POLITECNICO DI MILANO
Facoltà di Ingegneria Industriale
Corso di laurea in Ingegneria Aeronautica

VMTKGui: an innovative tool for anatomical three-dimensional reconstruction.

Relatore: Prof. Alfio Quarteroni . . . .
Correlatore: Dr. Simone Deparis . . . .

Tesi di laurea di:
Samuele Zampini 720564

Anno accademico 2010/2011

..alla Carletta..

Sommario

Lo scopo di questa tesi è quello di presentare lo sviluppo dell'interfaccia grafica *VMTKGui* da affiancare ad un software pre-esistente, the Vascular Modeling Toolkit (*VMTK*), per poter creare un tool in grado di processare un'immagine medica di tipo *DICOM* (DIgital COmmunication in Medicine) in modo da passare da essa a una ricostruzione tridimensionale della stessa che permetta di effettuare simulazioni numeriche su quello che è il flusso nei vasi, come è spiegato dettagliatamente in [5]. A questo scopo si sfruttano alcune funzioni offerte da un software opensource per l'analisi e visualizzazione di immagini (Visualization ToolKit, *VTK*). Lo scopo del tool che si intende realizzare è quello di poter ricostruire la geometria di una parte anatomica non banale (nello specifico il fine ultimo è quello di ricostruire la zona a cavallo della valvola mitrale, vale a dire ventricolo sinistro, valvola mitrale stessa e ultimo tratto di aorta ascendente) per poter successivamente fare un'analisi numerica del flusso in quell'area. Per la realizzazione dell'interfaccia grafica si utilizza il linguaggio *Python*: si è preferito andare in questa direzione dal momento che la totalità delle librerie *VMTK* ha un'interfaccia evoluta proprio in questo linguaggio.

Il presente lavoro è così suddiviso: nel primo capitolo viene presentato il progetto nelle sue linee generali. Viene dunque dedicato ampio spazio a quello che è lo stato dell'arte di *VMTK* per quel che concerne la visualizzazione di immagini in campo medico; di seguito sono presentati gli obbiettivi che ci si è posti all'inizio del lavoro; nell'ultima sezione, invece, si dà ampio spazio alle applicazioni pratiche che potrebbero prendere piede a partire dal tool sviluppato per poi sottolineare l'importanza del lavoro svolto.

Nel secondo capitolo si tratta della ricostruzione geometrica da un punto di vista teorico: si presentano, nella prima sezione, le possibili tecniche per ricostruire una certa geometria partendo da un'immagine in formato *DICOM*; nella seconda sezione sono presentati i dettagli del metodo di ricostruzione utilizzato in questo lavoro di tesi, vale a dire il Level Set Segmentation (*LSS*), individuando i vantaggi che questa tecnica presenta rispetto alle altre, mettendo in evidenza quali sono i motivi che ci hanno portato a una tale scelta.

Nel terzo capitolo, invece, si è redatto un manuale per l'utente del software *VMTKGui*. Si presentano i vari passi per arrivare ad una ricostuzione della geometria desiderata su cui verrà poi eventualmente costruita una mesh.
Per ogni step sono presentate le varie problematiche, il metodo di utilizzo del comando e, infine, l'applicazione dello stesso in un esempio pratico.

Nel quarto ed ultimo capitolo c'è una prima sezione in cui si discute un'attenta analisi di quelli che sono stati i risultati ottenuti in rapporto agli obbiettivi che ci si era posti in partenza. Inoltre, nella sezione finale si offre una discussione relativa a possibili sviluppi futuri, al fine di migliorare e ultimare il tool realizzato rendendolo disponibile a tut-

ti gli utilizzatori del software *VMTK*, sicuri del fatto che un tool con queste potenzialità possa suscitare interesse nell'ambito di simulazioni in campo medico.

Abstract

The main target of this thesis is to describe the development of the graphic interface, *VMTKGui*. This tool can be used with **V**ascular **M**odeling **T**ool**K**it (*VMTK*), an already-existing software used to precess medical images, in particular vascular vessels, as the name suggests in order to reconstruct a certain geometry. This graphic interface has the goal to partly automate the reconstruction of vascular geometries, starting from a medical *CT*-series in *DICOM* format. To do this, we use some libraries offered from another software, *VTK* which is an open-source, freely available software system for 3D computer graphics, image processing and visualization. The tool that we realize should be able to reconstruct the geometry we want (in this specific case we should reconstruct the left ventricle, the mitral valve and the final part of the ascending aorta) in order to build a three-dimensional mesh needed for numerical simulations of the flow in that area. The *GUI* has been coded in *Python*, for which *VMTK* provides a clear interface.

This work is divided into 4 chapters: in the first chapter an overview of the project is presented. The state of the art of medical images visualization is described at the very beginning. Furthermore, the targets of the project are pointed out; also, in the final section, some practical applications are discussed and the importance of the *VMTKGui* is clarified.

In the second chapter we give the theoretical details about the technique we decided to use, i.e. the Level Set Segmentation (*LSS*) technique. Thanks to this technique it is possible to reconstruct a 3D geometry that can be meshed and then used for simulations. In the first section of this chapter we present different algorithms for geometrical reconstruction, while in the second one we give the details about the *LSS* and its implementation and we explain why such a choice.

The third chapter contains the *VMTKGui* user's guide: all the steps are detailed. And, for each step, we list the problem, we give some hints about the command and its usage and, finally, we present a practical example (in this case we reconstruct the descending aorta).

Finally, in the fourth chapter, we analyze the results and we compare them with the goals we had at the beginning of this project. We also discuss possible future development in order to build a better tool and in order to make the tool available to the *VMTK*community. In fact, we are sure it could be interesting to have such a tool that anyone can use and develop: it could be a way to make *VMTK* more user-friendly. And it will inspire huge interest in the medical environment.

Ringraziamenti

> Quando ci si sente obbligati ad esprimere la propria gratitudine, si perde la metà della sua gioia.
>
> René Barjavel

Per prima cosa ci tengo a sottoloneare che la frase di R. Barjavel mi trova completamente d'accordo. È per questo che vi voglio svelare "un segreto": nello scrivere questa sezione ho il sorriso sulle labbra, proprio perché il mio vuole essere un grazie sincero e spontaneo.

I primi ringraziamenti, sono i più formali: ma anch'essi sinceri. Vorrei ringraziare chi mi ha aiutato in questi fantastici anni di università, dal primo esame fino alla stesura di questa tesi.

Il primo grazie va al professor Quarteroni. Sempre disponibile. Un riferimento per portare a compimento questo lavoro. La sua grande passione e il suo entusiasmo contagioso mi hanno permesso di crescere sia come uomo che come studente. Nonostante i suoi mille impegni non mi ha mai negato 5 minuti per un consiglio. Un grande professore e, prima ancora, una grande persona.

Poi, un ringraziamento speciale a Simone, il mio tutor. Mi ha seguito da vicino. Mi ha dato molti consigli, sempre con una precisione ed una puntualità "svizzere"; anche lui sempre impegnato, mi è stato accanto con la sua presenza costante. Grazie Simo!

Un grazie immenso - tanto sentito quanto doveroso - va' a Luca Antica, uno degli sviluppatori di *VMTK*. Con la sua pazienza e la sua esperienza mi ha aiutato a risolvere i problemi di carattere computazionale. Epici i nostri scambi di email nelle ore più improbabili. Paziente e competente.

Ancora, ringrazio **tutti** i professori che ho incontrato in questi cinque anni, nessuno escluso. Grazie per quello che mi avete insegnato. E grazie per come me lo avete insegnato. Ci sono stati - come è inevitabile -

momenti belli e momenti meno belli. La mia speranza è che tutti voi abbiate sempre agito in buona fede.

E tra tutti i professori, lasciate che dedichi un grazie particolare a Maurizio, per tutto. Un amico prima ancora che un prof.

E poi, come dimenticarmi dei miei compagni di università? È grazie a loro se questi anni sono stati anni magnifici. I compagni storici: Andre, Daniele, Fabi1, Sandro, Bob e Tia. Gli aerodinamici: Andre, Paolo e Tia. E i "Pingus-guys": Mehl, Luca and Mauri.

Dopo aver terminato con i ringraziamenti "professionali", vorrei dire un grazie a tutti coloro i quali **"rendono la mia vita bella da morire"**. Queste persone, nel mio caso, sono davvero tante: non vorrei dimenticare nessuno e, soprattutto, non vorrei cadere nella retorica.

Il primo grazie va alla Carletta. È lei che mi sopporta dal 15 Febbraio '85. Non saprei che dire se non che "è la mamma che ogni figlio vorrebbe". E un grazie speciale al Peter: da lassù fa sempre il tifo per me.

E un grazie speciale alla Cri: ci siamo presi per mano sette mesi fa. Con un unico obiettivo: non lasciarci mai. È con lei che ho scelto di dividere la mia vita. So che insieme andremo lontano, mano nella mano! Grazie Cri <3.

Poi un grande grazie al resto della mia grande famiglia: i fratelli (Ago, Teo e Pini), le sorelle (Arde, Simo e Paola), i cognati (Casti, Su e Sergio), la cognata (Laura) e i nipoti (Ste, Ele, Matti, Sofi, Lori, Sveva, Marc1, Saretta e Matilde + 2). Che mi supportano (e sopportano!) sempre. E che sopportano la mia (in)sana follia.

Ci sono poi gli amici, anzi, gli Amici: il primo grazie va a Gio e Tia. Sono loro gli Amici di una vita. Gli Amici storici, indivisibili: sono quelli che ci sono sempre. Ho conosciuto Gio quando avevamo 2 anni: nel 1987. È il mio settimo fratello. Da allora non ci siamo mai persi di vista.
E poi Tia, anch'egli Amico di una vita. La nostra amicizia è diventata ancora più forte dal giorno in cui ci siamo persi nel bosco. Avventuroso.

Poi un grazie a Carolina: tantissimi i momenti passati con lei. Dai bellissimi compleanni nella sua taverna, alle pizzate in "Freccia". E come dimenticare le sue chiamate disperate quando non riusciva a far funzionare la calcolatrice, il giorno prima dell'esame di matematica? Mitica.

E, a ruota, i "ragazzi della *Quinta C*". Siamo stati una classe di Amici prima ancora che di compagni di liceo. A tutti loro devo un grazie di cuore. La mia classe. Voglio nominarli, ad uno ad uno, in ordine alfabetico: Tia, Biso, Zambro, Chicca, Lucio, Santi, Gira, Giò, Mille, Marta, Sandro, Francis e Scandro. Memorabile la gita a Monaco con loro.

Un altro grazie alla Ce, un'artista: sempre presente, anche alla discussione della laurea triennale. Non ci sentiamo spesso: ma so che la Ce c'è (e scusate il gioco di parole)! Sempre e comunque. Thanks.

Vorrei ringraziare anche C-Mari, la new entry. In pochissimo tempo è diventata la mia "psicologa" di fiducia. Sempre pronta a dispensare

consigli: troppo buona!

Sono proprio queste le persone che mi hanno aiutato e mi aiuteranno in tutte le scelte più difficili. Sempre con un unico obiettivo: **to be happy**! È tutto. Spero davvero di non aver dimenticato nessuno: metto comunque le mani avanti dicendo un **“Grazie a tutti!”**. Grazie a tutti quelli che rendono la mia vita così meravigliosa. Grazie a chi mi vuole bene.

Acknowledgments

He who receives a good turn, should never forget it: he who does one, should never remember it.

Pierre Charron

First of all, let me say a more formal thanks to all the people who helped me in these fantastic years of university, from the first exam, till the redaction of this thesis.

The first thank goes to Professor Quarteroni: his availability and his passion helped me to write this thesis and to grow as a man and as an engineer. Despite his myriad of commitments, he has never denied me 5 minutes for advice. A great professor and, before that, a great person. An example.

Then a special thank to Simone, my tutor, who followed me closely. How can I forget his patient and his generosity? Also, he always gave me good advice. With a “Swiss” precision and punctuality: tough always busy, he helped me a lot with his constant presence. His help has been great for me. Thank you, Simo!

Moreover, thanks to Luca Antiga, a developer of *VMTK*: with his patience and his experience he has always helped me to solve problems - every time: during the day or in the middle of the night! Countless the e-mails that I sent him. Useful.

Again thanks to all the professors I met in these 5 years, no exceptions! Thanks for what you taught me, thanks for how you taught me and thank you because, after all, it is you and your help that allows me to graduate, today. There were good times and less beautiful moments, but that’s life: I hope (and I am sure about that!) that you always act in good faith.

And a special thanks to Maurizio: for everything. A friend before a

professor.

Furthermore, a thank to my uni-mates who have made these years, great years! The names are too many to do. I try to “group” them. The historical uni-friends: Andre, Remu, Fabi1, Sandro, Bob and Tia. The “aerodynamics”: Andre, Paolo, Tia and Luca. And the Pingus-guys: Mehl, Luca and Mauri. Brave!

That’s all for the “professinal” acknowledgments. Let me now say a more informal “thank” to “my world” outside university. All the people I am going to mention below help me - in everyday life - to be a better person.

The first thank, the biggest, is for *Carletta*, my mom. I say just one thing: she is the mom that every child would like to have. And a special thanks to Peter: from “somewhere over the rainbow” he always looks after his “baby”.

And a big thank, the biggest, to Cri. We have been walking together for seven months. And we do have a goal: to never leave each other. We decided to share our lives. We will go further and further. Together. Thanks Cri <3.

Also, a big thank to my big family: my brothers (Ago, Teo and Pini), my sisters (Arde, Simo e Paola), my brothers-in-law (Casti, Su e Sergio), my sister-in-law (Laura), the nephew (Ste, Matti, Lori, Marc1 - read MarcOne) and the nieces (Ele, Sofi, Sveva, Sara and Matilde). They always support (and bear!) me and my (in)sane madness.

Now my friends. Better, my Friends: the first thank goes to Gio and Tia. They are the whole-life-Friends. Gio is my seventh “brother”. We met in 1987 and there is nothing that can divide us.
Tia, the other old Friend: countless the adventures we lived together, everywhere in the world.

Furthemore, a big thank to Carolina: she is a great Friend. And I remember all the moments we shared. Our Pizza parties. And her phone-calls when the calculator did not work the day before the maths-exams. Unique.

Also, a special thank the the guys of the *Fifht C*: we were 14 Friends before being 14 classmates. Thanks Tia, Biso, Zambro, Chicca, Lucio, Santi, Gira, Giò, Mille, Marta, Sandro, Francis e Scandro. Incredible our trip to Munich :D..

Also, let me say thanks to my friend Ce: even tough we meet hardly ever, I know that Ce is there, ready to listen to me. Thanks.

One more thank to C-Mari, the new entry.. Now she is my favorite “psychologist”.. She is a very good person: often too good.

That’s all. I hope I have not forgotten anybody. Just in case, I finish this section with e **“Thank you”** to all people who care about me. And who makes my life a wonderful life!

Contents

Appendici

A Computational Fluid Dynamics

List of Figures

Prefazione

> Il vostro tempo è limitato, per cui non lo sprecate vivendo la vita di qualcun altro. Non fatevi intrappolare dai dogmi, che vuol dire vivere seguendo i risultati del pensiero di altre persone. Non lasciate che il rumore delle opinioni altrui offuschi la vostra voce interiore. E, cosa più importante di tutte, abbiate il coraggio di seguire il vostro cuore e la vostra intuizione. In qualche modo loro sanno che cosa volete realmente diventare. Tutto il resto è secondario.
>
> Steve Jobs

Implementare il codice che mi ha portato alla realizzazione dell'interfaccia grafica *VMTKGui* e, in seconda battuta, alla stesura della presente tesi, è stato per me motivo di orgoglio. Ritengo opportuno riassumere ciò che mi ha spinto ad una tale scelta, dal momento che l'argomento centrale di questa tesi è lontano **anni luce** dal campo aeronautico che, invece, è l'argomento centrale del corso di studi che - con oggi - finalmente porto a termine. Andrò quindi a discernere ed analizzare quelle che sono le obiezioni che è possibile muovere, spiegando i motivi che mi hanno spinto, nonostante ciò, a svolgere un lavoro di tesi di questo tipo.

In primo luogo, la mia scelta è stata dettata dalla passione per la numerica: tra tutti i corsi seguiti, quelli che hanno maggiormente destato il mio interesse sono stati i corsi a carattere computazionale. Ecco perché ho pensato di rivolgermi al Dipartimento di Matematica del Politecnico di Milano e, nello specifico, al *MOX*.

Si potrebbe obiettare che - pur restando in ambito numerico - avrei potuto scegliere un argomento prettamente aeronautico: vero, ma alla base della mia scelta c'era anche un forte desiderio di fare un'esperienza all'estero. E, dal momento che mi si è presentata questa occasione, non ho voluto lasciarmela sfuggire. Sono consapevole del fatto che questa non era l'unica scelta possibile, ma è stata sicuramente un'ottima scelta. E sono orgoglioso di aver vissuto questa esperienza.

