < 0.0) [ " 0 solution "
        NRoot = 0;
      ]
      ELSEIF (Delta = 0.0) [ " 1 solution "
        Root(1) = -b / (2.0D+00*a);
        NRoot = 1;
      ]
      ELSE [ " 2 solutions "
        IF (b <= 0.0) [ " if b = 0.0 then 2 opposed solutions (x1=-x2) "
          q = -0.5D+00 * (b - DSQRT(Delta));
        ]
        ELSE [
          q = -0.5D+00 * (b + DSQRT(Delta));
        ]
        Root(1) = q / a;
        Root(2) = c / q;
        NRoot = 2;
      ]
    ]
  ]
]
END; " End of subroutine SecondDegreeEqn "
*****
"                               FirstDegreeEqn                               "
*****
" Solves a first degree equation.                                           "
" Input:                                                                     "
"   . a, b: the coefficients of the equation ax + b = 0                     "
" Return:                                                                     "
"   . NRoot: the number of solutions.                                        "
"   . Root: an array that contains the list of solutions (0 or 1)          "

```

```

"
"*****"
SUBROUTINE FirstDegreeEqn(a, b, NRoot, Root);
  $IMPLICIT-NONE; " to make sure that all variables are declared "
  " parameters of the routine "
  $REAL a, b;
  $INTEGER NRoot;
  $REAL Root(1);

  "-----"
  " 1) to check if a solution exists "
  "-----"
  IF (a = 0.0) [
    NRoot = 0;
  ]
  ELSE [
    "-----"
    " 2) to calculate the solution "
    "-----"
    Root(1) = -b / a;
    NRoot = 1;
  ]
END; " End of subroutine FirstDegreeEqn "

```

Configuration File

```

#!/bin/csh
#       standard.configuration (SID 1.6 last edited 00/02/27)
# -----
#
#       When an EGSnrc code is compiled, the egs_compile script
#       calls this file to create the overall source code by
#       concatenating different files in a specified order. Generally
#       user codes will require extensions to this bare bones configuration
#       file. See the examples for dosrznrc etc.
#       If no configuration file is present on the user-code directory
#       (i.e. $HOME/egsnrc/user_code) then this file is used.
#
#       Note that order IS IMPORTANT since the last definition of a macro
#       is the one that is used.
#
#       catecho is a simple little script to concatonate the named file
#       and echo things to the terminal depending on whether EGS_PERT
#       is set or not.
#
#       The EGSnrc system has been structured to work with either the
#       RANLUX or the RANMAR random number generators. To switch
#       which rng to use, comment out the one not wanted (in 2 places).
#       The only difference to the user is that RANLUX requires a
#       luxury level (0 to 4) plus an initial seed (any positive integer)
#       whereas RANMAR needs two initial seeds between 1 and roughly 30,000.
#
#       This is part of the EGSnrc Code System
#       Copyright NRC 2000

echo "Entering $HEN_HOUSE/standard.configuration (SID 1.6)      "
echo "-----"
echo " "
echo " Using machine: $my_machine"
echo " "

echo "%L"                > .mortjob.mortran # Mortran switch to turn listing on
$HEN_HOUSE/catecho "$HEN_HOUSE/egsnrc.macros      " "egsnrc standard macros"
$HEN_HOUSE/catecho "$HEN_HOUSE/lib/$my_machine/machine.mortran" "machine macros"
#$HEN_HOUSE/catecho "$HEN_HOUSE/ranlux.macros    " "RNG macros"
$HEN_HOUSE/catecho "$HEN_HOUSE/ranmar.macros    " "RNG macros"

if ($?EGS_PERT != 1) echo "-----"

$HEN_HOUSE/catecho "$1.mortran                    " "user-code source"

if ($?EGS_PERT != 1) echo "-----"
#$HEN_HOUSE/catecho "$HEN_HOUSE/ranlux.mortran"    "RNG initialization"
$HEN_HOUSE/catecho "$HEN_HOUSE/ranmar.mortran"    "RNG initialization"

$HEN_HOUSE/catecho "$HEN_HOUSE/nrcaux.mortran     " "NRC auxilliary subs"
$HEN_HOUSE/catecho "$HEN_HOUSE/egsnrc.mortran    " "egsnrc subroutines"

echo " "
echo "-----"
echo "end of standard.configuration(SID 1.6). .mortan.mortjob created"
echo "-----"
echo " "

```

Input File

particle type	particle energy	histories
-1	0.025	4000000
-1	0.050	1500000
-1	0.075	750000
-1	0.100	400000
-1	0.150	160000
-1	0.200	100000
-1	0.250	90000
-1	0.300	40000
-1	0.350	30000
-1	0.400	28000
-1	0.450	24000
-1	0.500	20000
-1	0.600	17000
-1	0.700	15000
-1	0.800	15000

Output File

Absorbed fractions for irradiation from bone trabeculae.

Configuration No: 1

The calculation is performed for:

100 runs
40000 electrons per run
Total: 4000000 histories.
Initial kinetic energy: 0.025 MeV.

Absorbed fractions with 95% confidence intervals:

AF in bone:	0.99331129452813	+/- 0.00006901631965	(0.01%)
AF in marrow:	0.00602967415125	+/- 0.00006611645159	(1.10%)
AF outside:	0.00065686515317	+/- 0.00002202400057	(3.35%)
AF lost:	0.00000216616733	+/- 0.00000120616436	(55.68%)
Total AF:	0.99999999999989		

Configuration No: 2

The calculation is performed for:

100 runs
15000 electrons per run
Total: 1500000 histories.
Initial kinetic energy: 0.050 MeV.

Absorbed fractions with 95% confidence intervals:

AF in bone:	0.97749883116706	+/- 0.00021088383632	(0.02%)
AF in marrow:	0.02018993325492	+/- 0.00019559548235	(0.97%)
AF outside:	0.00230376700932	+/- 0.00006434810437	(2.79%)
AF lost:	0.00000746856873	+/- 0.00000356621659	(47.75%)
Total AF:	1.00000000000003		

Configuration No: 3

The calculation is performed for:

100 runs
7500 electrons per run
Total: 750000 histories.
