UNIVERSITÀ DEGLI STUDI DI ROMA TRE

MASTER DEGREE THESIS

Models from massive biomedical images via parallel computational topology

Author:
Salvati Danilo

Supervisor:
Prof. Paoluzzi Alberto

CVDLAB
Department of Engineering

March 6, 2016

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Introduction

This work has the aim to exploit some new generation techniques for the creation of three-dimensional models of biological parts. It is an extremely interesting field for the advancement of scientific research, because it allows to infer knowledge from data in a simple and effective way. Moreover, it brings together some of the most interesting sciences:

- Scientific Visualization
- Big Data
- High Performance Computing
- Computational Topology

Therefore, our aim is to show also how to link all these disciplines in a systematic way, introducing a new methodology which can shows and remove the problem related with the current state of the art. In fact, nowadays there are many technologies which permits to shows detailed images of biological parts. Several of these are commonly used in medical diagnostics, for example we can cite **PET** or **magnetic resonance**. They produces two-dimensional images of slices of the interesting organs, which are then collected in stacks for a rough three-dimensional representation. In addition, there are several techniques for the creation of a real three-dimensional model which involve **volumetric** reconstruction and **isosurfaces** reconstruction (which are extensively used in computer graphics). Their usefulness, lies in the high readability of the final result for research purposes on little-known structures. Furthermore, they can be used in combination with 3D printers to realize little models which can be useful for medical interventions. First problems, arises when the input grows in sizes. In fact the continuous progresses in scientific research, need images that have to be more and more detailed to fully understand the functioning of biological parts. However, these techniques we have cited above, have poor efficiency or are not detailed enough at highest resolutions.

The new methodologies introduced with this work, want to solve these problems. In particular, we will see a novel approach for the extraction of boundary of complex shapes which involves algebraic topology techniques. It is based on a representation schema called **LAR** (Linear Algebraic Representation) which is under development by several years at the Computational and Visual Design laboratory at the Roma Tre University. Its key features are the simplicity and the small space required to define every complex shape with only a description of vertices and the relationships between them in terms of edges, faces and cells. Moreover, we will see that algorithms for these structures are efficient in time and space with respect to traditional methods. Finally, to solve problems regarding huge volume of data we will show a technique for parallelization of computation which exploit some interesting characteristics of LAR.

In detail, this thesis is organized in the following way:

The first part describes the current state of art. It is divided into four chapters, the first two ones introduce the Scientific Visualization field and problems and opportunities related to the Big Data. The third chapter describes the fundamentals for **medical imaging** field comprehension and the fourth one shows the state of the art of Computer Graphics for the creation of three-dimensional models

The second part describes the methodologies adopted. It is divided into three chapters which show the fundamentals of topological algebra and the LAR representation schema, the principles of **High Performance Computing** and some important aspects of the language chosen for the implementation: **Julia**

The third part shows the structure and the functionalities of the application with three chapter which describe the overall architecture, the input manipulation and the conversion process from the stack of images to the three-dimensional model

The fourth part describes some case studies accompanied by a wide photographic documentation and the conclusions regarding the work results

Contents

Acknowledgements i

Introduction ii

I The problem and the current state of the art 1

1 The need for scientific visualization 2

1.1 Scientific visualization applications 4

2 The rise of big data era 6

2.1 Importance and value of data 6

2.2 Typical problems when dealing with big data 7

2.3 Big data collections and big data objects 9

3 Medical imaging visualization 11

3.1 Medical imaging basis 11

3.1.1 The electromagnetic spectrum 11

3.1.2 Image representation 12

3.2 Common imaging techniques 13

3.2.1 X-rays 13

3.2.2 Computed tomography 15

3.2.3 Magnetic resonance tomography 17

3.2.4 Ultrasound 18

3.2.5 Nuclear medicine and molecular imaging 20

3.3 The need for three-dimensional representations 21

4 Current state of art in computer graphics 23

4.1 Representation of shapes in graphics: meshes 23

4.1.1 Manifold and Nonmanifold meshes . . . 24
4.2 Typical structures for meshes representations . . . 26
4.2.1 Directed-edge structure . . . 26
4.2.2 Winged-edge structure . . . 28
4.2.3 Quad-edge structure . . . 29
4.2.4 Half-edge structure . . . 29
4.2.5 Memory requirements for mesh structures . . . 30
4.3 Volume rendering techniques . . . 32
4.3.1 Volume ray casting . . . 33
4.3.2 Maximum intensity projection . . . 34
4.4 Marching Cubes and iso-surfaces for data visualization . . . 34

II Methodologies and tools 38

5 A topological approach for data visualization: LAR 39
5.1 Fundamentals of topology . . . 39
5.1.1 Chain complexes . . . 43
5.1.2 Cell complexes . . . 44
5.2 LAR representation schema . . . 46
5.3 Some interesting topological operators . . . 50

6 Mastering big data using parallel computing 55
6.1 Principles of parallel computing . . . 55
6.2 Typical architectures for parallel computing . . . 57
6.2.1 Shared-memory systems . . . 57
6.2.2 Message-passing systems . . . 58
6.3 The Message Passing Interface . . . 59

7 The Julia language 61
7.1 Principal characteristics . . . 61
7.2 Parallel programming in Julia . . . 65

III The application **68**

8 Architecture of ImagesToLARModel **69**

8.1 Introduction to the application 69

8.2 Distributing the model in a grid 70

8.3 Exposed functionalities of the application 71

9 Data preparation **73**

9.1 Raw data extraction from images 73

9.2 Images resizing 74

9.3 Creating binary images 75

9.4 Images filtering 78

10 Images conversion **81**

10.1 The conversion pipeline 81

10.2 Converting pixels to voxels 82

10.3 Merging boundaries 83

10.4 Merging blocks 85

10.5 Smoothing 86

10.6 Creating the final model 88

IV Case studies and conclusions **89**

11 Studying the pipeline steps with an example **90**

12 Comparison between marching cubes and our algorithm **95**

13 Studying a three-dimensional model of a liver **98**

13.1 Hepatic portal system 98

13.2 The three-dimensional model of a liver 100

14 Conclusions **104**

List of Figures

1.1 Tufte example for pattern recognition . . . 3
1.2 Examples of scientific visualization applications in natural sciences . . . 4
1.3 Examples of scientific visualization application in geography . . . 5
1.4 Examples of scientific visualization applications in applied sciences . . . 5

2.1 The four V of big data . . . 8

3.1 X-ray image . . . 15
3.2 Sinogram . . . 16
3.3 Computed tomography of human brain . . . 17
3.4 MRI of human brain . . . 18
3.5 Color Doppler . . . 20
3.6 PET scan of a brain . . . 21
3.7 3D print of a tracheal splint . . . 22

4.1 Examples of triangular meshes . . . 24
4.2 Meshes orientation and normals . . . 25
4.3 Manifold and boundarylikes vertices . . . 25
4.4 Nonmanifold vertices . . . 26
4.5 The directed-edge structure . . . 27
4.6 The directed-edge structure for non-manifold vertices and edges . . . 28
4.7 The winged-edge structure . . . 29
4.8 Comparison between the quad-edge and half-edge structures . . . 30
4.9 Volume ray casting . . . 34
4.10 Maximum intensity projection . . . 35
4.11 Iso-boundary on a sample image . . . 35
4.12 All possible configurations in Marching Cubes . . . 36

4.13 Marching Cubes grid sizes 37
4.14 Marching Cubes quality 37

5.1 Homeomorphisms between a mug and a torus 42
5.2 Relationship between Mobius strip and a cylinder 42
5.3 Examples of standard n-simplices 43
5.4 A singular n-simplex 43
5.5 Operations on cell complexes 47
5.6 LAR representation schema 48
5.7 Relationships between chains and cochains 51
5.8 Cuboidal complex 54

6.1 Amdahl's Law 57

8.1 The grid used for parallel computation 70

9.1 Reading raw data from image 74
9.2 Some interesting resize cases 76
9.3 K-means clustering 78
9.4 Adjacency relationship for pixels 80

10.1 Transformation of a matrix resulting from a $2 \times 2 \times 2$ grid into an array 82
10.2 Sample models of $2 \times 2 \times 2$ blocks 83
10.3 Merging of boundary faces 84
10.4 Decomposition of a LAR model into seven parts 84
10.5 Removal of double faces from boundaries 85
10.6 Sample model with double vertices 86
10.7 Laplacian smoothing 86
10.8 Smoothing of a single block 87
10.9 Obj sample file 88

11.1 A three-dimensional representation of the Stanford Bunny 90
11.2 CT scans of the Stanford Bunny 91
11.3 Binary images obtained from the previous CT scans 91
11.4 Some blocks obtained from the images 92

11.5 Removal of duplicated boundaries from blocks 92
11.6 Squared bunny 93
11.7 Smoothed bunny 94
11.8 Rendering of Stanford bunny 94

12.1 Comparison between LAR and marching cubes 95
12.2 Topological errors in marching cubes 96
12.3 A close-up view of the LAR model 96
12.4 A detailed view of cerebral veins with LAR 97

13.1 A view of the hepatic portal system 99
13.2 Three-dimensional model of a pig liver 100
13.3 The original liver scan 100
13.4 Liver scan with threshold 32250 101
13.5 Liver scan after median filter application 101
13.6 The three-dimensional model of a liver 103

List of Tables

3.1 Sample values for HU 16

4.1 Comparison of several data structures 32

5.1 Incidence queries 50

7.1 benchmark times relative to C 64

7.2 Modules loading in Julia 65

For/Dedicated to/To my...

Part I

The problem and the current state of the art

Chapter 1

The need for scientific visualization

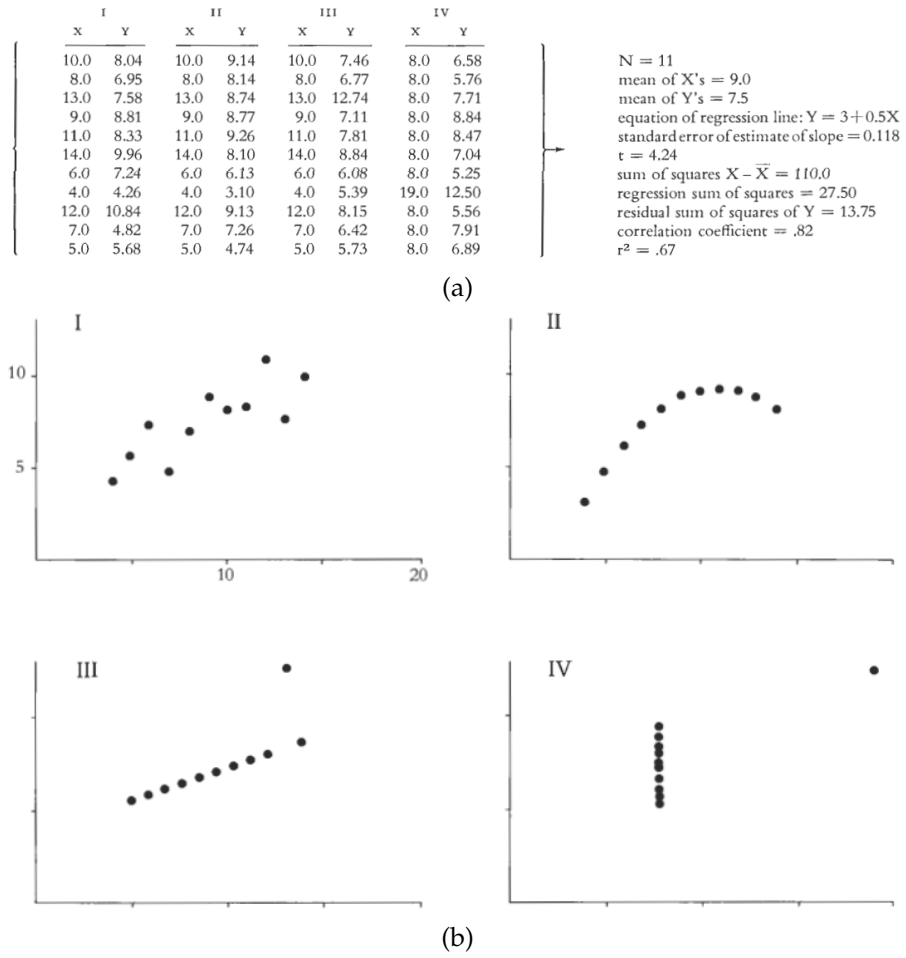
One of the computer science fields that is deeply changing in these years is *information visualization*. According to Wikipedia, it is the study of (eventually interactive) visual representations of abstract data with the purpose of finding knowledge from it. It is important because it exploits human vision, which is our fastest sense. As we can see in [18], the retina of a guinea pig can transfer 875,000 bits of information per second, and the same scientists estimate that a human eye can transfer data at the same speed of an Ethernet cable (approximately 100 Mb/s). For example, the data transfer of our ears is less than 100 bits per second. Moreover our vision is parallel and pre-attentive and permits us to recognize patterns in a simple way (see Figure 1.1).

Abstract data used in information visualization can be both numerical and non-numerical. However if the data is generated by scientific inquiry we say we have **scientific visualization**. Another definition can be taken from [23]:

Definition 1.1 (Information visualization)**.** *It's infovis [information visualization] when the spatial representation is chosen, and it's scivis [scientific visualization] when the spatial representation is given*

As shown, information visualization and scientific visualization are quite similar sciences although they are not identical. In fact, the former represents entities with no direct physical correspondence and devises a physical metaphor; the latter represents something physical or geometric. As this work is based on scientific visualization, and in particular on the representation of biological models, we will focus only on it.

Scientific visualization is an interdisciplinary branch of science. According to [13] it is:



I		II		III		IV	
X	Y	X	Y	X	Y	X	Y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

N = 11
mean of X's = 9.0
mean of Y's = 7.5
equation of regression line: Y = 3+0.5X
standard error of estimate of slope = 0.118
t = 4.24
sum of squares $X - \overline{X} = 110.0$
regression sum of squares = 27.50
residual sum of squares of Y = 13.75
correlation coefficient = .82
$r^2 = .67$

(a)

(b)

FIGURE 1.1: An example for pattern recognition taken from [26]. (a) Tabular data with a statistical analysis. (b) Visualization of the data. As we can see, we can find more relationships from data using information visualization.

"primarily concerned with the visualization of three-dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component".

As established from this definition, it can be considered a subset of computer graphics because of the several concepts taken from that discipline.

Now we can wonder why this science is so important. In effect, in these years we have observed continuous improvements in scientific fields. In particular computer science may have aided this progress because it permits to simply process data from other sciences, automating

processes in a way it was not possible in the past centuries. Moreover, the enormous improvements of technologies, provoked the increase of data originated by experiments and simulations; computer science (and in particular scientific visualization) has valid methods for mastering problems arising from their volume.

In the next part of this chapter, we will study some interesting characteristics of scientific visualization

1.1 Scientific visualization applications

Now we will observe some interesting application of scientific visualization, so that we can understand its importance.

Natural sciences Common applications of scientific visualization for the natural sciences are:

- Rendering of molecules
- Visualization of gravitational waves and other physical phenomena of interest
- Study of fluids

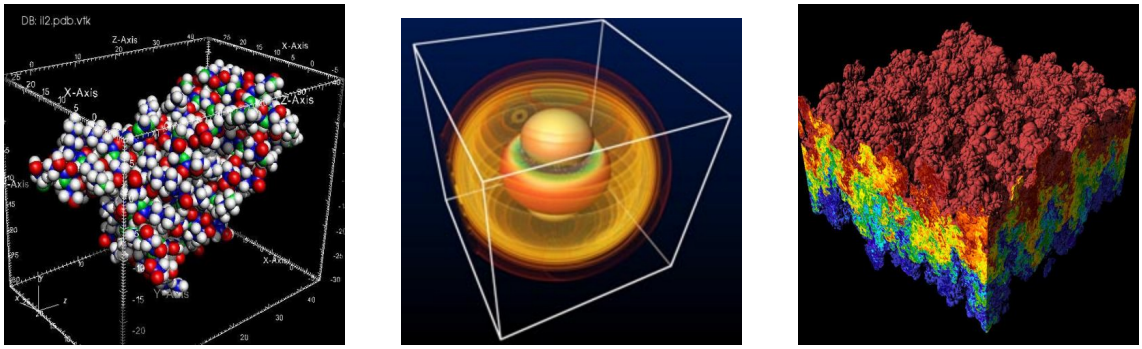


FIGURE 1.2: Examples of scientific visualization applications in natural sciences. (a) "Molecular rendering" by UCRL-WEB. (b) "Gravitywaves" by The Globus software creators Ian Foster, Carl Kesselman and Steve Tuecke. (c) "Rayleigh-Taylor instability" by Lawrence Livermore National Laboratory. All images were taken from Wikipedia

Geography Common scientific visualization applications for the natural sciences are:

- Rendering of terrains

- Simulations of atmospheric Anomalies
- Climate visualization

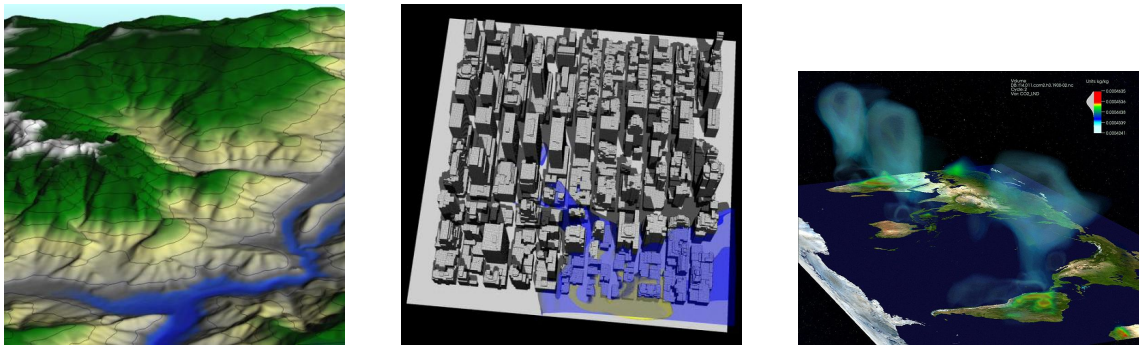


FIGURE 1.3: Examples of scientific visualization application in geography. (a) "Terrain rendering" by UCRL (b) "Atmospheric Anomaly in Times Square" by UCRL and Andrew Wissink Ph.D., LLNL. (c) "Climate visualization" by UCRL and Forrest Hoffman and Jamison Daniel of Oak Ridge National Laboratory. All images were taken from Wikipedia

Applied sciences Common scientific visualization applications for the natural sciences are:

- Visualization of medical images (some techniques will be studied in Chapter 3)
- Studies on materials

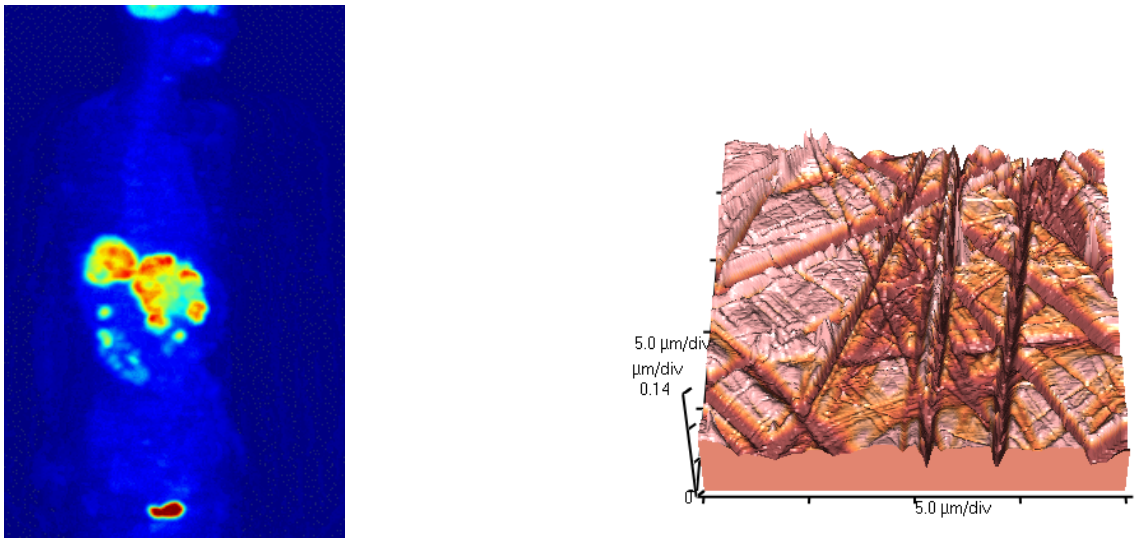


FIGURE 1.4: Examples of scientific visualization applications in applied sciences. (a) "Maximum intensity projection (MIP) of a whole body PET scan" by Jens Maus. (b) "Atomic force microscope topographical scan of a glass surface" by Chych. All images were taken from Wikipedia

Chapter 2

The rise of big data era

In the last years, the computer science world is witnessing the rise of the *big data* era. In fact world population is increasing continuously and, as shown in [27] at the beginning of January 2016 there were 7,392 billions of people. As the population grow, also data produced by people grow (thanks to the contemporary diffusion of technologies). According to an article published on Teradata Magazine in 2011 we can say that:

> "Big data exceeds the reach of commonly used hardware environments and software tools to capture, manage, and process it with in a tolerable elapsed time for its user population."

The next part of this chapter will show the importance of these data in the modern world and the new challenges they offer.

2.1 Importance and value of data

As we can see in [25], the digital universe is doubling in size every two years and by 2020 it will contain as many digital bits as the stars in the universe (about 44 zettabytes). These data come from a huge number of different sources like tweets, videos, sensors, etc..., and their union is important for modern systems (as in the *internet of things*). These data are defined and can be processed by software, for extraction of the various informations. Companies that will be able to master big data will gain an enormous economical power. Kaggle CEO Anthony John Goldbloom, says:

> "The bigger the data set you have, the more accurate the predictions about the future will be"

This means that our predictive systems work better when they have a lot of data to examine and this is the main reason for dealing with big data. In addition, from a scientific point of view, accurate experiments require a big amount of data to work on. An example is given by *Google Translate*. The popular translation service use collections of snippets of translations which are continuously debugged by users. It works fine because Google has tons of snippets from the indexed web pages, and the accuracy of the service improves as the training set grows.

Summing up the characteristics we have already seen, IBM data scientists break big data into four dimensions:

- **Volume**: big data volumes are typical in the order of Terabytes and are growing year by year
- **Variety**: big data come from a huge number of sources
- **Velocity**: speed in streaming data processing is essential, otherwise we will be overwhelmed by these big volumes. For example modern cars have about 100 sensors, so we need to be fast to process their informations to make the correct decision about driving. Another example come from finance world, in fact the New York Stock Exchange, captures 1 Terabyte of trading information that needs to be fast processed to make the right decision in the lesser time possible.
- **Veracity**: as already said we need big data to produce better predictions from our analysis systems. However, we need to have the best data possible in order to predict as *accurate* as possible. This dimension is so important that IBM data scientists said that poor data quality costs the US economy around 3.1 trillions of dollars a year

2.2 Typical problems when dealing with big data

In the earlier section, we have read about the advantages of big data. However, as shown in [25], the digital universe may actually be an obstacle for companies trying to become data-driven. There is too much information, it is too diverse, and it is too effervescent. Summing up we can say that:

- there is more heterogeneity

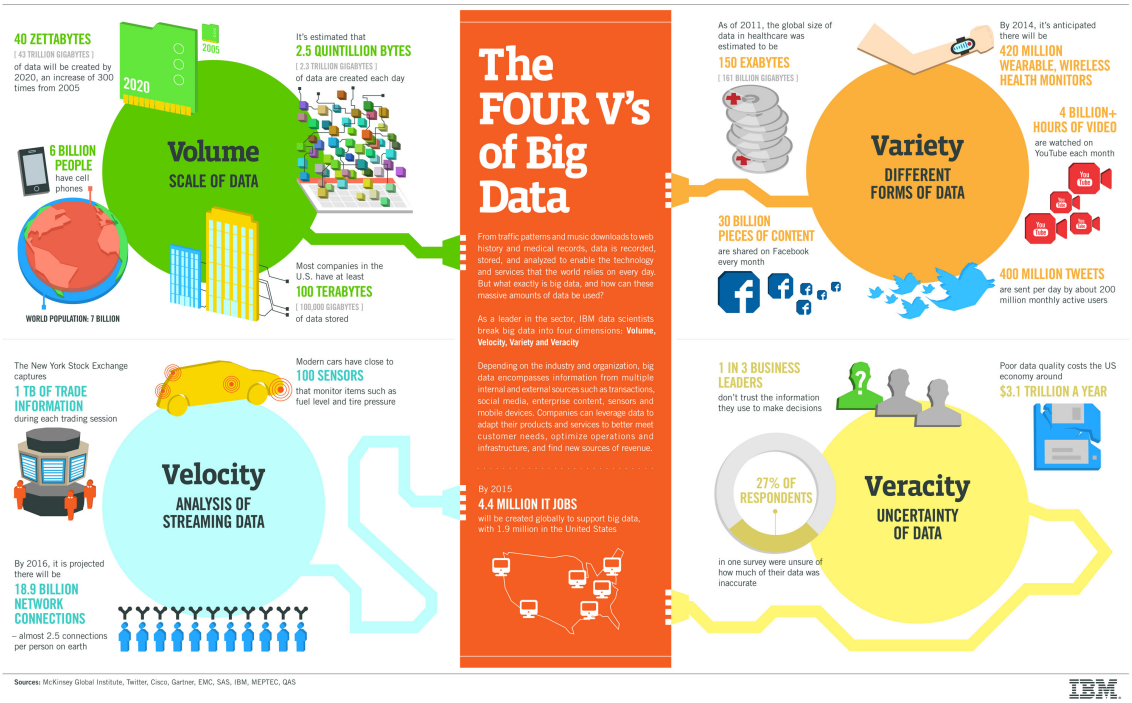


FIGURE 2.1: The four V of big data according to an infographic distributed by IBM data scientists

- data grows faster than energy in chip
- we still need humans to ask right questions for information extraction

These problems sometimes lead to wrong results from big data analysis. For example as pointed out in [10], *Google Flu Trends* service (which provides estimates of influenza activity by aggregating Google search queries), overestimated flu prevalence (sometimes it predicted twice as many doctors' visits). One of the problems was that people making searches about flu, knew very little about how to diagnose it. So searches for flu or flu symptoms may regard diseases that are similar to flu but are not. Moreover changes in Google search algorithm over time, have raised concerns about the meaning of its predictions.