Inoltre - come mi è stato più volte ripetuto nel corso di questi 5 anni - l'ingegnere è una figura caratterizzata da una forte duttilità: è questo che mi ha spinto ad andare a esplorare altri campi, lontani da quello aeronautico. Sempre animato da una forte curiosità e dal desiderio di imparare cose nuove: mi ritengo (citando alcuni versi della canzone intitolata "Addio", di Francesco cuccini) "un eterno studente perché la materia di studio sarebbe infinita e, soprattutto, perché so di non sapere niente".

Da ultimo, ma non per importanza, vi è un motivo più personale, che non ha nulla a che vedere con la didattica: vale a dire la mia passione per la medicina. Oltre ad essere stata la facoltà che avrei scelto se non avessi optato per ingegneria, il mio amore per la medicina si manifesta anche nel desiderio di poter salvare una vita. È vero, l'ingegnere può fare aerei sempre più veloci che permettano ad un medico di essere a migliaia di chilometri di distanza in poche ore. Oppure può costruire un ponte o un tunnel per collegare 2 città altrimenti divise da una profonda vallata o da un monte invalicabile. Ma - alla fine - è sempre il chirurgo che deve agire. Ecco perché ho deciso di dedicarmi allo sviluppo di questo software, pur consapevole del fatto che non sono un chirurgo, né mai lo sarò. Mi accontento dell'idea che un giorno un chirurgo sfrutterà il software che ho contribuito a sviluppare: questo è per me motivo di immensa gioia.

Concludo dicendo che sarei dispiaciuto se leggendo questa prefazione resterete delusi, in quanto non avete tra le mani una tesi "aeronautica". Vi invito comunque ad andare oltre e a non fermarvi qui; spero che, nello scrivere questo lavoro, sia riuscito a trasmettere almeno una parte della passione che ho dedicato ad esso e una parte dell'entusiasmo che si leggeva nei miei sorrisi quando le cose andavano bene, ma anche quando qualcosa non funzionava perfettamente.
Rimango a disposizione per ogni tipo di chiarimento e per ogni curiosità in merito al lavoro svolto.

Buona lettura,

Samuele Zampini
samuele.zampini@gmail.com

Preface

> Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary.
>
> Steve Jobs

I am really proud to have written the code that allowed me to create the *VMTKGui* graphic interface and to have published this thesis. In this preface I would like to sum up all the reasons that drove me in such a choice. It is a subject that is very very far from the main subjects that characterized my studies. Why such a choice? I think this is the right place where I can give you the motivations. I know you could object that this job is not linked to what I studied. But I think that - as you can read in the epigraph - everyone should follow his/her heart and intuition. That is what I did.

A great motivation that suggested me to accept this thesis that professor Quarteroni offered me is my love for the numerical subjects. That is why I asked to the Maths Department for a thesis.

The first objection you could move is that I could have found an aeronautical subject for a numerical research. That is true. But one more reason is that my dream was to go abroad for the thesis. And as soon as I heard about the opportunity to go to Lausanne I said: “Yes, I am ready to leave.” and I left. This was not the only possible choice. Maybe there were many other choices. I am sure that my choice has been a great choice and I am proud of what I did.

Furthermore, in these years of university, I was often told that an engineer should be a very ductile figure. So I decided to explore fields far from “airplanes”, which are the main subject of an aeronautical faculty. In fact I am very curious and (quoting some verses of the song “Addio” by Francesco Guccini)I am “an eternal student, since the field of study would be endless and above all because I know to know nothing”[1].

Also, there is a very personal motivation that I want to share with you: I like medicine. Not only because I would have chosen Medicine if I had not chosen Engineering, but also because the idea to be able to help other people is something that really makes me happy. It is true that an engineer could build a faster airplane that allows the surgeon to go from a place to another in a blink or he could build a bridge to reach a city in a couple of hours instead of two days. Also, engineers could create a tunnel to link to towns. But it is the surgeon who has the power and the possibility to save a patient or not. He is the surgeon who can save a person. I am also aware that I am not a surgeon and I will never be a surgeon. However, it is enough to know that a person will be saved thanks to the software I have contributed to develop, to make me proud and happy of what I have done through this report.

Finally, let me tell you that I would be very sorry if, reading this preface, you will be disappointed since you are reading a non aeronautical report. In this case, I will invite you to go further in reading this thesis, hoping that reading it, you will feel the passion and the enthusiasm I tried to convey while I was writing it.
Obviously, feel free to contact me if you have any question or curiosity that deals with the developed work.

Enjoy your reading,

Samuele Zampini
samuele.zampini@gmail.com

---

[1] “Francesco Guccini, eterno studente, perché la materia di studio sarebbe infinita e soprattutto perché so di non sapere niente”

1. Introduction

> If you can't solve a problem, then there is an easier problem you can solve: find it.
>
> George Polya

The aim of this first chapter is to give to the reader a general overview of the work developed in this thesis, without any technical detail. Details that - as we can see - are analyzed in the next chapters (read chapters 2 and 3). This first chapter is divided in four sections: in section 1.1 we discuss about the state of the art of the visualization in the medical applications and we offer a description of the different techniques available to acquire data.

In section 1.2 we define the targets we want to reach with our tool. Here, the final goals are presented and we discuss about the targets of both the *VMTKGui* and the *VirtualValveStent* project. What it is important to remind is that the final target of the *VirtualValveStent* project is not a goal we want to reach through this report; nevertheless, we would like to pave the road for those who may be interested in developing such a tool.

Finally, in section 1.3, we discuss about the importance of the tool we develop in this thesis, describing its main features and its possible applications. The key point of this section is that the reader could appreciate the ductility of the tools developed in this report.

1.1 *Medical visualization: the state of the art*[1]

Angiographic image acquisition techniques (see [40] for further details) provide detailed anatomic data on vascular structures. Graphic workstations usually linked to Computed (Axial) Tomography (*C(A)T*) or Magnetic Resonance (*MR*) scanners are used in the clinical practice to produce tridimensional patient-specific representations on the basis of the acquired data. The techniques used today for such purposes are in general not adequate for accurate geometric analysis and Computational Fluid Dynamics (*CFD*) which we address here, since the main effort is directed toward producing high-quality visual feedback rather than accurate geometric modeling of a particular anatomic structure: physicians, in fact, prefer a better graphic visualization instead of an accurate geometry. In our case, vascular modeling requires, as a first step, the extraction of vascular wall position from medical images. In this very first section it is useful to give a brief overview of the principal imaging techniques used in the clinical practice for the acquisition of tridimensional anatomy of vascular segments. Since we use data coming from *CT* images, we give a detailed description of this technique in the first subsection. In the second and third ones we give a list of other widely used techniques with very short descriptions.

1.1.1 Computed Tomography

Computed Tomography (for further details see [43] and [6]) imaging is also known as *CAT*-scanning, where the “middle” A stands for *axial*. The word tomography is from the Greek words “tomos” meaning “slice” or “section” and “graphia” meaning “describing”: in fact, we reconstruct the volume we are interested in taking many “shots”, slice by slice.
CT-scanning is a quite recent technique, since it was invented in 1972 by Godfrey Hounsfield (see [37]) of EMI Laboratories, and by Allan Cormack (see [34]) of Tufts University. Hounsfield and Cormack were later awarded the *Nobel Peace Prize* for their contributions to medicine and science.
The first clinical *CT*-scanners were installed between 1974 and 1976. The original systems were dedicated to head imaging only, but “whole body” systems with larger patient openings became available in 1976. *CT*-scanning became widely available by about 1980. There are now about 30 000 *CT*-scanners installed worldwide.
The first *CT*-scanners developed by Hounsfield in his lab at EMI took several hours to acquire the raw data for a single scan or “slice” and took days to reconstruct a single image from this raw data. The latest multi-slice CT systems can collect up to 4 slices of data in about 350 ms and reconstruct a 512 x 512-matrix image from millions of data points in less than a second. An entire chest (forty 8mm slices) can be scanned

---

[1] Please, refer to [17], [15], [13], [14], [9], [8] e [**?**] for details about the state of the art in the clinical imaging.

in five to ten seconds using the most advanced multi-slice *CT*-scanning system.
During its 35-year history, *CT*-scanning has made great improvements in speed, patient comfort, and resolution. As *CT*-scanning times have gotten faster, more anatomy can be scanned in less time. Faster scanning helps to eliminate artifacts from patient motion such as breathing or peristalsis even tough we still have a certain noise in the reconstruction we do. But this noise could be attenuated as analyzed further, thanks to some post-processing tools. Here, we are referring to the "smoothing" tool that eliminates the noise introduced during the acquisition.

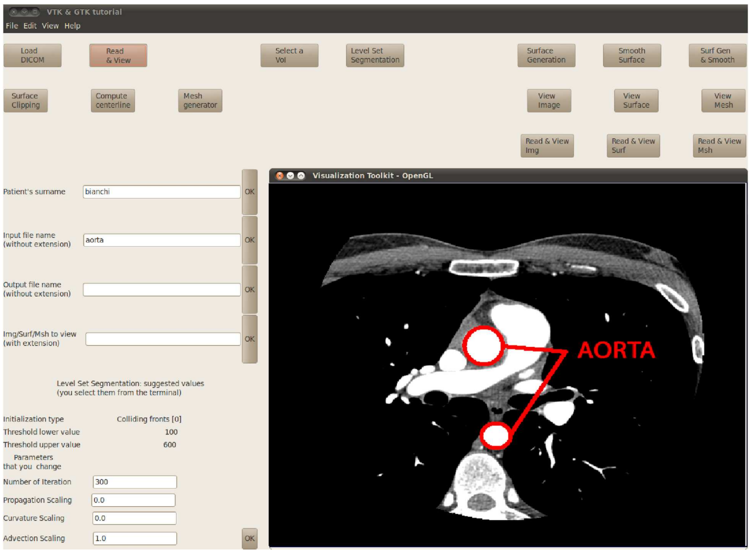
Tremendous research and development has been made to provide excellent image quality for diagnostic confidence at the lowest possible x-ray dose. Nowadays, in fact, it is possible to get very detailed images in a quite short time. Also, it is possible to have a colored reconstruction and this is certainly a great advantage both for the doctors and for the patients who would like to understand something more about their situation.

Why is *CT*-scanning becoming more and more popular? One of the great advantages is that it is a non-invasive technique: it is used widely both for the diagnose and also to simulate the surgery. In fact, thanks to *CT*-scanning, the doctor can both simulate a surgery (in order to analyze all the difficulties he may have to face) and make a clear and sure diagnose about some malformations or diseases, avoiding the biopsy, which is an invasive operation.
Furthermore, this kind of visualizations are very useful if you have to insert a stent (e.g. the *VirtualValveStent* project) in a patient's vessel: in fact, through visualizations, it is possible to reconstruct the geometry, to virtually build the stent and to virtually install it properly. And this helps the doctor and allows him to "study" in advance the surgery he is going to do. This means that - generally - the doctor has an idea about what he is going to find, making the operation be safer.
Also, in these days, imaging visualizations are used for a mathematical purpose, too: in fact, thanks to this visualization it is possible to obtain high-fidelity reconstruction of any anatomic area we are interested in. Once we get this reconstructions, it is possible to build a mesh either over the surface or in the volume and to analyze either the blood flow in the vessel or the air flow through a cavity, depending on what we need to study. That is another point that makes the visualizations very very important nowadays.

How does the *CT*-scanning technique work? To better understand this point, with animations, please visit [7]. This is a technique that consists of imaging cross-sections of a body using series of X-ray measurements taken at many different angles around it. The intensity of X-rays passing through the scanned body is attenuated according to the density of tissues that the rays encounters, so that the line integral of tissue density is measured. The source and the detector rotate around the subject and register a row of X-ray measurements for each angle during the rotation. In helical CT scanners, the subject is continuously moved through the plane of the rotating source and detector, so that an helical trajectory around the subject is generated. In multi-row helical CT scanners, the actual state of the art scanners, the signal is acquired simultaneously from up to four rows, thus optimizing the trade-off between acquisition time, z-resolution and noise.

The signal resulting from the acquisition is called sinogram, and is represented as a series of images with detected attenuation on the x-axis and rotation angle on y-axis. The image is then reconstructed by solving the inverse Radon transform (see [42]) on the image grid. The reconstructed image contains attenuation values expressed by Hounsfield units (HU), for which water is conventionally represented by 0. Consequently, fat is associated with negative values, connective tissue by low positive values and calcium by high positive values. Scanners today available in the clinical context allow in-plane resolution smaller than 0.5mm and slice thickness smaller than 1mm of anatomical structures in a single breath-hold.

In the image 1.1 we have an example of a thoracic CT. It is possible to see the different HU values that indicates the different tissues. We have pointed out the ascending and the descending aorta, which is the area that we analyze in this report.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 1.1. Example of a thoracic *CT*-scanning, got with the *VMTKGui* tool. It is possible to appreciate the different HU values depending on different kind of tissues.

1.1.2 Magnetic Resonance

There are two more important techniques that are used in the clinical practice; they are the magnetic resonance and the ultrasound imaging. In this subsection we give a brief description of the first one, i.e. the magnetic resonance, while the second one, i.e. the ultrasound imaging, is presented in the next subsection (see Subsection 1.1.3).

Magnetic resonance (see [39] for further details) is based on the measurement of relaxation times of the net magnetization vectors induced in tissues when a magnetic field at a given frequency is applied. Net magnetization occurs because the magnetic momentum of nuclei, whose Larmor frequency (see [38]) is that of the applied field, tends to orientate along the direction of the field. In medical imaging the Larmor frequency of hydrogen nuclei is usually employed. When the magnetic field is ceased, the magnetic momentum of hydrogen nuclei return to equilibrium by local field inhomogeneities and by interaction with other relaxing nuclei, and the net magnetization returns to 0. Both relaxation processes are exponential, and their time constants (named T1 and T2), which are characteristic of different tissues, can be measured. As a result, images can reflect proton density, T1, T2 or a mixture of these quantities. Localization of the voxels from the decaying magnetization signal (or Free Induction Decay, FID [36]) is accomplished by additional magnetic fields, acting as gradients along the directions if imaging axes. Signal is collected in lines of data in the frequency domain (the k-space), and is then brought to the image domain by Fourier transform techniques. Several sequences of application of magnetic fields and gradients are available, which offer great flexibility in imaging different anatomical structures.

It is important to underline that potentially harmful effects of magnetic fields have not been demonstrated, so that MR is considered a non-invasive investigation, as well. Recent developments of MR imaging techniques allow 1mm resolutions within a single breath hold: speed and resolution are lower than the CT ones, but they are good, too and this makes MR technique a very good and used technique. Also, there is an advantage of MR over CT: in fact, in the MR technique the imaging planes can be oriented by changing the direction of gradients, hence you could choose the orientation you like best.

1.1.3 Ultrasound imaging

Ultrasound imaging (see [41] for further details) is based on the generation of ultrasounds from a piezoelectric transducer which is then used as a receiver for the waves reflected at the interfaces between two tissues with different acoustic impedance. The brightness modulation imaging modality produces morphologic images which represent the echogenity of the tissues and of tissue interfaces. Recently, image processing techniques allow to combine acquired 2D images into volumes, yielding 3D ultrasound. Although image resolution still cannot compete with CT or MR angiography and the investigation is limited to superficial vessels, such acquisition method is attractive for the possibility of accurately discerning the morphology of the vessel wall, and for the relative inexpensiveness and flexibility of the technique.

A useful imaging modality is Doppler ultrasound, which accounts for the frequency shift of waves reflecting over flowing red blood cells. Thanks to this technique, velocity measurements can be performed real-time with high temporal resolution in a sample volume.

1.1.4 How to store data

A very good reference to refer to, in order to understand how the data acquisition works, is [1]. To sum up, let's say that for our purpose, what is important is to store the data we have acquired. And we have to do it in a smart way. The interesting point is that, independently from the image acquisition technique employed (either *CT*, MR or ultrasound), the acquired 3D images, usually in the form of stacks of 2D images, are stored on workstations linked to the scanners, and must be transferred to calculators for processing. This has to be accomplished without loss of information. Since medical images are usually represented by a number of gray levels greater than 256, 8-bit image formats, such as TIFF, handled by usual image-editing software are not suited for the representation of medical data. Hence, we need something different, also because due to the number of informations associated with acquisition, such as patient's and investigator's data, image number, image position, image resolution, acquisition time, acquisition modality and scan parameters, the need of a consistent way to handle such amounts of data has led to the definition of a standard for communication and storage of medical image data, the *DICOM* format. Images stored in the *DICOM* format contain a header which, in turn, encloses a number of tags organized into groups, followed by image data, which for *CT* and MR is represented in signed 16-bit label. By reading the tags and the image data it is possible to reconstruct the acquired volume from the image stack without the need of further information. For a detailed description of the *DICOM* standard see [23].