Initial kinetic energy: 0.075 MeV.

Absorbed fractions with 95% confidence intervals:

AF in bone:	0.95491201423482	+/- 0.00040740401832	(0.04%)
AF in marrow:	0.04031746338839	+/- 0.00038150759641	(0.95%)
AF outside:	0.00476833823425	+/- 0.00013964192324	(2.93%)
AF lost:	0.00000218414255	+/- 0.00000254679064	(116.60%)
Total AF:	1.00000000000002		

Configuration No: 4

The calculation is performed for:

100 runs
4000 electrons per run
Total: 400000 histories.
Initial kinetic energy: 0.100 MeV.

Absorbed fractions with 95% confidence intervals:

AF in bone:	0.92854649007421	+/- 0.00066711400750	(0.07%)
AF in marrow:	0.06392629697798	+/- 0.00059583728289	(0.93%)
AF outside:	0.00752385130459	+/- 0.00025566895967	(3.40%)
AF lost:	0.00000336164322	+/- 0.00000440673630	(131.09%)
Total AF:	1.00000000000000		

Configuration No: 5

The calculation is performed for:

100 runs
1600 electrons per run
Total: 160000 histories.
Initial kinetic energy: 0.150 MeV.

Absorbed fractions with 95% confidence intervals:

AF in bone:	0.86504602932233	+/- 0.00145269970286	(0.17%)
AF in marrow:	0.12025641296942	+/- 0.00142428920914	(1.18%)
AF outside:	0.01469184239012	+/- 0.00049345346303	(3.36%)
AF lost:	0.00000571531813	+/- 0.00000685698743	(119.98%)
Total AF:	1.00000000000000		

Configuration No: 6
 The calculation is performed for:
 100 runs
 1000 electrons per run
 Total: 100000 histories.
 Initial kinetic energy: 0.200 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.79571285457180 +/- 0.00192810313685 (0.24%)
 AF in marrow: 0.18034722582199 +/- 0.00179216815942 (0.99%)
 AF outside: 0.02393362121279 +/- 0.00091634435259 (3.83%)
 AF lost: 0.00000629839342 +/- 0.00001024354460 (162.64%)
 Total AF: 1.00000000000000

Configuration No: 7
 The calculation is performed for:
 100 runs
 900 electrons per run
 Total: 90000 histories.
 Initial kinetic energy: 0.250 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.72644780069761 +/- 0.00208291946364 (0.29%)
 AF in marrow: 0.23886296895719 +/- 0.00205071941137 (0.86%)
 AF outside: 0.03466651572280 +/- 0.00103139254715 (2.98%)
 AF lost: 0.00002271462241 +/- 0.00002694249315 (118.61%)
 Total AF: 1.00000000000000

Configuration No: 8
 The calculation is performed for:
 100 runs
 400 electrons per run
 Total: 40000 histories.
 Initial kinetic energy: 0.300 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.66416879779816 +/- 0.00346753135663 (0.52%)
 AF in marrow: 0.28893848801219 +/- 0.00344028637981 (1.19%)
 AF outside: 0.04685660249643 +/- 0.00164469320800 (3.51%)
 AF lost: 0.00003611169322 +/- 0.00005147278892 (142.54%)
 Total AF: 1.00000000000000

Configuration No: 9
 The calculation is performed for:
 100 runs
 300 electrons per run
 Total: 30000 histories.
 Initial kinetic energy: 0.350 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.60961078621342 +/- 0.00487011595718 (0.80%)
 AF in marrow: 0.33032297885657 +/- 0.00438452174336 (1.33%)
 AF outside: 0.06000634172027 +/- 0.00240703409728 (4.01%)
 AF lost: 0.00005989320973 +/- 0.00008305369959 (138.67%)
 Total AF: 1.00000000000000

Configuration No: 10
 The calculation is performed for:
 100 runs
 280 electrons per run
 Total: 28000 histories.
 Initial kinetic energy: 0.400 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.57034840275876 +/- 0.00406045250552 (0.71%)
 AF in marrow: 0.35644857717845 +/- 0.00387191785372 (1.09%)
 AF outside: 0.07320302006279 +/- 0.00267734846387 (3.66%)
 AF lost: 0.00000000000000 +/- 0.00000000000000 (NaN %)
 Total AF: 1.00000000000000

Configuration No: 11
 The calculation is performed for:
 100 runs

240 electrons per run
 Total: 24000 histories.
 Initial kinetic energy: 0.450 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.53177586194940 +/- 0.00400951065797 (0.75%)
 AF in marrow: 0.38199290907899 +/- 0.00375361511295 (0.98%)
 AF outside: 0.08620931992536 +/- 0.00276046091160 (3.20%)
 AF lost: 0.00002190904625 +/- 0.00004205380030 (191.95%)
 Total AF: 1.00000000000000

Configuration No: 12
 The calculation is performed for:
 100 runs
 200 electrons per run

Total: 20000 histories.
 Initial kinetic energy: 0.500 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.49998302697170 +/- 0.00530974392424 (1.06%)
 AF in marrow: 0.39552975698030 +/- 0.00491120607333 (1.24%)
 AF outside: 0.10448721604800 +/- 0.00327492109512 (3.13%)
 AF lost: 0.00000000000000 +/- 0.00000000000000 (NaN %)
 Total AF: 1.00000000000000

Configuration No: 13
 The calculation is performed for:
 100 runs
 170 electrons per run

Total: 17000 histories.
 Initial kinetic energy: 0.600 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.45849637330882 +/- 0.00464693626587 (1.01%)
 AF in marrow: 0.41282425401427 +/- 0.00451149277843 (1.09%)
 AF outside: 0.12867926536766 +/- 0.00446463084505 (3.47%)
 AF lost: 0.00000010730925 +/- 0.00000021032613 (196.00%)
 Total AF: 1.00000000000000

Configuration No: 14
 The calculation is performed for:
 100 runs
 150 electrons per run

Total: 15000 histories.