Another problem with big data is output misunderstanding. In fact after we have applied an analysis method on our big data, we have to be sure about conclusions; sometimes we could find strange correlations, unexpected results (like Google Flu Trends) or we risk to bad interpret the data.

We can also have technical problems when dealing with big data. They substantially regard computation speed and **memory consumption**. In particular the latter is the worst one, because we could tolerate waste of time in a lot of applications, but we cannot tolerate an algorithm that does not work when input grow too much. A common solution to these problems consists in *scale out* our architectures distributing the algorithm as we will see in Chapter 6. Another solution consist in choosing a better analysis technique. As we have already seen in Chapter 1, human vision has a great ability to synthesize big data. For this reason scientific visualization can represent a great help for these problems and in the following parts of this work we will see how it can be useful in medical research

2.3 Big data collections and big data objects

Now we will consider some issues in managing big data in the scientific visualization field. As we can see in [7], this is an old problem (the paper was published in 1997), nowadays still to be solved. According to this paper problems with big data come from:

- **Big data collections**
- **Big data objects**

Big data collections are aggregates of many datasets. The size of data may typically exceed the capacity of storage (usually disks), so it can be partitioned between several storage devices. Generally speaking, big data collections present the following problems:

- Data are distributed among multiple sites
- Data reside in heterogeneous databases
- Data are generally not self-describing
- Often they do not have meta-data
- There may be poor data locality for the queries
- Data access time may be quite high due to partitioning

Instead, big data objects typically comes from large-scale simulations and are usually multi-dimensional datasets. For example in computational fluid dynamics, a common data structure is

a set of curvilinear grids with computed values at the vertices of the grids.
Data objects, typically present the following problems:

- Multi-dimensional data structures are not adequately supported by database technology
- A single data object may not fit in the main memory or, in the worst case, it may be too big for local disks
- Latencies and bandwidths between the data store and main memory must be managed carefully

Obviously we can also have the combined problem: *big collections of big objects*.

Now we have understood the main problems that arise when dealing with big data especially in scientific visualization field. This knowledge will help us further on as we will learn of the methodologies for mastering big data complexity. In particular, we will see how to *scale out* our architectures using parallel programming, solving all problems regarding the volume of the data

Chapter 3

Medical imaging visualization

Now we will see an interesting application of scientific visualization which can also be an excellent test case for approaches in Chapter 5: creation of three-dimensional models from *medical images*. In fact, this is a good application, because the typical medical image contains a large amount of data. As pointed out by [24], the average CT procedure generates more than 2 million voxels (see Section 3.1) per patient examination and also other imaging techniques produce similar amounts of data. Moreover algorithms used for 3D medical imaging have great computational cost, even at moderate resolution.

A lot of the material that will be presented in this Chapter comes from [4], which represents an excellent source for the image processing world applied to medicine. It describes also the importance of the use of radiological images for clinical research. In fact, the author points out that their applications range from neuroscience, biomechanics and biomedical engineering to clinical routine tasks such as: the visualization of huge datasets provided by computed tomography systems, the manufacturing of patient-specific prostheses for orthopedic surgery or the planning of dose distributions in oncology. Here we will present the basis for understanding of techniques used to produce the images.

3.1 Medical imaging basis

3.1.1 The electromagnetic spectrum

Whoever wants to seriously study medical imaging should have a basic knowledge about the *electromagnetic spectrum*. In fact, most images are generated by recording physical properties of

tissues that are exposed by a certain type of electromagnetic (or sometimes mechanical) solicitation. Every radiation consists of *photons* (little "particles of lights") with an energy equal to $E = h\nu$ where *h* is the *Planck constant* ($6.626 \cdot 10^{-34} Js$) and ν is the frequency of the wave. How we can see, as *h* is a constant the energy of a wave is entirely described by its frequency. As a consequence, [4] uses the electromagnetic radiation frequencies for the definition of the following classification:

- $1 - 10^4$ Hz: it is the range of frequency of alternating current and it is used for **electrical impedance tomography**
- $10^4 - 10^8$ Hz: frequencies used for broadcasting and for the **magnetic resonance imaging (MRI)**
- $10^8 - 10^{12}$ Hz: microwaves used in oven, they are not used in medical imaging
- $10^{12} - 7 \cdot 10^{14}$ Hz: these are the frequencies of the infrared, used for **infrared imaging**
- $4.6 \cdot 10^{14} - 6.6 \cdot 10^{14}$ Hz: spectrum of visible light. It is used for **light microscopy** and for **histological imaging**, **endoscopy** and **optical coherence tomography**
- $4 \cdot 10^{14} - 10^{18}$ Hz: ultraviolet frequencies which are used in **fluorescence imaging**
- $10^{18} - 10^{18}$ Hz: X-rays, which are used for the **x-rays images**
- Beyond 10^{20} Hz: these are the frequencies of $\gamma-$radiation, which is usually a product of radioactive decay. Since $\gamma-$radiation may penetrate tissues easily it is widely used in nuclear medicine (**SPECT** and **PET**).

As we can see, medical imaging uses a large part of the electromagnetic spectrum. However there is a widespread imaging modality that relies on a different physical principle: **ultrasound**. These techniques will be studied further on.

3.1.2 Image representation

A digital image is represented by numerical values associated with positions in a regular grid. A single position is usually called **pixel** and its value is a gray value or a color. Anyhow, in medical image processing we often have three-dimensional representations. So, we can associate the numerical value mentioned above with a point in a three-dimensional space calling it **voxel**. Therefore an image is a discrete mathematical function which maps a vector from a two (or three)

dimensional space to a number. As we can read in [4], formally speaking an image is represented as:

$$I(x, y, z) = \rho \tag{3.1}$$

where x, y and z are the coordinates of the voxel (or pixel in a two-dimensional space) and ρ is the associated color value. We can also look at an image as a matrix (this kind of representation is the one we will extensively use in our application). So we can have:

$$I = \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ \rho_{21} & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{m1} & \rho_{m2} & \cdots & \rho_{mn} \end{pmatrix} \tag{3.2}$$

Now we will discuss the actual physical meaning of ρ. That scalar represents the gray scale value in a non-color image, but it can vary depending on the imaging technique chosen. For example, in x-ray imaging it can be the absorption of high-energy electromagnetic radiation from the x-ray tube in the patient. So we can say that data from a single image are *mathematical functions of measured values*. In general we can have several shades of gray between *black* (which represent no signal at all) and *white* (the maximum signal). However, the human eye cannot perceive so many details in gray levels (in effect it can only perceive approximately 100 shades of gray). As a consequence when processing medical images, we need techniques for manipulations of intensity which we will study in depth in Part 3 when we will examine the implementation of common operations on images.

3.2 Common imaging techniques

In this section, we will examine some of the most important techniques used to obtain medical images

3.2.1 X-rays

X-rays are generated when fast electrons interact with metal objects, in fact when they are accelerated by a high voltage the kinetic energy is partially transformed into electromagnetic energy. Usually, x-ray machines provide a great number of electrons which focus on a small spot in a

metallic material. Obviously, when we increment the kinetic energy of the electrons the intensity and the energy are both increased. According to [4] the X-ray spectrum so generated consists of two types of radiation:

- **Characteristics x-radiation**: it is generated when the electron interacts with one of the target atom. If the kinetic energy is sufficient to ionize the target atom, causing the removal of an electron from the inner shell of the atom with the consequently falling of outer-shell electrons into the inner shell, then we will observe this type of radiation. It is called characteristic because it is "characteristic" to each element
- **Bremsstrahlung or braking radiation**: it is generated when a fast electron interacts with the nucleus of a target atom. In fact, when an electron that avoids the orbital electrons while passing an atom can come sufficiently close to the nucleus for interaction, it is slowed down and deviated from its course causing a loss of kinetic energy and the release of an x-ray photon. As opposed to characteristic radiation its energy distribution is continuous

The typical source of X-rays is the *X-ray tube*. It consists of a *vacuum tube*, a *cathode* for electron creation and the target *anode*.

Now we can look at how it is possible to measure X-ray radiation as we will be able to create our images. The attenuation of X-rays follows the *Beer-Lambert* law:

$$\frac{dI}{ds} = -\mu I(s) \tag{3.3}$$

Which has the following solution:

$$I(s) = I_0 \cdot e^{-\mu s} \tag{3.4}$$

In these relationships I denotes the *intensity* of the electrons beam (where I_0 is the initial number of electrons), s is the thickness of the absorber and μ is the *linear attenuation coefficient* (which is material dependent). Generally speaking, the attenuation of the X-rays comes from three effects:

- The photoelectric effect
- The Compton effect
- The pair production effect

So we can write that $\mu = \mu_{photo} + \mu_{Compton} + \mu_{pair}$.

This attenuation is very important and is at the basis of X-ray imaging as we are interested in different attenuation in tissues. In fact, we have seen in Equation 3.3 that it is governed by the atomic number of the material, by the tissue density, photon energy and material thickness (which has an exponential effect on the intensity). For example, bones have high attenuation properties, so they are easy to distinguish from soft tissues. As a consequence, since the early days of X-ray, bone imaging was the most common, while other tissues and vessels are much difficult to visualize. To solve this problem one can use computer assisted contrast enhancement, or a *contrast media* (which is a substance administered to the patient with high or low attenuation properties). We can also have problems with scatter radiation, noise and blur, which reduces the signal to noise ratio for our images. The first is a problem when the target is thick, for example for a thorax [4] argue that the fraction of scattered radiation is more than 50% of the available imaging radiation, which results in a loss of contrast. To reduce this effect anti-scatter grids can be positioned between patient and detector. Instead, noise can be generated by statistical fluctuations of the photons (*quantum noise*). It can be reduced by increasing the number of photons; however this also increases the applied dose with incremented health risks for the patient. Other sources of noise come from the instruments used for imaging. Finally the blur comes from the finite size of the focal spot (which is not a single point as supposed in the theory)

FIGURE 3.1: An X-ray image of an elbow. Image taken from Wikipedia

3.2.2 Computed tomography

Now we will examine another important imaging technique: **computed tomography**. In this technique, a X-ray beam penetrates a slice of the patient, and a detector on the other side measures the intensity of the residual beam. This procedure is then repeated with different angles of the detector, obtaining projections from many different directions and a two-dimensional integral

of attenuation. From these intensity profiles, it is possible to reconstruct a two-dimensional image, displaying a gray value that is proportional to the attenuation. The voxel value is given in *Hounsfield units (HU)*.

TABLE 3.1: Sample values for HU

HU	**Material**
-1000	Air
0	Water
>1000	Metals
Between 50 and 100	Bone tissue

Now we can focus on the results obtained by this process. The raw data that results from series of X-ray scans is called *sinogram*, which is simply a display of all different projections for a given slice stacked together.

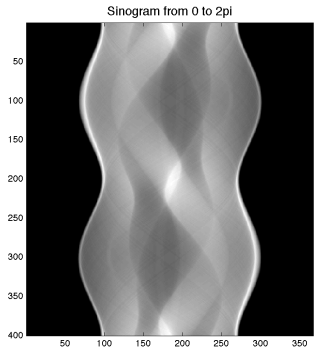


FIGURE 3.2: A sinogram from a CT. It has sinusoidal undulations due to the CT tube rotations around the patient

However, the sinogram is not sufficient for interpretation yet. So once the scan data has been acquired, it must be processed using a form of *tomographic reconstruction*, which produces the series of images we were looking for.

Because CT scanners offer *isotropic* (uniform in all orientations) or near isotropic, resolution, display of images does not need to be restricted to the conventional axial images. Instead, it is possible for a software program to build a volume by "stacking" the individual slices one on top of the other. We will discuss techniques for the volume rendering in Chapter 4 but it is important to remember that the idea of stacking the individual slices is at the basis of the work discussed in this thesis.

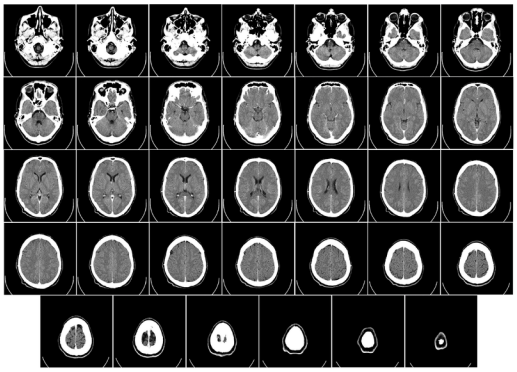


FIGURE 3.3: Computed tomography of human brain, from base of the skull to top.
Image taken from Wikipedia

3.2.3 Magnetic resonance tomography

Now we will discuss images obtained using the **magnetic resonance tomography**. It works exploiting the magnetic moment of ^{1}H nuclei bound in tissue water and fat molecules. With this technique, electromagnetic radiation in the radiofrequency range is transmitted to the body using a large magnet. The nuclei respond to the excitation by emission of RF signals with the same frequency, whose amplitude and phase are used for image construction. Information on slice location is obtained by superimposing spatial field gradients to the main magnetic field. As every voxel has a different frequency or phase we can divide signals coming from a single portion.

Imaging with this technique uses three independent processes:

- **Selection of a slice**. With the application of a gradient on a given direction, we can observe that the frequency of the atoms linearly changes along that direction, so the body inside the magnet can be subdivided in parallel planes. When we apply the radiofrequency impulse it will excite only one plane leaving the other in the equilibrium state.

- **Frequency codification**. If we apply a gradient after the radiofrequency impulse and during the signal acquisition, the emission frequency of the protons varies linearly on the space. So the acquired signal is the sum of signals at different frequencies that can be obtained with a Fourier transform. When we associate a position with a single frequency we obtain the localization on a single dimension. However we need the phase codification for spin localization

- **Phase codification**. When we apply a gradient in the second spatial dimension after the radiofrequency impulse and before the acquisition, the spins along that direction will acquire

a phase which can be easily measured. A single phase codification is not sufficient to obtain spatial information so the procedure should be repeated several times.

As a consequence of these processes, different impulse **sequences** lead to different images. The chosen sequence also determines the physical property that is represented into the resulting images. We will say that images representing different physical properties have different **contrast**

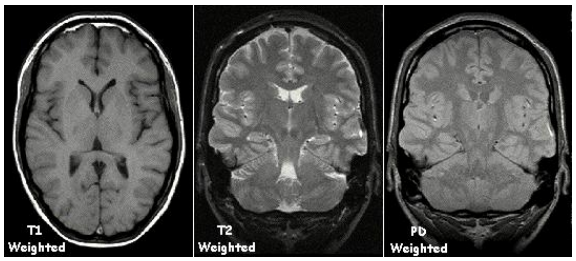


FIGURE 3.4: MRI images of a brain with different contrasts. (a) T_1 weighted (b) T_2 weighted (c) PD weighted. Image taken from Wikipedia

3.2.4 Ultrasound

Now we will see how to exploit sound for the creation of our medical images. First of all, we should remember that sound is a longitudinal pressure wave based on compression and decompression of the carrier medium. Therefore, the speed of sound c depends on the compressibility and density of the carrier medium and its value is:

$$c = \lambda\nu \tag{3.5}$$

For tissues (excluding bones where the speed is higher) the average value is *c = 1540 m/s*. While passing the body, the acoustic signal can be affected by *reflection* on tissue boundaries, *refraction* on border structures and *scattering* (which occurs on structures with sizes comparable to the wavelength of sound and with different density). These effects cause misinterpretation in positions and extension of the studied structures. Also signal attenuation can have a role in this process.

Now we can see how to produce the acoustic signals for the images. They are generated using the *inverse piezoelectric effect*. According to the *direct piezoelectric effect* phenomenon, voltage can be measured in certain solids under deformation; on the other hand, we have the inverse effect when applying voltage to these solids we cause the deformations.
When we want to produce images, A piezoelectric US transducer transforms an electrical pulse

into an acoustic signal which penetrates the object and is reflected on internal structures. The *echoes* of the reflected pulses are detected by the transducer and converted back to electronic signals. This imaging technique is called **amplitude modulation** or **A-mode ultrasound imaging**. The percentage of reflected waves is dependent from the difference in acoustic impedance of the tissue involved. As this difference is usually small, the main part of the sound energy passes the boundary surfaces, making possible localization of organs lying behind each other by measuring the temporal distance between the echoes. The distance z of a boundary surface is:

$$z = \frac{ct}{2} \tag{3.6}$$

Where t is the return time of the echo signal. However, echoes with longer run time are weaker due to absorption so they should be amplified.

As the A-mode technique only provides one-dimensional information, other imaging methods were developed. One of the most important is **B-mode** or **brightness modulation**. With this technique the echo amplitudes are converted to gray values in a two-dimensional image.

US imaging is also able to display flow speeds of fluids (such as blood) exploiting the *Doppler effect*. This effect, consists in a shift of frequencies of a wave when its source is moving away from (or moving towards) an observer. A typical example is the shift in sound waves when an ambulance approaches with a siren. In US imaging, red blood cells can be seen as a source of ultrasound waves when they reflect the input signal. The frequency difference which results from the Doppler effect is:

$$\Delta\nu = 2\nu(\pm\frac{v}{c}) \tag{3.7}$$

Where v is the velocity of blood. For Doppler ultrasound usually two techniques are used:

- **Color Doppler**: we have informations regarding mean speed of the medium (good for a wide volume study)
- **Gated Doppler**: we obtain the spectrum of every speed in the medium (well suited for a detailed study)

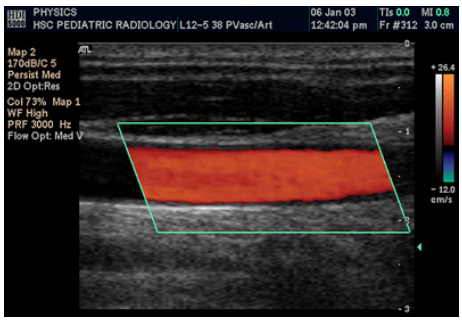


FIGURE 3.5: Color Doppler of a carotid artery. As we can see Ultrasound images usually have a low resolution (we typically have 256×256 with 8 bit per pixel). Image taken from Wikipedia

3.2.5 Nuclear medicine and molecular imaging

Now we will examine several medical imaging techniques which exploit *nuclear medicine*. The main difference with x-ray diagnostics, is that the ionizing radiation is emitted by the patient rather than an outside instrument. For visualization of metabolic activities in organs we commonly use the *tracer method*. With this method the radio nuclide (the *tracer*) is bound to a chemical complex taken up by the organ like any metabolic substance, for example ^{99}Tc is used in most applications. After it is injected, the tracer releases $\gamma-$radiations due to the decay of the nuclide which are detected and displayed as image intensities. Observing the distribution of this activity, we can visualize the metabolism of an organ. As a consequence, these methods show the physiological function of the system, not its structure. So we can merge images obtained with these techniques with images taken with other modalities to acquire more anatomical informations. The common term used for imaging the distribution of radioactivity is **scintigraphy**.

A particular scintigraphy based technique is **SPECT**, which uses rotating $\gamma-$cameras. Projections of the activity distribution are recorded at discrete angles and used for reconstruction of a three-dimensional volume. Another technique is **PET**, which uses positron (β^+) emitters as tracers. These kind of tracers are nuclides with an excess number of protons compared to the number of neutrons, so they are unstable and decay by converting a proton p into a positron e^+, a neutron n and a neutrino ν, as stated by the following equation:

$$p \rightarrow e^+ + \nu + n \tag{3.8}$$

This is known as the β^+ *decay*. We cannot detect directly the positrons as their free path before

annihilation is very small, however we can detect the $\gamma-$rays coming from the annihilation of the positron with an electron (which produces exactly two $\gamma-$quanta). To distinguish the annihilation $\gamma-$quanta from quanta resulting from other processes, we can use *co-incidence detection*. So only events where two quanta are detected simultaneously are counted.

Image reconstruction with PET is similar to the one of CT even thought in this case the raw data set is of lower quality

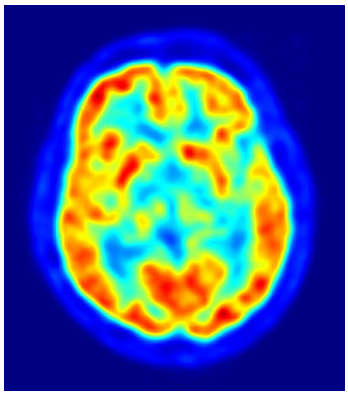


FIGURE 3.6: PET scan of a human brain. Image taken from Wikipedia

3.3 The need for three-dimensional representations

At this point, we have collected a general overview of the medical imaging field. Therefore, we can examine typical uses in the everyday world. [24] offers a good case of study. In this example are collected 63 MRI image slices of the head from a patient with intractable seizure activity. From these slices the doctor cannot disclose an abnormality. After the reconstruction of a three-dimensional model of the MRI, a study reveals flattening in the gyri of the lower motor and sensory strips. So the doctor asks for a second study using a PET to portray the metabolic activity of the brain. Using these images it is also possible to build a three-dimensional model of the average cortical metabolic activity. According to it, the patients has hypermetabolic activity; however as the PET has a poor resolution, the location of the abnormality cannot be correlated with surface anatomical landmarks. So the doctor has to combine the 3D MRI model with the 3D PET model, using post-hoc image registration techniques to align the two sets of data. With these representations, the doctor can also use a surgery test program to simulate surgery on the brain

model and after the operation the patient's seizure activity cease.

Another interesting use of three-dimensional models consists in the **printing of human parts with a 3D printer**. The following case study comes from [28]. In February of 2012, a medical team at the University of Michigan carried out an unusual operation on a three month old boy. In fact he was born with a rare condition called *tracheobronchomalacia*, which causes weakness in the tissue of the airway. This made breathing very difficult causing also blockage of vital blood vessels. The area of weak tissue would have somehow needed to be repaired or replaced with a dangerous operation for a patient so young. As a consequence, they decided to print a *tracheal splint* with a 3D printer. The researchers began by taking a CT scan of the baby's chest which they converted into a three-dimensional model. Using this model they designed and printed the splint with biocompatible materials. After the implantation, problems with the tracheobronchomalacia disappeared. As we can see, without a technique for the three-dimensional representation of models, the researchers would have not been able to build the splint.

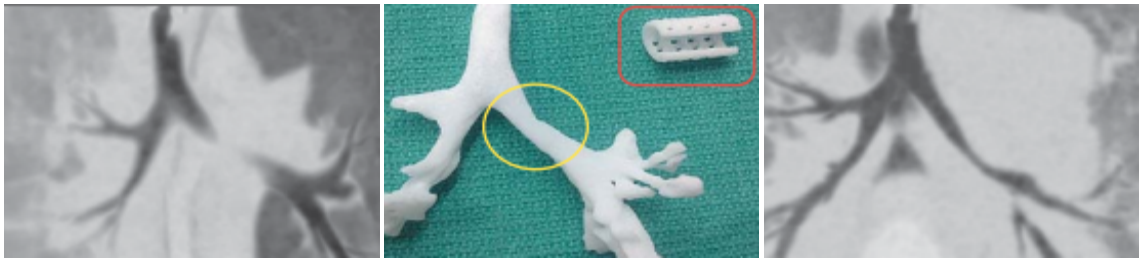


FIGURE 3.7: 3D print of a tracheal splint. (a) The 3D printed models for the airway and for the splint. (b) The airway in expiration before placement of the splint. (c) The airway in expiration one year after placement of the splint. Images taken from [28]

As we can read in [16], three-dimensional models can also be used for postoperative investigations. For example in that work the authors created a model from helical CT scans of the temporal bone for investigations in cochlear implantation.

However these are only a few examples of things we can do with three-dimensional representations of human parts. Later we will study a good technique to produce perfect models

Chapter 4

Current state of art in computer graphics

In this chapter, we will examine the state of art of **computer graphics**. Our aim is to understand techniques which are actually used for visualization of the three-dimensional biomedical models, and to provide a basis for comprehension of the innovative approaches we will show further on. A lot of this material comes from [17], which is a classic book for this field

Here, we will also show the actual techniques used for the generation of volumes and surfaces from the biomedical images so we will see the difference with our approach

4.1 Representation of shapes in graphics: meshes

The first thing we will examine is the most widely used representation of shapes in graphics: **meshes**. At first we will focus on *triangle meshes* but we could define meshes with n vertices. Triangle meshes simply consist of many triangles joined along their vertices to form a surface. They have some interesting properties, but the most important is **coplanarity** of vertices of a single triangle.

Two interesting operations on meshes are *subdivision* (where a single triangle is replaced with several small triangles for smoothing a shape) and *simplification* (when a mesh is replaced with another that is similar but has a more compact representation). As a consequence of these properties and operations, we are able to represent whichever shape approximating it with triangles (see Figure 4.1), we will have an evidence of this when we will see models from our application.

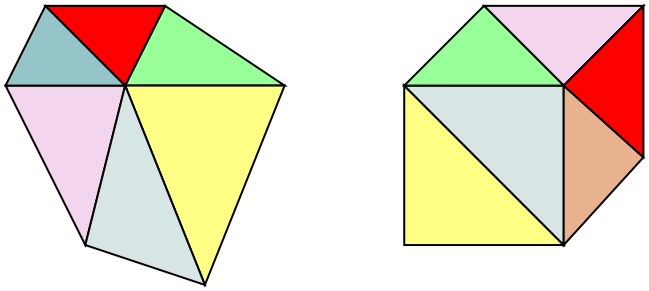


FIGURE 4.1: Creation of shapes with composition of triangular meshes. As we can see using this composition we can create whichever shape we want. At the same time we can decompose every complex shape into triangles

Now we can focus on the representation of three-dimensional meshes. The simpler data structure (and the one we will use later) for description of these meshes consists in a *list of vertices* and in a *list of triangles*. With this structure we can also infer edges of the meshes using topological techniques. With this structure, insertion of triangles and vertices is fast ($O(1)$) but deletion of vertices is slow (it is $O(n)$ because we need to find all associated triangles).