1.2 *Targets*

After the brief overview just presented, let's focus on our targets.

Through this thesis we want to write and test a procedure that - starting from a *CT*-scanning series - allows to faithfully reconstruct the geometry of a certain area. This target could be achieved using the existing software *VMTK*: the interesting point that makes the *VMTKGui*[2] tool innovative is that, thanks to it, we provide a user friendly graphical interface, that does not exist in *VMTK*. The latter, in fact, is handled by command line. Commands are often long and non-intuitive: that is why the physicians or the scientists may find it difficult to use *VMTK*. Also, this tool will be available for the *VMTK*-community, making it easier and faster to reconstruct a certain area, starting from a *CT*-scanning

---

[2] As the name suggests, this tool is developed starting from *VMTK*

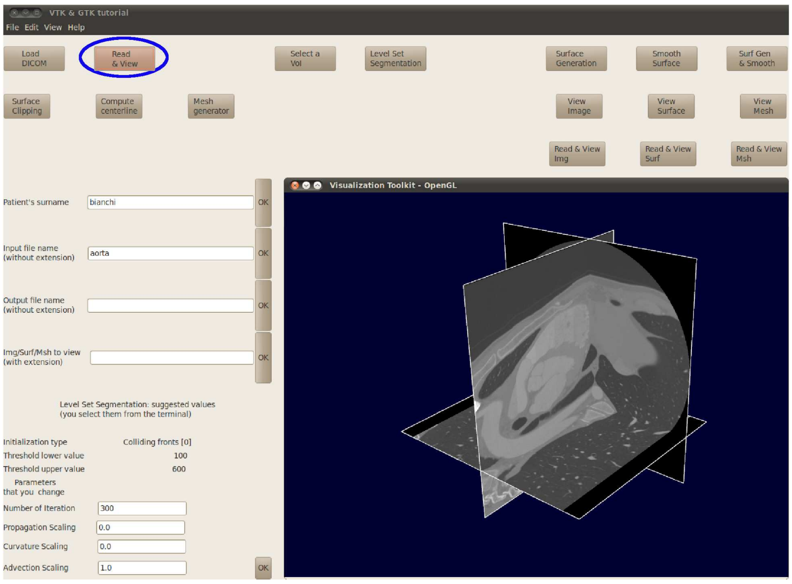
series, since “everything that deals with a *GUI* is welcome in our community” as a *VMTK*-developer stated.
To get our goal - as you can see reading the code in appendix B - we use *Python* language: this choice seems to be the right one, since this language is very intuitive and flexible and - above all - since *VMTK* has the proper interfaces to do this.

To sum up, the first step is to build a *GUI*: the intent of this work includes the identification of clear interfaces between *VMTK* and the user.
Once the data are loaded into *VMTK*, it is possible to act on it, in order to smooth it, since the surface reconstruction by *LSS* (i.e. the algorithm we use to reconstruct the geometry we need, see 2) is pretty noisy. Furthermore, in order to make it easier to reconstruct and to compute the centerlines, we use another *VMTK* function, the “clipping”, that allows to get a well-defined inlet and outlet surfaces (all this features are detailed further, in chapter 2).

Hence, to present *VMTKGui*, we say that the it is a smart and an innovative way to manage the most of the *VMTK* functions (the missing ones can be added) thanks to a user friendly graphic interface; we decided to start from this point since a *GUI* is the worst lack we noticed in the *VMTK* software.

Another important target that we reached through *VMTKGui* is the possibility to build a mesh readable from different finite element solvers. So that, such a tool could be used from the researchers in order to build meshes over a “real geometry” to do a patient *ad-hoc* numerical simulation. This makes *VMTKGui* tool a tool used in the pre-processing for a FE analysis.

In the Figure 1.2 it is possible to observe how the *GUI* is with all its features that will be discussed and detailed in chapter 3.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 1.2. An overview of the *GUI* where it is possible to appreciate all the features available through *VMTKGui* tool.

Let's now discuss about the long term goal, named *VirtualValveStent*, that is a somewhat ambitious project. This is the reason for which we decided to go ahead step by step, decomposing the original project into three sub-targets. Even tough in this thesis we develop the *VMTKGui* software, we think it is better (for a better understanding of the main project and also to make it possible its future developments) to give a short description of the final target, too.
It is clear, once again, that *VirtualValveStent* is a great project, but it is very ambitious and too heavy to be developed in a master thesis.

After the creation of a *GUI* (thanks to the *VMTKGui*), there are two more steps:

- insertion of the stent above the reconstructed geometry;
- possibility - thanks to a joint control - to choose the best inserting-point for the stent.

As you can appreciate, *VirtualValveStent* (i.d. of the second and the third sub-target) is a very ambitious project and it is going to be both a medical and an engineering tool. In fact, to place the stent in the right position you have to put the center of the stent itself in the center of the aortic valve. Also, once found the exact position, we have to find the right orientation: this is possible once we compute the centerlines of the stent and of the aorta. In this case, we make them fit and what we get is the exact positioning of the stent.
Furthermore, as far as the third sub-target is concerned, we can find out the best inserting point for the stent. This means that the operation will be the least invasive possible. Also, thanks to this trick, the doctor can simulate many ways to insert the stent, choosing the best one. This possibility to make different simulations makes the software *VirtualValveStent* very interesting. Also, the doctor has to choose and to perform the simulation that offers the best results in terms of feasibility, safety and comfort (both for the patient and for him). One more step could be an automation of this process, thanks to a joint control that minimizes a certain figure of merit properly studied.

1.3 *The importance of VMTKGui and its possible applications*

On the one hand, the *VMTKGui* is only an intermediate step toward the *VirtualValveStent* and one could wonders if it is smart to develop and use such a tool. Of course, the answer is "Yes, it is!". Why can we assert this? The reply is pretty easy: in fact *VMTKGui* is a flexible tool that can be used in many applications. One of its best feature is the possibility to get a detailed and *high-fidelity* geometrical reconstruction: thanks to this tool, in fact, we can get a surface in the tridimensional space. And this surface can be used as the starting point by people interested in generating a mesh over a certain surface or in a certain volume, starting from a *CT*-series. This is why - as it is explained in the chapter 4, section 4.2 - we think that the software has to be developed and specialized in order to make it easier and faster to load a *DICOM*-series, to extract the volume we are interested in and to generate the corresponding surface/volume mesh. After we get a good mesh - and this is the key point - we can start with a numerical simulation. In fact, nowadays, the computer power is growing faster and faster and the Computational Fluid Dynamics (*CFD*: for further details visit, for example, [35] and A) is becoming more and more important and used. In fact, *CFD* is spreading in many different fields, also in medicine, for example.

And the base for a good *CFD* simulation is to have a good mesh. That is why the possibility to generate a mesh makes *VMTKGui* a very ductile tool.

2. The Level Set Segmentation

> You do not really understand something unless you can explain it to your grandmother.
>
> Albert Einstein (1879-1955)

This is the theoretical chapter[1] of this report where a detailed description of the Level Set Segmentation (LSS) technique is given. In the first section of this chapter it is appropriate to introduce different techniques that could be used to extract a tridimensional reconstruction, starting from a *CT*-series. What we would like to achieve is to underline all the pros and the cons of each technique, in comparison with the LSS technique.

2.1 *Techniques for reconstruction of tridimensional models*

In this section we give a brief overview of the techniques that can be used in order to get a tridimensional reconstruction of a certain geometry starting from *DICOM* medical images. Medical images describe the region of interest by level of grey. Each level correspond to a different density, hence to a different tissue. Therefore, geometry reconstruction relies on identifying borders between different grey levels. There are at least three important techniques for geometry reconstruction:

- contouring;
- parametric deformable models (e.g. snakes and balloons);
- implicit models (e.g. *LSS*).

In this section let's examine the contouring and the parametric deformable model; the implicit models are discussed in Section 2.2. As far

---

[1] To edit this chapter we revise Antiga [1] and Sethian's [26] PhD thesis.

as contouring is concerned, the idea that stands behind this method is that - in a rough approximation - different tissues correspond to different grey levels in the CT-scanning. Hence, the easiest approach is to create a surface in correspondence of a certain grey-level. The advantage of this technique is that it can be performed on a single two-dimensional image constituting the three-dimensional volume or directly on the tridimensional image. The most popular contouring algorithm is the *marching cubes* algorithm[2]. This algorithm has been introduced by Lorensen et. al [20] in 1987 and, although quite old, this technique is still widely used; it is also used to validate newer and more sophisticate improvements [3] or alternatives [44].
Let's now discuss about how this algorithm works: we start from a three dimensions CT-series and we set a level of gray that allows us to extract the tissues we are interested in. Then we create a grid and it is possible to label image grid vertices, i.e. the voxels, as above or below the given level by reading their scalar value and comparing it with the chosen level. Let's now consider the set of one voxel-wide cubes, which is defined by eight neighboring voxels: hence, it is possible to select which are the cubes intersected by the isosurface of the value of interest as the ones in which some of the voxels are labeled above and the rest below. The idea at the base of this algorithm is that the above and below voxels of each cube can be partitioned by a set of triangles whose vertices lie on cube edges in a finite number of ways, called *cases*. That is why we can construct a table of these cases: this table should contain all topological configurations of above and below voxels and triangles partitioning them, regardless the exact position of triangle vertices along cube edges (as in Figure 2.1; it is important to underline that this picture only has a topological meaning. Triangle vertices must be considered in a generic position along cube edges and their coordinates are found by interpolation).

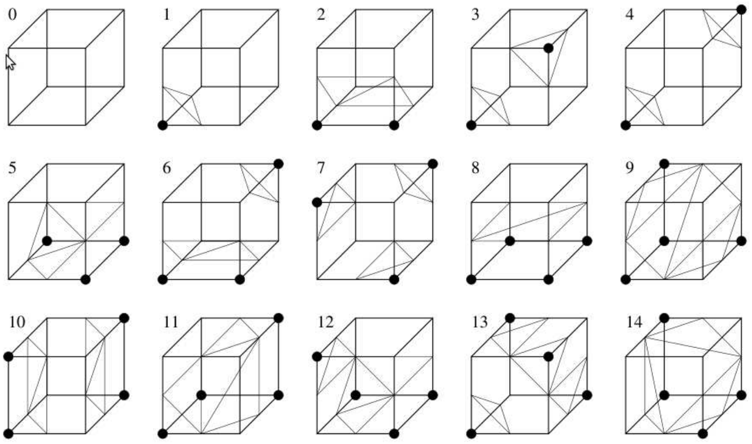


Figure 2.1. Marching cubes cases (courtesy of L. Antiga, see [1]).

---

[2] This is the technique implemented to initialize the geometry we are interested in.

A great advantage of this technique is that - in most cases - there is only one possible configuration of surface triangles: this means that the surface reconstruction can be performed independently for each cube constructed in the contoured image, retrieving the proper surface configuration from the case table. To find the exact position of the triangle vertices along the cube edges we use a linear interpolation of voxel scalar values on cube vertices. The surface produced by marching through the whole image volume is therefore first-order sub-voxel accurate.
Figure 2.1 shows the table of cases reduced to 15 by symmetries. Now a technical detail arises: in fact, for a small number of cases, namely 3, 6, 7, 10, 12 and 13 in the Figure 2.1 there is more than one possibility of constructing a surface partitioning above and below voxels. These are called complementary cases and they are represented in Figure 2.2.

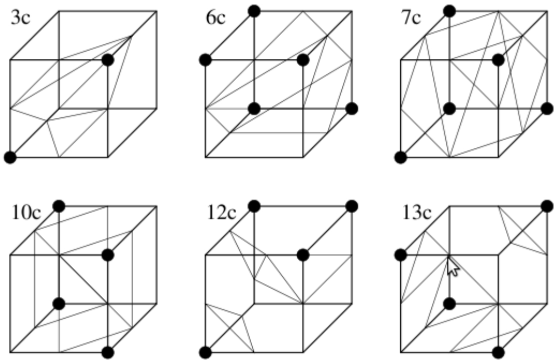


Figure 2.2. Marching cubes complementary cases (courtesy of L. Antiga, see [1]).

These complementary cases are generated from the base case by swapping above and below voxels. This possibility generates ambiguity in surface reconstruction, since an arbitrary choice among ambiguous cases can give rise to changes in surface topology (e.g. holes). The way to avoid this problem is to introduce a set of rules. From a theoretical point of view, the rules can be avoided if contouring problem statement, besides image and contouring level, also includes the definition of different connectivity neighborhoods for above and below voxels, so that above voxels that are considered connected would not necessarily considered connected if labeled below. This way above/below swapping does not produce indistinguishable configurations in terms of cube voxel partitioning, therefore resolving ambiguity. In practice, it is not always possible to choose the proper neighborhoods for the whole three dimensions image prior to constructing the isosurface, so the use of local rules is often preferred.
This technique presents some limitations. Nevertheless we often use it, since angiographic tridimensional images represent scalar fields sampled on regular grids and this marching cubes algorithm is a good way to reach our goal: in fact it is possible to apply it to construct the isosurface

Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 2.3. Change in surface position and topology due to a slight change in contouring level. These levels are A: 215HU; B: 235HU.

located over the transition from vessel lumen to the surrounding tissue. Thanks to its simplicity, this approach is currently implemented in software applications for radiological visualization. Though it is very useful for the purpose of visualization, it is not adequate if we are interested in reconstructing an accurate geometric for the *CFD* analysis. The major limitation is that resulting surface depends on the scalar value we choose for the contouring and this choice is made by the operator who, being a man, can make a mistake. In fact there is not any fixed scalar level for vessel lumen boundary, but this level depends on different conditions (it is different from one patient to another, it depends on the environment and also on the quality of the *CT*-scan itself). Moreover, interfaces are the areas where the gray-level variation is bigger: this means that a slight change in contouring level may produce great changes in geometric and even topological features of the resulting surface, as shown in Figure 2.3, where models are generated by different levels contouring. One more problem is that a single contouring value may be not suitable for an angiographic acquisition if the contrast medium is used. In fact, the blood flow makes the concentration of the contrast medium change. Hence, the opacization level change, as well, time after time.
The same problem can be experienced in time of flight[3] and phase contrast[4] MR angiographic acquisitions, due to complex blood flow patterns altering the received signal.

One more issue involving isosurface extraction is the impossibility of selectively reconstruct vessels of interest ignoring branching vessels, or to avoid the reconstruction of calcified plaques on CT images (see Figure (2.4)), since they have a higher gray value than the contrast medium, without manually editing the source images. This task can introduce operator dependency if the edited regions extend over several serial images

---

[3] Time of Flight is a modality for the MR acquisition where flowing blood is excited in one plane and its signal acquired in a downstream plane. This allows to enhance saturated blood over the unexcited static tissue without using contrast agents, with the drawbacks of low spatial resolutions, slow acquisition and artifacts from complex flow

[4] Phase Contrast technique relies on velocity induced phase shifts of the transverse magnetization. Since in the presence of complex flow patterns velocity measurements can be affected by artifacts, acquisition is usually performed in relatively straight vessels. This technique provides information about different velocity components.

and involve geometrically complex regions. That's why other techniques have been developed, even tough each of them presents its pros and its cons and it is impossible to say which is the best way to acquire images. That is why CT-scan is still widely used, nowadays.

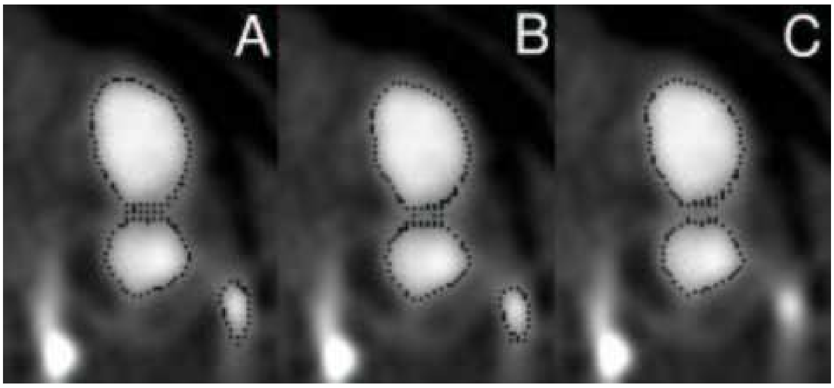


Figure 2.4. Problems linked to the contouring model: a manual editing is necessary.

Let's give now a brief description about the two most-widely used parametric deformable models. These models have been introduced since there is a lack of absolute correspondence between tissues and grey level of a CT-series. These models, in fact, are based on image features instead of absolute gray level ranges. The two most popular models are the *snakes* - for two-dimensional analysis - and its three-dimensional counterpart, i.e. the *balloons*. These models evolves in the image space and a Lagrangian approach is used: with the expression "Lagrangian approach" we mean that deformations are referred to the undeformed initial configuration and each material point is followed [45]. The next two subsections are dedicated to a description[5] of these two models.