 Initial kinetic energy: 0.700 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.42889098104990 +/- 0.00582369348153 (1.36%)
 AF in marrow: 0.41592364164424 +/- 0.00495134984339 (1.19%)
 AF outside: 0.15518537730585 +/- 0.00470649674780 (3.03%)
 AF lost: 0.00000000000000 +/- 0.00000000000000 (NaN %)
 Total AF: 1.00000000000000

Configuration No: 15
 The calculation is performed for:
 100 runs
 150 electrons per run

Total: 15000 histories.
 Initial kinetic energy: 0.800 MeV.
 Absorbed fractions with 95% confidence intervals:
 AF in bone: 0.40241561383268 +/- 0.00413365915638 (1.03%)
 AF in marrow: 0.41223860721924 +/- 0.00522417045957 (1.27%)
 AF outside: 0.18528821830799 +/- 0.00552135795794 (2.98%)
 AF lost: 0.00005756064010 +/- 0.00011281885459 (196.00%)
 Total AF: 1.00000000000000

REFERENCES

- Akabani G. 1993. Absorbed dose calculations in Haversian canals for several beta-emitting radionuclides. *J Nucl Med* 34: 1361-1366.
- Atkinson PJ. 1965. Changes in resorption spaces in femoral cortical bone with age. *J Pathol Bacteriol* 89: 173-178.
- Atkinson PJ. 1967. Variation in trabecular structure of vertebrae with age. *Calcif Tissue Res* 1: 24-32.
- Atkinson PJ and Woodhead C. 1973. The development of osteoporosis. A hypothesis based on a study of human bone structure. *Clin Orthop* 90: 217-228.
- Baker HH. 1989. Building surfaces of evolution: the weaving wall. *Int J Comp Vis* 3: 51-71.
- Ballon D, Jakubowski A, Gabrilove J, Graham MC, Zakowski M, Sheridan C, and Koutcher JA. 1991. In vivo measurements of bone marrow cellularity using volume-localized proton NMR spectroscopy. *Magn Reson Med* 19: 85-95.
- Ballon D, Jakubowski AA, Graham MC, Schneider E, and Koutcher JA. 1996. Spatial mapping of the percentage cellularity in human bone marrow using magnetic resonance imaging. *Med Phys* 23: 243-250.
- Beddoe AH. 1976a. The microstructure of mammalian bone in relation to the dosimetry of bone-seeking radionuclides. Leeds, UK: University of Leeds. Department of Medical Physics.
- Beddoe AH. 1976b. A quantitative study of the structure of mammalian bone. *J Anat* 122: 190.
- Beddoe AH. 1977. Measurements of the microscopic structure of cortical bone. *Phys Med Biol* 22: 298-308.
- Beddoe AH. 1978. A quantitative study of the structure of trabecular bone in man, rhesus monkey, beagle and miniature pig. *Calcif Tissue Res* 25: 273-281.
- Beddoe AH, Darley PJ, and Spiers FW. 1976. Measurements of trabecular bone structure in man. *Phys Med Biol* 21: 589-607.
- Berne RM, Levy MN, and editors. 1993. *Physiology*. St Louis, MO: Mosby Year Book.

- Bloomenthal J. 1988. Polygonization of implicit surfaces. *Comp Aided Geom Des* 5: 341-355.
- Bolch WE, Patton PW, Rajon DA, Shah AP, Jokisch DW, and Inglis BA. 2002a. Considerations of marrow cellularity in 3D dosimetric models of the trabecular skeleton. *J Nucl Med* 43: 97-108.
- Bolch WE, Patton PW, Shah AP, Rajon DA, and Jokisch DW. 2002b. Considerations of anthropometric, tissue volume, and tissue mass scaling for improved patient specificity of skeletal S values. *Med Phys* 29: 1054-1070.
- Bone HG, Cody DD, and Monsell EM. 1994. Application of quantitative computed tomography to Paget's disease of bone. *Semin Arthritis Rheum* 23: 244-247.
- Bouchet L, Bolch W, Stabin M, Eckerman K, Poston J, and Brill A. in press. Monte Carlo methods and mathematical models in the dosimetry of the skeleton and bone marrow. In: Zaidi, H.; Sgouros, G., eds. *Monte Carlo calculations in nuclear medicine: therapeutic applications*. Bristol, UK: Institute of Physics Publishing.
- Bouchet LG and Bolch WE. 1999. A three-dimensional transport model for determining absorbed fractions of energy for electrons within cortical bone. *J Nucl Med* 40: 2115-2124.
- Bouchet LG, Bolch WE, Howell RW, and Rao DV. 2000. S values for radionuclides localized within the skeleton. *J Nucl Med* 41: 189-212.
- Bouchet LG, Bolch WE, Wessels BA, and Weber DA. 1999a. Head and brain dosimetry: absorbed fractions of energy and absorbed dose per unit cumulated activity within pediatric and adult head and brain modes for use in nuclear medicine internal dosimetry. Reston, VA: The Society of Nuclear Medicine.
- Bouchet LG, Jokisch DW, and Bolch WE. 1999b. A three-dimensional transport model for determining absorbed fractions of energy for electrons within trabecular bone. *J Nucl Med* 40: 1947-1966.
- Boyde A, Maconnachie E, Reid SA, Delling G, and Mundy GR. 1986. Scanning electron microscopy in bone pathology: review of methods, potential and applications. *Scan Electron Microsc* 4: 1537-1554.
- Brodsky A. 1996. Review of radiation risks and uranium toxicity with applications to decisions associated with decommissioning clean-up criteria. Hebron, CT: RAS.
- Chan SL and Purisima EO. 1998. A new tetrahedral tessellation scheme for isosurfaces generation. *Computer & Graphics* 22: 83-90.

- Chao TC, Bozkurt A, and Xu XG. 2001a. Conversion coefficients based on the VIP-man anatomical model and EGS4-VLSI code for external monoenergetic photons from 10 keV to 10 MeV. *Health Phys* 81: 163-183.
- Chao TC, Bozkurt A, and Xu XG. 2001b. Organ dose conversion coefficients for 0.1-10 MeV electrons calculated for the VIP-man tomographic model. *Health Phys* 81: 203-214.
- Chen L and Herman GT. 1985. Surface shading in the cuberille environment. *IEEE Comput Graph Appl* 5: 33-43.