4.1.1 Manifold and Nonmanifold meshes

Some particular meshes are the **manifold meshes**. According to [17], we can introduce the following definition:

Definition 4.1 (Manifold meshes)**.** *A finite 2D mesh is a manifold mesh if the edges and triangles meeting a vertex v can be arranged in a cyclic order*
$t_1, e_1, t_2, e_2, \ldots, t_n, e_n$ *without repetitions such that edge e_i is an edge of triangles t_i and t_{i+1}*

This implies that for a manifold mesh every edge is *incident with at most two faces*. These structures are central to many parts of geometry and modern mathematical physics because they allows more complicated structures to be described and understood in terms of the relatively well-understood properties of Euclidean space.

Often, we should care about the orientation of triangles in a mesh (for example when we consider lighting). In fact, orientation is fundamental when we want to find the *normal vector* of the face and decide what is inside and what is outside our meshes. Moreover, when two adjacent triangles have consistently oriented normal vectors, then the edge they share will appear as (i, j) in one triangle and as (j, i) in the other. And if a manifold mesh can have its triangles oriented, it

can also be given consistently oriented normal vectors.

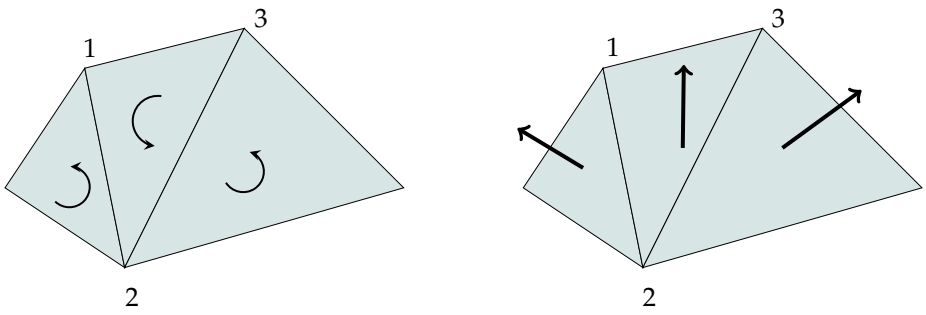


FIGURE 4.2: Meshes orientation and normals. As we can see in (a), for oriented faces we always have that for a common edge between two faces in the first we will have (i, j) and in the second we will have (j, i). In the example we can observe $(1, 2)$ and $(2, 1)$

More interesting than manifold meshes are meshes whose vertices are manifold or **boundary-like**. In the last case, vertices do not form a cycle but a chain, whose first and last elements share only one of their edges with other triangles in the chain. The other edge of the first (or the last) triangle that meets the vertex is contained only in the first (or the last) triangle, and not in any other triangle of the mesh. This unshared edge is called a **boundary edge**, and the vertex is called a **boundary vertex**. In general, an oriented mesh without boundary edges is called **closed**

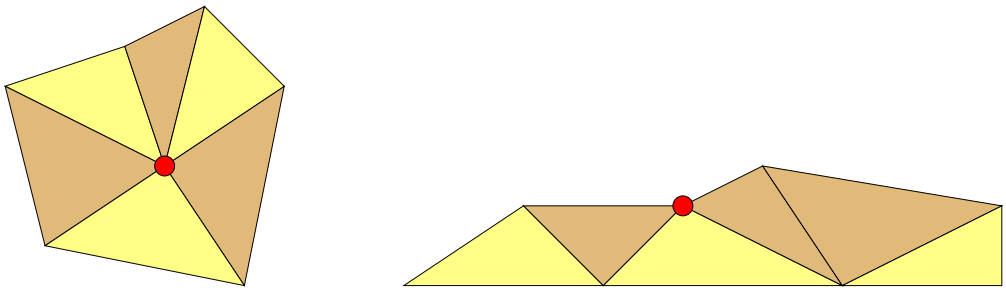


FIGURE 4.3: Manifold and boundarylikes vertices. (a) A manifold vertex (in red) has a cycle of triangles around it. (b) A boundarylike vertex v has a chain of triangles around it. All edges that contain v are boundary edges

Finally, we can define **nonmanifold vertices** as the vertices which are neither manifolds nor boundarylike.

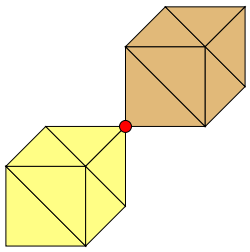


FIGURE 4.4: Nonmanifold vertices. The shared vertex (in red) between the two manifold shapes is nonmanifold.

4.2 Typical structures for meshes representations

Now we will examine some of the most common data structures used for mesh representation. We have already seen the simpler one, based on lists of vertices and faces and very advantageous because of its simplicity. Moreover with the concept of **shared vertices** between faces, we can save memory space. In fact, all triangles sharing a common vertex store pointers to the same physical instance. As a consequence a vertex in a certain position needs to be specified only once.

However a lot of common operations of computer graphics need more complex structures, that we will study in depth here.

4.2.1 Directed-edge structure

Usually, objects represented by triangle meshes contain isolated vertices or edges that are locally non-manifold, while many data structures are restricted to orientable two-dimensional manifold triangle meshes. So, these structures are not capable of storing such an object. Other structures which can store them, typically require a lot of memory because they have to handle non-manifolds all over the whole object. This is an important point because memory can be considered an *hard limit*, while time is only a *soft limit*. In general the implementation of a data structure is a compromise between memory requirements and performance gains. For this reason [6] shows a *scalable data structure*: the **directed-edge structure**.

We know that a triangle is represented by three direct edges, so the common edge between two adjacent triangles is composed by the following two direct edges:

$$v^a \rightarrow v^b$$

$$v^a \leftarrow v^b$$

where v^a and v^b are two vertices of a triangle. In the simplest version of the directed-edge structure, for every direct edge we store:

- v^a: start vertex
- v^b: end vertex
- e^{pv}, e^{nx}, e^{ng}: previous edge, next edge and neighboring edge

We can insert these values into an array where the i-th triangle is represented by direct edges in positions *3i*, *3i + 1*, *3i + 2*. As a consequence, we do not need to store explicit references between a single triangle and its edges. We can save up more memory, if we consider only triangular meshes. In fact, we can safely remove v^a for one edge e computing v^b for the previous edge of e. We can also find the next edge searching for the previous of the previous edge. So, we do not really need to store v^a and e^{nx}. With a similar strategy we can remove e^{pv} using constant-time computations.

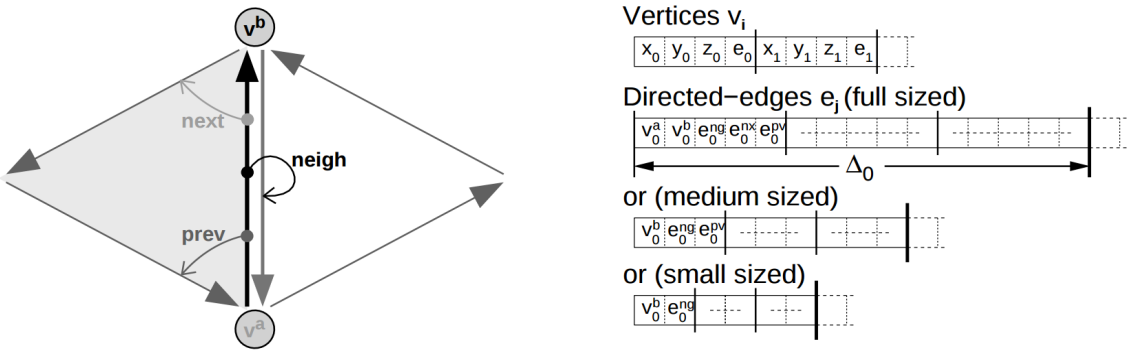


FIGURE 4.5: The directed-edge structure (a) A simple drawing of two triangular meshes with the interesting values for the data structure. Grey values can be exchanged with further computations (b) The three variants of the directed-edge data structure (note the different memory occupation). Images taken from [6]

However, as we have seen above we can have non-manifold vertices and edges in many applications. So [6] introduced little modifications to this structure assuming that the number of non-manifold entities is typically negligible compared to the complexity of the whole mesh. We can use the following strategy:

- For every couple of edges that represent a 2-manifold interior edge we can use zero or positive indexes.
- A directed edge on boundary is marked with a negative entry for e^{ng}

- Remembering that a 2-manifold vertex references one of its emanating edges, we can use the same strategy for non-manifold vertices and edges. A negative array-index indicates such a node and we can use this value to index a separated array which lists either all edges emanating from that vertex or the connected components attached to that vertex (as we can see in Figure 4.6)

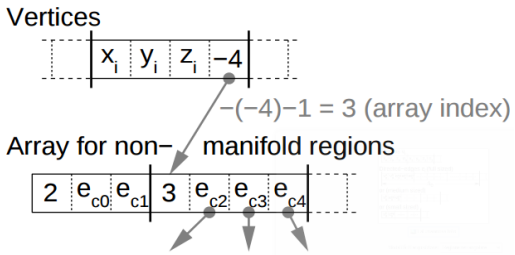


FIGURE 4.6: The directed-edge structure for non-manifold vertices and edges. Image taken from [6]

4.2.2 Winged-edge structure

Now we will see one of the oldest data structures proposed for mesh representation which was introduced by [3]: the **winged-edge structure**.

This structure explicitly describes the geometry and the topology of faces, edges and vertices when three or more surfaces share a common edge. Here is important to remember, that faces are ordered counter-clockwise with the respect of the innate orientation of the intersection edge. For this structure we need to save:

- Vertices of an edge e
- Left and right faces of e
- The predecessor and the successor of e when traversing its left face
- The predecessor and the successor of e when traversing its right face

In Figure 4.7 there is the representation schema.

How we can see, we need three arrays to memorize all informations about our meshes. In particular, observing the Figure 4.7 we can say that e^a, e^b and e^d are the *wings* of edge $v^a v^b$ which is called *winged*. This data structure is interesting because we can easily compute the relationships of adjacency between structures (some even in constant time).

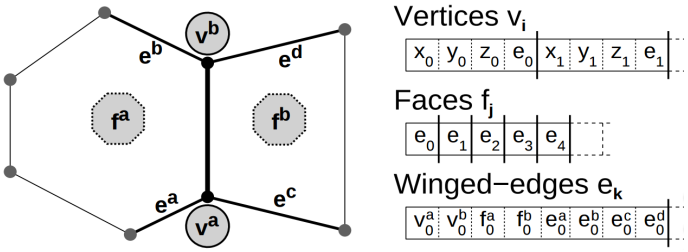


FIGURE 4.7: The winged-edge structure. Image taken from [6]

4.2.3 Quad-edge structure

The **quad-edge data structure** has been introduced by [14] and is a representation of the topology of a two-dimensional or three-dimensional map (which is a graph drawn on a closed surface). In this structure, we represent simultaneously the map, its dual and mirror image. The main idea behind the quad-edge structure, is the recognition that an edge in a closed mesh topology sits exactly between two faces and two vertices. As a consequence, it can represent a dual of the graph simply reversing the convention on what a vertex is and what a face. However, given this definition, we have that the quad-edge can only represents manifold edges. It is easy to use this structure iteration through the topology, so it can be useful for computation of adjacency relationships.

4.2.4 Half-edge structure

The **half-edge structure** (also known as **doubly connected edge list** or **DCEL**) has been introduced by [22] and has been designed to represent an embedding of a planar graph in the plane and polytopes in three-dimension. This data structure provides efficient manipulation of the topological information associated with the various objects (vertices, edges and faces); so it is commonly used to handle polygonal subdivisions of the plane (for example for computation of a Voronoi diagram). In the general case, a DCEL contains a record for each edge, vertex and face of the subdivision, and each record may contain additional information. As each edge usually bounds two faces, it is convenient to refer to an edge as two half edges. Each one bounds a single face and has a pointer to that face; so we can go to the other half-edge and traverse the other face. Moreover, each half-edge has a pointer to its origin vertex.

In addition, each vertex contains its coordinates and a pointer to an arbitrary edge that has the vertex as its origin, and each face stores a pointer to some half-edge of its outer boundary. It

also has a list of half-edges, one for each hole that may be incident with the face; however, if the vertices of faces do not hold useful information we can save space by avoiding to store them.

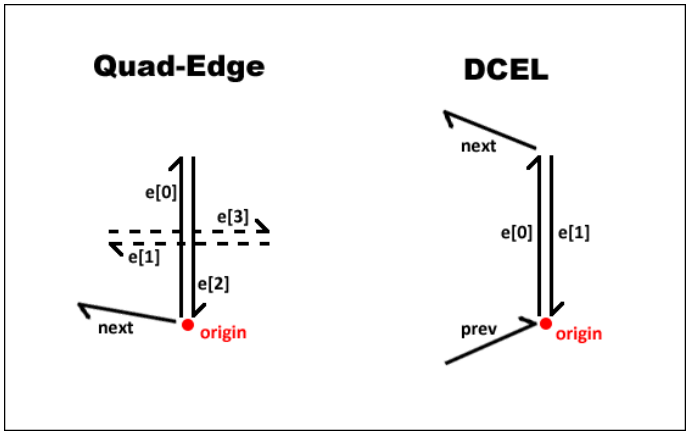


FIGURE 4.8: Comparison between the quad-edge and half-edge structures. Image taken from https://code.google.com/archive/p/cgprojects09/

4.2.5 Memory requirements for mesh structures

As we have seen in previous chapters, our aim is the manipulation of great volumes of data, so as size is an important variable, we need to study the memory requirements for the structures we have studied so far.

The simplest data structure is based on a list of vertices and a list of faces. Assuming that each integer is stored with four bytes, this representation uses $12V$ bytes for V vertices and $12T$ bytes for the T triangles. From the Euler's polyhedron formula we have that:

$$V - E + F = 2 - 2g \tag{4.1}$$

Where V is the number of vertices, E is the number of edges, F is the number of faces and g is the *genus* of the surface (the maximum number of cuttings along non-intersecting closed simple curves, without rendering the resultant manifold disconnected). As each triangle has three edges and each edge is shared by approximately two triangles the number of edges is $\frac{3}{2}T$. So we can write:

$$V - \frac{3}{2}T + T = 2 - 2g \tag{4.2}$$

Assuming that the genus is low with respect to the number of triangles, we can write:

$$V = 2 - 2g + \frac{1}{2}T \approx \frac{1}{2}T \tag{4.3}$$

So the number of vertices is twice the number of triangles. So the total memory (in bytes) needed by the initial data structure is:

$$12V + 12T = 6T + 12T = 18T \tag{4.4}$$

For the directed-edge data structure, each vertex uses 12 bytes for coordinates and four for an edge reference (faces are not represented). In the full sized version, each directed edge contains references to two vertices and three edges, using 20 bytes while in the medium and small sized versions we have respectively 12 and 8 bytes. So the three memory costs are:

$$16V + 20 \cdot 3T \approx 8T + 60T = 68T\ bytes \tag{4.5}$$

$$16V + 12 \cdot 3T \approx 8T + 36T = 44T\ bytes \tag{4.6}$$

$$16V + 8 \cdot 3T \approx 8T + 24T = 32T\ bytes \tag{4.7}$$

For the winged-edge structure each vertex uses 16 bytes, each face uses four bytes and each edge has four edge references, two face references and two vertex references. So we can write:

$$16V + 32E + 4T \approx 8T + 32\frac{3}{2}T + 4T = 60T\ bytes \tag{4.8}$$

We can make the same considerations for the quad-edge structure and the half-edge structure obtaining *30 bytes* and *38 bytes* for each triangle.

Finally, we can observe the Table 4.1 from [6], where the main aspects about memory used and time spent for the main relationships computations are summarized

TABLE 4.1: Comparison of several data structures. "M" indicates entities that are explicitly stored, "C" means that the information may be calculated in $O(1)$, and "-" marks entities that require computations of complexity $\gg O(1)$

Data structure	**bytes/△**	**Supports non-manifold**	**neigh-bor-hood**	**point refer-ence**	**prev. edge**	**next edge**
Individual triangles	36	yes	-	-	(M)	(M)
shared vertex	18	yes	-	M	(M)	(M)
shared vertex with neig.	32	no	M	M	(M)	(M)
winged-edge	60	no	M	M	M	M
full directed-edge	68	yes	M	M	M	M
medium directed-edge	44	yes	M	M,C	M	C
small directed-edge	32	yes	M	M,C	C	C

4.3 Volume rendering techniques

Volume rendering techniques are used for visualization of a two-dimensional projection of a three-dimensional *discretely sampled data set* (which is a *three-dimensional scalar field*). A typical example of these scalar fields is represented by 2D slice of medical images (seen in Chapter 3) which are also called *range images*. [8] points out a set of desirable properties for a volume rendering algorithm which we will report here:

- **Representation of range uncertainty**. The data in range images typically have asymmetric error distributions with primary directions along sensor lines of sight.
- **Utilization of all range data**, including redundant observations of each object surface.
- **Incremental and order independent updating**. Incremental updates allow us to obtain a reconstruction after each scan or small set of scans and to choose the next best orientation for scanning. Order independence is desirable to ensure that results are not biased by earlier scans. Together, they allow straightforward parallelization.
- **Time and space efficiency**. Complex objects may require many range images in order to build a detailed model. So they must be represented efficiently and processed quickly.
- **Robustness**. Outliers and systematic range distortions can create challenging situations for reconstruction algorithms. A robust algorithm needs to handle these situations without failures such as holes in surfaces and self-intersecting surfaces.

- **No restrictions on topological type**. The algorithm should not assume that the object is of a particular genus. Simplifying assumptions such as "the object is homeomorphic to a sphere" yield useful results in only a restricted class of problems.
- **Ability to fill holes in the reconstruction**. Given a set of range images that do not completely cover the object, the surface reconstruction will necessarily be incomplete. For some objects, no amount of scanning would completely cover the object, because some surfaces may be inaccessible to the sensor. In these cases, we desire an algorithm that can automatically fill these holes with plausible surfaces, yielding a model that is both "watertight" and esthetically pleasing.

All algorithms for volume rendering, uses two main strategies:

- Reconstruction from unorganized points
- Reconstruction that exploits the underlying structure of acquired data

In this section we will examine some of the most widespread techniques used for volume rendering to understand its key-features.

4.3.1 Volume ray casting

With this technique, we generate a ray for every image pixel. Using a simple *camera model*, the ray starts at the center of projection of the camera and passes through the pixel on the imaginary image plane floating in between the camera and the volume to be rendered. We can save time by clipping the ray by the boundaries of the volume. The data is interpolated at each sample point, and the process repeated until the ray exits the volume. The RGBA value obtained is converted to an RGB color and saved in the corresponding image pixel. The process is repeated for every pixel on the screen to form the complete image.

So entering in details, we can divide the volume ray casting technique into the following four steps:

1. **Ray casting**. For each pixel of the final image, we cast a ray through the volume
2. **Sampling**. Along the part of the ray of sight that lies within the volume, equidistant sampling points or samples are selected

3. **Shading**. For each sampling point, a gradient of illumination values is computed. The samples are then shaded according to their surface orientation and the location of the light source.

4. **Compositing**. After all sampling points have been shaded, they are composited along the ray of sight, obtaining the final color value.

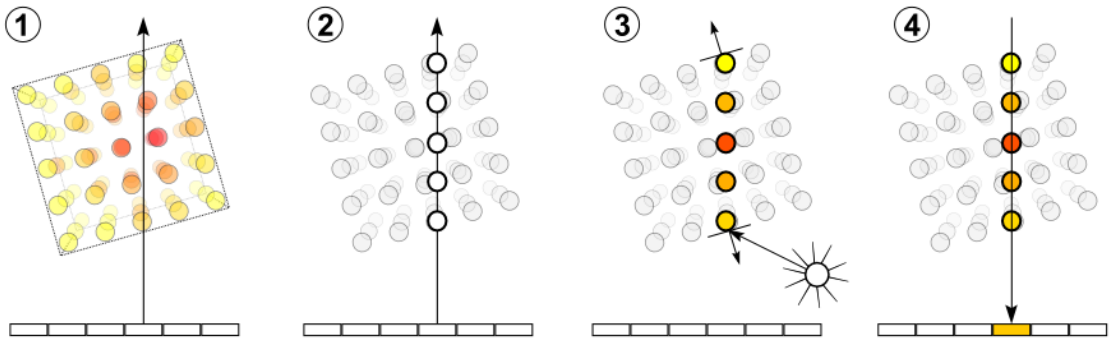


FIGURE 4.9: Volume ray casting phases. (1) Ray casting (2) Sampling (3) Shading (4) Compositing. Image taken from Wikipedia

4.3.2 Maximum intensity projection

A **maximum intensity projection** is a volume rendering method that projects, in the visualization plane, voxels with maximum intensity that fall in the way of parallel rays traced from the viewpoint to the plane of projection. This technique is computationally fast but the two-dimensional results do not provide a good sense of depth of the original data. To improve the sense of 3D, animations are usually rendered using several frames, creating the illusion of rotation

4.4 Marching Cubes and iso-surfaces for data visualization

In the previous section, we have seen how to generate three-dimensional representations from slices of images. These procedures are simple, inherently parallel and produce high quality renderings. However, they are quite slow (we usually have a lot of rays also in regions where they are not needed). So, when working on huge data sets, we have to use other techniques for generation of three-dimensional models. The most famous, and also the one compared with our method in the next parts, is **iso-surface rendering**. It is based on transformation of volumetric data sets

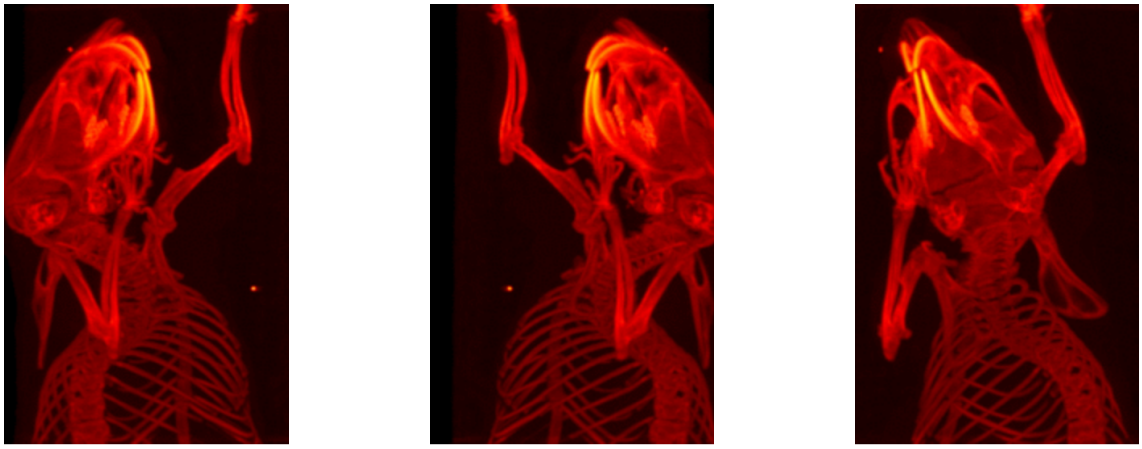


FIGURE 4.10: Maximum intensity projection. From (a) to (c) different views of a MIP of a mouse Images taken from Wikipedia

into a *surface model*. Now, we will see some definitions so we can better understand this method.

First of all we can define an **iso-surface** as a surface that represents points of a constant value within a volume of space. In mathematical words *it is a level set of a continuous function whose domain is three-dimensional space*. In Figure 4.11 there is an example showing an iso-value on a sample image

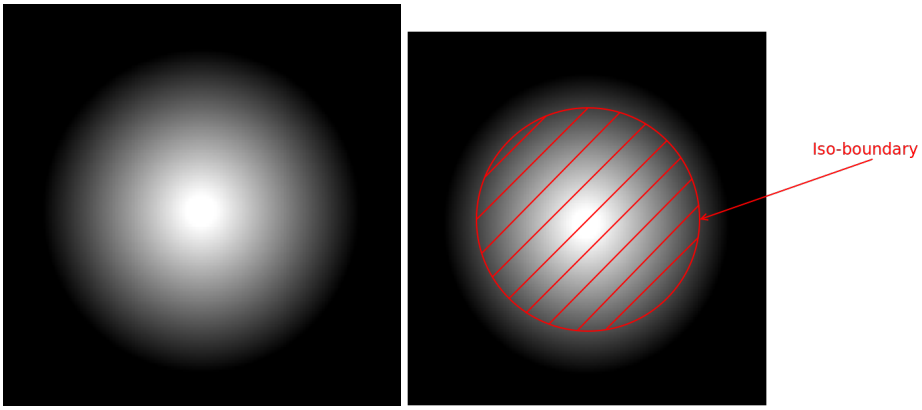


FIGURE 4.11: Iso-boundary on a sample image. The chosen iso-value determines the iso-boundary (in red) and obviously the iso-surface. The rendered part of the sphere will be inside the boundary

For the extraction of a polygonal mesh from an iso-surface, the most famous algorithm is **Marching Cubes**, which is fast and simple to implement and has been introduced by [20]. A description of the algorithm is given in Listing 4.1

ALGORITHM 4.1: Marching Cube algorithm

```
1  begin
2   foreach point p of the scalar field:
3     Take eight neighbor locations of p (thus forming an imaginary cube)
4   foreach cube c:
5     if all 8 points of c are above or below the iso-surface:
6       ignore points
7     else:
8       assume surface is linear inside c and emit triangles representing it
9   return mesh composed by all emitted triangles
10 end
```

The heart of the algorithm is in the computation of triangles into the cubes. All possible polygon configurations inside the cubes are $2^8 = 256$, since the single vertex can either be positive or negative (above or under the surface). However, many of these are equivalent so in [20] are *fifteen unique cases*, which are listed in Figure 4.12

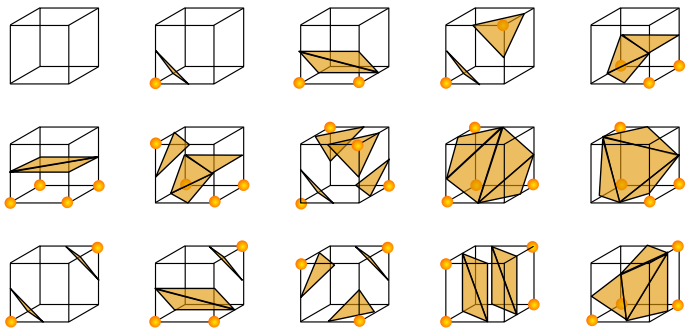


FIGURE 4.12: All possible configurations in Marching Cubes as published by [20]. The simplest pattern (the upper left one) occurs if all vertex values are above (or below) the selected value and produces no triangles. The next one occurs if the surface separates one vertex from the other seven and so on. Image taken from Wikipedia

So, we can generate triangles simply observing which one of the cases we have for the current cube according to the intersections of vertices with the cubes edges. However we can have problems for cases with "rippling" signs where we can have at least two correct choices for where the correct contour should pass. This problem was solved using an extended table without ambiguities that uses 33 configurations.