2.1.1 Snakes

In this subsection we give an overview of the "snakes" model, which is developed in [45]. This technique may be used for two-dimension problems, only. A *snakes* is a parametrized curve evolving on the basis of image feature and intern constraints and not on absolute values. The key point is that - in a two dimensions setting - any closed curve evolve in a circle and finally collapses into one point since its curvature shrinks it. So, *snakes* are usually initialized as closed lines surrounding regions of interest. Then they move and shrink until the functional gets the desired result. In the blood vessel reconstruction, indeed, we can also initialize an "internal *snakes*" inside the lumen of the vessel itself. Then we inflate it until when the desired value (the HU value corresponding to the wall) is encountered. These active contours are often used for a three dimensions reconstruction - e.g. for vessels - even tough snakes

---

[5] We give a qualitative description. For technical details see [1].

are in two dimensions. And this is possible since we can reconstruct a stack of two dimensions images and then we put them together to build a tridimensional reconstruction. The snakes are widely used since they are very easy to be implemented. Of course they present some limitations: the shape retrieval is not simple. The image slice direction is not aligned with the vessel axis and it could be difficult to reconstruct accurately the vessel. Also, it is not easy to construct a surface from a set of two dimensions contours, especially if there is a bifurcation. In fact, through the interpolation, it may happen that the fidelity of the reconstruction is not high enough for a well-done *CFD* simulation. That is the reason that leads the *VMTK* developers toward a different choice.

2.1.2 Balloons

Balloons are the three dimensions counterparts of the *snakes*. In this case is not true that each surface evolves into a sphere before collapsing in a point. That is why - as far as balloons technique is concerned - we usually perform an “inverse” process: starting from a small surface, we inflate it until the functional reaches the set value. This is also commonly used for initialization since it is enough to give a point inside the vessel (let’s say its axis) and then we inflate it. The advantage is that - using balloons for the blood-vessels reconstruction (see [19]) - we can deal with the tridimensional geometry directly. This is a gain in speed and in operator independence. But there is also a drawback: the parametric nature of balloons may make it necessary to introduce some constraints on their evolution in order to avoid interpolation problem after a too large deformation. Also, there is a problem in the merging area between two balloons. In fact this area requires an *ad-hoc* parametrization. Details about this problem are discussed in [11] and [22]. Please refer to the mentioned articles.

Although these methods are widely used, we prefer - in accordance with the *VMTK* software - using the *LSS* technique that is described in the next section.

2.2 *Theoretical concepts behind the **LSS***

The *LSS* is an implicit technique and it is a good alternative to parametric deformable models presented in the previous section (e.g. *snakes* and *balloons*). These models are scalar functions defined in $\mathbb{R}^2$ or $\mathbb{R}^3$ whose isosurface of level k is the model of interest. *LSS* technique is introduced from an Eulerian point of view, while equations that rule the deformable models and written in the Lagrangian approach.
Let’s anticipate the content of this section: we discuss the evolution equations for implicit models, hence the very first step is to switch from the Lagrangian to the Eulerian point of view. This means that we have to give the deformations referring to the deformed configuration at the step before.

Also, we give some numerical and implementation details and in the final subsection an application of the *LSS* technique to vessels modeling is presented. One of the great advantage of this technique is that the equations we write are independent from the number of dimensions. That's why we present the three-dimensional equations only. Another reason is that the tridimensional model is the model implemented in *VMTK*.

Also, we suggest the reader to carefully read a very important section (i.e. Section 2.7) in Antiga's thesis [1] that report e validation for the *LSS* method. Validation that has been done using synthetic images of cylinders with different features (e.g. resolution, orientation and noise level). The function studied is the following:

$$F(r) = C\left(\frac{1}{2} - \frac{1}{1+e^{-a(r-R)}}\right)$$

where R is the radius of the cylinder, which is the distance between the symmetry axis and the zero level set of $F(r)$, and controls the steepness of the sigmoid at the inflection point (the gradient modulus at the inflection point is a), and C is a scale factor. For results and more details see the cited reference (i.e. Antiga's thesis [1]).

2.2.1 Evolution equation

The first thing to do is to introduce the evolution equation. Hence, the first step is to switch from the Lagrangian approach to the Eulerian one. We discuss about the three-dimensional problem, assuring that the extension to a different number of dimensions is easy and leads to the same equations. We start from the equation we got for the balloons (details are available in [1]), in the Lagrangian form:

$$\frac{\partial \mathbf{S}}{\partial t} = w_1 G(\mathbf{S})\mathbf{N} + w_2(\mathbf{S_{rr}} + \mathbf{S_{ss}}) - w_3 \nabla \mathbf{P}(\mathbf{S}) \tag{2.1}$$

where:

- t is time;
- $\mathbf{S}$ is the time-evolving surface, such that $\mathbf{S}(t) = \{\mathbf{x} | F(\mathbf{x}, t) = k\}$;
- $\mathbf{x}$ is a point in $\mathbb{R}^3$;
- $G(\mathbf{S})$ is scalar inflation speed;
- $\mathbf{N}$ is the outward surface normal;
- $(\mathbf{S_{rr}} + \mathbf{S}_{ss})$ is an average second-order smoothing term;
- $\mathbf{P}$ is the attraction term. It is a scalar potential function that takes into account image features (e.g. $\mathbf{P}(\mathbf{x}) = -|\nabla \mathbf{I}(\mathbf{x})|$);
- w_1, w_2 and w_3 are three coefficients that allow the user to optimize the function.

To switch from the Eulerian to the Lagrangian equations we have to remember that a surface evolving in time could be written as $\mathbf{S} : \mathbb{R}^2 \times \mathbb{R}^+ \to \mathbb{R}^3$. And it also can be represented as an isosurface of k-level of a scalar function: $F : \mathbb{R}^3 \times \mathbb{R}^+ \to \mathbb{R}$.
Being $\mathbf{S}$ the k-level set of F over time, we can write:

$$\frac{\partial F(\mathbf{S},t)}{\partial t} = -\nabla F(\mathbf{S},t) \cdot \frac{\partial \mathbf{S}}{\partial t} = -|\nabla F(\mathbf{S},t)| \cdot \frac{\partial \mathbf{S}}{\partial t} \cdot \mathbf{N}, \tag{2.2}$$

where $\mathbf{N} = \frac{\nabla F}{|\nabla F|}$ is the outward normal to level sets. It is important to underline that equation (2.2) is nothing but the implicit counterpart of the "balloons equations", written at the beginning of this section (see equation (2.1)). However this equation has the great advantage that the description of the embedded version of $\mathbf{S}(t)$ does not require a global parametrization, but relies only on local geometric properties of $F(\mathbf{x},t)$. We have to rewrite the equation (2.1) in the Eulerian form, starting from the Lagrangian one. Following [1] and [26], after some technical details we can write the localized level sets equation for $F(\mathbf{x},t)$, yielding

$$\frac{\partial F(\mathbf{x},t)}{\partial t} = -w_1 G(\mathbf{x})|\nabla F| + 2w_2 H(\mathbf{x})|\nabla F| + w_3 \nabla \mathbf{P} \cdot \nabla F, \tag{2.3}$$

which is the equation that represents a deformable surface (e.g. a balloon) embedded as a level set of a scalar field evolving in time. In this formulation there is a great advantage that is absent with a balloon model: there is no parametrization. Hence, our geometry could deform freely and we do not need to use any re-parametrization strategies or *ad-hoc* merging rules. Another advantage is that if we are in a three dimensions setting, equation (2.3) can be solved on the regular grid by one of the classical numerical methods as is detailed in subsection 2.2.2. To better understand the different terms of Equation 2.3, once again we suggest the reader to refer to Antiga's work [1] where the definitions introduced by Malladi [21] are explained.

2.2.2 Numerical approximation

To solve the problem we rearrange the equation (2.3), writing it in a slightly different way, namely

$$\frac{\partial F(\mathbf{x},t)}{\partial t} = -w_1 G(\mathbf{x})|\nabla F| + 2w_2 G(\mathbf{x}) H(\mathbf{x})|\nabla F| + w_3 \nabla P \cdot \nabla F, \tag{2.4}$$

where we weight the curvature term by function G, so that smoothing effect is stronger in regions of lower image gradient magnitude which are zones where less image features are present.
To solve equation (2.4) many methods are available. We can use, e.g., the finite difference method where the image domain is used as the structured grid for the problem discretization. It is possible to observe that equation (2.4) is an equation in the Hamilton-Jacobi class. The general form of this class of equations is:

$$\frac{\partial F(\mathbf{x},t)}{\partial t} + \mathcal{H}(\mathbf{x}, \nabla F(\mathbf{x},t)) = 0 \tag{2.5}$$

and its specialization for our problem becomes:

$$\frac{\partial F(\mathbf{x},t)}{\partial t} = G(\mathbf{x},t)|\nabla F(\mathbf{x},t)| \tag{2.6}$$

which is a class of nonlinear, hyperbolic PDEs. Let's apply a forward scheme for the numerical approximation.

It is known that hyperbolic equations have a particular behavior: information flows from the direction of front advancement. Hence, using a central two-sided finite differences could lead to instabilities. Instabilities that arise in the $\nabla F(\mathbf{x})$ approximation if we have a region where two level set fronts moving along incident directions merge: in this case we have an incorrect result despite arbitrary grid refinement, because information from each side of first-order discontinuity is averaged in calculating gradient values. This error can propagate to the neighboring points and instabilities may arise. To remedy to this inconvenience, Sethian et al. (see [26]) proposed to use an up-wind method, that is to say:

$$\begin{aligned}
F_x|_{\mathbf{x}_{ijk}} &\approx D^{+x}_{ijk} = \frac{F(i+1,j,k) - F(i,j,k)}{h}\\
F_x|_{\mathbf{x}_{ijk}} &\approx D^{-x}_{ijk} = \frac{F(i,j,k) - F(i-1,j,k)}{h}\\
F_y|_{\mathbf{x}_{ijk}} &\approx D^{+y}_{ijk} = \frac{F(i,j+1,k) - F(i,j,k)}{h}\\
F_y|_{\mathbf{x}_{ijk}} &\approx D^{-y}_{ijk} = \frac{F(i,j,k) - F(i,j-1,k)}{h}\\
F_z|_{\mathbf{x}_{ijk}} &\approx D^{+z}_{ijk} = \frac{F(i,j,k+1) - F(i,j,k)}{h}\\
F_z|_{\mathbf{x}_{ijk}} &\approx D^{-z}_{ijk} = \frac{F(i,j,k) - F(i,j,k-1)}{h}
\end{aligned} \tag{2.7}$$

Let's analyze the equations above: even though they are a first-order accurate equations, these expressions can approximate regions with a cusp-solution. Being a one-sided approximation, we need to introduce a numerical viscosity, which does not have any physical meaning, but that allows us to solve the problem. The solution converges to the exact solution for $h \to 0$, but it is a slow convergence (see [24]). Higher-order schemes can be used, with all the difficulties that they imply.

Using up-wind finite difference for the level set approximation, we get:

$$F(\mathbf{x},t+\Delta t) = F(\mathbf{x},t) - [max(G(\mathbf{x}),0)\nabla^{+} + min(G(\mathbf{x}),0)\nabla^{-}]\Delta t. \tag{2.8}$$

For the definition of ∇^{+} and ∇^{-}, please refer to [1].

As demonstrated in [25], this scheme yields stable viscosity solution to equation (2.6).

Another numerical issue is that, in equation (2.3) we have to deal with the second-order derivatives. These are the terms related to the level sets curvature $H(\mathbf{x})$ for the smoothing term. But second-order derivatives are naturally diffusive, that is why we can use - for these derivatives - the central finite differences.

There is one more numerical issue that has to be analyzed: the discretization in time. Discretization of Equation (2.4) leads to an equation which is not unconditionally stable, so that Δt must respect the Courant-Friedrich-Levy (CFL) condition, i.e. (see [24]) there must be a relationship between the space and the time spacing:

$$\max\left(\frac{\partial F}{\partial t}\right) \leq \frac{h}{\Delta t}$$

that in this specific case becomes:

$$\Delta t \leq \frac{h}{\max(G(\mathbf{x},t)|\nabla F(\mathbf{x},t)|)}$$

This is a very good method, but it inevitably presents its drawback: if on the one hand, embedding the parametric deformable surface into a scalar function allows to achieve topology independence and let us menage great deformations, on the other hand there is the needing of defining a function on the whole image volume instead of a parametric function. Hence, the complexity of the problem depends on three dimensions image size rather than on model surface size. Computationally, this is very expensive and it can be impossible to gain real time control on model evolution, tough computer performances are growing very fast.

One of the possible solution is the *Sparse Field Approach* (as demonstrated by Whitaker, see [33]): we get our objective by tracking the set of voxels, called active set, intersected by the level set of interest (usually the 0-level set) at each time step, as well as two layers of voxels around the active set, to compute the required derivatives.

Thanks to this idea, the problem is computationally much cheaper: in fact we have to update only the voxels in the active set, and the complexity depends, once again, on model size. One more advantage of the *Sparse Field Approach* is that 0-level set position can be estimated from the values of $F(\mathbf{x})$ and $\nabla F(\mathbf{x})$ of the voxels in the active set, so that all the image-based quantities can be computed with a good accuracy, i.e. sub-voxels accuracy.

To sum up the *Sparse Field Approach*, we introduce the following scheme:

i. initialize layers, by finding zero crossing;
ii. compute solution change for active layer voxels based on upwind finite differences and Newton's method for 0-level set location;
iii. Compute time increment based on CFL condition;
iv. Add (solution change*time increment) to active layer voxels;
v. update the active set;

vi. update adjacent layers starting from the inner ones reconstructing the signed distance transform;

vii. repeat operation ii) to vi) until RMS change is smaller than the tollerance;

viii. contour 0-level set;

ix. end.

2.2.3 Implementation

In this subsection we explain how the *LSS* solution is implemented in *VMTK*.

First of all, images were transferred from *CT*-scanners and read using a simple *DICOM* reader implemented by the *VMTK*-developers as a subclass of vtkImageDataReader, on the basis of the *DICOM* standard protocol.

The key point, as already stated, is the step that allows to reconstruct a tridimensional model, starting from the *CT*-series. The marching cubes algorithm, used for the inizialization, was implemented in C++ (and then translated in Python) using *VTK* 4.1, on the basis of what R. Whitaker implemented in VISPACK library. Also, *VTK* classes were mainly used to provide basic data structures (vtkStructuredGrid and vtkPolyData), input/output operations, image gradient computation (vtkImageGradient) and the contouring algorithm (vtkMarchingCubes).

The sparse field *LSS* algorithm was implemented as a *VTK* filter, derived from vtkStructuredPointsToStructuredPointsFilter class. A based *sparse field approach* level sets solver (vtkLevelSetsMachinery) class was constructed as a derived class of vtkStructuredPointsToStructuredPointsFilter class, which provided the sparse field level sets mechanism. The methods for the computation of image-based terms in level sets equation were kept virtual, and were implemented in a derived class (vtkLevelSets), inherited from vtkLevelSetsMachinery. Class vtkLevelSets takes as input an image, which is used for level sets initialization, and returns as output the same image at the end of the evolution. Level sets initialization can also be performed from the points of a vtkPolyData object, as in the case of initialization from centerlines. Two more images are used as input for the computation of the inflation (a scalar field, such as source image gradient magnitude) and the force (a vector field, such as source image gradient) terms. The evolution parameters and the number of iterations to perform with that parameters are also set. At the end of the specified number of iterations it is possible to change parameter values.

Single vessel evolution is performed instantiating multiple vtkLevelSets classes, and the merging step is handled outside the vtkLevelSets classes.

2.2.4 Application to blood vessel tridimensional modeling

As far as the application of this level set method to the blood vessel three dimensions modeling are concerned, it is important to underline that this technique is widely applied, e.g.: [2], [12], [29], [30] and [31] and references therein. This methods starts by performing the initialization inside the vessel we are interested in. Carefully looking at equation (2.4), we note that single points turn into spheres, due to the inflation term. These spheres could merge together and the surface during the evolution (and, also, at the end of the process) can be extracted by contouring the 0-level set of $F(\mathbf{x}, t)$ using a proper method.

About the inflation term, it is deactivated when the 0-level set gets very close to the wall of the vessel. In that region the attraction term is activated in order to ensure the convergence.

There are many methods (see [4], [27] and [28]) that have been proposed to automate *LSS* evolution, from the initialization till the convergence on the wall. Nevertheless, dealing with generic vessel segments, we think that it is better to perform a *LSS* evolution driven by an operator, in order to make it easier the control when acquisition artifacts affect the angiographic images. This is a good way to avoid the application of Gaussian-smoothing filters to the images for the calculation of image-based terms and, this way, we reduce the effects that those filters could have on reconstruction accuracy. Also, this method allows the operator to receive feedback on *LSS* evolution:

- plots of 0-level set over serial angiographic images or their gradient magnitude images;
- intermediate polygonal surfaces rendering;
- the maximum value of *LSS* speed.