- Chevalier F, Laval-Jeantet AM, Laval-Jeantet M, and Bergot C. 1992. CT image analysis of the vertebral trabecular network in vivo. *Calcif Tissue Int* 51: 8-13.
- Chung HW, Chu CC, Underweiser M, and Wehrli FW. 1994. On the fractal nature of trabecular structure. *Med Phys* 21: 1535-1540.
- Chung HW, Hwang SN, Yeung HN, and Wehrli FW. 1996. Mapping of the magnetic-field distribution in cancellous bone. *J Magn Reson B* 113: 172-176.
- Chung HW, Wehrli FW, Williams JL, Kugelmass SD, and Wehrli SL. 1995a. Quantitative analysis of trabecular microstructure by 400 MHz nuclear magnetic resonance imaging. *J Bone Miner Res* 10: 803-811.
- Chung HW, Wehrli FW, Williams JL, and Wehrli SL. 1995b. Three-dimensional nuclear magnetic resonance microimaging of trabecular bone. *J Bone Miner Res* 10: 1452-1461.
- Ciarelli MJ, Goldstein SA, Kuhn JL, Cody DD, and Brown MB. 1991. Evaluation of orthogonal mechanical properties and density of human trabecular bone from the major metaphyseal regions with materials testing and computed tomography. *J Orthop Res* 9: 674-682.
- Clayman CB. 1995. *The human body: an illustrated guide to its structure, function and disorders*. New York, NY: Jonathan Reed.
- Cline HE, Lorensen WE, Ludlke S, Crawford CR, and Teeter BC. 1988. Two algorithms for three-dimensional reconstruction of tomograms. *Med Phys* 15: 320-327.
- Cody DD, Flynn MJ, and Vickers DS. 1989. A technique for measuring regional bone mineral density in human lumbar vertebral bodies. *Med Phys* 16: 766-772.
- Cody DD, Goldstein SA, Flynn MJ, and Brown EB. 1991. Correlations between vertebral regional bone mineral density (rBMD) and whole bone fracture load. *Spine* 16: 146-154.

- Cody DD, McCubbrey DA, Divine GW, Gross GJ, and Goldstein SA. 1996. Predictive value of proximal femoral bone densitometry in determining local orthogonal material properties. *J Biomech* 29: 753-761.
- Coleman R. 1969. Random paths through convex bodies. *J Appl Prob* 6: 430-441.
- Cowin SC. 1989. Bone mechanics. Boca Raton, FL: CRC Press.
- Crawford CR, Santos E, Weinstein ID, Teeter BC, Cline HE, and Lorensen WE. 1988. 3-D imaging using normalized gradient shading. In: Proceedings of the annual international conference of the IEEE Engineering in Medicine and Biology Society. New Orleans, LA, USA: 1: 412-413.
- Cristy M. 1980. Mathematical phantoms representing children of various ages for use in estimates of internal dose. Oak Ridge, TN: Oak Ridge National Laboratory. ORNL/NUREG/TM-367.
- Cristy M and Eckerman KF. 1987. Specific absorbed fractions of energy at various ages from internal photon sources. Oak Ridge, TN: Oak Ridge National Laboratory. ORNL/TM-8381.
- Custer RP and Ahlfeldt FE. 1932. Studies on the structure and function of bone marrow variations in cellularity in various bones with advancing years of life and their relative response to stimuli. *J Lab Clinical Med* 17: 951-962.
- Darley PJ. 1968. Measurement of linear path length distribution in bone marrow using a scanning technique. In: eds. Symposium on Microdosimetry. Ispra, Italy: EAEC. EUR 3747.
- Darley PJ. 1972. An investigation of the structure of trabecular bone in relation to the radiation dosimetry of bone-seeking radionuclides. Leeds, UK: University of Leeds. Department of Medical Physics.
- Delibasis KS, Matsopoulos GK, Mouravliansky NA, and Nikita KS. 2001. A novel and efficient implementation of the marching cubes algorithm. *Comp Med Imag Graph* 25: 343-352.
- Dixon WT. 1984. Simple proton spectroscopic imaging. *Radiology* 153: 189-194.
- DOE. 1995. Environmental management 1995. [U.S. Department of Energy]. Springfield, VA: National Technical Information Service: Publisher. DOE/EM-0228.
- Doi A and Koide A. 1991. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans Commun Elec Inf Syst* E-74: 214-224.
- Durand EP and Ruegsegger P. 1992. High-contrast resolution of CT images for bone structure analysis. *Med Phys* 19: 569-573.

- Düurst MJ. 1988. Letters: Additionnal reference to "Marching Cube". *Computer Graphics* 22: 72-73.
- Eckerman KF. 1985. Aspects of the dosimetry of radionuclides within the skeleton with particular emphasis on the active marrow. In: *Proceedings of the Fourth International Radiopharmaceutical Dosimetry Symposium*. Oak Ridge, TN: ORAU; CONF-851113. 514-534.
- Eckerman KF, Ryman JC, Taner AC, and Kerr GD. 1985. Traversal of cells by radiation and absorbed fraction estimates for electrons and alpha particles. In: *Proceedings of the Fourth International Radiopharmaceutical Dosimetry Symposium*. Oak Ridge, TN: ORAU; CONF-851113. 67-81.
- Eckerman KF and Stabin MG. 2000. Electron absorbed fractions and dose conversion factors for marrow and bone by skeletal regions. *Health Phys* 78: 199-214.
- Eckerman KF, Westfall RJ, Ryman JC, and Cristy M. 1993. Nuclear decay data files of the dosimetry research group. Oak Ridge, TN: Oak Ridge National Laboratory.
- Ellis RE. 1961. The distribution of active bone marrow in the adult. *Phys Med Biol* 5: 225-258.
- Engelke K, Graeff W, Meiss L, Hahn M, and Delling G. 1993. High spatial resolution imaging of bone mineral using computed microtomography. Comparison with microradiography and undecalcified histologic sections. *J Invest Radiol* 28: 341-349.
- Farrell EJ. 1983. Color display and interactive interpretation of three-dimensional data. *Computer Graphics and Image Processing* 27: 356-366.
- Flynn MJ and Cody DD. 1993. The assessment of vertebral bone macroarchitecture with X-ray computed tomography. *Calcif Tissue Int* 53: S170-175.