As we can see, a correct choice for the grid size has a great influence on the final result as in

Figure 4.13. In fact, choosing a finer grid we have more cubes and the interpolation of surface produces a better result.

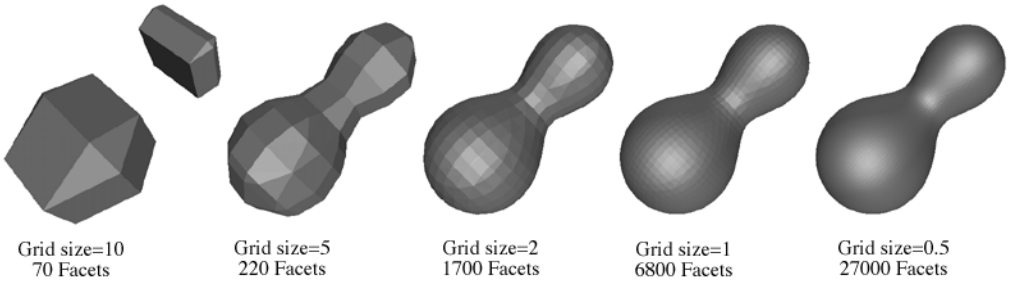


FIGURE 4.13: Marching Cubes grid sizes. How we can see in this example the final result of the Marching Cubes vary according with the grid size. Obviously choosing a finer grid we obtain a better result thus resulting in a slower computation. Image taken from [5]

Marching Cubes is useful for its simplicity, however despite this feature, this algorithm produces low quality meshes as we can observe in Figure 4.14

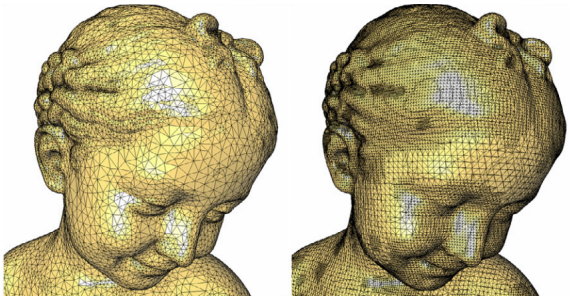


FIGURE 4.14: Marching Cubes quality. On the left there is a model taken with an advanced reconstruction technique. On the right we have a lesser quality model taken with the Marching Cubes algorithm. Image taken from [2]

Another problem arises when we do not have a filled space of voxels, because any interpolated value may reduce the validity of the resulting surface. Moreover, the resulting mesh is not connected so we need to weld adjacent voxels together to get a closed mesh. Finally, we should remember that the model obtained with this method is just an approximation of the initial scalar field, in next parts we will introduce a method which will return perfect models from our data.

Part II

Methodologies and tools

Chapter 5

A topological approach for data visualization: LAR

Now we will study several different methodologies for the creation and managing of enormous geometric models. At first, we will see some methodological instruments which are at the basis of a powerful library for geometric modeling developed at the *Computational and Visual Design* laboratory in the Roma Tre University: **LAR**. It is essentially based in algebraic topology techniques, which will be introduced here.

As a consequence, the chapter will first do a general overview of the topology field, then it will study in detail the LAR representation schema and some interesting topological operators

5.1 Fundamentals of topology

In this section we will give a general overview of **topology** field. In mathematics, topology is concerned with the properties of space that are preserved under **continuous deformations**, such as *stretching* and *bending*, but **not** *tearing* and *gluing*. Topology comes from geometry and set theory and the first theorems of the field comes from the Euler's *Seven Bridges of Königsberg Problem* and from the *Polyhedron Formula* (which we have already seen in Chapter 4.2.5) . It has many subfields:

- **General topology**: covers the foundational aspects of topology and investigates properties of topological spaces and concepts inherent to topological spaces

- **Algebraic topology**: tries to measure degrees of connectivity using algebraic constructs such as homology and homotopy groups
- **Differential topology**: deals with differentiable functions on differentiable manifolds. It is closely related to differential geometry
- **Geometric topology**: studies manifolds and their embeddings in other manifolds. A particularly active area is low-dimensional topology which studies manifolds of four or fewer dimensions

Now we can study in detail this field giving some definitions. First of all from [19] we can get the definition of a **topological space** as a set satisfying three properties

Definition 5.1 (Topological space)**.** *Let X be a set and let $\mathcal{U}$ be a collection of subsets of X satisfying:*

1. $\emptyset \in \mathcal{U}$, $X \in \mathcal{U}$
2. *the intersection of two members of $\mathcal{U}$ is in $\mathcal{U}$*
3. *the union of any number of members of $\mathcal{U}$ is in $\mathcal{U}$*

*Such a collection $\mathcal{U}$ of subsets of X is called a **topology** for X. The set $\mathcal{U}$ together with X is called a **topological space** and is denoted by $(X,\mathcal{U})$ which is often abbreviated to T or just X. The members $U \in \mathcal{U}$ are called the **open sets**[1] of T. Elements of X are called **points** of T*

A typical example of topological space is the **metric space**:

Definition 5.2 (Metric space)**.** *A **metric space** is an ordered pair (M, d) where M is a set and d is a metric on M, i.e., a function $d\colon M \times M \to \mathbb{R}$ such that for any $x, y, z \in M$ the following holds:*

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

The most famous example of a metric space is the **Euclidean space** (that it consequently is a topological space). So every object contained in a plane or in a space (such as polygons and

[1] Intuitively, we say that a set is *open* if we can move slightly in every direction from every point of the set without leaving it. The complementary of an open set is a **closed set**

fractals) is a topological space

As we have seen above, topology studies spaces but also deformations that are applied on them. So we can introduce the notion of **continuous functions**:

Definition 5.3 (Continuous function)**.** *A function $f\colon X \to Y$ between two topological spaces is said to be **continuous** if for every open set U of Y the inverse image $f^{-1}(U)$ is open in X*

For example the identity function and the constant function are continuous functions. When two topological spaces are equivalent we say there is an **homeomorphism**:

Definition 5.4 (Homeomorphism)**.** *Let X and Y be topological spaces. We say that X and Y are **homeomorphic** if there exist inverse continuous functions $f\colon X \to Y$, $g\colon Y \to X$. We write $X \cong Y$ and say that f and g are **homeomorphisms** between X and Y*

An equivalent definition is:

Definition 5.5 (Homeomorphism(2))**.** *A function $f\colon X \to Y$ between two topological spaces (X, T_x) and (Y, T_y) is called a **homeomorphism** if it has the following properties:*

1. *f is a bijection*
2. *f is continuous*
3. *the inverse function f^{-1} is continuous (f is an open mapping)*

In other words, according to this definition, we have a bijective function between X and Y which links the open sets of X with the open sets of Y. For example we can say that a square and a circle are *homeomorphic* and so a mug and a torus, because we can define a function which map one space to the other. Thus, using homeomorphism we can define new topological spaces from known ones.

Now we can introduce another important definition for the construction of a topological space from a given one:

Definition 5.6 (Quotient space)**.** *Suppose that $f\colon X \to Y$ is a surjective mapping from a topological space X onto a set Y. The **quotient topology** on Y with respect to f is the family $\mathcal{U}_f = \{U; f^{-1}(U)$ is open in $X\}$*

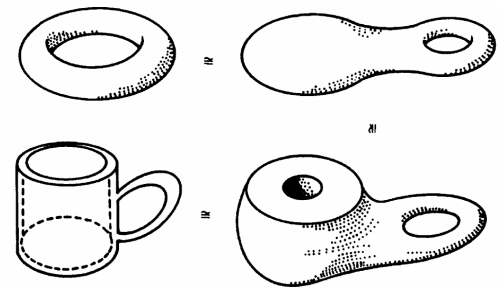


FIGURE 5.1: Homeomorphisms between a mug and a torus. Image taken from [19]

[19] have a good example of a quotient space. We can consider the space: $C = \{(x, y, z) \in \mathbb{R}^3; x^2 + y^2 = 1, |z| \leq 1\}$

which represents a cylinder. Let M be the set of unordered pairs of points in C of the form $\{p, -p\}$, i.e: $M = \{\{p, -p\}; p \in C\}$

Since we have a natural surjective map from C to M we can give M the quotient topology; the result is called a **Mobius strip**.

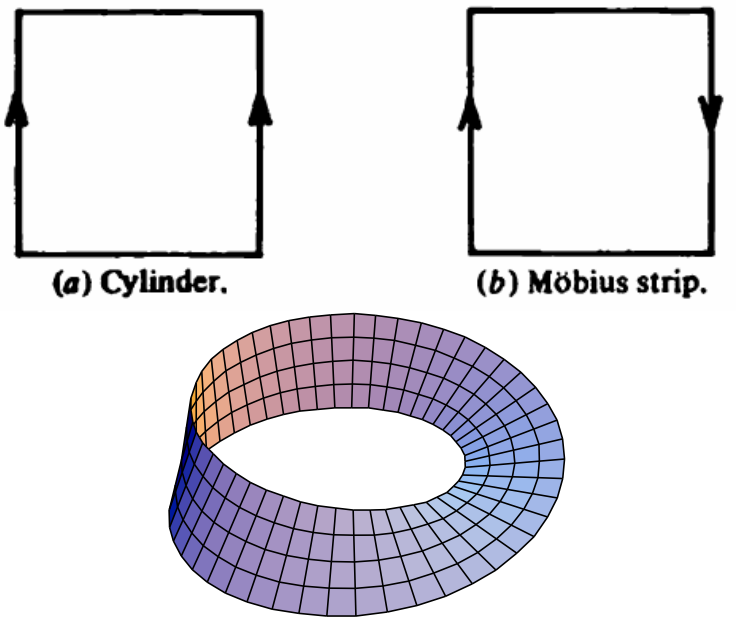


FIGURE 5.2: (*First row*) Flat view of a Mobius strip and a cylinder. (*Second row*) three-dimensional view of a Mobius strip. Images taken from [19] and Wikipedia

At this point, we have seen the main definitions for topology and some ways to generate topological spaces. In next subsections we will study some interesting topological spaces which will be used in later parts

5.1.1 Chain complexes

Now we can introduce a very important construct of the algebraic topology the **chain complexes**. They are algebraic means of representing the relationships between *cycles* and *boundaries* in various dimensions of a topological space. To fully understand the theory behind chain complexes we should introduce some other interesting definitions:

Definition 5.7 (Standard n-simplex)**.** *The **standard n-simplex** Δ_n is defined to be the following subspace of $\mathbb{R}^{n+1}$:*

$\Delta_n = \{x = (x_0, x_1, \ldots, x_n) \in \mathbb{R}^{n+1}; \sum_{i=0}^{n} x_i = 1, x_i \geq 0, i = 0, 1, \ldots, n\}$

The points $v_0 = (1, 0, \ldots, 0), v_1 = (0, 1, 0, \ldots, 0), \ldots, v_n = (0, 0, \ldots, 0, 1)$ are called the vertices of Δ_n

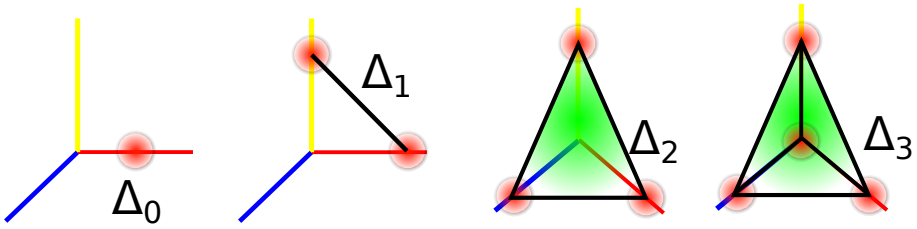


FIGURE 5.3: Examples of standard simplices. Δ_0 is a point, Δ_1 is a line, Δ_2 is a triangle and Δ_3 is a tetrahedron

Definition 5.8 (Singular n-simplex)**.** *Let X be a topological space. A **singular n-simplex** in X is a continuous map $\varphi\colon \Delta_n \to X$*

So a singular 0-simplex is a point in X, while a singular 1-simplex is a path in X.

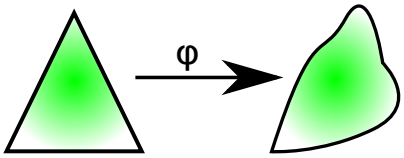


FIGURE 5.4: A singular n-simplex.

Now we can define the **singular n-chain**:

Definition 5.9 (Singular n-chain)**.** *A singular n-chain in X is an expression of the form $\sum_{j \in J} n_j \varphi_j$ where $\{\varphi_j; j \in J\}$ is the collection of all singular n-simplexes in X (with J some indexing set) and $n_j \in \mathbb{Z}$ with only a finite number of $\{n_j; j \in J\}$ being non-zero*

In other words we can say that a simplicial k-chain is a linear combination of k-simplices. The set $S_n(X)$ of singular n-chains in X form an abelian group[2] with the addition defined by

[2]We say that an abelian group is a group where its binary operation is commutative

$\sum n_j\varphi_j + \sum m_j\varphi_j = \sum(n_j + m_j)\varphi_j$. This group has extremely interesting properties which we will exploit in following parts and that makes it a fundamental construct for LAR library.

Now that we have a definition of a chain complex, we can also define the **cochain complex**:

Definition 5.10 (Singular n-cochain)**.** *Given a topological space X and an abelian group G, we define the group $C^n(X;G)$ of* ***singular n-cochains with coefficients in G*** *to be the dual group of the singular chain group $C_n(X)$. Thus an n-cochain $\varphi \in C^n(X;G)$ assigns to each singular n-simplex $\sigma\colon \Delta_n \to X$ a value $\varphi(\sigma) \in G$. Since the singular n-simplices form a basis for $C_n(X)$, these values can be chosen arbitrarily hence n-cochains are exactly equivalent to functions from singular n-simplices to G.*

This definition is taken from [15]. From a computer science point of view, cochains offer a mechanism for representation of quantities associated with combinatorial and discrete representations. As a consequence, we can use them to represent physical properties for our domain.

5.1.2 Cell complexes

Another fundamental construct of algebraic topology, are **cell complexes**, which provides a useful procedure for the construction of a topological space. As we can see in [15] we can use the following procedure:

1. Start with a discrete set X^0, whose points are regarded as **0-cells**

2. Inductively, form the **n-skeleton** X^n from X^{n-1} by attaching n-cells e^n_α via maps $\varphi_\alpha\colon S^{n-1} \to X^{n-1}$. This means that X^n is the quotient space of the disjoint union $X^{n-1} \coprod_\alpha D^n_\alpha$ of X^{n-1} with a collection of n-disks D^n_α under the identifications $x \sim \varphi_\alpha(x)$ for $x \in \partial D^n_\alpha$[3]. Thus as a set, $X^n = X^{n-1} \coprod_\alpha e^n_\alpha$ is an open-disk.

3. One can either stop this inductive process at a finite stage, setting $X = X^n$ for some $x < \infty$, or one can continue indefinitely, setting $X = \bigcup_n X^n$. In the latter case X is given the **weak topology**: A set $A \subset X$ is open (or closed) iff $A \cap X^n$ is open (or closed) in X^n for each n

A space X constructed in this way is called a **cell complex** or **CW complex**. If $X = X^n$ for some n, then X is said to be finite-dimensional, and the smallest such n is the **dimension** of X, the maximum dimension of cells of X. We could prove that a n-dimensional cell is homeomorphic to a *n-dimensional ball*

[3] ∂D^n_α is the boundary of the n-disk as we will see in section 5.3

In addition, [15] gives us interesting examples of cell complexes. For example a 1-dimensional cell complex $X = X^1$ is a **graph** in algebraic topology. It consists of vertices (the 0-cells) to which edges (the 1-cells) are attached. The two ends of an edge can be attached to the same vertex. Another example consists in considering the sphere S^n which is a cell complex with just two cells. These cells are e^0 and e^n, where the n-cell is attached by the constant map $S^{n-1} \to e^0$. This is equivalent to regarding S^n as the quotient space $D^n/\partial D^n$.

We can also give the following definition:

Definition 5.11 (Subcomplex)**.** *A **subcomplex** of a cell complex X is a closed subspace $A \subset X$ that is a union of cells of X. Since A is closed, the characteristic map of each cell in A has image contained in A, and in particular the image of the attaching map of each cell in A is contained in A, so A is a cell complex in its own right. A pair (X, A) consisting of a cell complex X and a subcomplex A will be called a **CW pair**.*

According to [15], we can also define various operations on cell complexes. We have:

Products. If X and Y are cell complexes, then $X \times Y$ has the structure of a cell complex where whose cells are the products $e^m_\alpha \times e^n_\beta$ where e^m_α ranges over the cells of X and e^n_β over the cells of Y

Quotients. If (X, A) is a CW pair consisting of a cell complex X and a subcomplex A, then the quotient space X/A inherits a cell complex structure from X. The cells of X/A are the cells of $X - A$ plus one new 0-cell, which is the image of A in X/A. For a cell e^n_α of $X - A$ attached by $\varphi_\alpha \colon S^{n-1} \to X^{n-1}$, the attaching map for the corresponding cell in X/A is the composition $S^{n-1} \to X^{n-1} \to X^{n-1}/A^{n-1}$

Suspension. For a space X, the suspension SX is the quotient of $X \times I$ obtained by collapsing $X \times \{0\}$ to one point and $X \times \{1\}$ to another point.

Join. Given X and a second space Y, one can define the space of all line segments joining points in X to points in Y. This is the join $X * Y$, the quotient space of $X \times Y \times I$ under the identifications $(x, y_1, 0) \sim (x, y_2, 0)$ and $(x_1, y, 1) \sim (x_2, y, 1)$. Thus we are collapsing the subspace $X \times Y \times \{0\}$ to X and $X \times Y \times \{1\}$ to Y.

Wedge Sum. Given spaces X and Y with chosen points $x_0 \in X$ and $y_0 \in Y$, then the wedge sum $X \vee Y$ is the quotient of the disjoint union $X \sqcup Y$ obtained by identifying x_0 and y_0 to a single point.

Smash Product. Inside a product space $X \times Y$ there are copies of X and Y, namely $X \times \{y_0\}$ and $\{x_0\} \times Y$ for points $x_0 \in X$ and $y_0 \in Y$. These two copies of X and Y in $X \times Y$ intersect only at the point (x_0, y_0), so their union can be identified with the wedge sum $X \vee Y$. The smash product $X \wedge Y$ is then defined to be the quotient $X \times Y / X \vee Y$

In Figure 5.5 there is a graphical representation for all these operations.

5.2 LAR representation schema

As stated in [11] (and also in the first chapters of this work), present-day computational problems in science and technology must deal with increasingly complex geometric information. The evolution of 3D geometric representations can be generally understood in terms of graph-based data structures representing one of several possible cell complexes partitioning either the boundary or the interior of the represented model. However, we can make a lot of assumptions about cell complexes and graph representations, so standardization seem difficult to reach. Thus the specialized data structures that have been used since now, are no longer adequate for dealing with this problems.

[11] provide a solution to these problems with a new representation schema which supports all topological constructions and queries useful in typical cellular decomposition of space: **LAR**. Formally, LAR uses the standard definitions of the mod-2 cell complexes. So we have *n-chains* which are sets of *n-cells*. The standard basis of the $\mathbb{Z}_2$-linear space C_d of n-chains is provided by singletons of n-cells; each n-cell is represented by a map $C_n \rightarrow \mathbb{Z}_2 C_0$, i.e. by a row of a binary characteristic matrix M_n. Obviously, each n-chain in C_n can be generated by a $(\mathbb{Z}_2)$-linear combination of M_n rows. This formulation can be extended to n-cochains that represent any possible field over the chains.

How we can see in Figure 5.6, characteristic matrices are very sparse for actual chain complexes, so they can be represented in the **Compressed Sparse Row** (CSR) format in order to save space. This format is widely used in computer science, and consists in using three one-dimensional arrays. The first one save the non-zero values as they are traversed in a row-wise fashion, while the second stores the columns indices of the values. The last one stores the locations of the values array that start a row. Moreover, if we have a binary matrix we can safely delete the first array, as the non-zero value can only be 1. In addition this data structure is faster

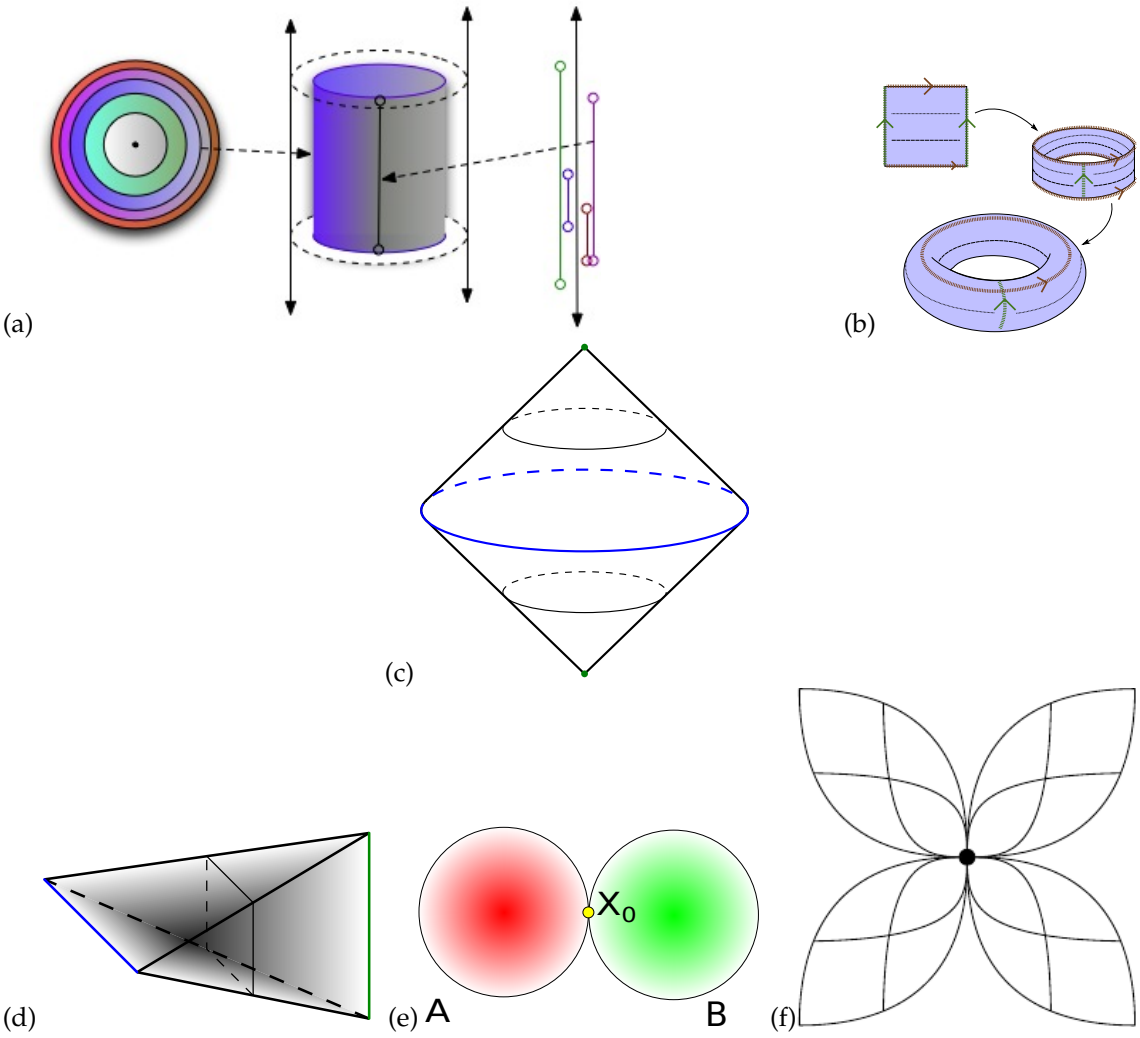


FIGURE 5.5: Operations on cell complexes.
(a) A product topology. How we can see, a cylinder can be obtained with a product between a circle and a line. Image taken from scienceblogs.com
(b) A quotient topology. Intuitively we have a quotient topology when we glue spaces points together. For example we can obtain a torus by gluing opposite sides of a square. Image taken from mathroughguides.wikidot.com
(c) A suspension topology. The original space is in blue and the collapsed end points are in green. Image taken from Wikipedia.
(d) Geometric join of two line segments. The original spaces are in blue and green. The join is a three-dimensional solid in black. Image taken from Wikipedia
(e) A wedge sum topology obtained by attaching two circumferences by a point
(f) Smash product. In this example we build a smash product starting from two segments. The product topology produces a square and then the vertical and the horizontal lines are collapsed to a point. Image taken from www.math.ntnu.no

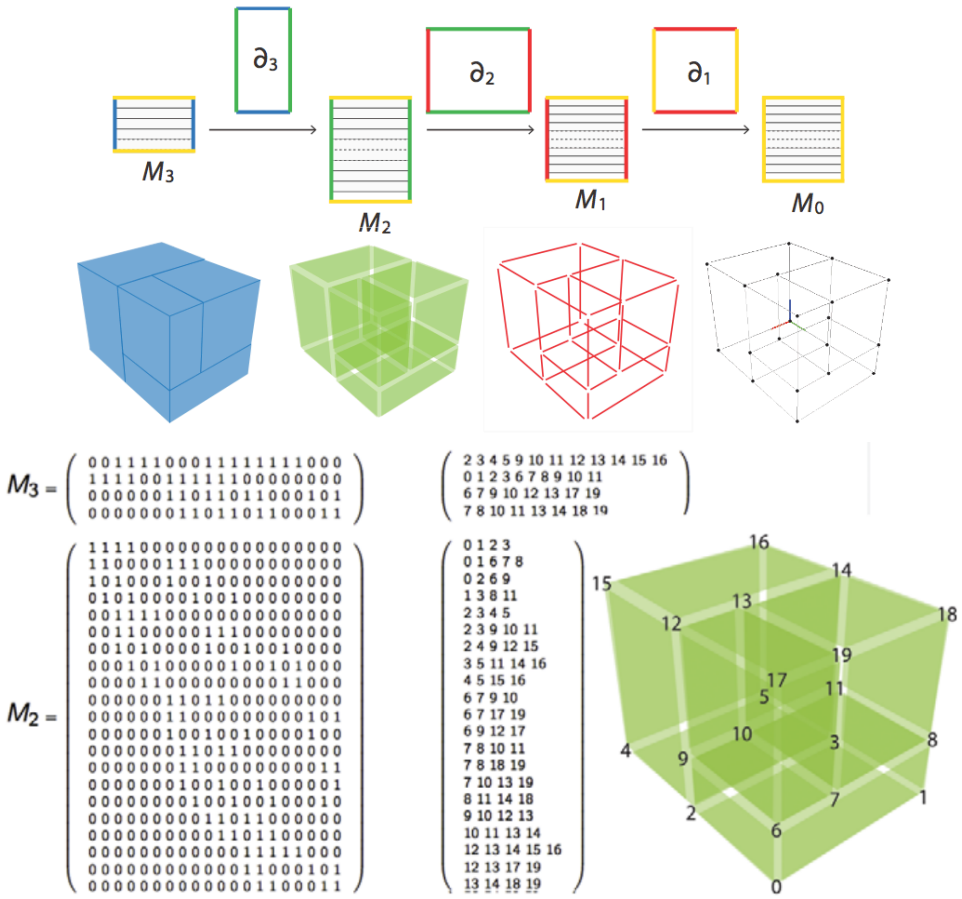


FIGURE 5.6: LAR representation schema. We can see the relationship between chains of different dimensions and how to represent them using a characteristic matrix. Images taken from [11]

than a traditional one, because the **sparse matrix-vector multiplication** (SpMV) is linear in the size of the output.