This latter quantity is employed to automatically detect convergence in the last phase of *LSS* evolution, the one driven by attraction potential, after a convergence tolerance has been fixed.

This method presents a problem if we perform it over an entire vessel tract that can include branches and vessels of different scales: in this case level sets evolution can become difficult to tune, because vessel wall will be approached earlier in smaller vessels than in bigger ones, so that we can not use a single set of parameters for all scales. Using a single level sets evolution for an entire branching vessel tract presents another problem, linked to the CFL condition. Δt, in fact, may become very small because we encounter an image region of low gradient, and this slows down the evolution for the remaining portion of the model. But solutions to both the problems can be found, either by including locally adaptive w_1, w_2 and w_3 parameters in equation (2.4) or by adaptive grids in which CFL condition is satisfied acting on h, by locally modifying the image grid (i.e. a bigger "h" for a bigger vessel). But this approach is not easy to be implemented. Following what the *VMTK* developers did, we instead adopt a simpler approach, made possible by the particular nature of our problem: since evolution parameters are dependent on vessel scale, we let level sets evolve into single vessels, or into groups of vessels of similar scale. Thanks to the implicit nature of *LSS*, we later merge the $F_i(\mathbf{x})$ functions resulting from N single vessel evolutions, and finally extract model surface by contouring the merged $F_m(\mathbf{x})$ function with the *marching cubes* algorithm. Since in the *sparse field approach* the *LSS* represents the *signed distance function* from the 0-level set, with negative values inside the model and positive values on the rest of the domain, merging of N level sets scalar fields is performed selecting their minimum value

$$F_m(\mathbf{x}) = \min_{i \in [1,N]} F_i(\mathbf{x}) \tag{2.9}$$

The evolution of *LSS* into vessels of similar scale has two beneficial effects. The first is the increased ease of interactively setting evolution parameters (e.g. to switch from inflation to attraction to gradient magnitude ridges). The second is that similar solution changes are computed over the domain, so that more adequate time step values are chosen by the CFL condition, effectively speeding up evolution.

3. *VMTKGui* user's guide: from *DICOM* to mesh generation

> The theory is when you know everything and nothing works. The practice is when everything works and nobody knows why. We have put together theory and practice: there is nothing that works.. and nobody knows why!
>
> Albert Einstein

This chapter is the *VMTKGui user's guide*: this is the crux of this job. The structure of this chapter is the following[1]:

- in the first one we discuss about the problems and the difficulties we met, while developing the code;
- the second subsection offers the instructions to use a certain tool;
- in the third one, an example[2]
  In this example we start from the very beginning (i.e. we begin launching the tool and loading the *DICOM*-series, till the mesh creation, step by step). The patient in the example is Mr. Bianchi and we extract a branch

---

[1] Each section (except the first one because it is not technical) is divided into 3 subsections.

[2] To write and develop the third subsection at each step, the paper by J. Bonnemain (see [5]) has been studied and followed; we reproduced the same steps that Bonnemain did with *VMTK* with our new tool. As it is shown in the mentioned paper, through *VMTK* it is possible to reconstruct the desired geometry and then to build a mesh over it. Also, as already discussed, the problem is that we need to learn lots of non-simple and non-intuitive commands to be written in the command line. What we would like to do with this tool is to do the same things that *VMTK* does but, and this is a *VMTK*'s great advantage, creating a *GUI*. So that, users who are not either computer scientists or engineers could use this tool without learning any command, but simply reading what the buttons in the new interface say and following this user's guide.

of the descending aorta from a thoracic *CT*-scan; that's why the file we read is named "aorta" and we give the same name to the file we write.

Before writing the *VMTKGui user's guide*, it is important to give an overview of the structure of the code. This wants to be simply an introduction. You can contact the author if you are interested in the Doxygen documentation, to better understand the structure of the code.

Let's start with a brief overview of the implemented files:

- **vmtkgui.py**: this is the main file. It is used to build a link between the different files and to execute the program. In order to be a good main file it has to be short and clear. It is divided in two sections: in the first section there are the calls to the different functions, while in the second one there is the command to execute the program;
- **vmtkinterface.py**: as the name suggests, this file is the interface between the *GUI* built with *VMTKGui* and the *VMTK* software. This module contains all the *VMTK* functions needed for the *VMTKGui* tool;
- **canvas3D.py**: this module is made up of two classes. This two classes has to create an area to draw the *DICOM* images in and the 3D reconstruction. Also, through this module it is possible to manage the interaction between the mouse and the window where we are representing the reconstruction;
- **vmtkcenterlineswithrenderer.py**: this file is very important to make it possible to use the same render. Without this file, using the "original" *VMTK* function vmtkcenterlines, we create a new render when we invoke it. Since we need to use the same renderer we define this new function that allows us to have a function with the same features of the original one, using the same window;
- **gui.glade**: this is the file that allows to build the *GUI*. It is a file automatically generated by *Glade* and it is structured in a style that is xml-like.

As one can see, there are not many files involved in this algorithm: that means that our goal to keep the most of the original software (i.e. *VMTK*) has been achieved. And this is certainly a good starting point, in order to have a "clear and user friendly" tool.

Let's now start, *step-by-step*, to explain the usage of the software.

3.1 *Pre-requisites, installation and launching the tool*

The tool *VMTKGui* is built for a *UNIX*-based system[3]. So, a first problem is that you must use Linux to run the tool. Also, *VMTKGui* does

---

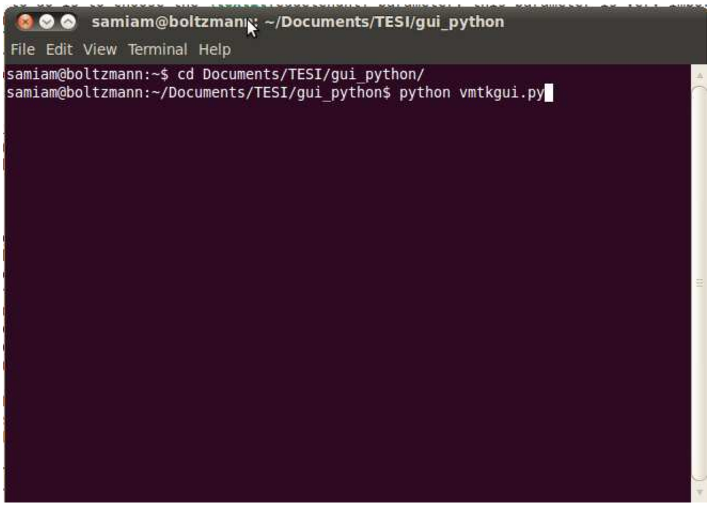
[3] It is important to underline that in this report we work on a Debian-based system. In this case the software used has *ad-hoc* packages for Debian-based distros that make it very easy to install the software needed. For other distros, you may need to compile the packages.

not need any installation: it is enough to launch in from the command line, once all the pre-requisites are satisfied. Hence, let's analyze these pre-requsites:

- *VMTK*: the latest version is downloadable from *http://www.vmtk.org*. On the same page one can also find the instructions to install and to use this software;
- *Python*: see *http://www.python.it* in order to download and install it;
- *VTK*: once again, on *http://www.vtk.org* there are all the information needed to download and install *VTK*.

When all the software needed is installed, it is possible to download the files of the tool *VMTKGui* and to copy them into the desired folder. Once all the files are properly copied, the user has to open a terminal and change the directory, entering the folder where the *VMTKGui* files have been saved. Everything is now ready and we can launch/use the tool.
To launch the tool the right command to write in the terminal (see Figure 3.1) is

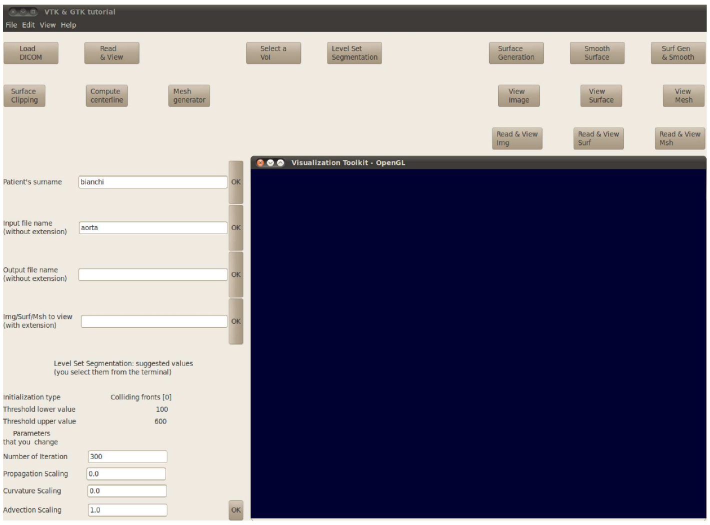
python vmtkgui.py.



```
samiam@boltzmann:~$ cd Documents/TESI/gui_python/
samiam@boltzmann:~/Documents/TESI/gui_python$ python vmtkgui.py
```

Figure 3.1. This is the command used to launch the *VMTKGui*'s *GUI*.

The *GUI* appears (see Figure (3.2)) and the user can start using the software.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

Initialization type	Colliding fronts [0]
Threshold lower value	100
Threshold upper value	600

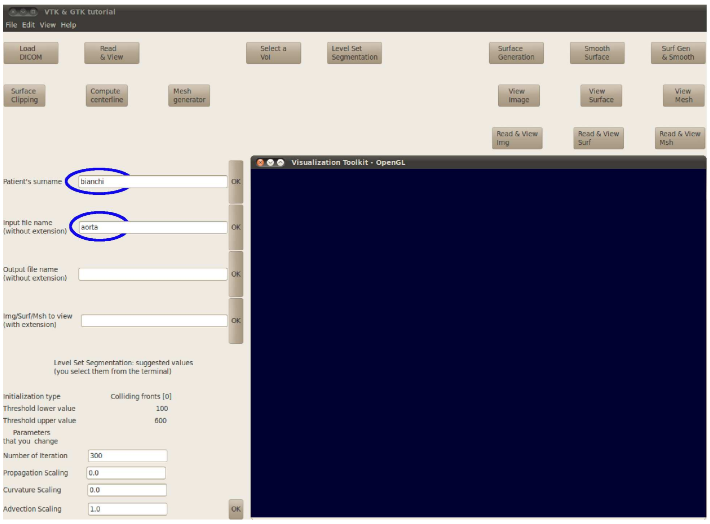
Parameters that you change

Number of Iteration	300
Propagation Scaling	0.0
Curvature Scaling	0.0
Advection Scaling	1.0

Figure 3.2. This is the *GUI* as it appears once we launch the tool *VMTKGui*.

3.2 *Choosing the patient and loading the DICOM images*

It is now necessary to insert the patient's name and the name of the file we want to analyze. In this example we want to examine the descending aorta and our patient is Mr. Bianchi. The first thing to do is to write "bianchi" in the right line (see Figure (3.3)) and press "Ok". Then, we can choose the name that we want to give to our file (e.g. "aorta"[4] as you can see in the same Figure (3.3)) and we press, once again, "Ok" to confirm our choice.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

Initialization type	Colliding fronts [0]
Threshold lower value	100
Threshold upper value	600

Parameters that you change

Number of Iteration	300
Propagation Scaling	0.0
Curvature Scaling	0.0
Advection Scaling	1.0

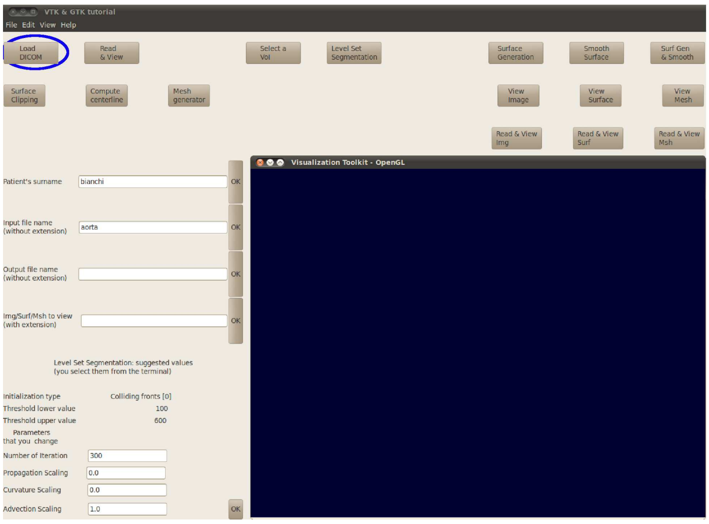
Figure 3.3. Insertion of the patient's name and of the name of the file we want to analyze.

---

[4] Note that the user does not have to care about the file extensions and the suffix: the tool provides both of them automatically, in a smart and standard way.

The second box that deals with the "Output file name" (see Figure (3.3)) may be left void. In this case, the file we create takes the same root-name of the *DICOM*-series.

We now have to load the chosen *DICOM*-series: it is now enough to click on the **Load *DICOM*** button, as shown in Figure (3.4).



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

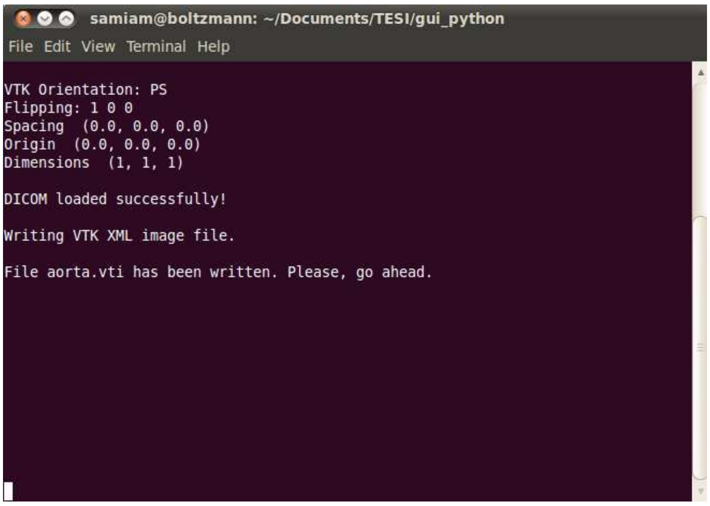
Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.4. Button that we have to press in order to load our *DICOM* series, once that all the preliminaries have been done.

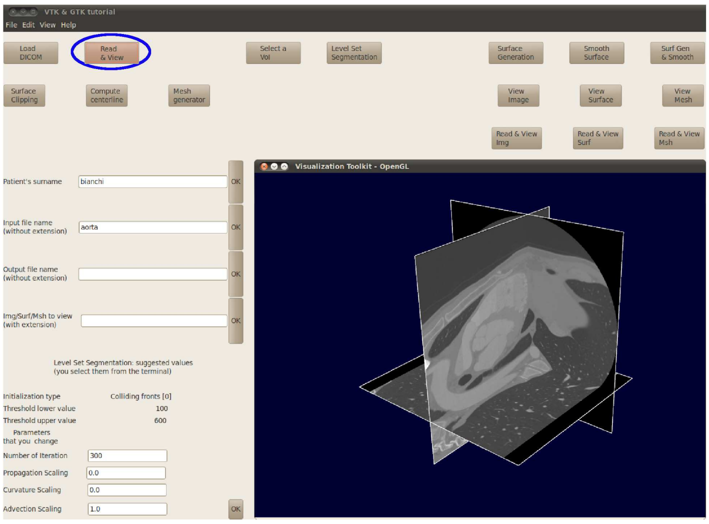
We have to wait till in the terminal we can read:
DICOM loaded successefully: please go ahead
(as you can see in Figure (3.5)).



```
VTK Orientation: PS
Flipping: 1 0 0
Spacing  (0.0, 0.0, 0.0)
Origin  (0.0, 0.0, 0.0)
Dimensions  (1, 1, 1)

DICOM loaded successfully!

Writing VTK XML image file.

File aorta.vti has been written. Please, go ahead.
```

Figure 3.5. This message confirms that the *DICOM* has been loaded successefully.

Once the *DICOM* is loaded, we can view the *CT* by clicking on the "Read & View" button (see Figure (3.6)). All the preliminaries have been done. We are now ready to go ahead.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]

Threshold lower value: 100

Threshold upper value: 600

Parameters that you change

Number of Iteration: 300

Propagation Scaling: 0.0

Curvature Scaling: 0.0

Advection Scaling: 1.0

Figure 3.6. This is the visualization of the *DICOM*-series we load.

3.3 *Preparing the tridimensional reconstruction*

Now the *DICOM* is loaded and we can visualize it. The following step that the user has to do is to create a tridimensional reconstruction in order to make it possible to generate a file that can be used with *VMTK* and to create the mesh on it. As we can see in the next subsection, this is a key point and some problems arose, while facing it.