- Foo TK, Shellock FG, Hayes CE, Schenck JF, and Slayman BE. 1992. High-resolution MR imaging of the wrist and eye with short TR, short TE, and partial-echo acquisition. *Radiology* 183: 277-281.
- Glover GH and Schneider E. 1991. Three-point Dixon technique for true water/fat decomposition with B0 inhomogeneity correction. *Magn Reson Med* 18: 371-383.
- Gordon CL, Webber CE, Christoforou N, and Nahmias C. 1997. In vivo assessment of trabecular bone structure at the distal radius from high-resolution magnetic resonance images. *Med Phys* 24: 585-593.
- Goulet RW, Goldstein SA, Ciarelli MJ, Kuhn JL, Brown MB, and Feldkamp LA. 1994. The relationship between the structural and orthogonal compressive properties of trabecular bone. *J Biomech* 4: 375-389.

- Grampp S, Majumdar S, Jergas M, Newitt D, Lang P, and Genant HK. 1996. Distal radius: in vivo assessment with quantitative MR imaging, peripheral quantitative CT, and dual X-ray absorptiometry. *Radiology* 198: 213-218.
- Guglielmi G, Selby K, Blunt BA, Jergas M, Newitt DC, Genant HK, and Majumdar S. 1996. Magnetic resonance imaging of the calcaneus: preliminary assessment of trabecular bone-dependent regional variations in marrow relaxation time compared with dual X-ray absorptiometry. *Acad Radiol* 3: 336-343.
- Hahn M, Vogel M, Pompesius-Kempa M, and Delling G. 1992. Trabecular bone pattern factor-a new parameter for simple quantification of bone microarchitecture. *Bone* 13: 327-330.
- Hamacher KA and Sgouros G. 2001. Theoretical estimation of absorbed dose to organs in radioimmunotherapy using radionuclides with multiple unstable daughters. *Med Phys* 28: 1857-1874.
- Hartmann E. 1998. A marching method for triangulation of surfaces. *The Visual Computer* 14: 95-108.
- Herman GT and Liu HK. 1979. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing* 9: 1-21.
- Herman GT and Udupa JK. 1983. Display of 3-D digital images: computational foundations and medical applications. *IEEE Comput Graph Appl* 3: 39-45.
- Hilton A, Stoddart AJ, Illingworth J, and Windeatt T. 1996. Marching triangles: range image fusion for complex object modelling. In: *Proceedings. International Conference on Image Processing. Lausanne Switzerland*: 2: 381-384.
- Hindmarsh M, Owen M, Vaughan J, Lamerton LF, and Spiers FW. 1958. The relative hazards of stontium-90 and radium-226. *Br J Radiol* 31: 518-533.
- Hohne KH and Bernstein R. 1986. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging* MI-5: 45-47.
- Howell RW. 1994. The MIRD schema: from organ to cellular dimensions. *J Nucl Med* 35: 531-533.
- Hwang SN, Wehrli FW, and Williams JL. 1997. Probability-based structural parameters from three-dimensional nuclear magnetic resonance images as predictors of trabecular bone strength. *Med Phys* 24: 1255-1261.
- ICRP. 1979. *Limits for intakes of radionuclides by workers*. Oxford, UK: International Commission on Radiation Protection. Publication 30.

- ICRU. 1984. Stopping powers for electrons and positrons. Bethesda, MD: International Commission on Radiation Units and Measurements. Report 37.
- ICRU. 1992. Photon, electron, proton and neutron interaction data for body tissues. Bethesda, MD: International Commission on Radiation Units and Measurements. Report 46.
- Jara H, Wehrli FW, Chung H, and Ford JC. 1993. High-resolution variable flip angle 3D MR imaging of trabecular microstructure in vivo. *Magn Reson Med* 29: 528-539.
- Jokisch DW. 1997. Nuclear magnetic resonance imaging as a tool for studying beta-dosimetry in trabecular bone and red marrow regions. Gainesville, FL: University of Florida. Nuclear and Radiological Engineering.
- Jokisch DW. 1999. Beta particle dosimetry of trabecular region of a thoracic vertebra utilizing NMR microscopy. Gainesville, FL: University of Florida. Nuclear and Radiological Engineering.
- Jokisch DW, Bouchet LG, Patton PW, Rajon DA, and Bolch WE. 2001a. Beta-particle dosimetry of the trabecular skeleton using Monte Carlo transport in 3D digital images. *Med Phys* 28: 1505-1518.
- Jokisch DW, Patton PW, Inglis BA, Bouchet LG, Rajon DA, Rifkin J, and Bolch WE. 1998. NMR microscopy of trabecular bone and its role in skeletal dosimetry. *Health Phys* 75: 584-596.
- Jokisch DW, Patton PW, Rajon DA, Inglis BA, and Bolch WE. 2001b. Chord distributions across 3D digital images of a human thoracic vertebra. *Med Phys* 28: 1493-1504.
- Kawrakow I. 2000. Accurate condensed history Monte Carlo simulation of electron transport. I. EGSnrc, the new EGS4 version. *Med Phys* 27: 485-498.
- Kellerer AM. 1971. Considerations on the random traversal of convex bodies and solutions for general cylinders. *Radiat Res* 47: 359-376.
- Kinney JH, Lane NE, and Haupt DL. 1995. In vivo, three-dimensional microscopy of trabecular bone. *J Bone Miner Res* 10: 264-270.
- Kleerekoper M, Nelson DA, Flynn MJ, Pawluszka AS, Jacobsen G, and Peterson EL. 1994a. Comparison of radiographic absorptiometry with dual-energy x-ray absorptiometry and quantitative computed tomography in normal older white and black women. *J Bone Miner Res* 9: 1745-1749.

- Kleerekoper M, Nelson DA, Peterson EL, Flynn MJ, Pawluszka AS, Jacobsen G, and Wilson P. 1994b. Reference data for bone mass, calciotropic hormones, and biochemical markers of bone remodeling in older (55-75) postmenopausal white and black women. *J Bone Miner Res* 9: 1267-1276.
- Knoll GF. 2000. Radiation detection and measurement. New York, NY: John Wiley & Sons, Inc.