Now we can study in deep the LAR representation from a formal point of view using concepts from [11]. First of all we need to consider a cellular complex $\Lambda(X)$ such that $X = \Lambda_0 \cup \cdots \cup \Lambda_d$. We can define $M_p \in \mathbb{Z}_2^{m \times n} (0 \leq p \leq d)$ as the characteristic matrices, with number of rows equal to the number of p-cells and number of columns equal to the number of 0-cells:

$$m = k_p = \#\Lambda_p, \qquad n = k_0 = \#\Lambda_0 \tag{5.1}$$

Each $m_{ij} \in M_p$ tells us whether the vertex $v_j \in \Lambda_0$ is contained on the boundary of the cell $\lambda_i \in \Lambda_p$ or not. So the LAR schema is a map from a mathematical model of a solid to its computer

representation:

$$LAR\colon \mathbb{C}h(\Lambda(X)) \to (M_p)_{p=1}^{d} \tag{5.2}$$

So we can have the following definition from [11]:

Definition 5.12 (LAR representation)**.** *LAR represents chain complexes $\mathbb{C}h$, supported by a finite cellular complex $\Lambda(X)$, by $d-$tuples of binary CSR matrices, where d is the dimension of the space X*

For a cellular $d-$complex $\Lambda(X)$, all the M_p matrices have the same number of columns, so we can use standard matrix transposition and multiplication. Thus we can easily compute *boundary* and *coboundary* operators (see Section 5.3) and topological relations between cells.

Summing up, we have seen that this new representation schema, make us able to define cell complexes in a very simple way.

In addition, we can consider a common situation where a topological structure is a regular $d-$complex (so every cell is contained in some $d-$cell). In this case, [11] define a **reduced LAR representation** under the following assumptions:

- LAR contains both the $d-$cells of a regular decomposition of a $d-$space and of its complement
- Any two cells intersect on a connected cell

In this case the highest dimensional $CSR(M_d)$ matrix is a valid reduced LAR representation in the sense that all lower-dimensional M_p matrices and operators may be computed from the matrix M_d. Therefore we can say that $CSR(M_d)$ **fully characterizes the chain complex**.

Another representation (which is defined in [9]), is the **BRC representation**. It based on arrays of integers, with no requirement of equal length for the component arrays. Each component array, corresponds to a matrix roe and contains the indices of columns that store a 1 value. In [9] we can

find the following example:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \mapsto \quad \texttt{BRC}(A) = \begin{array}{l} \texttt{[[1,7],} \\ \texttt{[2],} \\ \texttt{[0,3,9],} \\ \texttt{[0,6],} \\ \texttt{[5,6,7],} \\ \texttt{[2,4,8],} \\ \texttt{[],} \\ \texttt{[1,7,9],} \\ \texttt{[3,8],} \\ \texttt{[1,2,4]]} \end{array}$$

In next section we will see how to define topological operators in order to manipulate them.

5.3 Some interesting topological operators

Now we can define some interesting operators for manipulation of our topological structures. First of all we are interested in incidence queries that arise in a cellular decomposition $\Lambda(X)$ of a two-dimensional space. We can have nine relations:

TABLE 5.1: Incidence queries

$VV: C_0 \to C_0$	$EV: C_0 \to C_1$	$FV: C_0 \to C_2$
$VE: C_1 \to C_0$	$EE: C_1 \to C_1$	$FE: C_1 \to C_2$
$VF: C_2 \to C_0$	$EF: C_2 \to C_1$	$FF: C_2 \to C_2$

As we can see in [11], with LAR we can compute these relations with only SpMV multiplications:

$$VV = VE \circ EV = EV^{\mathsf{T}} \circ EV \Rightarrow [VV] = M_1^{\mathsf{T}} M_1$$

$$VE = EV^{\mathsf{T}} \Rightarrow [VE] = M_1^{\mathsf{T}}$$

$$VF = FV^{\mathsf{T}} \Rightarrow [VF] = M_2^{\mathsf{T}}$$

$$EV[VV] = M_1$$

$$EE = EV \circ VE = EV \circ EV^{\mathsf{T}} \Rightarrow [EE] = M_1 M_1^{\mathsf{T}}$$

$$EF = EV \circ VF = EV \circ FV^{\mathsf{T}} \Rightarrow [EF] = M_1 M_2^{\mathsf{T}}$$

$$FV[FV] = M_2$$

$$FE = FV \circ VE = FV \circ EV^{\mathsf{T}} \Rightarrow [FE] = M_2 M_1^{\mathsf{T}}$$

$$FF = FV \circ VF = FV \circ FV^{\mathsf{T}} \Rightarrow [FF] = M_2 M_2^{\mathsf{T}}$$

Now we will study one of the most important operators in topological algebra: the **boundary operator**. Intuitively, a boundary of a subset S of a topological space X is the set of points which can be approached both from S and from the outside of S. So it is a set of points that belong to S but do not belong to its interior. Thus using the boundary operator we are able to pass from a $d-$dimensional chain to a $(d-1)-$dimensional one. [19] gives us a good definition of the boundary operator:

Definition 5.13 (Boundary operator)**.** *The **boundary operator** $\partial\colon S_d(X) \to S_{d-1}(X)$ is defined by:*

$$\partial = \partial_0 - \partial_1 + \partial_2 - \cdots + (-1)^d \partial_d = \sum_{i=0}^{d} (-1)^i \partial_i$$

This leads to the definition of two interesting subgroups of $S_d(X)$:

Definition 5.14 (Cycles)**.** *A singular $d-$chain $c \in S_d(X)$ is a $\mathbf{d}-$**cycle** if $\partial_c = 0$. The set of $d-$cycles in X is denoted by $Z_d(X)$*

Definition 5.15 (d-boundaries)**.** *A singular $d-$chain $b \in S_d(X)$ is a $\mathbf{d}-$**boundary** if $b = \partial_e$ for some $e \in S_{d+1}(X)$. The set of $d-$boundaries in X is denoted by $B_n(X)$*

We can also define the **coboundary operator** (δ^d) on cochains as the dual of the boundary operator. Using it we can pass from a $(d-1)-$dimensional complex to a $d-$dimensional one. In Figure 5.7 there is a representation of the relationships between chains and cochains of different dimensions:

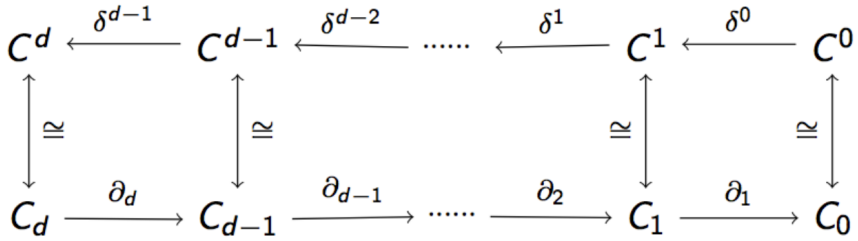


FIGURE 5.7: Relationships between chains and cochains

At this point we are interested in computing these operators in an efficient way. As always, [11] provides an interesting algorithm based on the LAR representation schema. First of all, we

should consider the incidence map $\ell^p_{p-1}\colon C_p \to C_{p-1}$ and its matrix $[\ell^p_{p-1}] = M^p_{p-1} = M_{p-1}M_p^{\mathsf{T}}$. The entry $M^p_{p-1}[i,j]$, stores the number k of common vertices between the cells μ^i_{p-1} and λ^j_p, where $\mu \in \Lambda_{d-1}$ is the common facet between cell complexes and $\lambda \in \Lambda_d$ represents a single cell. In fact we have: $M^p_{p-1}[i,j] = \sum_{h=0}^{k_0-1} M_{p-1}[i,h] \cdot M_p[j,h] = \#(\mu^i_{p-1} \cap \lambda^j_p) = k$. So if we want to compute the **unoriented boundary** operator, we can use the following algorithm:

ALGORITHM 5.1: Unoriented boundary algorithm

```
1  begin
2   CSR(M^p_{p-1}) = CSR(M_{p-1})CSR(M_p^T)
3   foreach i in 0 ≤ i ≤ k_{p-1} − 1:
4     k = #μ^i_{p-1}  //Number of nonzero elements for row i of CSR(M_{p-1})
5     foreach j in 0 ≤ j ≤ k_p − 1:
6       if M^p_{p-1}[i,j] = k:
7         ∂_p[i,j] = 1
8       else:
9         ∂_p[i,j] = 0
10  return ∂_p
11 end
```

By duality, we can obtain the coboundary operator as the transposition of ∂_p operator. For a better understanding of this algorithm we can see an example of ∂_2 computation from [11]. First of all we have to consider the cell complex in Figure 5.8 with the characteristics matrices M_1 and M_2. According to the algorithm we have that $M^2_1 = M_1M_2^{\mathsf{T}}$. So if:

$$M_2 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} M_1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

we can obtain:

$$M_1^2 = \begin{pmatrix} 2 & 1 & 0 & 1 & 0 & 2 \\ 1 & 2 & 0 & 0 & 1 & 2 \\ 2 & 2 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 2 & 1 & 2 \\ 2 & 0 & 1 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 & 2 & 2 \\ 0 & 2 & 1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 1 & 2 & 2 & 0 & 1 & 0 \\ 1 & 0 & 2 & 2 & 1 & 0 \\ 2 & 1 & 2 & 1 & 0 & 0 \end{pmatrix} \Rightarrow \partial_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

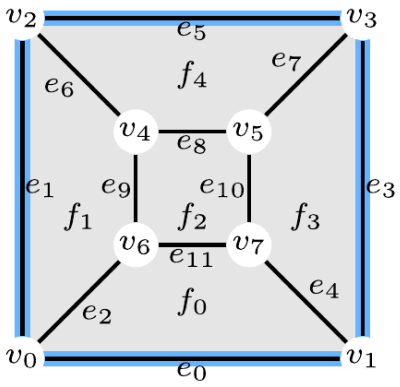


FIGURE 5.8: Cuboidal complex. Image taken from [11]

Chapter 6

Mastering big data using parallel computing

Usually we design our softwares thinking that they will run on a single *executor*. However the increase of data sizes, and the consequent needs for architectures that can do efficient computations, determined the birth of a new programming model: **parallel computing**. The main idea at the basis of this model is that if a single computer is not able to compute operations on a huge quantity of data, so we can connect more and more computers in *parallel* to get our computation done. Indeed, by now we have parallelism at several levels of our architectures. For example modern CPUs are designed with the aim of executing parallel code and also GPUs can be used for this purpose. However, in this work we are mainly focused on **clusters** of supercomputers, as we will see in detail in Section 6.2. In this chapter we will also study one of the most important tools for parallel computing: MPI

6.1 Principles of parallel computing

As we have seen in previous chapters, big data are deeply changing computer science and all its branches. So to deal with this new challenge we need to rethink our programming models and, how we have read in the introduction to this chapter, **parallel computing** can be an answer to these requests. According to [12], parallel computing is simply an answer to the following problem: if there are n operations to be done ans they would take time t on a single processor, can they be done in time t/p on p processors?.

We use a parallel computer for two important reasons: to *have access to more memory* and to *obtain higher performance*. While it is simple to understand the gain in memory, as the total memory is the sum of the individual memories, the speed is harder to characterize. First of all we can give the following definition:

Definition 6.1 (Speedup)**.** *Given T_1 as the best time to solve a problem in a single processor, and T_p as the best solving time for the same problem with p processors, we define the **speedup** S_p as: $S_p = \frac{T_1}{T_p}$.*

As we can see in the definition, we usually use different algorithms on different architectures in order to obtain the most efficient code possible. In the ideal case, $T_p = \frac{T_1}{p}$ (**perfect speedup**) but in practice is very difficult to obtain it, so $S_p \leq p$. To measure how far we are from the ideal speedup we can introduce the **efficiency** as $E_p = \frac{S_p}{p}$. As we can see $0 < E_p \leq 1$.

One of the reasons why we do not have perfect speedup, is that parts of a code can be inherently sequential. Supposing that 5% of a code is sequential, then the time for that part cannot be reduced. Thus the speedup on that code is limited to a factor of 20. This phenomenon is described by the **Amdahl's Law**. According to [12] we can use the following definition:

Definition 6.2 (Amdahl's law)**.** *Let F_s be the sequential fraction and F_p be the parallel fraction of a code. Then $F_p + F_s = 1$. The parallel execution time T_p on p processors is the sum of the part that is sequential ($T_1 F_s$) and the part that can be parallelized ($T_1 \frac{F_p}{p}$):*

$$T_p = T_1(F_s + \frac{F_p}{p}) \tag{6.1}$$

When we have $p \to \infty$, the parallel execution time tends to $T_1 F_s$, so the upper bound for the speedup is $S_p \leq \frac{1}{F_s}$.

Amdahl's law is optimistic. In fact code parallelization give a speedup, but it also introduces communication overhead between processes. So if T_c is the time spent on communication, we have:

$$T_p = T_1(F_s + \frac{F_p}{p}) + T_c \tag{6.2}$$

Assuming we have a fully parallelized code, we can write:

$$S_p = \frac{T_1}{T_1/p + T_c} \tag{6.3}$$

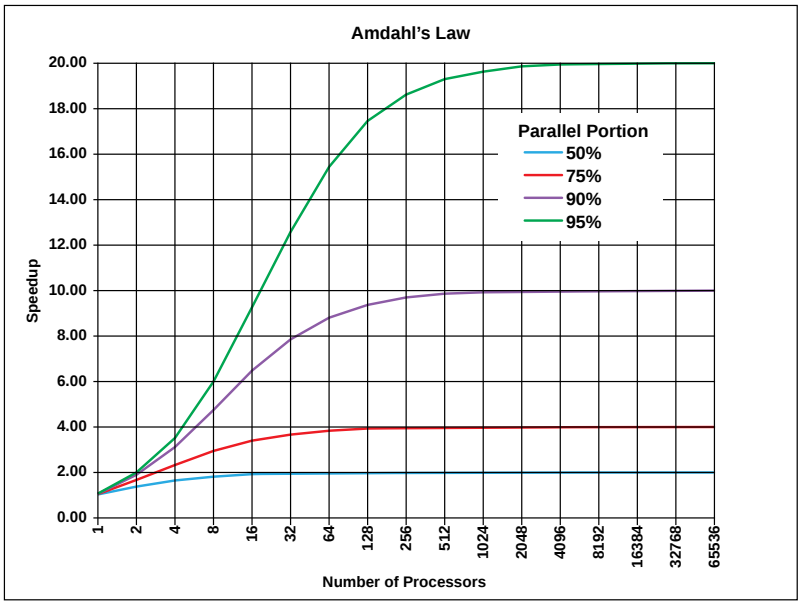


FIGURE 6.1: Amdahl's Law. Image taken from Wikipedia

Thus, to be close to p we need $T_c \ll T_1/p$ or $p \ll T_1/T_c$.

As a consequence of the Amdahl's Law, our speedup is limited, so we do not have a great improvement with large-scale supercomputers. So now we are wondering why we should use parallel computing. In fact, when dealing with big data, we are not only interested in speedup but also in *memory gain*. So, using this technique we are sure that our algorithms can finish returning a result. In this work, a primary importance is given to the improvement of memory occupation, as the three-dimensional models that we will create can be huge.

6.2 Typical architectures for parallel computing

Now we are interested in studying the most common architectures for parallel computing. A good source for this argument is [21]. How we can see there, these architectures are commonly divided into two groups: **shared-memory systems** and **message-passing systems**

6.2.1 Shared-memory systems

With this architecture, many CPUs share the same physical memory. So we talk about **MIMD**, which stands for *Multiple Instruction Multiple Data*. In fact we have many different independent CPUs which access different memory locations at any given time. In these days, shared-memory

systems are very common even on consumer PCs and mobile phones.

As an example we can consider a program which has a global variable X and a local variable Y on this hardware. If the compiler assigns location 200 to X, then all processors will have that variable in common. In fact, any processor which issues a memory operation on location 200 will access the *same physical memory cell*. On the other hand, each processor will have its own run-time stack. All stacks are in shared memory, but they are accessed separately as each CPU has a different value in its stack pointer. Thus each processor will have its own independent copy of the local variable Y.

From a parallel point of view, each execution of a program requires parallel accessing of memory in order to avoid slowdowns. In parts this is handled by having a cache at each CPU, but it is also facilitated by dividing the memory into separate modules or *banks*. The division can be done with two main strategies:

- **High-order interleaving**: consecutive words are in the same bank (except at boundaries). It is widely used when dealing with matrices.
- **Low-order interleaving**: consecutive addresses are in consecutive banks (except when we get to the right end). It is widely used in GPUs

For a better understanding of shared-memory systems we can do the following example. Suppose we have to implement a matrix-vector multiplication between a vector X and a matrix A producing vector Y. The arrays for A, X and Y are held in common by all nodes. Then, each node multiply its assigned rows of A times X and place the result directly in the proper section of Y. From a technical point of view, these operations are usually done with *threads*.

6.2.2 Message-passing systems

In message-passing systems, we have a certain number of independent CPUs each with its own independent memory. All processors communicate using some kind of network protocol. A typical environment for these systems is a **cluster** of different PCs. The idea consists in taking several PCs and networks and divide the computation sending **messages** to them containing their part of input data. Obviously, we need a fast network to avoid slowdowns due to network communication. The common choice today is *Infiniband* as the traditional TCP/IP networks are quite

slow. The common paradigm used in message-passing systems is the **scatter/gather paradigm**. According to this paradigm, we have a *manager* node which sends out chunks of work to the other nodes, which are called *workers*. When they have finished their work, workers send back to the manager the results which will be collected and assembled.

Now we can consider the same example as above of matrix-vector multiplication. With this architecture one node (for example node 0) distributes the rows of A to the other ones, so that each node receive a different set of rows, and the vector X to all nodes. Each node then multiply X by its own assigned rows of A and then send back the results to node 0 which will collect all parts and store in Y the final result

As this type of architecture is the one chosen for our software, we will describe it more deeply when we will see the most important technology for implementation of a message-passing system: MPI.

6.3 The Message Passing Interface

According to [21], the **message-passing interface** or **MPI**, is a set of API called from user programs which provides a communication protocol for message-passing systems. So it is able to abstract all communications between processes. In MPI abstraction, when we write a program which will be run on four machines of the clusters, every machine executes its own copy of the program and following official terminology we say we have four **processes**. Though the nodes are all running the same program, they will likely be working on different parts of its data. This is called the **Single Program Multiple Data** (**SPMD**).

From an architectural point of view, one of the most important parts of MPI is the **communicator**. It connects group of processes in the MPI session giving to each one an independent identifier which is called **rank**. By default the rank 0 process belongs to the MPI process that starts the program. Moreover the communicator arranges these processes in an ordered topology.

MPI also provides powerful methods for **point to point communications**. An example is `MPI_Send` function, which allows one process to send a message to a second process. These type

of communications are useful in irregular communication (for example a master-slave architecture where the master sends new data task to a slave whenever the prior task is completed).

Moreover MPI provides functions for **collective communications**:

- `MPI_Bcast`: The root sends messages from a buffer to all other processes
- `MPI_Scatter`: The root has a buffer message and splits it into n parts (where n is the number of processes), then send each part to the corresponding process.
- `MPI_Gather`: It is the opposite of scatter, in fact the root fills a buffer concatenating n messages
- `MPI_Reduce`: As in the gather function the root fills the buffer with n messages. However, the collected data is then "reduced" using an associative function and the function returns the resulting value-

Chapter 7

The Julia language

In this work has been used the **Julia language** as the main programming language. As we can see from its website[1], it is a high-level and high-performance language for technical computing, with a syntax that is well known for everybody uses scientific programming languages. It also provides a compiler, distributed parallel execution and a mathematical function library. We chose it because of these characteristics and because it permits to parallelize programs in a very simple way.

In next sections we will see in detail its characteristics and the code used for parallel computation.

7.1 Principal characteristics

As we can see in the Julia website, this language has the following characteristics:

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Dynamic type system: types for documentation, optimization, and dispatch
- Good performance, approaching that of statically-compiled languages like C
- Built-in package manager
- Lisp-like macros and other metaprogramming facilities
- Call Python functions: use the PyCall package

[1] http://julialang.org/

- Call C functions directly: no wrappers or special APIs
- Powerful shell-like capabilities for managing other processes
- Designed for parallelism and distributed computation
- Coroutines: lightweight "green" threading
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- MIT licensed: free and open source

From the syntax point of view, Julia takes inspiration from various dialects of *Lisp*, including *Scheme* and *Common Lisp*, and it shares many features with *Dylan* and *Fortress*. Is is also possible to implement **metaprogramming** using macros. They are necessary because they are executed when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code before the full program is run. We can see the difference in the following example taken from the official documentation:

```
1  > macro twostep(arg)
2      println("I execute at parse time. The argument
3              is: ", arg)
4      return :(println("I execute at runtime.
5             The argument is: ", $arg))
6    end
7
8  > ex = macroexpand( :(@twostep :(1, 2, 3)) );
9    "I execute at parse time. The argument
10      is: :((1,2,3))"
11
12 > typeof(ex)
13   Expr
14
15 > ex
16   :(println("I execute at runtime. The argument is:
17     ",$(Expr(:copyast, :(:((1,2,3)))))))
```

```
18
19  > eval(ex)
20    "I execute at runtime. The argument is: (1,2,3)"
```

Macro are invoked using the following syntax:

```
1  @name expr1 expr2 ...
```

In addition, Julia includes an interactive session shell called **REPL**, which can be used to make experiments and fast code testing. For example we can write:

```
1  julia> p(x) = 2x^2 + 1; f(x, y) = 1 + 2p(x)y
2  julia> f(0, 4)
3  9
```

All these commands can also be written in a script file with `.jl` extension and executed in the shell with:

```
1  user@pc: julia <filename>
```

As we have seen in the list above Julia has good performances, which are achieved using **just-in-time (JIT)** compilation, implemented using *LLVM*. This compiler, combined with the language's design allow it to approach the C performances. In table 7.1 we can see a little benchmark coming from the official website:

Julia supports also modular applications. In particular, modules in Julia are separate global variable workspaces. They are delimited by the keywords:

```
1  module Name
2  ...
3  end
```

With modules, one can create top-level definitions without worrying about name conflicts. Within a module, it is possible to control which names from other modules are visible (via importing), and specify which names are intended to be public (via exporting). In the official documentation we can find the following example:

TABLE 7.1: benchmark times relative to C (smaller is better, C performance = 1.0).
Table taken from the official documentation

	fib	**parse-int**	**quick-sort**	**mandel**	**pi-sum**	**rand-mat-stat**	**rand-mat-mul**
Fortran	0.70	5.05	1.31	0.81	1.00	1.45	3.48
Julia	2.11	1.45	1.15	0.79	1.00	1.66	1.02
Python	77.76	17.02	32.89	15.32	21.99	17.93	1.14
R	533.52	45.73	264.54	53.16	9.56	14.56	1.57
Matlab	26.89	802.52	4.92	7.58	1.00	14.52	1.12
Octave	9324.35	9581.44	1866.01	451.81	299.31	30.93	1.12
Mathematica	118.53	15.02	43.23	5.13	1.69	5.95	1.30
Javascript	3.36	6.06	2.70	0.66	1.01	2.30	15.07
Go	1.86	1.20	1.29	1.11	1.00	2.96	1.42
LuaJIT	1.71	5.77	2.03	0.67	1.00	3.27	3.27
Java	1.21	3.35	2.60	1.35	1.00	3.92	2.36

```
1 module MyModule
2 export x, y
3 x() = "x"
4 y() = "y"
5 p() = "p"
6 end
```

In this module we export the x and y functions (with the keyword `export`), and also have the non-exported function p. In Table 7.2 there are several different ways to load the Module and its inner functions into the current workspace.

Obviously, we can use file scripts for defining multiple modules. Moreover, including the same code in different modules provides mixin-like behavior. One could use this to run the same code with different base definitions, for example testing code by running it with "safe" versions of some operators:

```
1 module Normal
2 include("mycode.jl")
3 end
```

TABLE 7.2: Modules loading in Julia. Table taken from the official documentation

Import command	**What is brought into scope**
`using MyModule`	All exported names (x and y), MyModule.x, MyModule.y and MyModule.p
`using MyModule.x, MyModule.p`	x and p
`using MyModule: x, p`	x and p
`import MyModule`	MyModule.x, MyModule.y and MyModule.p
`import MyModule.x, MyModule.p`	x and p
`import MyModule: x, p`	x and p
`importall MyModule`	All exported names (x and y)

```
4
5 module Testing
6 include("safe_operators.jl")
7 include("mycode.jl")
8 end
```

7.2 Parallel programming in Julia

How we have seen in the earlier chapter, modern computer posses more than one CPU and several computers can be combined together in a cluster. Moreover we have seen the big advantages that this type of architecture can give us. For its parallel features, Julia have chosen an environment based on *message passing* (so we have multiple processes which run in separate memory domains). However, the Julia's implementation of message passing is a bit different from other environments such as MPI. In fact communication is generally "one-sided", so the programmer needs to explicitly manage only one process in a two-process operation. In addition these operations do not seems "message send" and "message receive" but resemble higher-level operations like calls to user functions.