3.3.1 Problems

The first problem deals with the *CT*. In fact, it was born for visualization analysis and not for reconstruction ones. Hence, the geometry resolution is often too thin and the fidelity of the reconstruction may be a problem. That is way interpolation is needed. Also, as explained in the theoretical chapter (see Chapter 2), the method to get the reconstruction (e.g. the *LSS*) implies non-simple maths.

One more difficulty is that often we deal with a big volume and we need a much smaller one for our analysis. That is way - in order to reduce the the costs - the selection of a “Volume of Interest” is a smart tool to reduce the size of the problem.

Also, we have to face with the complexity of the already existing code. In fact, there is a very nested code that does not allow us to completely manipulate the source code.

3.3.2 Usage

To get our reconstruction, some steps are needed:

- the first step is to select a "Volume of Interest": for example, if we are interested in the reconstruction of the mitral valve and we have a thoracic CT, it would be useless (and too expensive, computationally speaking) to work on a big file. That's why the "Volume of Interest" feature is very simple but very useful: it allows the user to select a certain volume of interest and to reduce the file size, simply by clicking on the proper button.

  This makes the following operations faster. Though it is not a compulsory operation, it is very useful;
- once we have saved a certain volume of interest, we have to be able to make a tridimensional reconstruction of the geometry that we want to analyze. This is done thanks to the *LSS* technique. For the theory behind this technique the user should read chapter 2 and all the bibliography associated. What must be underlined here is that before clicking on the "*LSS*" button we have to select some parameters[5]). Hence, the first thing to do is to choose the requested parameters and to press. Then we can make the reconstruction start, by pressing the proper button as we can see in the next subsection.

  Now the user has to interact a bit with the terminal since it is necessary to press *y/n* once in order *to/not to* accept the result and to press *y/n* once, *to/not to* initialize another branch. Once again, to save what we have done and to quit it is necessary to press *q*;
- The last thing to do is to press the "Surface Generation" button in order to create a .vtp surface from the .vti image. Now we do have a surface and we are ready to go to the next step, that is to say to create the mesh, over the surface and/or in the volume. It is very important to be able to create both a surface mesh and a volume mesh since we could need either the former or the latter, depending on the simulation we are interested in.

  After having created the surface it is necessary to smooth it and clip both the inlet and the outlet sections: in fact, the surface reconstruction has been done starting from the *DICOM* image and this image is pretty noisy since it takes a certain time to scan all the sections during the *CT*-scanning. Also, it is important to clip the inlet and the outlet sections: this operation makes it easier both to compute the centerline and to generate a mesh over the surface and/or in the volume (see the section 3.4). To smooth the surface it is enough to click on the right button, while to clip it, it is necessary to interact a bit with the *GUI* after having pressed the "i", in order to be able to activate the image (to launch the clipping command it is necessary to click on the clipping button). The procedure is similar to the selection of a volume of interest.

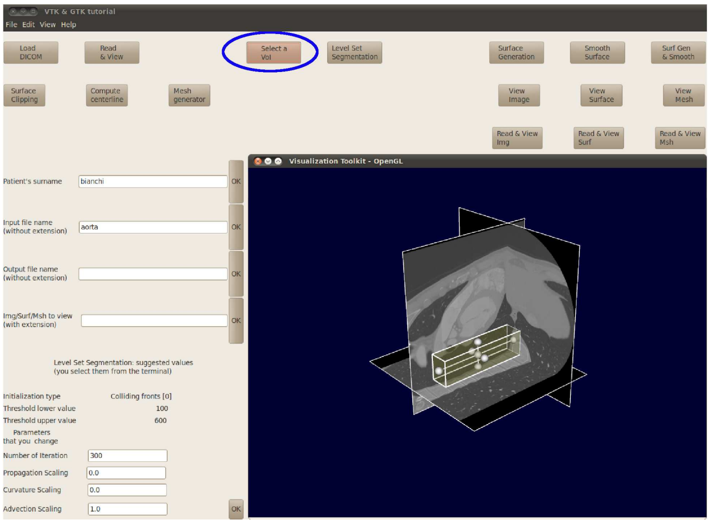
[5] It is important to point out that some parameters could be chosen directly in the *GUI*, while others have to be chosen from the command line.

A parallelepiped appears, we place it properly and we press the space button to clip the surface.

3.3.3 Example

In this subsection we present an example in order to understand how the reconstruction (explained in the previous subsection) works. This step allows the user to convert the file from the *DICOM*-standard format to the *VMTK* format. Once again, thanks to the *VMTKGui*, this is a very easy step, since it is enough to click on the right buttons and interact a little bit with the terminal, as explained in the following list.

- the first sub-step is to select a “Volume of Interest”: since in this example we exctract a brench of the descending aorta, we are interested in the volume that includes it. Hence, we press the “Select a VoI” and then we press “i” to start our interaction with the image. Then we have to select a parallelepiped that includes the descending aorta (see Figure (3.7)), resizing and moving it with few clicks[6].



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

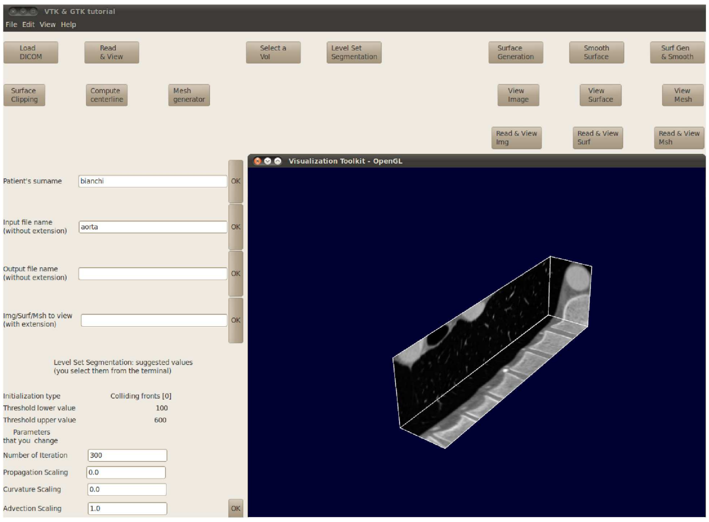
Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.7. In the figure above it is possible to notice the parallelepiped that cointains the part of the geometry we are intereste in.

Once the parallelepiped is chosen it is necessary to press “q”, in order to quit and save a smaller and lighter file that appears like the image in the Figure (3.8);

---

[6] With a middle-click of the mouse we can translate the parallelepiped, while with a left-click on the “little spheres” we can resize it.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

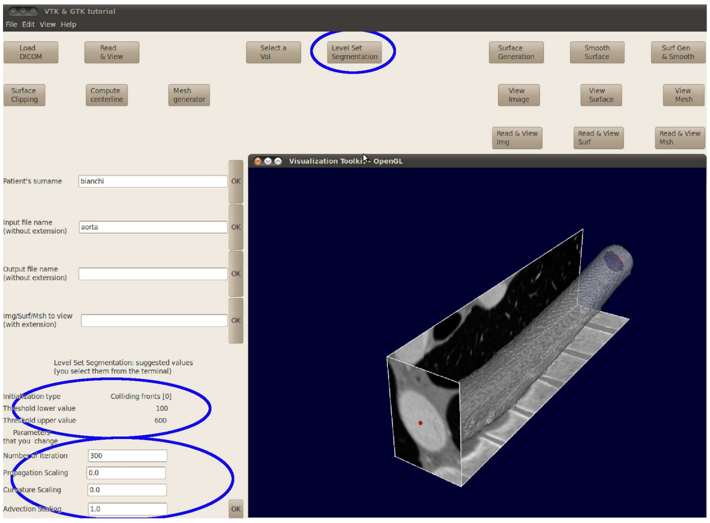
Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.8. This is the visualization of the Volume of Interest only.

- once we select a certain volume of interest, we have to be able to make a tridimensional reconstruction of the area we want to analyze. This is done thanks to the *LSS* technique. Actually, this is a theoretical step since, practically, we simply need to press the "Level Set Segmentation" button and everything is done automatically. We also have to interact a little bit with the command line since we have to choose some parameters (suggested in the *GUI*, see Figure (3.9) and - finally - we could see the initialization of the surface we are trying to reconstruct (see again Figure (3.9));



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

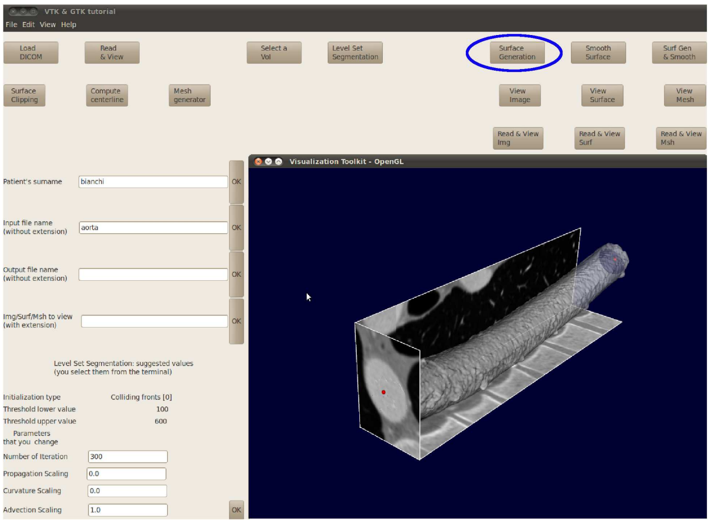
Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.9. In the picture above there is the initial guess for the *LSS* procedure.

- the last thing to do is to press on the "Surface Generation" button (see Figure 3.10) in order to create a surface (the file extension is .vtp) from the .vti image.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

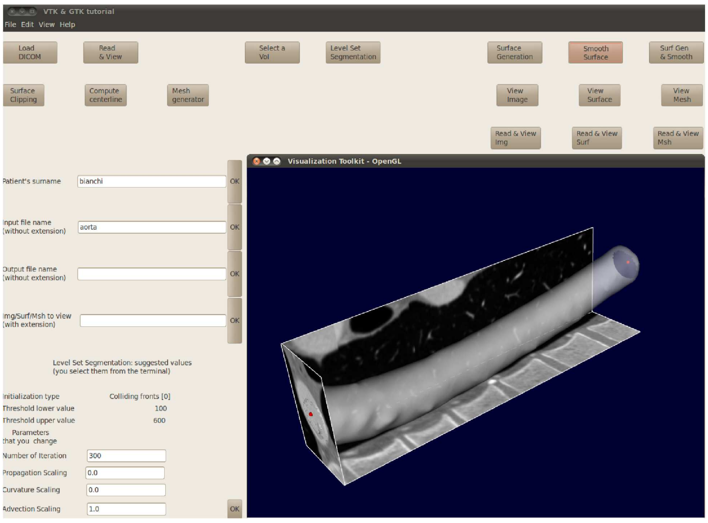
Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.10. Circled in blue the button to generate the surface.

Now we do have a surface and we have to smooth it (see fig. 3.11) and to clip both the inlet and the outlet section (see the procedure to clip a surface in the 2 Figures (3.12) and (3.13): in the first one it is possible to see the parallelepiped which indicates the region that must be clipped while in the second picture the reader could enjoy the clipped surface).



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values (you select them from the terminal)

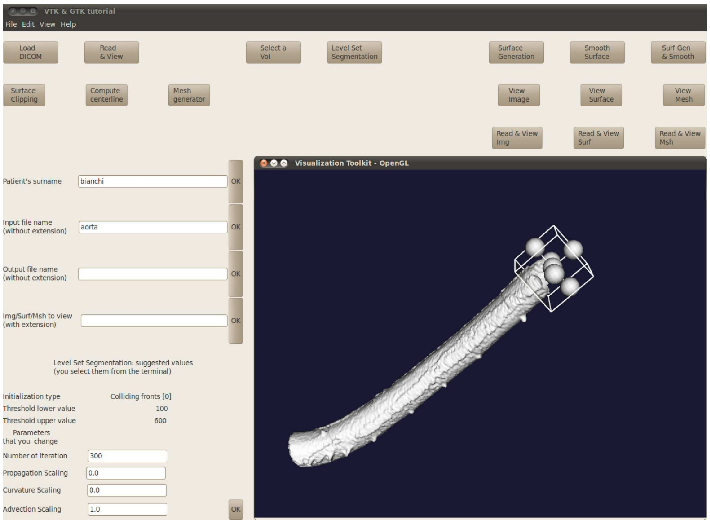
Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change

Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.11. In the picture above there is the smoothed surface.

In the pictures of Figures (3.12) and (3.13) we show the procedure to clip the inlet section, only. Of course, the same can be done for the outlet section, too.



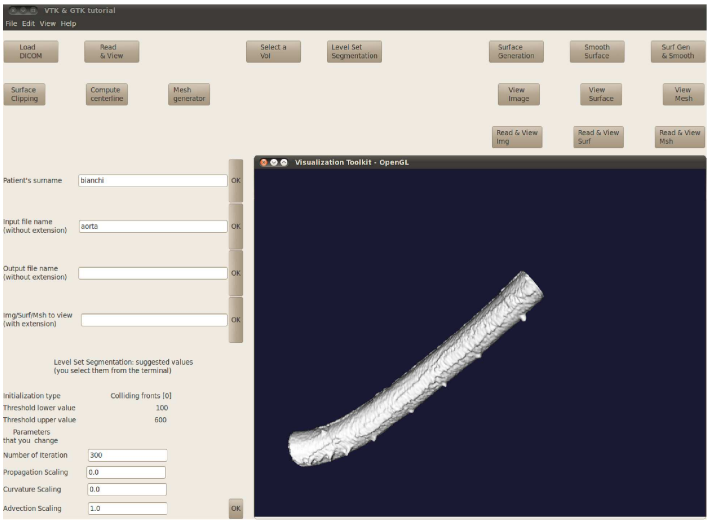
Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.12. In the picture above there is the clipping parallelepiped.



Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.13. In the picture above there is the clipped surface: the clipping has been done before smoothing the surface.

3.4 *Computing centerline(s) and generating mesh*

In this section we deal with two very important features: the computing of the centerlines and the mesh generation. Computing the centerline of a vessel is very useful since centerlines are a powerful tool to describe vessels geometry.
As far as the meshes are concerned, it is clear that they are very importand for a *FEM* analysis and/or for the study of a *CFD* simulation.

3.4.1 Problems

The two features detailed in this section present some criticism:

- Computing of the centerlines: the idea of computing the centerlines is very simple from a theoretical point of view. In fact, it is enough to connect the centers of the vessel at each secion/slide[7]. Practically, this is very difficult since we do not have a vessel the is straight, perpendicular to the *CT*-slides and that does not have a circular section. So, the definition of the center itself is not clear and unique.
  Also, since the dimensions could be very small, a small absolute error could be a big relative one, thus producing a wrong computation of the centerlines. Hence, the results are far from the exact ones. E.g., thinking about the insertion of a stent in a certain point, having a wrong centerlines means to insert the stent with the wrong orientation with all the problems that this can cause.
- Mesh generation: due to the complexity of the geometry, the mesh generation is a crucial point. We have to get the right compromise between a good space-resolution and a not-too-big mesh. Also, since the surface may be noisy, with an accurate choice of the mesh parameters, we could get the good effect of a smoothing.

3.4.2 Usage

3.4 To compute the centerline of a vessel is matter of seconds: just a single click on the right button and a window appears. In this window (as it is shown in the next section) there is the tridimensional reconstruction we did with clipped inlet and outlet. Each inlet and outlet section has a number id: the user has to indicate in the command line which are the inlet sections and which are the outlet ones. If the software could not recognize any inlet/outlet sections, it is possible to select them manually. This is done by positioning the mouse where we want to place the inlet/outlet section(s) and pressing the space bar.

Finally, the last step we could do, is the mesh creation: this step is very important since the mesh generation is the starting point for the *FE* analysis. By default this feature creates a volume mesh, but - if necessary - we could generate a surface mesh, too. The goal of this step is to create a mesh that is supported by different *CFD* solvers. It is enough to click on the right button in order to generate the desired mesh. What we have to do is to choose the *edgelenght* parameter: this parameter is very important since it is the absolute nominal length of a surface triangle edge. Once we have generate the mesh it is possible to transform it in the right format, according to the *CFD* solvers the user wants to use.