- Kuhn JL, Goldstein SA, Feldkamp LA, Goulet RW, and Jasion G. 1990. Evaluation of a microcomputed tomography system to study trabecular bone structure. *J Orthop Res* 8: 833-842.
- Link TM, Majumdar S, Konermann W, Meier N, Lin JC, Newitt D, Ouyang X, Peters PE, and Genant HK. 1997. Texture analysis of direct magnification radiographs of vertebral specimens: correlation with bone mineral density and biomechanical properties. *Acad Radiol* 4: 167-176.
- Link TM, Majumdar S, Lin JC, Augat P, Gould RG, Newitt D, Ouyang X, Lang TF, Mathur A, and Genant HK. 1998a. Assessment of trabecular structure using high resolution CT images and texture analysis. *J Comput Assist Tomogr* 22: 15-24.
- Link TM, Majumdar S, Lin JC, Newitt D, Augat P, Ouyang X, Mathur A, and Genant HK. 1998b. A comparative study of trabecular bone properties in the spine and femur using high resolution MRI and CT. *J Bone Miner Res* 13: 122-132.
- Loevinger R and Berman M. 1968a. A formalism for calculation of absorbed dose from radionuclides. *Phys Med Biol* 13: 205-217.
- Loevinger R and Berman M. 1968b. MIRD Pamphlet No. 1: a schema for absorbed dose calculations for biologically-distributed radionuclides. *J Nucl Med* 9: 7-14.
- Loevinger R, Budinger TF, and Watson EE. 1991. MIRD Primer for absorbed dose calculations. New York, NY: The Society of Nuclear Medicine.
- Lord BI. 1990. The architecture of bone marrow cell populations. *Int J Cell Cloning* 8: 317-331.
- Lorensen WE and Cline HE. 1987. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics* 21: 163-169.
- Majumdar S and Genant HK. 1995. A review of the recent advances in magnetic resonance imaging in the assessment of osteoporosis. [Review]. *Osteoporos Int* 5: 79-92.

- Majumdar S, Genant HK, Grampp S, Newitt DC, Truong VH, Lin JC, and Mathur A. 1997. Correlation of trabecular bone structure with age, bone mineral density, and osteoporotic status: in vivo studies in the distal radius using high resolution magnetic resonance imaging. *J Bone Miner Res* 12: 111-118.
- Majumdar S, Link TM, Augat P, Lin JC, Newitt D, Lane NE, and Genant HK. 1999. Trabecular bone architecture in the distal radius using magnetic resonance imaging in subjects with fractures of the proximal femur. *Magnetic Resonance Science Center and Osteoporosis and Arthritis Research Group. [Review]. Osteoporos Int* 10: 231-239.
- Majumdar S, Newitt D, Mathur A, Osman D, Gies A, Chiu E, Lotz J, Kinney J, and Genant H. 1996. Magnetic resonance imaging of trabecular bone structure in the distal radius: relationship with X-ray tomographic microscopy and biomechanics. *Osteoporos Int* 6: 376-385.
- Marieb EN. 1998. *Human anatomy and physiology*. Menlo Park, CA: Benjamin/Cummings Science Publishing.
- Marsaglia G and Zaman A. 1991. A new class of random number generators. *A Appl Prob* 1: 462-480.
- Matveyev SV. 1994. Approximation of isosurface in the Marching Cube: ambiguity problem. In: *IEEE Proceedings on Visualisation '94 (Cat. No.94CH35707)*. Washington, DC, USA: 288-292.
- McCubbrey DA, Cody DD, Peterson EL, Kuhn JL, Flynn MJ, and Goldstein SA. 1995. Static and fatigue failure properties of thoracic and lumbar vertebral bodies and their relation to regional density. *J Biomech* 28: 891-899.
- McDevitt MR, G GS, Finn RD, Humm JL, Jurcic JG, Larson SM, and Scheinberg DA. 1998. Radioimmunotherapy with alpha-emitting nuclides. *Eur J Nucl Med* 25: 1341-51.
- Mechanik N. 1926. Studies of the weight of bone marrow in man. *Zeitschrift fur die Gest Anatomy* 79: 58-99.
- Montani C, Scateni R, and Scopigno R. 1994a. Discretized marching cubes. In: *Proceedings. Visualization '94*. Washington, DC, USA: 1: 281-287.
- Montani V, Scateni R, and Scopigno R. 1994b. A modified look-up table for implicit disambiguation of marching cubes. *The Visual Computer* 10: 353-355.
- Montani V, Scateni R, and Scopigno R. 2000. Decreasing isosurface complexity via discrete fitting. *Comp Aided Geom Des* 17: 207-232.

- Mosekilde L. 1986. Age-related changes in vertebral trabecular architecture assessed by a new method. *Bone* 9: 247-250.
- Mosekilde L. 1989. Sex differences in age-related loss of vertebral trabecular bone mass and structure - biomechanical consequences. *Bone* 10: 425-432.
- Muller R, Hildebrand T, and Ruegsegger P. 1994. Non-invasive bone biopsy: a new method to analyse and display the three-dimensional structure of trabecular bone. *Phys Med Biol* 39: 145-164.
- Muller R and Ruegsegger P. 1996. Analysis of mechanical properties of cancellous bone under conditions of simulated bone atrophy. *J Biomech* 29: 1053-1060.
- Natarajan BK. 1994. On generating topologically consistent isosurfaces from uniform samples. *The Visual Computer* 11: 52-62.
- Nielson GM and Hamann B. 1991. The asymptotic decider: resolving the ambiguity in marching cubes. In: *IEEE Proceedings on Visualisation '91* (Cat. No.91CH3046-0). San Diego, CA: 83-91, 413.
- Ning P and Bloomenthal J. 1993. An evaluation of implicit surface tilers. *IEEE Comput Graph Appl* 13: 33-41.
- Odgaard A, Andersen K, Melsen F, and Gundersen HJ. 1990. A direct method for fast three-dimensional serial reconstruction. *J Microsc* 159: 335-342.
- Ouyang X, Selby K, Lang P, Engelke K, Klifa C, Fan B, Zucconi F, Hottya G, Chen M, Majumdar S, and Genant HK. 1997. High resolution magnetic resonance imaging of the calcaneus: age-related changes in trabecular structure and comparison with dual X-ray absorptiometry measurements. *Calcif Tissue Int* 60: 139-147.