The most important primitives, are **remote references** and **remote calls**. The former is an object that can be used from any process to refer to an object stored on a particular process, while the latter is a request by one process to call a certain function on certain arguments on another

process. A remote call returns a remote reference to its result and *return immediately*, while its process proceeds to the next operation. It is possible to wait for a remote call to finish by calling the function `wait` on its remote reference (or `fetch` if we are waiting for its value).

Now we can examine some useful functions for parallel programming. The simpler function is `remotecall()`, which takes as parameters the *index of the process that will do the work*, the function that will be invoked and its parameters. It is considered a low-level interface that provides a finer control on the software. However most parallel programming in Julia, does not reference specific processes or the number of processes available. So usually Julia programmers uses the `@spawn` macro, which operates on an expression and choose where to do the operation. For example we can write:

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

Note the use of `1 .+ fetch(r)` instead of `1 .+ r`. This is because we do not know where the code will run, so in general a `fetch()` might be required to move r to the process doing the addition. In this case, `@spawn` is smart enough to perform the computation on the process that owns r, so the `fetch()` will be a no-op.

Now we have to do a final note on code availability. In fact, according to Julia parallel architecture, the code must be available on any process that runs it. For example we can observe this code from the official documentation:

```
julia> function rand2(dims...)
           return 2*rand(dims...)
       end

julia> rand2(2,2)
2x2 Float64 Array:
 0.153756  0.368514
```

```
 1.15119    0.918912

julia> @spawn rand2(2,2)
RemoteRef(1,1,1)

julia> @spawn rand2(2,2)
RemoteRef(2,1,2)

julia> exception on 2: in anonymous: rand2 not defined
```

As as consequence, we have to be aware of this fact when loading code from modules. Generally speaking, we can have the following cases:

- `include("ModuleName.jl")` loads the file on just a single process
- `using ModuleName` causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement

In addition, we can force a command to run on all processes using the `@everywhere` macro. For example we can write:

```
@everywhere using ModuleName
```

In conclusion, here we have seen the most interesting features of Julia language (in particular the ones which are involved in parallel computing). At this point, the reader should have understood why we have chosen this language and the decisions at the basis of the developed library

Part III

The application

Chapter 8

Architecture of ImagesToLARModel

From this chapter, our attention will be focused on the application realized for this work. Its purpose is to extract three-dimensional models from medical images in a efficient way, trying to reduce the memory occupation as much as possible. In particular, in this chapter we will see the application architecture and the ideas at the basis of its behavior; later we will examine in detail the algorithms used

8.1 Introduction to the application

How we have seen above, the application described in this work, which is called **ImagesToLARModel**, has the aim to produce three-dimensional models from medical images. In Chapter 3 and Chapter 4, we have seen that there are many techniques for producing these models. However they have some inconvenience, for example they can be slow or produces only approximated models. In particular, when dealing with huge volumes of data, people use iso-surfaces with the marching cubes algorithm so our comparisons will be done with it.

Now we can talk about our technique. How we have seen in Chapter 6, a three-dimensional model can be represented using the LAR representation schema. Its advantages are the small space required for memorization and the speed provided by algorithms based on topological algebra. In addition the LAR schema is **topologically correct**, so we are sure that the resulting model is *perfectly correspondent to the given data*. We can be sure of this assumption because it is proved by the topological theory, that provides us a great foundation for all possible practical applications. Moreover this means that LAR schema is *independent from the shape* of the complex,

so using the same algorithm we can process whichever model (even with holes).

How we can see in [1], in the past had already been developed a first prototype for the extraction of three-dimensional models from images. However that application was not specifically designed for a parallel environment, so it was not able to manage models so big that they cannot enter in memory. ImagesToLARModel can solve this problem using parallel algorithms applied to the LAR schema. In next sections we will see how to do this.

8.2 Distributing the model in a grid

How we have seen earlier, ImagesToLARModel is able to exploit a parallel architecture for extracting models from medical images. In particular we are focused on a *cluster environment*. The strategy adopted for the problem decomposition among all processes is very simple. In fact, it is sufficient to draw a grid on our images and distribute the resulting images **blocks** to all processes. After a progressive refinement, the blocks are then merged obtaining the final three-dimensional representation in the *wavefront obj* format.

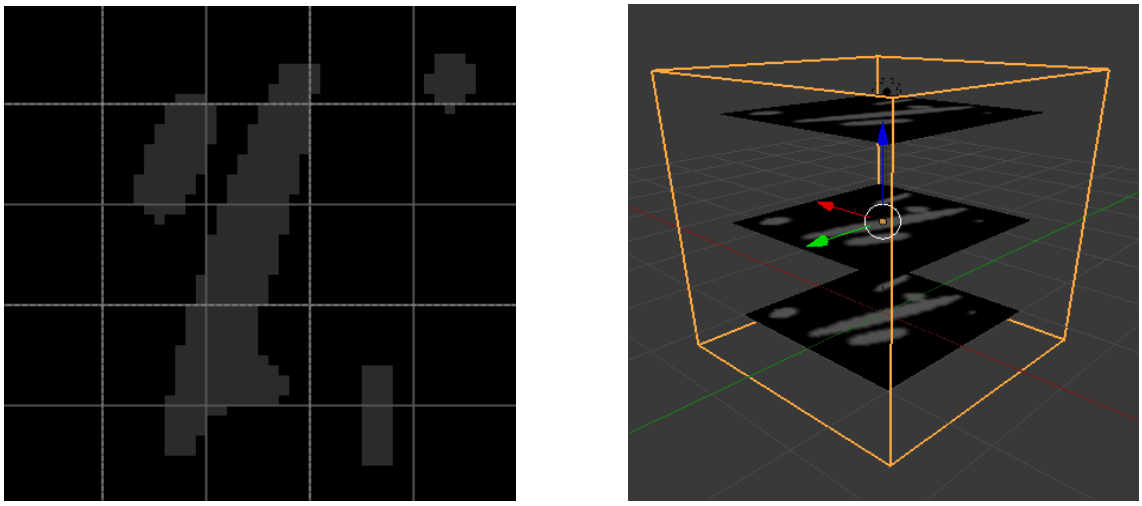


FIGURE 8.1: The grid used for parallel computation. (a) 2D grid on a single image (b) 3D grid for the stack of images

Now we can give the following definitions, which will be used in next parts of this thesis:

- **Grid:** is the subdivision of the entire stack of images, with sizes defined by the user.
- **Block:** is a single cell of the grid

- **blockDx, blockDy, blockDz:** are the dimensions of a single block
- **xBlock:** is the x-coordinate of a block
- **yBlock:** is the y-coordinate of a block
- **zBlock:** is the z-coordinate of a block

xBlock and *yBlock* are defined on a single image, while *zBlock* is defined on different images; in the application it is also referred with the terms **StartImage** and **EndImage**, which indicate the first image and the last image of that block respectively.

So, we can see that every single processes now transform only a single block at time (which will contain at most blockDx $\times$ blockDy $\times$ blockDz voxels). Obviously the dimension of a block can be set by the user according to the image sizes and the characteristics of the computing infrastructure.

8.3 Exposed functionalities of the application

Now we can focus on the architecture of the application. We have already said that our application need to take a stack of medical images and produce a three-dimensional representation. Thus this has been divided into two main parts, the first manipulate the stack using image processing techniques while the second does the conversion process.

So our application will expose two functions:

- `prepareData`: take a folder with the stack of images and write on disk the same images after manipulations (filtering, clustering, etc...)
- `convertImagesToLARModel`: take the folder with the pre-processed images and converts them to the final model

These two exposed functions have the responsibility to call all the modules used by the software and both work with JSON configuration files. A full explanation of the parameters and their meaning will be introduced in the next chapters.

Now we can see the modules used:

- **ImagesToLARModel.jl**: it is the main module for the software, which takes input parameters and start images conversion and does the image processing
- **ImagesConversion.jl**: it is called by `ImagesToLARModel.jl` module and controls the entire conversion process calling all other modules
- **GenerateBorderMatrix.jl**: it generates the boundary operator for grid specified in input, saving it in a JSON file
- **PngStack2Array3dJulia.jl**: it is responsible of all functions involving images
- **Lar2Julia.jl**: it contains a small subset of LAR functions written in Julia language
- **LARUtils.jl**: it contains utility functions for manipulation of LAR models
- **Smoother.jl**: it contains function for smoothing of LAR models
- **Model2Obj.jl**: it contains function that can read and write obj models

Chapter 9

Data preparation

In the previous chapter we have seen the main principles and the organization at the basis of our application for the construction of three-dimensional biomedical models. Now we are interested describing the processes which transforms a common medical image into data usable for further conversions.

9.1 Raw data extraction from images

How we have already seen, the input for our software consists in a stack of medical images in various formats. First thing we have to do is reading the binary data from images and use those values to represents voxels. This process follows these steps:

1. Open images and read binary data
2. Convert images into greyscale format
3. Resize images (according to values chosen from the user)
4. Filter noise from images
5. Transform input images into binary images

Now we can see in detail every single step used for input transformation. The first is trivial and is based on knowledge we have seen in Chapter 3. In fact we know that every images contains raw data associated with its pixels (or voxels when considering the three-dimensional stack), as we can see in a sample image in Figure 9.1

The only problem could comes from the different image formats existing for these purposes; however the software resolved it using `ImageMagick`, a popular image processing library that

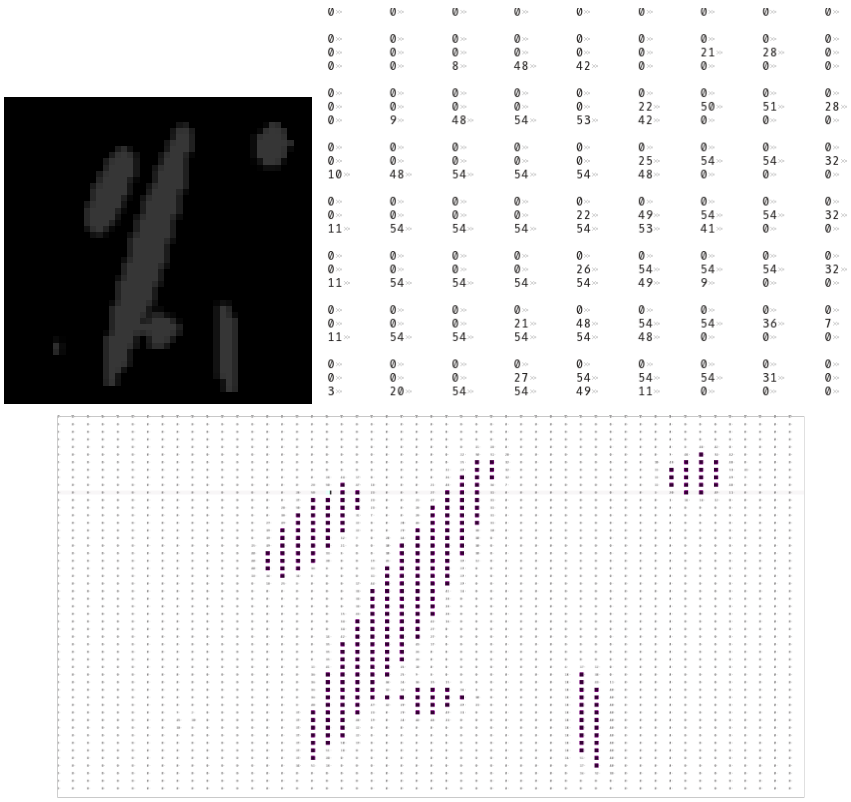


FIGURE 9.1: Reading raw data from image. (a) Original greyscale image (b) A view of raw data array (c) The entire raw data array with main color highlighted

can manage a huge number of different formats. The second step is trivial too and can be solved with a simple image processing library (as the previously cited `ImageMagick`).

In next sections we will see in detail the remaining steps which are also the more interesting.

9.2 Images resizing

A functionality that can be useful when dealing with medical images is image resizing. In fact we often have images larger than the parts we are interested in, so we can gain a great speedup with removal of useless data. Moreover, how we will see next, the boundary operator depends on the size of a single block which in turn depends on the size of images. So sometimes we would to increase the image dimension, in order to better divide the image into blocks and use a particular boundary operator matrix.

When the user wants to resize an image, passes to the software the list of desired boundaries: *[[xcropStart,xcropEnd], [ycropStart,ycropEnd], [zcropStart,zcropEnd]]*. For now we can focus on resizes on x and y axis. In Figure 9.2 we can see some resize cases.

Reducing dimensions of image is very simple; we just need to load the raw data and select only a slice. Instead, for image extension we need to concatenate rows and columns containing only zero values. This means that the increment of data does not imply an increment of sizes for the final model, as empty zones are not represented.

On the z axis the algorithm used is quite different. In fact the z-dimension is determined by the number of images. When we want to reduce the stack we just need to remove images, when we want to increase the stack dimension we just need to add black images at the end.

9.3 Creating binary images

When converting from the two-dimensional representation with images to the three-dimensional model, we need binary images. We need this format because we have to be able to unequivocally recognize what to represent in the final result and what we can safely remove. However, we know that in a common medical image every pixel represent a grayscale value and in general we can have several different values. So we need algorithms for mapping of many values into two values, say $0x00$ and $0xff$. Moreover, as we can read in [4], physical properties of tissues as recorded by medical imaging devices, do not correlate completely with the anatomic boundaries of certain organs. The problem of the identification of certain regions in an image, is very common in medical imaging processing field and it is referred to as **segmentation**. So we can use segmentation techniques to find only the right regions for our model contemporary creating binary images.

The most basic approach, and the one we will see here, is to divide the image in areas that contain interesting information and areas that are not interesting by making a binary classification based on gray levels. For example from Table 3.1 we can see that bones usually have an image density ρ above 50 HU. So we can set a **threshold** in order to save only pixels that have ρ greater than that value. In Listing 9.1 we have the implementation of the thresholding algorithm for our images

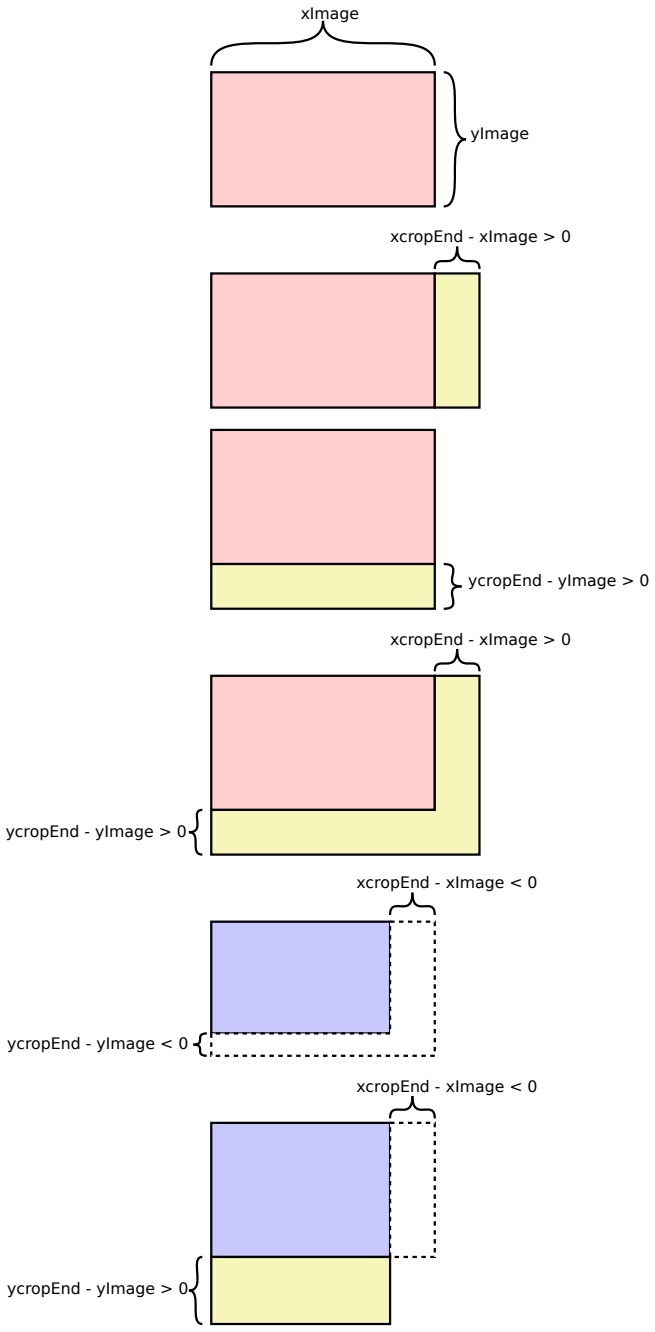


FIGURE 9.2: Some interesting resize cases. (a) The original image (b) Extension on the x dimension (c) Extension on the y dimension (d) Extension on both dimensions (e) Crop of both x and y (f) Crop of x and extension of y

ALGORITHM 9.1: Image Thresholding

```
begin
 for i in length(image):
   for j in length(image[0]):
     if image[i][j] > threshold:
       image[i][j] = 0xff
     else:
       image[i][j] = 0x00
 return image
end
```

However, if the user does not want to set a threshold (or he does not know a value for it), can create the binary images by using **clustering**. This is a definition for clustering (taken from Wikipedia):

Definition 9.1 (Clustering)**.** *The clustering is the task of grouping a set of objects in such a way that objects in the same group (called a **cluster**) are more similar (in some sense or another) to each other than to those in other groups (clusters).*

So the clustering is not one specific algorithm but the task to be solved. It can be achieved by many algorithm but here we will see only the **k-mean algorithm**. Here, we have a set of *observations* $(x_1, x_2, \ldots, x_n)$, where each observation is a d-dimensional real vector. We want to partition the n observations into $k \leq n$ sets $S = \{S_1, S_2, \ldots, S_k\}$ in order to minimize the within-cluster sum of square distances. In Listing 9.2 there is the pseudocode:

ALGORITHM 9.2: K-means algorithm

```
begin
  Initialize cluster centroids m_1^(1),...,m_k^(1)
  do until <convergence-condition>:
    S_i^(t) = {x_p: ||x_p - m_i^(t)||^2 <= ||x_p - m_j^(t)||^2 forall j, 1 <= j <= k}
                                                    \\Assignment step
    m_i^(t+1) = 1/|S_i^(t)| * sum_{x_j in S_i^(t)} x_j \\ Update step
end
```

We reach the convergence condition when the assignments no longer changes. In Figure 9.3 there is an example of k-means computing.

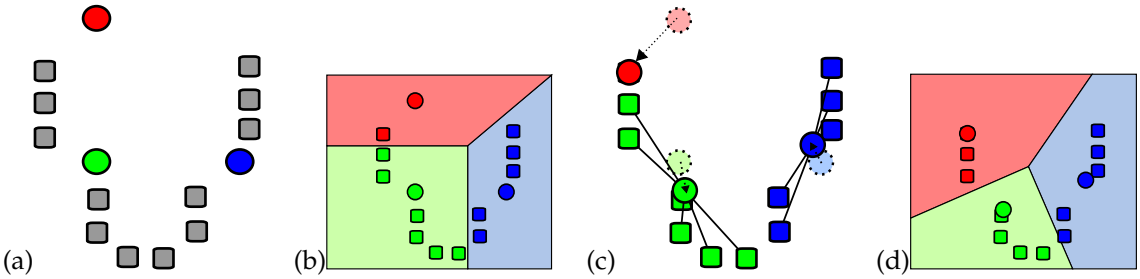


FIGURE 9.3: K-means clustering. (a) k initial "means" (in this case k=3) are randomly generated within the data domain (shown in color). (b) k-clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means. (c) The centroid of each of the k-clusters becomes the new mean. (d) Steps b and c are repeated until convergence has been reached. Images taken from Wikipedia

In our application, we have only two cluster and we will assign the $0x00$ and $0xff$ values depending on which cluster the pixel are.

9.4 Images filtering

Now we are interested in improving quality of our images. So for example we have implemented a filter for removal of noise from the image. Our choice fell on a **median filter**, because it preserves better the image edges. The main idea is to iterate on every pixel of the image replacing each value with the median of neighbors. The pattern of neighbors is called **window** which slides over the entire image, and we can have a lot of possible patterns (such as boxes or crosses). For a better understanding we can see the following example with a window of size three. Given a one-dimensional vector $x = [4\ 65\ 8\ 3]$, the filtered output y will be:

$y[1] = Median[4\ 4\ 65] = 4$

$y[2] = Median[4\ 65\ 8] = 8$

$y[3] = Median[65\ 8\ 3] = 8$

$y[4] = Median[8\ 3\ 3] = 3$

So $y = [4\ 8\ 8\ 3]$. How we can see, when we consider the boundaries we repeat the first or the last value. However other choices are possible, for example we could not consider boundaries or fetching entries for other places. A simple implementation of a median filter is given in Listing 9.3.

ALGORITHM 9.3: Median filter

```
1 begin
2     edge_x = ⌊(window width / 2)⌋
3     edge_y = ⌊(window height / 2)⌋
```

```
4     for x from edge_x to image width − edge_x :
5       for y from edge_x to image height − edge_y :
6         i = 0
7         for f_x from 0 to window width:
8           for f_y from 0 to window height:
9             window[i] = inputPixelValue[x + f_x − edge_x][y + f_y − edge_y]
10            i++
11        sort entries in window
12        outputPixelValue[x][y] = window[(window width·window height)/2]
13    return outputPixelValue
14  end
```

However when we have a lot of noise this filter could not work well. In fact, it removes or preserves pixels without seeing if they truly contains a useful information (especially when the noise is concentrated on a big region of the image). So in the software we have implemented a particular filter that consider groups of adjacent pixels on all the stack, removing only ones that have a small size (thus assuming that they are not interesting for our purposes). As a consequence, it is the first part that see our input as a three-dimensional model. The main idea consists in visiting adjacent pixels as they were nodes of a graph using a **Depth First Search**. The pseudocode is:

ALGORITHM 9.4: DFS visit

```
1  begin
2    S.push(vertex) // S is a stack and v is the first vertex
3    while S is not empty:
4      v = S.pop()
5      if v is not labeled as discovered:
6        visited.push(v) // visited contains all visited vertices
7        label v as discovered
8        foreach edge in adjacentEdges(v):
9          S.push(w)
10   return visited
11 end
```

The most important part there is the $adjacentEdges$ function, which describes the *adjacency condition* for the given graph and it changes for every particular application. This is a function

valid for every type of graph. What can change from an application to another is the *adjacency condition*. The adjacency condition used in this case is trivial. In fact we assume that:

Definition 9.2 (Adjacency condition for pixels)**.** *Given a pixel with coordinates (xPixel, yPixel, zPixel) we consider only the following adjacent pixels:*

$z \in [zPixel - 1, zPixel + 1]$

$coordinates = \{(xPixel - 1, yPixel, z), (xPixel + 1, yPixel, z), (xPixel, yPixel - 1, z), (xPixel, yPixel + 1, z)\}$

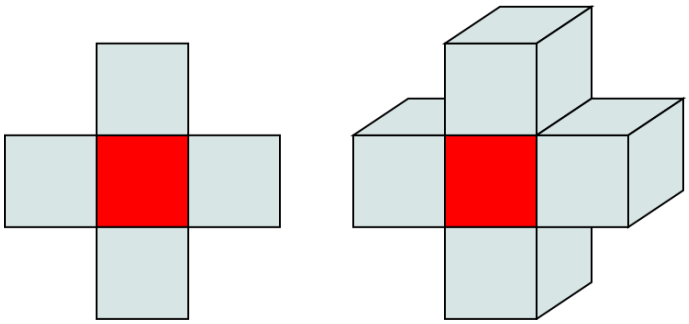


FIGURE 9.4: Adjacency relationship for pixels. (a) A 2D view of the relationship. (b) A 3D view of the relationship (where the z-axis is determined by previous and next pictures in the array stack

Summing up, the code for the three-dimensional filter is the following:

ALGORITHM 9.5: 3D filter

```
1 begin
2   foreach pixel p:
3     if p was not visited:
4       visited = DFS(p)
5       if visited.length < threshold:
6         mark all pixels in visited as 0x00
7       mark all nodes in visited as visited
8 end
```

Chapter 10

Images conversion

In the previous Chapter we have see how to prepare input for our software. Now we will focus on the proper process conversion. How we will see here, for the conversion we have a **pipeline of transformations** which progressively refines the results. Firstly we will give an overall overview of the pipeline, then we will see in detail each step.