This is done by typing

---

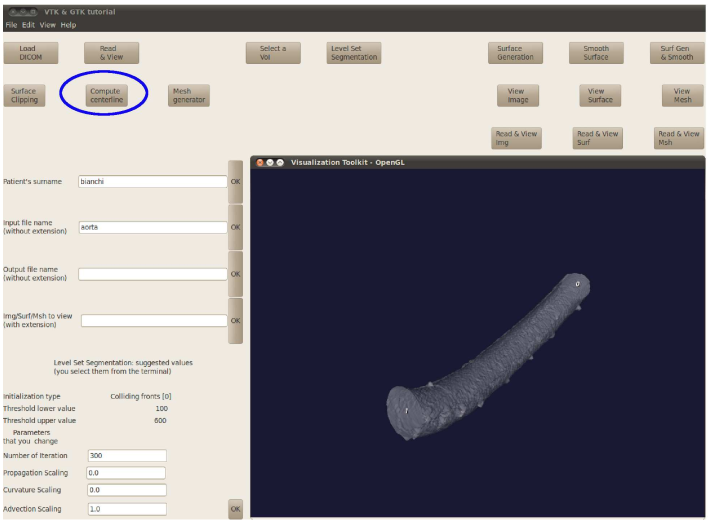
[7] This could appear very simple since we have a *CT*-scanning.

```
vmtkmeshwriter -ifile file_name.vtu -entityidsarray
CellEntityIds -ofile file_name.lifev
```

in the command line[8].

3.4.3 Example

As explained in the subsection 3.4, both to compute centerlines and to generate a mesh are important features that the tool *VMTKGui* offers: with a single click on the proper button (see fig. 3.14 in the previous section) we can compute the centerline of the descending aorta.



Patient's surname: bianchi

Input file name (without extension): aorta

Output file name (without extension):

Img/Surf/Msh to view (with extension):

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

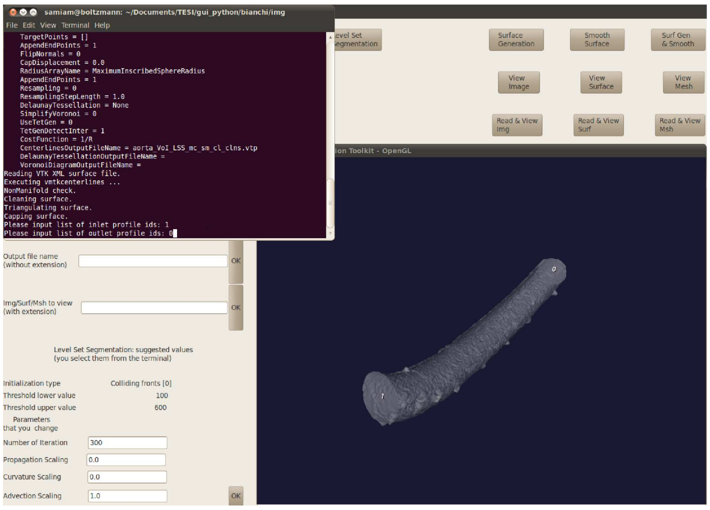
Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.14. The figure shows the button (circled in blue) to compute the centerline(s).

Also, in Figure (3.15), we can see both the vessel reconstruction with the ids and the terminal where we have to insert which is/are the inlet(s) (0, in this case) and which is/are our outlet(s) (1, in the example presented here) are shown and the surface as it appears before the centerline(s) is/are computed.

---

[8] As it is shown in the section 4.2, one of the target is to insert a button in the *GUI* that allows to convert the mesh and to make it possible to choose the mesh dimension without modifying the source code.



```
    TargetPoints = []
    AppendEndPoints = 1
    FlipNormals = 0
    CapDisplacement = 0.0
    RadiusArrayName = MaximumInscribedSphereRadius
    AppendEndPoints = 1
    Resampling = 0
    ResamplingStepLength = 1.0
    DelaunayTessellation = None
    SimplifyVoronoi = 0
    UseTetGen = 0
    TetGenDetectInter = 1
    CostFunction = 1/R
    CenterlinesOutputFileName = aorta_VoI_LSS_mc_sm_cl_clns.vtp
    DelaunayTessellationOutputFileName =
    VoronoiDiagramOutputFileName =
Reading VTK XML surface file.
Executing vmtkcenterlines ...
NonManifold check.
Cleaning surface.
Triangulating surface.
Capping surface.
Please input list of inlet profile ids: 1
Please input list of outlet profile ids: 0
```

Output file name (without extension)

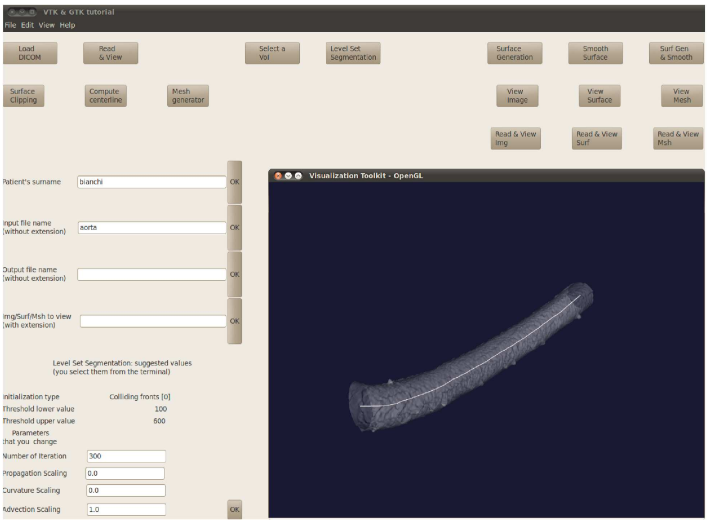
Img/Surf/Msh to view (with extension)

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type Colliding fronts [0]
Threshold lower value 100
Threshold upper value 600
Parameters that you change
Number of Iteration 300
Propagation Scaling 0.0
Curvature Scaling 0.0
Advection Scaling 1.0

Figure 3.15. This figure shows the reconstructed geometry with the ids used to identify each possible inlet/outlet. There is also the terminal where we have to insert the inlet(s) and the outlet(s).

In this window there is the tridimensional reconstruction we did with capped inlet and outlet. Each inlet and outlet section has a number id: the user has to write in the command line which are the inlet sections and which are the outlet ones. In this example - since we are dealing with a simple geometry - we have only 2 sections that could be either the inlet or the outlet of the flow. After having chosen which is the inlet and the outlet, the tool starts to compute centerline(s) and the result is presented in Figure (3.16).



Patient's surname bianchi

Input file name (without extension) aorta

Output file name (without extension)

Img/Surf/Msh to view (with extension)

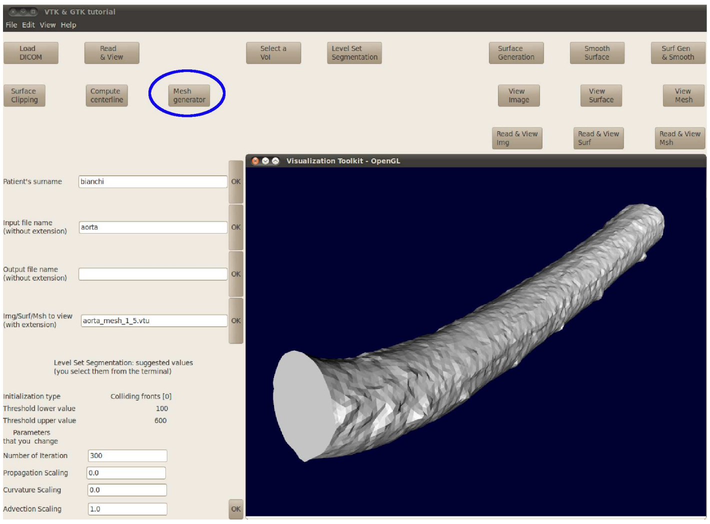
Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type Colliding fronts [0]
Threshold lower value 100
Threshold upper value 600
Parameters that you change
Number of Iteration 300
Propagation Scaling 0.0
Curvature Scaling 0.0
Advection Scaling 1.0

Figure 3.16. This is the reconstruction of the centerline of the brench of the descending aorta we have exctracted.

Also, if we are interested in a *FE* analysis and not in simulating the insertion of a stent, we do not need the centerline computation, but we

would need to generate a mesh.
We have, by default, the generation of a volume mesh but we can also get a surface mesh[9], when the surface mesh is our goal. What we would like to do is to create a mesh that is readable from any *CFD* solver. It is enough to click on the right button (see Figures (3.17) and (3.18)) in order to generate a mesh. What we have to do is to choose the edgelenght parameter: this parameter is very important since it is the absolute nominal length of a surface triangle edge (see the difference beetween the Figure (3.17) and Figure (3.18): in these two pictures the only parameter that changes is the absolute edgelenght. As one can immediately notice, the dimension of the triangles in the two pictures is very different and some details are better displayed in the thicker mesh reconstruction).



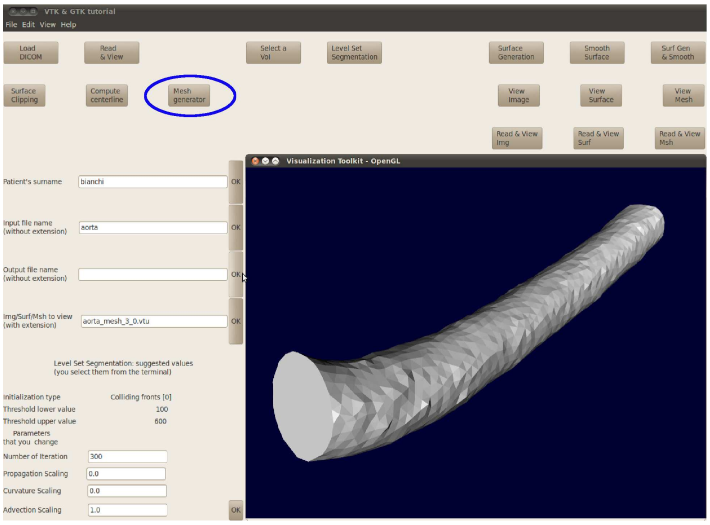
Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension): aorta_mesh_1_5.vtu

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

Figure 3.17. In this figure we choose a *edgelenght* parameter of 1.5.



Patient's surname: bianchi
Input file name (without extension): aorta
Output file name (without extension):
Img/Surf/Msh to view (with extension): aorta_mesh_3_0.vtu

Level Set Segmentation: suggested values
(you select them from the terminal)

Initialization type: Colliding fronts [0]
Threshold lower value: 100
Threshold upper value: 600

Parameters that you change
Number of Iteration: 300
Propagation Scaling: 0.0
Curvature Scaling: 0.0
Advection Scaling: 1.0

---

[9] This option has not been implemented yet, as you can read in the chapter 4, where conclusion and future developements are given.

Figure 3.18. In this figure we choose a *edgelenght* parameter of 3.0.

4. Conclusions and future developments

> The future belongs to those who believe in the beauty of their dreams.
>
> Anna Eleanor Roosevelt (1884-1962)

Besides the theoretical concepts (chapter 2) and the user's guide (chapter 3), it is very important to write a conclusive chapter in which we presents a final digest of this report and where we can discuss about the possible future developments to improve the tool presented here. Also, this is a section in which we present an honest balance between the ideal targets vs. the caught ones.

That being so, this final chapter is divided into two sections: the first section deals with a balance between the targets we wanted to reach vs. the target we reached, explaining which are the reasons that did not allow us to catch all the goals we wanted to reach. In the second one we provide some cues in order to make it possible, for the interested reader, to develop and improve the *VMTKGui* tool and to realize the *VirtualValveStent* tool[1].

4.1 *Conclusions*

Honesty: this is the keyword of this section. Here we sum up all the results we got and we analyze which are the targets we did not reach, trying to explain the reasons that prevent us from reaching them.

The very first thing to say is that we are very happy and proud of what we did and that we are sure that this tool is very useful.

Moreover, let's start saying that this project started with the idea to create the *VirtualValveStent* tool and not the *VMTKGui* tool. The original request came from "Symetis", a Swiss firm which deals with stent

---

[1] The author strongly believe in the *VirtualValveStent* project and is available for any question about the implemented algorithm. Feel free to contact him, if interested

fabrication. We immediately observed that this was a too heavy project to be developed in a master thesis. That is why we decided to divided the master project in three different sub-projects. And we decided to develop only the first of this sub-targets. This has been done to avoid to begin with a too demanding work that we would have given up in the middle of the project itself.
Also, there is something to say about the programming language used: we started modifying the *C++* libraries and not the *Python* ones. But it was more complicated and that choice did not offer any particular advantage. That is why we have found it very useful and smart to change our mind, working with *Python*. In fact, *VMTK* has the right interface for a *Python*-coding.
Furthermore, since the source code is terribly nested, we have found it very difficult to supplement the new code with the already existing files and it took a long time to cope with this matter. That is why it has been impossible, up to now, to remove each kind of interaction with the command line. As we will see later (read section 4.2), this is going to be a very challenging point to be developed in the future.

What it is very important to underline is that we spent much time and many resources to develop such a tool, with the hope that it could be a help in the medical field. Also, we hope that the reader could find it useful the “Users’ Guide” available in chapter 3 and also the *Python* code reported in Appendix B.

To end this report, the author wants to assure that he works honestly and with nothing but a target: to make something useful and available for other people. This could sound crazy, but he thinks that he is not “a number” (i.e. the graduation score): that is why he decided to develop such a tool that does not match with an aeronautical contest.

4.2 *Future developments*

In this final section, all the possible future developments are summarized: this part is very important since it has the hope to avoid to make this report and above all the *VMTKGui* tool a mere loss of time. We strongly believe in this tool and in the possible developments it offers. That is why we think it would be mindless to throw to the wind what we have done. About this section, it is divided into two subsections: the first deals with the short-term targets, while in the second one the longer-term goals are listed.

4.2.1 Short-term targets

In this first subsection the short-term targets are presented: these targets deal with the presence of some bugs in the code. Bugs that can be fixed soon. Also, there are some improvements that require a pretty simple work in order to be done. Here is a list:

- the *VMTKGui* tool can be considered complete. Nevertheless, some features could be improved. For example, it could be useful to add the possibility to choose the *edgelenght* parameter directly from the *GUI*, avoiding to modify the source code (even tough it is nothing but a number that must be changed);
- another feature that we want to insert in the *VMTKGui* tool is the button that allows to choose to generate either the surface or the volume mesh;
- also, it would be smarter to have pop-up windows (since they draw the user's attention much more than a black command-line) instead of the an interaction with the terminal (which is boring and, at times, not so "visible"). We write here this target, even tough we suppose it probably takes a long time to completely eliminate any interaction with the command line, since the source-code is very nested;
- also, it could be useful (but this is a not-so-important goal) to create a nicer graphic. Even tough this is not a conceptual goal, this would make the software more user-friendly and the *VMTK*-community and the users in general should appreciate it much more.

Once the short-term goals are reached, we can state that *VMTKGui* is complete and finished. Let's now check the longer-terms targets.

4.2.2 Long-term targets

Here, we collect the long-term goals that will be hopefully reached from any interested user. Our dream is to see, soon or later, the *VirtualValveStent* project ended and used for the purpose it has been thought. Let's sum up all the long-term targets that must be achieved:

- we must shape the stent properly, in order to have a correct simulation of the operation that we have to do. Actually, this is a simple target, since once the surgeon gives to the user the shape of the stent he is interested in, it is matter of minutes to reproduce it;
- once the stent has been modeled, it is important to insert it in the same window where we have our geometry reconstruction in order to be able to simulate the operation;
- once we get these two targets, the most has been done and we will have reached the purposes we wanted to reach with the *VirtualValveStent* project. However, there is one more step we could do, that is to say that we can automate recognizing the different geometry, giving a certain color depending on the HU value. This helps to make it easier to identify the different part of the area we are studying. We think it is honest to state here that there already exist *open-source* codes that do this (e.g. 3D Slicer, Osirix[2]), but we prefer to embed this feature in our own tool.

---

[2] Actually Osirix is free only for the Mac-OSX users

Once we will have reached all this goals, we have a good software that allows to reconstruct a certain geometry starting from a *DICOM*-series. Also, we can specialize ourselves in the mitral valve area reconstruction. This will allow us to simulate the *transcatheter mitral valve replacement* that was the original target. The author's hope is that this will become available soon. He also thinks that - thanks to the help of many people[3] - he paved the road to reach that goal. The next step is to find volunteers interested in developing this tool. We are aware that it is not matter of days or weeks, but it will be a great satisfaction to complete this project and to make it available worldwide.

[3] I would like to dedicate here a special thank to Dr. Simone Deparis for his great help and to Professor Alfio Quarteroni for the chance heofferes me.

Appendici

A. Computational Fluid Dynamics

Computational Fluid Dynamics (*CFD*) is the art of replacing the partial differential equations (which represent conservation laws for the mass, momentum and energy) by a set of algebraic equations which can be solved digitally.

CFD provides a qualitative (and sometimes even quantitative) prediction of fluid flows, using a mathematical modeling first, then a numerical method and, finally, running software tools.

As we can see in the Figure A.1, *CFD* is nowadays used in many different fields, from aeronautical problem, to biomedical one; from automotive, to civil engineer.