- Parfitt AM, Mathews CH, Villanueva AR, Kleerekoper M, Frame B, and Rao DS. 1983. Relationships between surface, volume, and thickness of iliac trabecular bone in aging and in osteoporosis. Implications for the microanatomic and cellular mechanisms of bone loss. *J Clin Invest* 72: 1396-1409.
- Patton PW. 1998. Nuclear magnetic resonance assessment of chord distributions for trabecular bone dosimetry: the effects of sample freezing and thawing. Gainesville, FL: University of Florida. Nuclear and Radiological Engineering.
- Patton PW. 2000. NMR microscopy for skeletal dosimetry: an investigation of marrow cellularity on dose estimates. Gainesville, FL: University of Florida. Nuclear and Radiological Engineering.
- Patton PW, Jokisch DW, Rajon DA, Shah AP, Myers SL, Inglis BA, and Bolch WE. 2002a. Skeletal dosimetry via NMR microscopy: Investigations of sample reproducibility and signal source. *Health Phys* 82: 316-326.

- Patton PW, Rajon DA, Shah AP, Jokisch DW, Inglis BA, and Bolch WE. 2002b. Site-specific variability in trabecular bone dosimetry: Considerations of energy loss to cortical bone. *Med Phys* 29: 6-14.
- Payne BA and Toga AW. 1990. Surface mapping brain function on 3D models. *IEEE Comput Graph Appl* 10: 33-41.
- Poston J, Bolch W, and Bouchet L. 2002. Mathematical models of human anatomy. In: Zaidi, H.; Sgouros, G., eds. Monte Carlo calculations in nuclear medicine: therapeutic applications. Bristol, UK: Institute of Physics Publishing.
- Press WH, Teukolsky SA, Vetterling WT, and Flannery BP. 1992. Numerical recipes in C. Cambridge, UK: Cambridge University Press.
- Rajon DA. 1999. Trabecular bone dosimetry: assessment of minimum voxel size for nuclear magnetic resonance imaging. Gainesville, FL: University of Florida. Nuclear and Radiological Engineering.
- RSI. 2001. Interactive Data Language (IDL 5.5).
- Rubin P and Scarantino CW. 1978. The bone marrow organ: the critical structure in radiation-drug interaction. *Int J Radiat Oncol Biol Phys* 4: 3-23.
- Ruegsegger P, Koller B, and Muller R. 1996. A microtomographic system for the nondestructive evaluation of bone architecture. *Calcif Tissue Int* 58: 24-29.
- Samaratunga RC, Thomas SR, Hinnefeld JD, Kuster LCV, Hyams DM, Moulton JS, Sperling MI, and Maxon HR. 1995. A Monte Carlo simulation model for radiation dose to metastatic skeletal tumor from Rhenium-186(Sn)-HEDP. *J Nucl Med* 36: 336-350.
- Sebag GH and Moore SG. 1990. Effect of trabecular bone on the appearance of marrow in gradient-echo imaging of the appendicular skeleton. *Radiology* 174: 855-859.
- Sgouros G. 1993. Bone marrow dosimetry for radioimmunotherapy: theoretical considerations. *J Nucl Med* 34: 689-694.
- Sgouros G, Stabin M, Erdi Y, Akabani G, Kwok C, Brill AB, and Wessels B. 2000. Red marrow dosimetry for radiolabeled antibodies that bind to marrow, bone, or blood components. *Med Phys* 27: 2150-2164.
- Shen S, DeNardo GL, Sgouros G, O'Donnell RT, and DeNardo SJ. 1999. Practical determination of patient-specific marrow dose using radioactivity concentration in blood and body. *J Nucl Med* 40: 2102-2106.

- Shen S, Meredith RF, Duan J, Macey DJ, Khazaeli MB, Robert F, and LoBuglio AF. 2002. Improved prediction of myelotoxicity using a patient-specific imaging dose estimate for non-marrow-targeting (90)y-antibody therapy. *J Nucl Med* 43: 1245-1253.
- Shirley P and Tuckman A. 1990. A polygonal approximation to direct scalar volume rendering. *Computer Graphics* 24: 63-70.
- Siegel JA, Wessels BW, Watson EE, Stabin MG, Vriesendorp HM, Bradley EW, Badger CC, Brill AB, Kwok CS, Stickney DR, Eckerman KF, Fisher DR, Buchsbaum DJ, and Order SE. 1990. Bone marrow dosimetry and toxicity for radioimmunotherapy. *Antibody, Immunoconjugates, Radiopharm* 3: 213-233.
- Snyder BD, Piazza S, Edwards WT, and Hayes WC. 1993. Role of trabecular morphology in the etiology of age-related vertebra fractures. *Calcif Tissue Int* 53: S14-S22.
- Snyder WS, Ford MR, Warner GG, and Watson SB. 1974. A tabulation of dose equivalent per microcurie day for source and target organs of an adult for various radionuclides. Oak Ridge, Tennessee: Oak Ridge National Laboratory.
- Snyder WS, Ford MR, Warner GG, and Watson SB. 1975. "S" absorbed dose per unit cumulated activity for selected radionuclides and organs. New York: Society of Nuclear Medicine. MIRD Pamphlet No. 11.
- Song HK, Wehrli FW, and Ma J. 1997. In vivo MR microscopy of the human skin. *Magn Reson Med* 37: 185-191.
- Spiers FW. 1949. The influence of energy absorption and electron range on dosage in irradiated bone. *Br J Radiol* 22: 521-533.
- Spiers FW. 1951. Dosage in irradiated soft tissue and bone. *Br J Radiol* 24: 365-370.
- Spiers FW. 1963. Interim report on the determination of dose to bone marrow from radiological procedures. *Br J Radiol* 36: 238-240.
- Spiers FW. 1966a. Dose to bone from strontium-90: Implications for the setting of maximum permissible body burden. *Radiat Res* 28: 624-642.
- Spiers FW. 1966b. Dose to trabecular bone from internal beta-emitters. In: *First International Congress of Radiation Protection*. Rome, Italy: Pergamon Press. 1: 165-172.