10.1 The conversion pipeline

We have already seen in Chapter 8.2, in the conversion process we divide our input into *blocks* which are manipulated in parallel. In particular, we execute several transformations for each block that progressively refine the result. As a consequence we can see that from an architectural point of view `ImagesToLARModel` is an application based on the pattern **pipes and filters**, where each conversion step represent a filter whose output is the input for the next one. Actually we have the following steps:

- **Pixel to voxels transformations**
- **Boundaries merge**
- **Block merge**
- **Smoothing**
- **Final model creation**

Each step works on a block of the model, so the pseudocode for the definition of a step is the following:

ALGORITHM 10.1: Single step of the conversion pipeline

```
1  begin
2    beginImageStack = 0
3    endImage = beginImageStack
4    for zBlock from 0 to image depth/blockDz − 1:
5      startImage = endImage
6      endImage = startImage + blockDz
7      for xBlock from 0 to image width/blockDx − 1:
8        for yBlock from 0 to image height/blockDy − 1:
9          parallel execute stepFunction on b = (xBlock, yBlock, zBlock)
10 end
```

How we can see from the pseudocode, for the implementation of a step, the user have to give the sizes of the block

10.2 Converting pixels to voxels

Now we can see the first step of the conversion pipeline: **conversion from pixels to voxels**. First of all, we need to load only the binary data of the image stack corresponding to the current block obtaining a cuboid geometry of the block. At this point, we can transform the binary matrix into an array. In Figure 10.1 there are two examples showing how this transformation works.

$$\begin{pmatrix} 0^0 & 0^2 \\ 0^1 & 0^3 \end{pmatrix} \begin{pmatrix} 1^4 & 0^6 \\ 1^5 & 1^7 \end{pmatrix} \rightarrow \; 0^0 \quad 0^1 \quad 0^2 \quad 0^3 \quad 1^4 \quad 1^5 \quad 0^6 \quad 1^7$$
$$\begin{pmatrix} 0^0 & 0^2 \\ 0^1 & 0^3 \end{pmatrix} \begin{pmatrix} 0^4 & 1^6 \\ 1^5 & 1^7 \end{pmatrix} \rightarrow \; 0^0 \quad 0^1 \quad 0^2 \quad 0^3 \quad 0^4 \quad 1^5 \quad 1^6 \quad 1^7$$

FIGURE 10.1: Transformation of a matrix resulting from a $2 \times 2 \times 2$ grid into an array (with cells indexes) (a) First example (b) Second example

We have already seen that the input for this step consists in binary images which will contain only $0x00$ and $0xff$ values. Obviously we are only interested in non-empty values, so what we have to do now is to iterate on data and get only indices of the array where the value is $0xff$. Consequently, we have obtained a set of pixels that belongs to the model. If we want a real three-dimensional representation, we only have to map a pixel to a voxel transforming flat squares into cube geometries.

So far we have obtained a list of cuboidal cells inside a block; however in our representations we want only **boundary cells**, so we have to find them. How we have seen in Chapter 5.3 it is simple to compute the boundaries using LAR. So at this point we have to convert the list of cells into a LAR model. This is possible by creating a basis for a cell complex (formed by cuboids into a grid of the same size of the image grid with integer coordinates that vary from $(0, 0, 0)$ to $(blockDx, blockDy, blockDz)$) and saving only cubes whose indices are the same contained in the list we have computed earlier. Final boundary computation can be done computing a **boundary matrix** for the block of sizes $(blockDx, blockDy, blockDz)$ and, as it depends only on those dimensions, it can be saved somewhere and reused with other models.

All this process is **embarrassingly parallel**, which means that every block can be processed independently of the others.

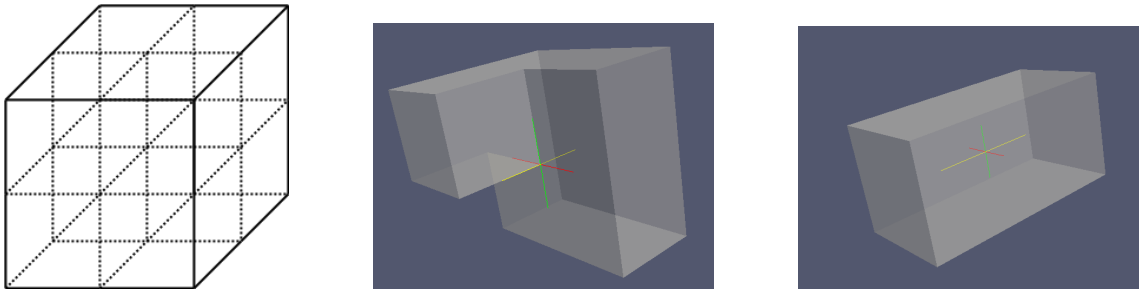


FIGURE 10.2: Sample models of $2 \times 2 \times 2$ blocks. (a) Basis for a $2 \times 2 \times 2$ block. (b) and (c) Representations of sample geometries obtained from the previous basis

10.3 Merging boundaries

At the end of the previous step, we have obtained at most $\frac{image - width}{blockDx} \times \frac{image - height}{blockDy} \times \frac{image - depth}{blockDz}$ blocks (some of them could be empty). However because of every single block is processed independently of the other we still have boundaries between them. In fact, the boundary operator works only on a single block so we have to *manually remove faces between blocks*. A simple procedure is explained in the following pseudocode and in Figure 10.3:

ALGORITHM 10.2: Removal of internal boundaries

```
1 begin
2   foreach block b:
3     rb = next block on the right
```

```
    tb = next block on the top
    fb = next block on the front
    merge right boundary of b with left boundary of rb
    merge top boundary of b with bottom boundary of tb
    merge front boundary of b with back boundary of fb
end
```

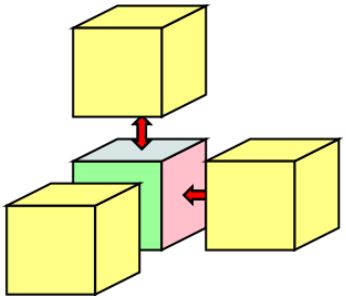


FIGURE 10.3: Merging of boundary faces. For a single block we need adjacent blocks on the right, top and front

How we can see from the pseudocode, the first thing we will need is to recognize boundaries from the interior of a block. In fact, we can decompose a single block into seven parts (listed in Figure 10.4) observing the coordinates of every face.

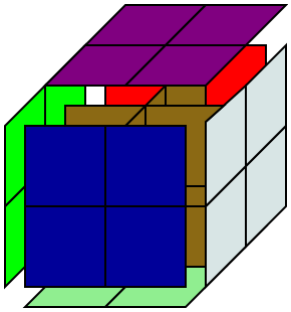


FIGURE 10.4: Decomposition of a LAR model into seven parts: the inside model (brown), the left boundary (green), the right boundary (light blue), the top boundary (purple), the bottom boundary (light green), the front boundary(blue), the back boundary (red)

Now we can focus on the merge procedure. First of all we need to remove double vertices from models. To achieve this goal, we need to iterate on the vertices array and find and remove them saving the indices of the removed ones at the same time. Then we have to reindex the faces in order to remove links to the removed vertices. At this point, we can find faces with the

same coordinates and remove them from both blocks we are merging. In Listing 10.3 there is the pseudocode for the procedure we have described here.

ALGORITHM 10.3: Merging of two boundaries

```
1 begin
2   concatenate V1 and V2
3   concatenate FV1 and FV2
4   remove double vertices from V1 + V2
5   reindex vertices indices in FV1 + FV2
6   remove double faces in FV1 + FV2
7 end
```

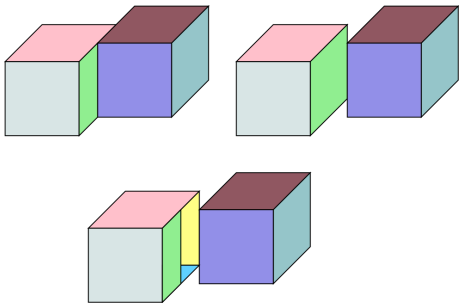


FIGURE 10.5: Removal of double faces from boundaries. (a) Two adjacent blocks (b) The same blocks exploded on x axis (c) Result of the removal on the exploded blocks

10.4 Merging blocks

At the end of the previous step, we have obtained models representing the inner parts of a single block and the boundaries remained after removal of duplicates. Now we are interested in merge all these models into blocks removing duplicated vertices between them, in order to save space and prepare the model to the next step. In fact, the process that produces the models uses adjacent cuboids with duplicated vertices. An example is given in Figure 10.6, where we can see the boundary of a model obtained from a $2 \times 2 \times 2$ grid

As in the previous step we have to concatenate models, remove the repeated coordinates and reindex vertices in faces.

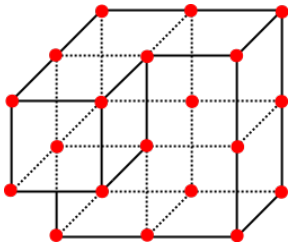


FIGURE 10.6: A sample model taken from a $2 \times 2 \times 2$ grid with double vertices between faces in red (remember that we have only the boundaries faces for the model)

10.5 Smoothing

At the end of the previous step we have obtained independent blocks without double vertices. However, these blocks are not so good to see, as they have squared edges (remember that they are composed by attached cuboids). As the medical parts usually have rounded edges, now we want to **smooth** the three-dimensional model.

There are many different algorithms for mesh smoothing, the simpler and the one it has been used in the library is **laplacian smoothing**. Here, for each vertex in a mesh, a new position is chosen according to local information (such as the coordinates of neighbors) and the vertex is moved there. If that mesh is topologically a rectangular grid (so each internal vertex is connected to four neighbors) then this operation produces the *Laplacian* of the mesh.

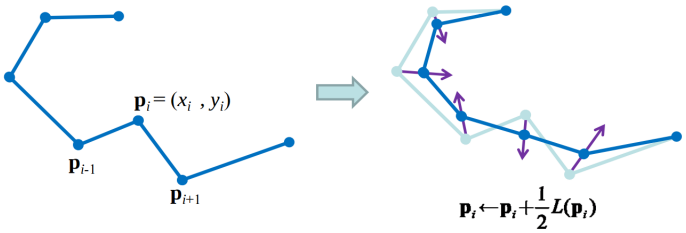


FIGURE 10.7: Laplacian smoothing (picture taken from the *Geometry Processing Algorithms* course at Stanford University)

As we can see from Figure 10.7, with substitution of every vertex position with the mean of the neighbors positions, we can obtain a curve with smoothed edges. This procedure can be repeated many times. This is the pseudocode for laplacian smoothing:

ALGORITHM 10.4: Laplacian smoothing

```
1 begin
2   foreach vertex i:
```

```
3       x̄_i = 1/N Σ_{j=1}^{N} x̄_j
4   end
```

In that procedure, N is the number of adjacent vertices to vertex i, $\bar{x}_j$ is the position of the j-th adjacent vertex and $\bar{x}_i$ is the new position for i. How we can see, computation of adjacent vertices is a very important task. However, using LAR it become simple as we just need to compute the VV relation.

Now we can focus on smoothing for our blocks. One great problem that arises from our subdivision is the computation of adjacent vertices on boundaries. In fact we cannot load the entire model into memory due to the enormous sizes. The solution to this problem consists in loading also the near blocks to the one we want to smooth, execute the algorithm and than save only vertices of the block. In Figure 10.8 there is a graphical explanation for the algorithm while in Listing 10.5 there is the pseudocode.

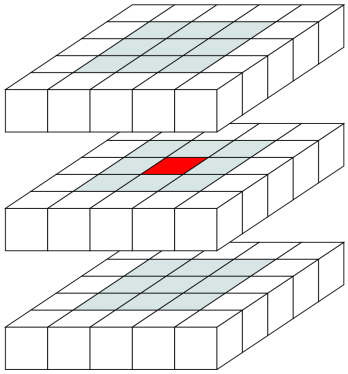


FIGURE 10.8: Smoothing of a single block. The red block at the center of the figure is the current one, while the other twenty six colored ones are the blocks that will be part of the model which will be smoothed for this iteration

ALGORITHM 10.5: Smoothing of a block

```
1  begin
2    foreach block b:
3      Merge b with its near blocks
4      Compute VV relation
5      Execute smoothing on the resulting model
6      Save new vertices only for b
7  end
```

10.6 Creating the final model

At the end of the previous step, we have obtained smoothed blocks. Now, as we want to study the entire model, we have to merge all these blocks in a unique file. The file format chosen is the **wavefront obj** format, which is simple and widespread. The syntax used is the following:

- All vertices are described with their coordinates and written on a single row according to the following syntax:

  `v xCoord yCoord zCoord`

- All faces are described with their vertex index (calculated from their row) according to the following syntax:

  `f vertex1 vertex2 ...vertexn`

In Figure 10.9 there is an example of an obj file

```
v 0 0 0
v 1 0 0
v 0 1 0
v 1 1 0
v 0 0 1
v 1 0 1
v 1 1 1
f 1 2 4 3
f 5 6 8 7
f 1 2 6 5
```

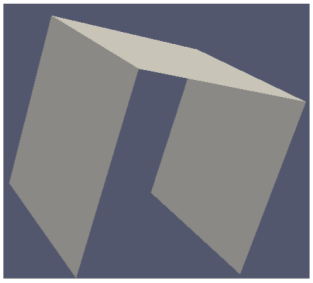


FIGURE 10.9: Obj sample file

We can see that this kind of representation is very similar to the LAR representation schema, so we just have to read every element of V and FV (in the list of indices format) and write them on disk.

At the end of this step, we will have the full model in obj format which can be observed on every three-dimensional viewer.

Part IV

Case studies and conclusions

Chapter 11

Studying the pipeline steps with an example

Now we can start to see some interesting examples for a complete understanding of our technique and of software functionalities. In the example that we will introduce in this chapter, we will study in detail the output of every conversion step. In particular, here we will use one of the classic example of computer graphics: the **Stanford Bunny**.

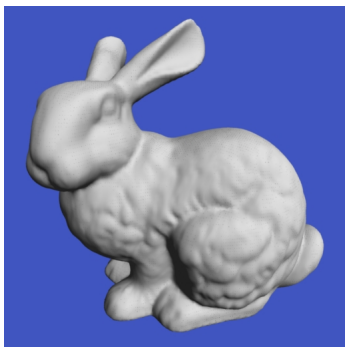


FIGURE 11.1: A three-dimensional representation of the Stanford Bunny (realized by Peter Lindstrom)

In the *Stanford volume data archive* we can find a CT scan of a terra-cotta bunny which we will use to create our three-dimensional model. The data provided is in a raw 512×512 format with 361 slices stored as 16 bit pixels. First of all we have to read this data and create the images (the results are in Figure 11.2).

At this point, we have to prepare the data for conversion. In particular, as we can see in Figure 11.3, we have applied a threshold on a data at the value of 37000 HU and a crop on y-axis cutting out the last 142 pixels and on z-axis cutting out 5 images:

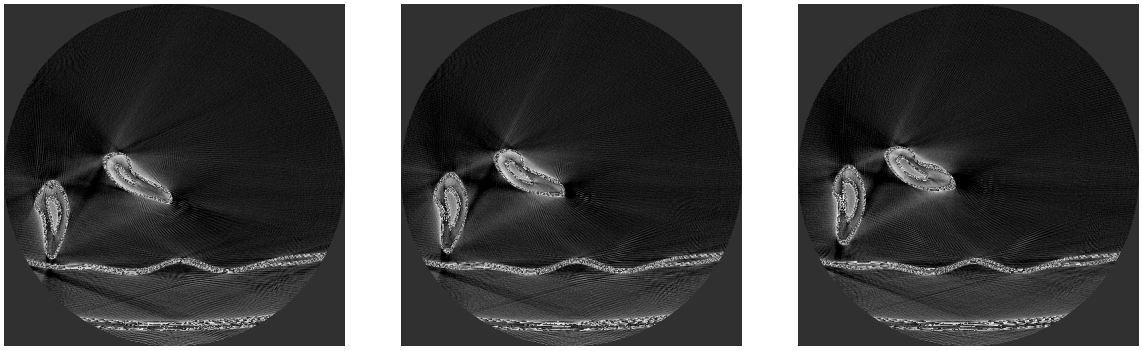


FIGURE 11.2: CT scans of the Stanford Bunny

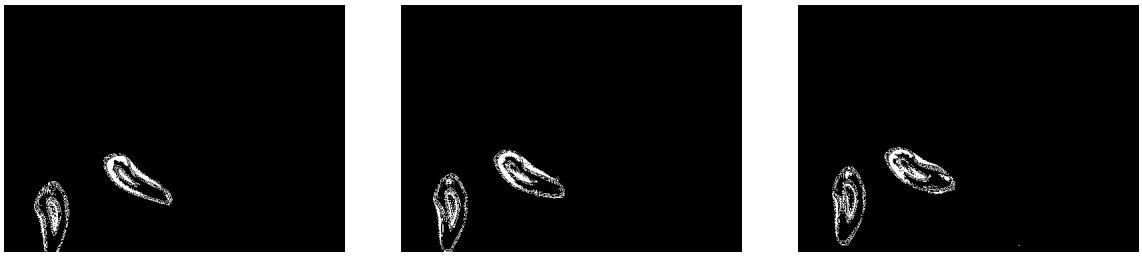


FIGURE 11.3: Binary images obtained from the previous CT scans. Threshold set at 37000 HU

At the end of this step we have obtained 356 images with size 512×370, so now we can start the conversion pipeline. First of all we have to choose the sizes of the grid. In this case, we have chosen a grid of $64 \times 10 \times 4$. The first step consists in the computation of the boundary chain for every block. In Figure 11.4 we can see two sample blocks

Now we can delete double vertices and boundaries between blocks. In Figure 11.5 we can see the effect of the removal of the boundaries on the previous blocks.

Now if we want to merge these blocks, we will obtain a three-dimensional model with squared edge (see Figure 11.6).

The next step of the conversion pipeline solve this problem applying the laplacian smoothing to our bunny. In Figure 11.7 we can see the smoothed model.

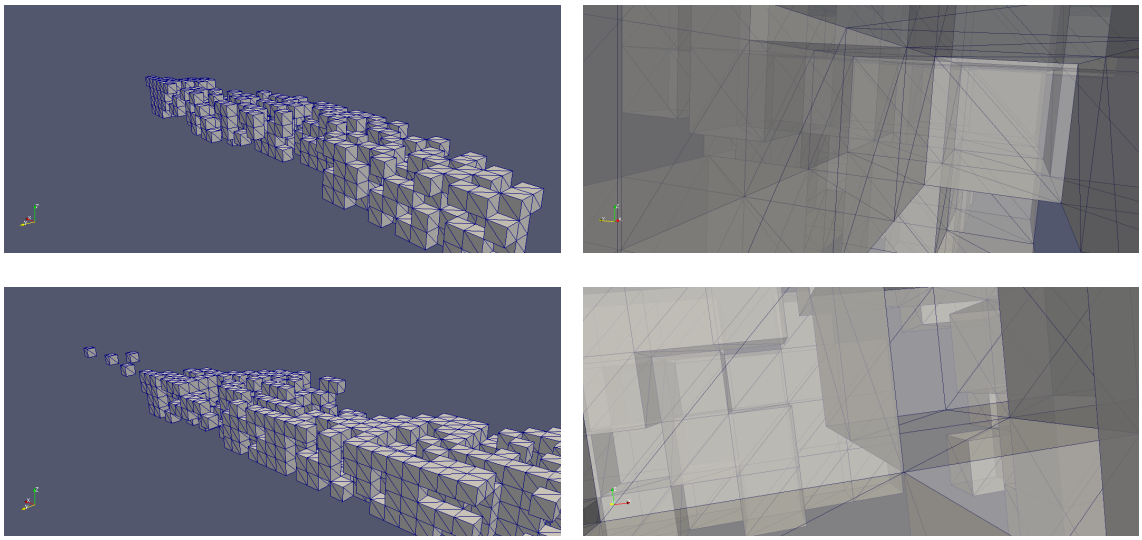


FIGURE 11.4: Some blocks obtained from the images. On the two rows we have different sample blocks. Notice how every block is empty at its internal.

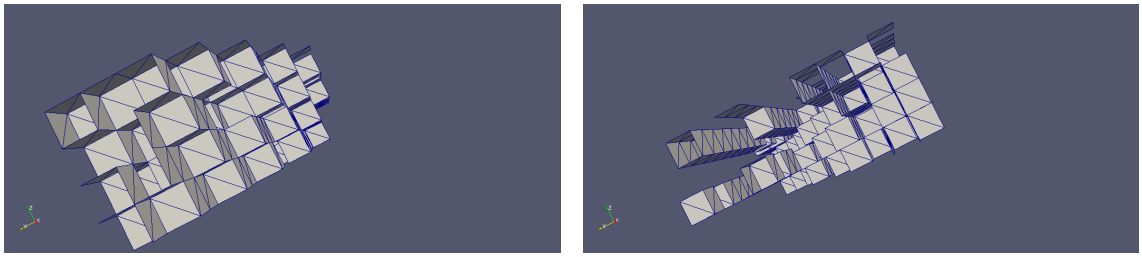


FIGURE 11.5: Removal of the duplicated boundaries from blocks. The two figures refer to the two blocks in Figure 11.4

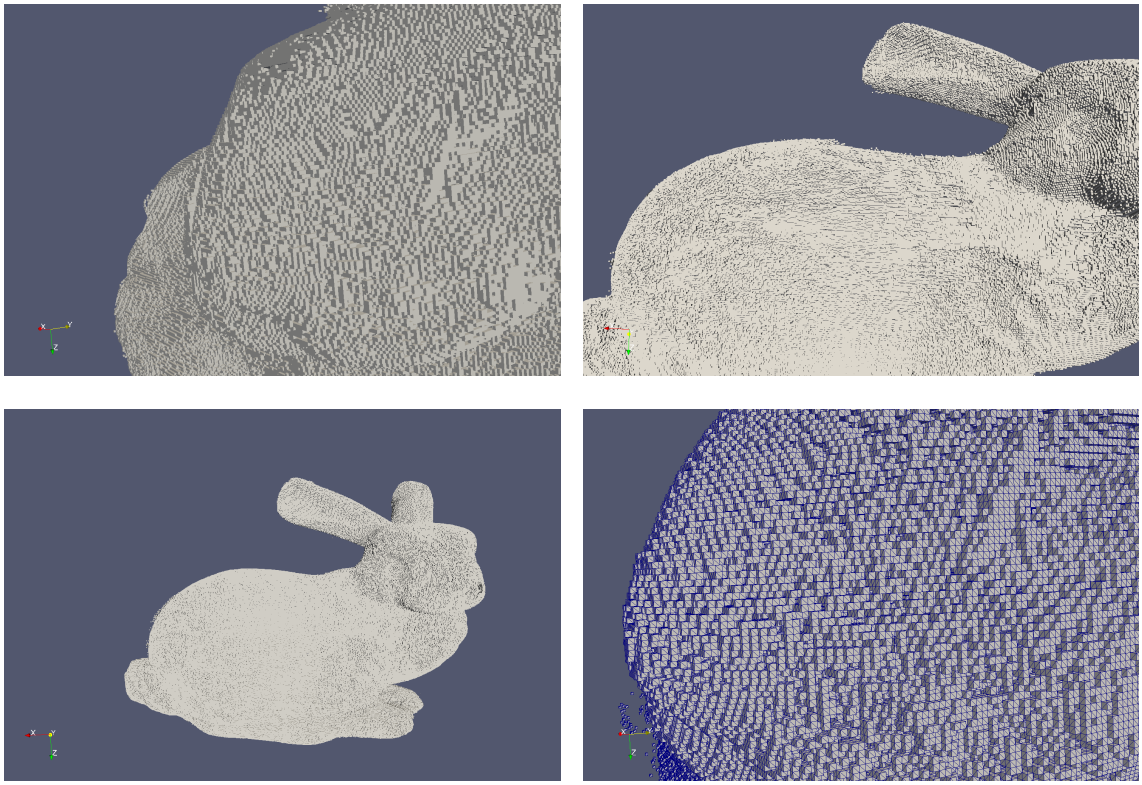


FIGURE 11.6: Squared bunny

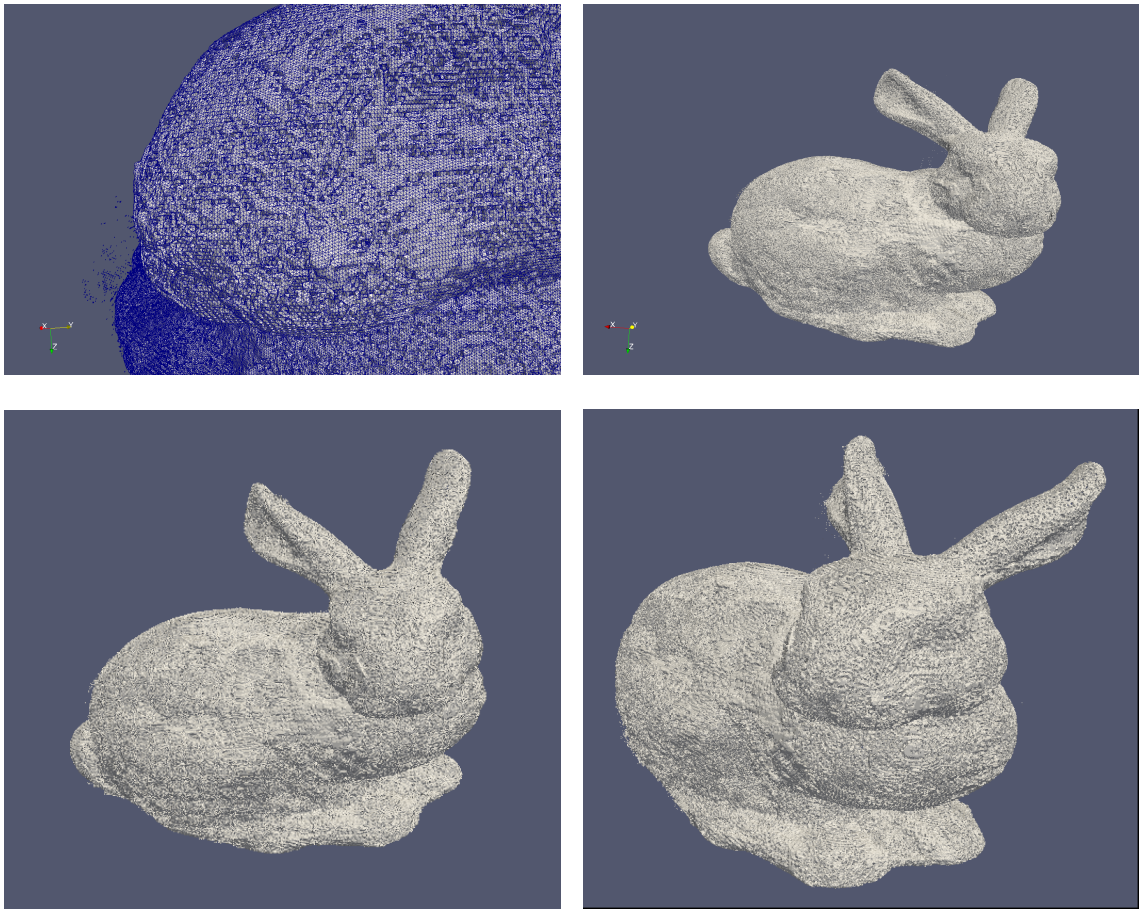


FIGURE 11.7: Smoothed bunny

FIGURE 11.8: Rendering of Stanford bunny. This is the same model created above and rendered with Blender

Chapter 12

Comparison between marching cubes and our algorithm

In the previous chapter, we have studied the conversion process with a classical example. Now we will study in detail a three-dimensional representation of cerebral veins. For now we are not interested in a study from a medical point of view, in fact our objective is to compare our technique with the commonly used ones. In Figure 12.1 we can see the veins representation with our technique and the same model created with the marching cubes algorithm.