- Spiers FW. 1967. Beta particle dosimetry in trabecular bone. In: *Delayed effects of bone-seeking radionuclides*. Salt Lake City: University of Utah Press, 95-108.
- Spiers FW. 1968. Dosimetry at interfaces with special reference to bone. *Ispra, Italy: EAEC Report EUR 3747*.

- Spiers FW. 1969. Determination of absorbed dose to bone and red bone marrow. In: Medical radionuclides: Radiation dose and effects. Oak Ridge, TN: U. S. Atomic Energy Commission; AEC Symposium Series 20, 347-367.
- Spiers FW, Baddoe AH, and Whitwell JR. 1978a. Mean skeletal dose factors for beta-particle emitters in human bone, part 1: volume-seeking radionuclides. *Br J Radiol* 51: 622-627.
- Spiers FW and Beddoe AH. 1977. 'Radial' scanning of trabecular bone: consideration of the probability distributions of path lengths through cavities and trabeculae. *Phys Med Biol* 22: 670-680.
- Spiers FW, Whitwell JR, and Baddoe AH. 1978b. Calculated dose factors for the radiosensitive tissues in bone irradiated by surface-deposited radionuclides. *Phys Med Biol* 23: 481-494.
- Stabin M, Watson E, Cristy M, Ryman J, Eckerman K, Davis J, Marshall D, and Gehlen M. 1995. Mathematical models and specific absorbed fraction of photon energy in the nonpregnant adult female and at the end of each trimester of pregnancy. Oak Ridge, TN: Oak Ridge National Laboratory.
- Stabin MG. 1996. MIRDose: personal computer software for internal dose assessment in nuclear medicine. *J Nucl Med* 37: 538-546.
- Takahashi T. 1994. Atlas of the human body. New York, NY: HarperCollins Publishers, Inc.
- Tortora GJ. 1992. Principles of human anatomy. New York, NY: HarperCollins Publishers.
- Treece GM, Prager RW, and Gee AH. 1999. Regularised marching tetrahedra: improved iso-surface extraction. *Computer & Graphics* 23: 583-598.
- Turner CH. 1992. On Wolff's law of trabecular architecture. *J Biomech* 25: 1-9.
- Turner JE. 1995. Atoms, radiation and radiation protection. New York, NY: John Wiley & Sons, Inc.
- Udupa JK and Ajjanagadde VG. 1990. Boundary and object labelling in three-dimensional images. *Comput Vis Graph Image Process* 51: 355-369.
- Van-Gelder A and Wilhelms J. 1994. Topological considerations in isosurface generation. *ACM Transactions on Graphics* 13: 337-375.
- Vaughan J. 1960. The relation of radiation dose to radiation injury in the skeleton, with special reference to strontium-90. In: Radioisotopes and bone. Princeton, NJ: F. A. Davis Company, 317-333.

- Vaughan J. 1975. The physiology of bone. Oxford: Clarendon Press.
- Vaughan JM. 1973. The effects of irradiation on the skeleton. London: Oxford University Press.
- Wehrli FW, Hwang SN, Ma J, Song HK, Ford JC, and Haddad JG. 1998. Cancellous bone volume and structure in the forearm: noninvasive assessment with microimaging and image processing. *Radiology* 206: 347-357.
- Wessels M, Mason RP, Antich PP, Zerwekh JE, and Pak CY. 1997. Connectivity in human cancellous bone by three-dimensional magnetic resonance imaging. *Med Phys* 24: 1409-1420.
- Whitehouse WJ and Dyson ED. 1974. Scanning electron microscope studies of trabecular bone in the proximal end of the human femur. *J Anat* 118: 417-444.
- Whitehouse WJ, Dyson ED, and Jackson CK. 1971. The scanning electron microscope in studies of trabecular bone from a human vertebral body. *J Anat* 108: 481-496.
- Whitwell JR. 1973. Theoretical investigations of energy loss by ionizing particles in bone. Leeds, UK: University of Leeds. Department of Medical Physics.
- Whitwell JR and Spiers FW. 1976. Calculated beta-ray dose factors for trabecular bone. *Phys Med Biol* 21: 16-38.
- Williams JL and Lewis JL. 1982. Properties and an anisotropic model of cancellous bone from the proximal tibial epiphysis. *J Biomech Eng* 104: 50-56.
- Wong EC, Jesmanowicz A, and Hyde JS. 1991. High-resolution, short echo time MR imaging of the fingers and wrist with a local gradient coil. *Radiology* 181: 393-397.
- Wyvill G, McPheeters C, and Wyvill B. 1986. Data structure for soft objects. *The Visual Computer* 2: 227-234.
- Xu XG, Chao TC, and Bozkurt A. 2000. VIP-Man: an image-based whole-body adult male model constructed from color photographs of the Visible Human Project for multi-particle Monte Carlo calculations. *Health Phys* 78: 476-486.
- Zankl M and Wittmann A. 2001. The adult male voxel model "Golem" segmented from whole-body CT patient data. *Radiation and Environmental Biophysics* 40: 153-162.
- Zanzonico P and Sgouros G. 1997. Predicting myelotoxicity in radioimmunotherapy: what does dosimetry contribute? *J Nucl Med* 38: 1753-1754.
- Zhou C, Shu R, and Kankanhalli MS. 1994. Handling small features in isosurface generation using Marching Cubes. *Computer & Graphics* 18: 845-848.

Zhou Y, Chen W, and Tang Z. 1995. An elaborated ambiguity detection method for constructing isosurfaces within tetrahedral meshes. *Computer & Graphics* 19: 355-364.

BIOGRAPHICAL SKETCH

Didier A. Rajon was born in Jallieu, France, on May 15, 1964. Didier is the son of Paul and Marie-Thérèse Rajon. Didier graduated from the “Université Mendes France” of Grenoble in 1984 and started a career in computer science until 1993 when he enrolled at the “Institut National Polytechnique de Grenoble.” He graduated from the “École Nationale Supérieure de Physique de Grenoble” in 1997. After a 6-month internship at the University of Florida, Didier enrolled at UF in January 1998 to pursue a Master of Science degree with a concentration in health physics. He graduated in December 1999 and since then has been pursuing his doctoral research for which this dissertation is a partial requirement.