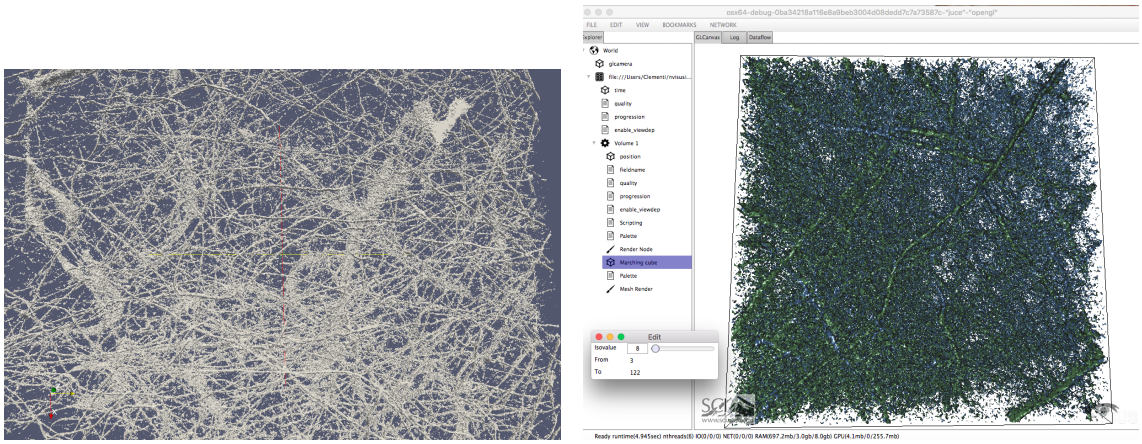


FIGURE 12.1: Comparison between LAR and marching cubes. (a) A three-dimensional model of cerebral veins taken with our software. (b) The same model obtained from marching cubes and visualized in *Visus* visualizer

From that figure, we can see that these two models seems similar when they are observed at a certain distance. However in Figure 12.2 we can see that the reality is different.

How we can see in that figure, while from a certain distance the models seems similar, a zoomed view shows all the errors caused by the marching cubes algorithm. In particular we can

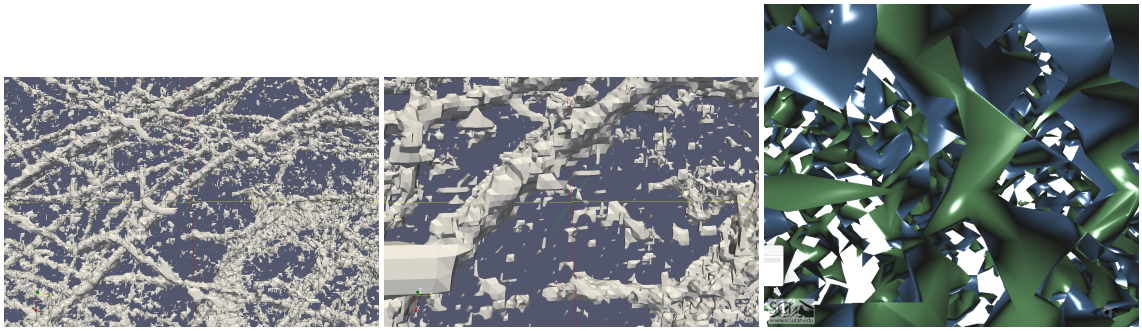


FIGURE 12.2: Topological errors in marching cubes. (a) and (b) Some images showing a zoomed portion of our three-dimensional model. (c) A zoomed view of the veins obtained from the marching cubes algorithm.

see that the topology of a single vein is completely lost. With LAR, instead, we do a *topologically perfect* extraction saving the correct shape of the represented objects. In Figure 12.3 we can see that every connected component (even noise) is closed.

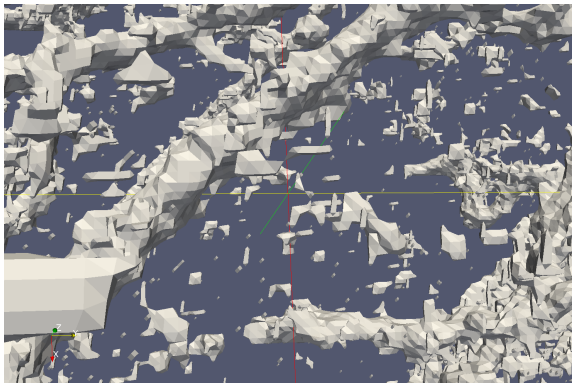


FIGURE 12.3: A close-up view of the LAR model

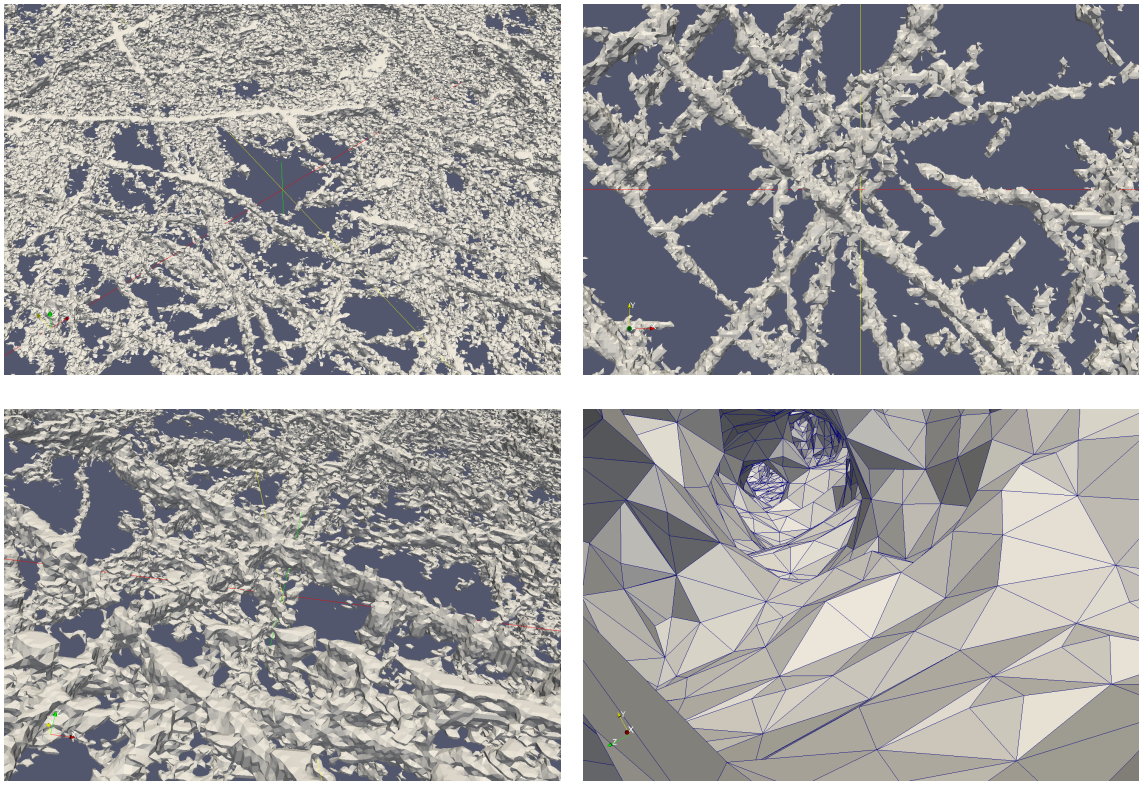


FIGURE 12.4: A detailed view of cerebral veins obtained with LAR. We can see all component are topologically correct (for example in (d) we can wee the interior of a vein)

Chapter 13

Studying a three-dimensional model of a liver

13.1 Hepatic portal system

The **hepatic portal system**, is the systems which includes the **hepatic portal vein** and its tributaries. It it responsible for directing blood from parts of the gastrointestinal tract to the human liver; in fact the substances absorbed in the small intestine travel first to the liver and then continue to the heart. Not all of the gastrointestinal tract is part of this system, in fact it extends from the lower portion of the esophagus to the upper part of the anal canal.

Blood flow to the liver is unique in that it receives both *oxygenated* and *deoxygenated* blood. So, the partial gas pressure of oxygen (pO_2) and the perfusion pressure of portal blood, are lower than in other organs of the body. Blood passes from branches of the portal vein through cavities between "plates" of hepatocytes (the cells of the liver tissue) called **sinusoids**. In addition, blood also flows from branches of the hepatic artery and mixes in the sinusoids to supply the cells with oxygen. This mixture goes through the sinusoids and collects in a central vein which drains into the hepatic vein. Then the hepatic vein drains into the **inferior vena cava**. The hepatic artery provides 30 to 40% of the oxygen to the liver, while only accounting for 25% of the total liver blood flow. The remaining part, comes from the partially deoxygenated blood from the portal vein.

The large veins that are considered part of the portal venous system are:

- Hepatic portal vein

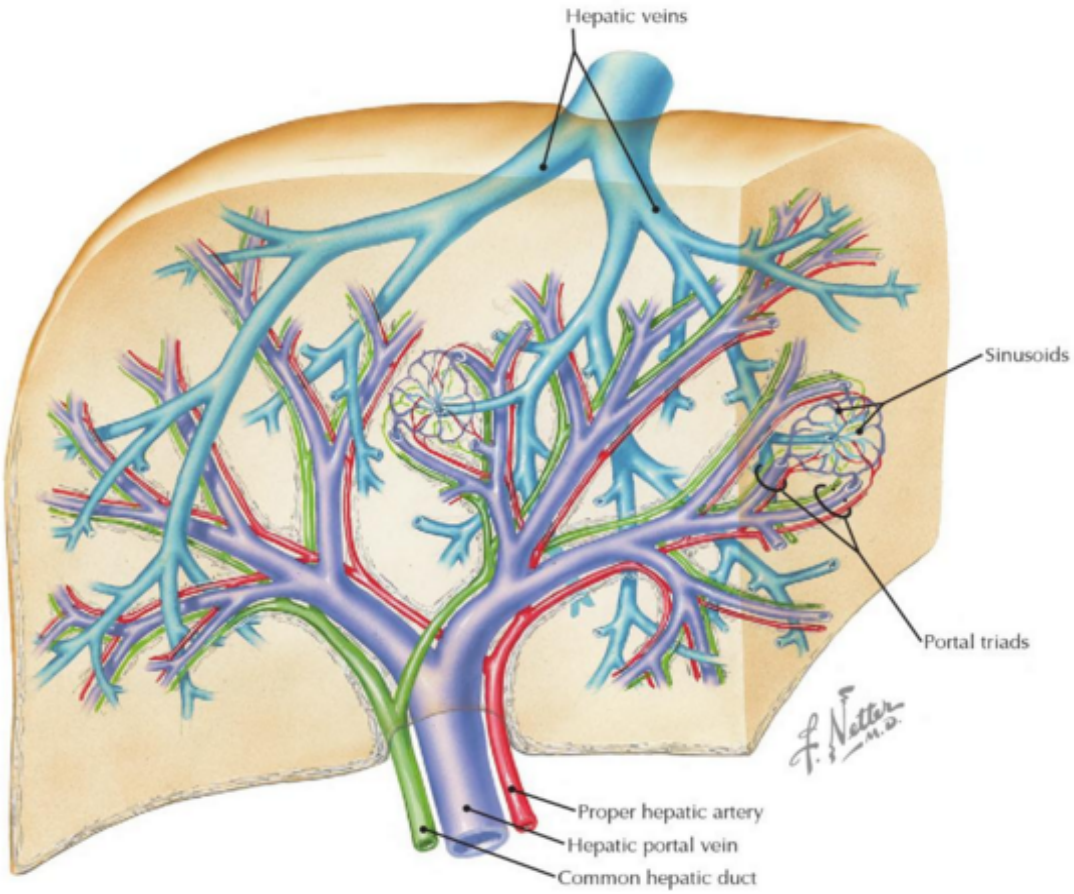


FIGURE 13.1: A view of the hepatic portal system. Image taken from *Netter's Concise Radiologic Anatomy*

- Splenic vein
- Superior mesenteric vein
- Inferior mesenteric vein

The superior mesenteric vein and the splenic vein come together to form the actual hepatic portal vein. The inferior mesenteric vein connects in the majority of people on the splenic vein, but in some people, it is known to connect on the portal vein or the superior mesenteric vein.

How we can see in Figure 13.2, the pig portal vein follows a path from right to left branching out to eight distinct segments.

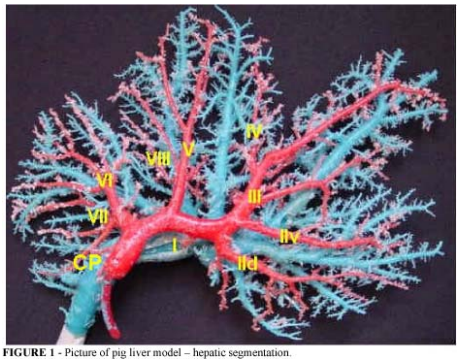


FIGURE 13.2: Three-dimensional model of a pig liver. We can see the hepatic segmentation into the eight segments

13.2 The three-dimensional model of a liver

Now we can observe a three-dimensional model of a pig liver, which is obtained from a electron microscopic scan taken from the West Bohemia University. This model could be used to perform computational fluid dynamics simulations of blood flow within the portal system. In addition, while the macroscopic structure of the hepatic vasculature is well studied, the microvasculature is not yet fully understood. This model can be useful to fulfill these issues. The entire dataset is composed by 994 DICOM images with sizes 992×1013 and resolution 0.004682^3 mm. This dataset has been already used in [1], obtaining a model with size $370 \times 228 \times 237$. In Figure 13.3 we can see an image taken from the original dataset.

FIGURE 13.3: The original liver scan

In [1], the computation was not parallelized so it was not possible to create huge models. With

our software, now we can create a greater model with sizes $992 \times 1012 \times 100$, with a boundary matrix of sizes $16 \times 4 \times 10$. The first thing we have to do is to apply a threshold on data. After a few tries, we have chosen the value 32250 which best suits to our input.

FIGURE 13.4: Liver scan with threshold 32250

As we can see in Figure 13.4, in these pictures there is a lot of noise so we have to reduce it using the median filter with window size 6.

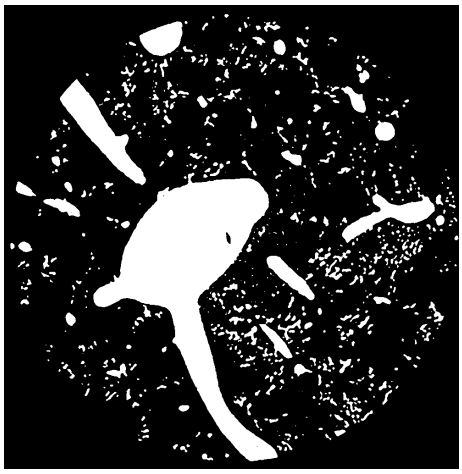


FIGURE 13.5: Liver scan after median filter application

Now we can see the result of the computation. In Figure 13.6 there are some screenshots taken from the model. The conversion took about 9 hours, using a cluster with 24 processes. At this point, we can observe the time used by each conversion step to obtain this final result:

- 31 minutes for the blocks creation
- 1 hour and 50 minutes for the boundaries merge

- 1 hour and 30 minutes for the blocks merge
- 3 hours for the smoothing
- 2 hours and 30 minutes for the creation of the output model

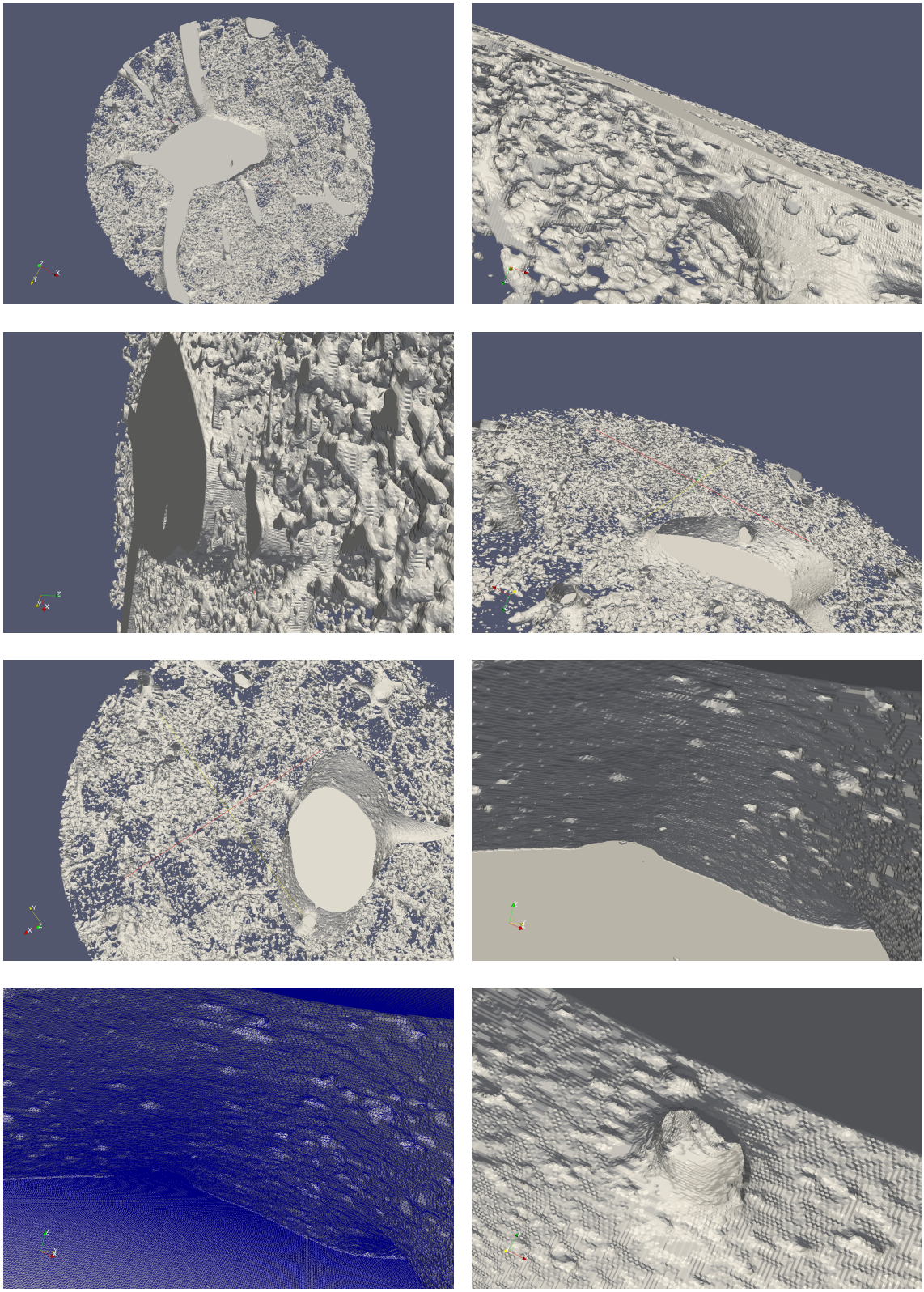


FIGURE 13.6: The three-dimensional model of a liver. Note that this model is topologically correct, in fact we can see that the hepatic portal vein is perfectly empty. (a) A view of the entire model. (b) and (c) Some close-up views of the model. (d) and (e) A detailed view of a section of the model (note that the veins are empty). (d),(e) and (f) The interior of the hepatic portal vein (you can see the triangulated structures)

Chapter 14

Conclusions

How we have read in previous Chapters, it has been possible to create a software for the extraction of three-dimensional models from medical images. In particular, we have seen how the combination between the topological approach and the High Performance Computing it has made possible the definition of extremely detailed three-dimensional models (as in Chapters 11, 12 and 13). Moreover, we have seen hot this reconstruction is topologically correct, returning us the true shape of the object instead of its simplified representation. This advantage become evident in Chapter 12, where we can compare the models obtained from marching cubes algorithm with the models obtained with the extraction of the boundary chain. In the first case it is not possible to see the interior of a single neuronal vein because the topology is not correct, in the other one it is possible.

However there are some open questions. In fact, until now, we have studied how to create in a efficient manner these models but we have not said nothing about visualization. In fact, in the previous examples the sizes of the final obj files were quite big, causing several problems. We need computer with a lot of memory space and the visualization is still slow. A solution can be found using new streaming algorithms which can adapt the output to the computer processing capabilities and to the detail level which is requested at that time.

Finally, although the results are quite appreciable (especially for the memory occupation), execution times could be further improved. In particular, the smoothing algorithm is quite slow. From an in-depth analysis, this fact depends on the search for adjacent vertices, which is done in linear time but is affected by the huge volume of data. A solution to this problem, is given from the adoption of an optimized pipeline conversion which can use less data at one time.

Bibliography

[1] Alberto Paoluzzi and Antonio DiCarlo and Francesco Furiani and Miroslav Jirik. "CAD models from medical images using LAR". In: vol. 13. To appear. 2015.

[2] P. Alliez et al. "Voronoi-based Variational Reconstruction of Unoriented Point Sets". In: *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*. SGP '07. Barcelona, Spain: Eurographics Association, 2007, pp. 39–48. ISBN: 978-3-905673-46-3. URL: http://dl.acm.org/citation.cfm?id=1281991.1281997.

[3] Bruce G. Baumgart. *Winged Edge Polyhedron Representation.* Tech. rep. Stanford, CA, USA, 1972.

[4] Birkfellner W. *Applied Medical Image Processing, Second Edition: A Basic Course*. Taylor & Francis, 2014. ISBN: 9781466555570. URL: https://books.google.it/books?id=9J7AAgAAQBAJ.

[5] P. Bourke. "Polygonising A Scalar Field". May 1994. URL: http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise.

[6] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. "Directed edges—A scalable representation for triangle meshes". In: *Journal of Graphics tools* 3.4 (1998), pp. 1–11.

[7] Michael Cox and David Ellsworth. "Managing big data for scientific visualization". In: *ACM Siggraph*. Vol. 97. 1997, p. 21.

[8] Brian Curless and Marc Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237269. URL: http://doi.acm.org/10.1145/237170.237269.

[9] CVD-Lab. *Linear Algebraic Representation*. Tech. rep. 13-00. Roma Tre University, 2013.

[10] David Lazer et al. "The Parable of Google Flu: Traps in Big Data Analysis". In: *Science 14* 343.6176 (Mar. 2014). DOI:10.1126/science.1248506, pp. 1203–1205. DOI: 10.1126/science.1248506.

[11] Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. "Linear algebraic representation for topological structures". In: *Computer-Aided Design* 46 (2014). 2013 {SIAM} Conference on Geometric and Physical Modeling, pp. 269–274. ISSN: 0010-4485. DOI: 10.1016/j.cad.2013.08.044. URL: http://www.sciencedirect.com/science/article/pii/S001044851300184X.

[12] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2014.

[13] Friendly Michael. *Milestones in the history of thematic cartography, statistical graphics, and data visualization*. 2009.

[14] Leonidas Guibas and Jorge Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi". In: *ACM Trans. Graph.* 4.2 (Apr. 1985), pp. 74–123. ISSN: 0730-0301. DOI: 10.1145/282918.282923. URL: http://doi.acm.org/10.1145/282918.282923.

[15] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002. ISBN: 9780521795401. URL: https://books.google.it/books?id=BjKs86kosqgC.

[16] Himi T. et al. "Three-Dimensional Imaging of the Temporal Bone Using a Helical CT Scan and Its Application in Patients with Cochlear Implantation". In: *ORL* 58.6 (1996), pp. 298–300. ISSN: 0301-1569. DOI: 10.1159/000276857.

[17] John F Hughes et al. *Computer graphics: principles and practice*. Pearson Education, 2013.

[18] Koch Kristin et al. "How Much the Eye Tells the Brain". In: *Current Biology* 16.14 (Jan. 2016). doi: 10.1016/j.cub.2006.05.056, pp. 1428–1434. ISSN: 0960-9822. DOI: 10.1016/j.cub.2006.05.056.

[19] Czes Kosniowski. *A First Course in Algebraic Topology*. Cambridge Books Online. Cambridge University Press, 1980. ISBN: 9780511569296. URL: http://dx.doi.org/10.1017/CBO9780511569296.

[20] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: http://doi.acm.org/10.1145/37402.37422.

[21] Norm Matloff. "Programming on parallel machines". In: *University of California, Davis* (2011).

[22] D.E. Muller and F.P. Preparata. "Finding the intersection of two convex polyhedra". In: *Theoretical Computer Science* 7.2 (1978), pp. 217–236. ISSN: 0304-3975. DOI: 10.1016/0304-3975(78)90051-8. URL: http://www.sciencedirect.com/science/article/pii/0304397578900518.

[23] Munzner Tamara. "Process and Pitfalls in Writing Information Visualization Research Papers". In: ed. by Kerren Andreas et al. Vol. 4950. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 134–153. ISBN: 978-3-540-70955-8. DOI: 10.1007/978-3-540-70956-5_6.

[24] M. R. Stytz, G. Frieder, and O. Frieder. "Three-dimensional Medical Imaging: Algorithms and Computer Systems". In: *ACM Comput. Surv.* 23.4 (Dec. 1991), pp. 421–499. ISSN: 0360-0300. DOI: 10.1145/125137.125155. URL: http://doi.acm.org/10.1145/125137.125155.

[25] *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. URL: http://www.emc.com/leadership/digital-universe/2014iview/index.htm.

[26] Tufte E.R. *The visual display of quantitative information*. The Visual Display of Quantitative Information v. 914. Graphics Press, 1983. URL: https://books.google.it/books?id=tWpHAAAAMAAJ.

[27] *World population counter*. URL: http://www.worldometers.info/world-population.

[28] David A. Zopf et al. "Bioresorbable Airway Splint Created with a Three-Dimensional Printer". In: *New England Journal of Medicine* 368.21 (2013). PMID: 23697530, pp. 2043–2045. DOI: 10.1056/NEJMc1206319. eprint: http://dx.doi.org/10.1056/NEJMc1206319. URL: http://dx.doi.org/10.1056/NEJMc1206319.