The governing equation for a fluid, according to the fundamental laws of mechanics are the equation for the conservation of the mass and the equation for the conservation of the momentum:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{A.1}$$

and

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \rho g + \nabla \tau_{ij}. \tag{A.2}$$

Also, there is the energy equation. These three equations form a set of coupled nonlinear PDEs. And this system, in the most of the engineering problems, can not be solved analytically. However, we can solve this system thanks to the Computational Fluid Dynamics (*CFD*).

To justify this appendix, it is important to underline that biomedical engineering is a field that is growing rapidly and it uses *CFD* for studies about the circulatory and respiratory systems. As an example, we show below (see Figure A.2) a blood pump that plays the role of heart in the open-heart surgery.

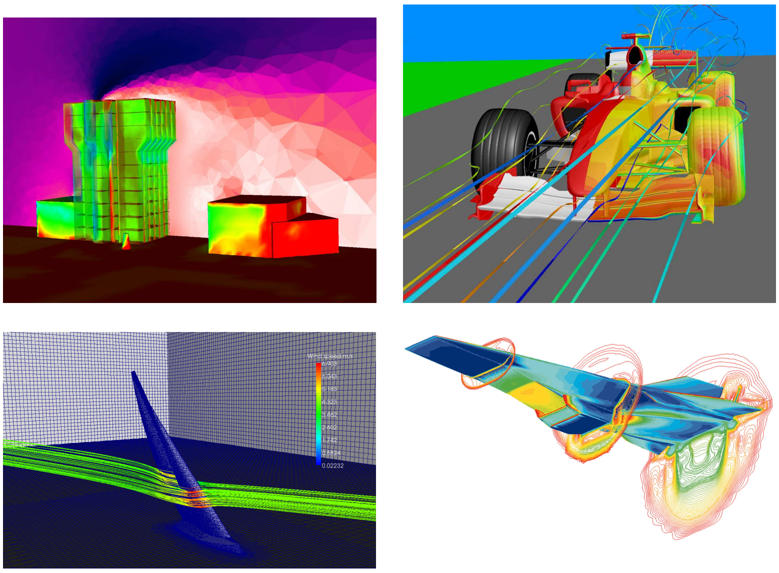


Figure A.1. Some of the possible applications *CFD* can be used for.

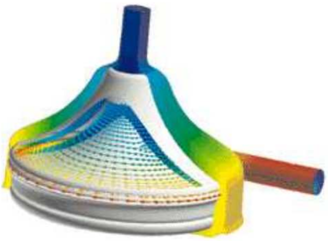


Figure A.2. *CFD* in blood pump that plays the role of heart in the open-heart surgery.

To solve problems in *CFD*, it's necessary to specify the problem, i.e. its geometry, the flow conditions and the requirements for simulation (precision level, require time, solution of the interest parameters).

So, why is *CFD* so important? In *CFD* it's possible to model and evaluate the fluid flow performance in our study model. The application of *CFD* simulations offers a set of advantages when compared to theoretical and experimental studies:

- it has purchase and operation low cost;
- it offers detailed information about the fluid flow studied;
- it allows the quick change of parameters in the flow analysis;
- it allows simulating flows in detailed and complex geometries and study phenomena impossible to do in experimental model (ex: explosions).

But, an important warning we want to remark is that the *CFD* does not substitute the theory and experience; we should always make an approach of three, to interpret the results. The comparison with experience or with simple cases, the solution of which is known, gives us the accuracy achieved by the simulation. In fact it could be dangerous to perform a simulation without any experiment and/or theoretical comparison.

A.1 *How CFD works*

The strategy of *CFD* is to replace the continuous problem domain (with infinite degrees of freedom) with a discrete domain (with a limited number of dof).

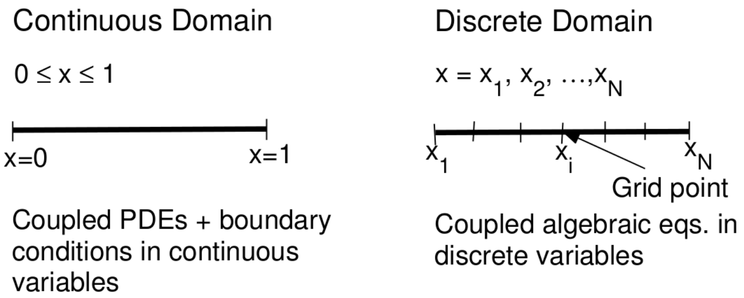


Figure A.3. Continuous and discrete domain: from the analytical to the numerical problem.

Analyzing the two domains (the continuous and the discrete one) in Figure A.4, and calling q a certain quantity we are interested in, in the continuous case we know

$$q = q(x), 0 < x < 1,$$

while in the discrete problem we only know the value of q in the different nodes, i.e.

$$q_i = q(x_i), i = 1, 2, .., N$$

So, with *CFD* we know the values of the relevant variable at the grid points. We have to interpolate, to get the values at different locations. On the other hand, both the equations themselves and the boundary conditions are continuous: we have to extract the values in the grid points. Hence, we have a large system of coupled algebraic equations.

A.2 *Discretization: “Finite Difference method” vs. “Finite Volume method”*

In this section we will introduce the two techniques used to discretize the system of PDEs.

A.2.1 Finite Difference method

In this subsection (since this appendix wants to be an introduction to *CFD*) we analyze a 1D linear equation, i.e.:

$$\frac{du}{dx}+u=0; 0\le x\le 1; u(0)=1 \tag{A.3}$$

Let's consider the grid in Figure A.4.

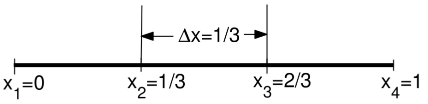


Figure A.4. Grid used to solve the Equation (A.3).

Now we can write the Equation A.3 on the grid points, thus leading to the following:

$$\left(\frac{du}{dx}\right)_i+u_i=0$$

We have to evaluate $\left(\frac{du}{dx}\right)_i$. We can write u_{i-1} in a Taylor's series:

$$u_{i-1}=u_i-\Delta x\left(\frac{du}{dx}\right)_i+\mathcal{O}(\Delta x^2),$$

that leads to:

$$\left(\frac{du}{dx}\right)_i=\frac{u_i-u_{i-1}}{\Delta x}+\mathcal{O}(\Delta x),$$

where $\mathcal{O}(\Delta x)$ is the truncation error. Excluding higher order terms we can write:

$$\frac{u_i-u_{i-1}}{\Delta x}+u_i=0,$$

which is an algebraic equation.

A.2.2 Finite Volume method

In the finite volume method for discretization, we have to introduce the "cells" and the "nodes". If you look at Figure A.5, you can see a cell and the for vertexes are called nodes.

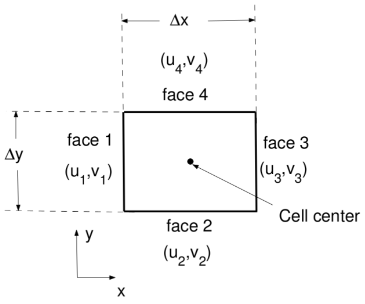


Figure A.5. Rectangular cell used for the finite volume approach.

Using this approach, we have to apply the integral form of the conservation equations. As far as the mass conservation is concerned, we can write:

$$\int_s \mathbf{u} \cdot \mathbf{n} \, dS = 0. \tag{A.4}$$

This equation implies that the net volume flow inside the control volume is 0. Considering the cell of Figure A.5, the velocity at the face i is $\mathbf{u}_i = u_i \hat{\imath} + v_i \hat{\jmath}$. Applying the mass conservation (through Equation (A.4)) to the control volume (i.e. the cell), we have:

$$-u_1 \Delta y - v_2 \Delta x + u_3 \Delta y + v_4 \Delta x = 0.$$

Once again, we have an algebraic equation that can be solved.

And the same can be done both with the momentum and the energy equation. In this case - as in the “Finite Difference” approach - we have an algebraic system that can be solved.

A.3 *Main difficulties*

It is clear that *CFD* must have its drawback, too. We analyze here the 2 main problems that may arise while performing a *CFD* analysis. The first problem is the numerical stability of the method used. The second, is the turbulence modeling.

A.3.1 Numerical Stability

We just give here a brief introduction to this problem. Depending on which kind of method we are using, i.e. either implicit or explicit, we may have to face with the numerical instability.

Explicit method, in fact, needs a temporal step that must be smaller than a certain value. This value is known as the **Courant number**, C. The CFL[1] condition states that $C \equiv \frac{c\Delta t}{\Delta x} \leq 1$.

If the limitation due to the CFL condition is too restrictive, we can use an implicit method. Of course, we have a fee to be paid: we have to solve a system of algebraic equations, instead of solving each equation detached from the others.

A.3.2 Turbulence Modeling

Finally, let’s introduce the problem of the turbulence modeling. The most of the flows are turbulent[2], hence the turbulence must be considered when we are studying a high Reynolds number flow.

---

[1] Courant, Friedrichs and Lewy

[2] Laminar flows are characterized by smoothly varying velocity fields in space and time in which individual “laminae” (sheets) move past one another without generating

The equations governing a turbulent flow are, as for the laminar flows, the Navier-Stokes equations. The problem is that - in this regime - the solution is much more complicated (due to fluctuations of the variable we want to evaluate). The first idea is to solve the equations with the Direct Numerical Simulation (DNS). This means that we resolve all the spatial and temporal scales. This offer an accurate results, but it is very expensive, computationally speaking.

The alternative to DNS are the Reynolds Averaged Navier Stokes (RANS) equations. They care about the mean velocity and pressure. But models are needed, in order to close the problem. And the key-point is to be able to model the small scales properly. There are myriads of turbulence models. And each *CFD* solver allows us to choose the most appropriate model.

The most used is the $k - \epsilon$ model: this mean that we model two turbulen parameters, i.e. the turbulent kinetic energy and the turbulent energy dissipation.

Since this appendix wants to be nothing but an introduction, we recommend the interested reader to refer to [10], [18], [32], [16] and [35].

---

cross currents. These flows arise when the fluid viscosity is sufficiently large to damp out any perturbations to the flow that may occur due to boundary imperfections or other irregularities. These flows occur at low-to-moderate values of the Reynolds number.

B. The implemented code

In this chapter we list all the files needed to run the tool we have developed. This is a very technical appendix and it is addressed to people who want to develop and study what there is under the hood.

B.1 *vmtkgui.py*

This is the “main” file and it contains all the links between the different files.

```
## @package vmtkgui
#  This is the main module. To be a good main, it needs to be short and clear: that is why it simply calls the functions we need. All the functions are contained in different modules.
# At the end of the module, there is the execution of the module.

#!/usr/bin/python

try:
        import gtk
        import gtk.glade
except:
        sys.exit(1)

# from vmtk import pypes

import vmtkinterface
import vtkcones
import math
import canvas3D

from gtk import gdk
import vtk
from vtk import *

## This class contains a certain number of functions that are invoked when a button is clicked. And the target of this functions is simply to call the corresponding function in another file. The functions in the new file (see vmtkinterface.py) do what the users wants to do clicking on the button.
class VmtkGui:

```

```
    ## The constructor.
    #  @param self The object pointer
    def __init__(self): # (original version)


        #Set the Glade file
        self.gladefile = "gui.glade"

        #This is the command to open my GUI
        self.wTree = gtk.glade.XML(self.gladefile) #(original version)

        #Create our dictionay and  connect it
        dic = { "on_name_in_clicked" : self.call_namein,
                "on_name_out_clicked" : self.call_nameout,
                "on_file_view_clicked" : self.call_fileview,
                "on_patient_clicked" : self.call_patient,
                "on_load_DICOM_clicked" : self.call_loadDICOM,
                "on_img_viewer_clicked" : self.call_imageviewer,
                "on_VOI_clicked" : self.call_VOI,
                "on_Lev_Set_Seg_clicked" : self.call_LSS,
                "on_surf_gen_clicked" : self.call_SurfGen,
                "on_sm_sur_clicked" : self.call_SmoothSurf,
                "on_gensm_sur_clicked" : self.call_GenSmoothSurf,
                "on_surf_clipp_clicked" : self.call_SurfClipp,
                "on_CenterLines_clicked" : self.call_CenterLines,
                "on_Mesh_Gen_clicked" : self.call_MeshGen,
                "on_viewimg_clicked" : self.call_viewimg,
                "on_viewsurf_clicked" : self.call_viewsurf,
                "on_viewmesh_clicked" : self.call_viewmesh,
                "on_readviewimg_clicked" : self.call_readviewimg,
                "on_readviewsurf_clicked" : self.call_readviewsurf,
                "on_readviewmsh_clicked" : self.call_readviewmsh,
                "on_setvalues_clicked" : self.call_setvalues,
                "on_MainWindow_destroy" : gtk.main_quit,
                "on_imagemenuitem5_destroy" : gtk.main_quit}


        self.wTree.signal_autoconnect(dic)

        # defining new Canvas
        self.vtkda =canvas3D.Canvas3D(None)
        self.vtkda.show()

        # adding Canvas to gtk
        hbox = gtk.HBox(spacing=5)
        hbox.show()
        hbox.pack_start(self.vtkda)

        # setting window to widget
        self.window = self.wTree.get_widget("hbox2")
        self.window.connect("destroy", gtk.main_quit)
        self.window.connect("delete_event", gtk.main_quit)
        self.window.add(hbox)
```

```
        self.window.set_size_request(800, 600)
        # self.window.show()

        self.ren = vtk.vtkRenderer()
        self.ren.SetBackground(.95, .95, .95)

        self.vtkda.GetRenderWindow().AddRenderer(self.ren)

        # define instance of  VmtkInterfacevtkWindow
        # self.vtkcones = vtkcones.vtkWindow(self.wTree, self.ren)
        self.vtkcones = vtkcones.vtkWindow(self.ren)

        # define instance of  VmtkInterface
        self.vmtkinterface = vmtkinterface.VmtkInterface("img",self.ren, self.vtkda)

        self.vmtkinterface.input_filename = None

    ## Function used to read the name of the patient
    # @param widget gdk widget
    def call_patient (self, wTree):
        self.vmtkinterface.read_patient(self.wTree)

    ## Function used to read the name of the file we want to use
    # @param widget gdk widget
    def call_namein (self, wTree):
        self.vmtkinterface.read_in_name(self.wTree)

    ## Function used to read the name of the file we want to use
    # @param widget gdk widget
    def call_nameout (self, wTree):
        self.vmtkinterface.read_out_name(self.wTree)

    ## Function used to read the name of the file we want to use
    # @param widget gdk widget
    def call_fileview (self, wTree):
        self.vmtkinterface.file_to_view(self.wTree)

    ## Function used to read the name of the file we want to use
    # @param widget gdk widget
    def call_setvalues (self, wTree):
        self.vmtkinterface.setvalues(self.wTree)

    ## Function used to load a DICOM file and to convert the DICOM in a proper format for vmtk
    # @param self The object pointer
    # @param widget gdk widget
    def call_loadDICOM (self, widget):

        # if not self.vmtkinterface.input_filename:
        #     self.vmtkinterface.input_filename = "example"
        # if not self.vmtkinterface.output_filename:
        #     self.vmtkinterface.output_filename = self.vmtkinterface.input_filename

        # if not self.vmtkinterface.input_filename:
```

```
         #    print "Insert␣the␣name␣of␣the␣file␣that␣you␣want␣to␣load"
        # else:
        self.vmtkinterface.loadDICOM(self.vmtkinterface.input_filename, self.vmtkinterface.output_filename)

    ## Function used to view the image (after havindg loaded the DICOM file and converted it in a more suitable format)
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_imageviewer (self, widget):
        # if not self.vmtkinterface.input_filename:
        #      print "Insert␣the␣name␣of␣the␣file␣that␣you␣want␣to␣load"
        # else:
        self.vmtkinterface.test = 1
        self.vmtkinterface.imagereader(self.vmtkinterface.input_filename)
        self.vmtkinterface.imageviewer()

    ## Function used to view the image (after havindg loaded the DICOM file and converted it in a more suitable format)
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_readviewimg(self, widget):
        self.vmtkinterface.flag = 0
        self.vmtkinterface.read_view()

    ## Function used to view the image (after havindg loaded the DICOM file and converted it in a more suitable format)
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_readviewsurf(self, widget):
        self.vmtkinterface.flag = 1
        self.vmtkinterface.read_view()

    ## Function used to view the image (after havindg loaded the DICOM file and converted it in a more suitable format)
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_readviewmsh(self, widget):
        self.vmtkinterface.flag = 2
        self.vmtkinterface.read_view()

    ## Function the allows the user to exctract a Volume of Interest
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_VOI (self, widget):
      self.vmtkinterface.VOI(self.vmtkinterface.output_filename)

    ## This function just performs a Level Set Segmentation, i.e. it adjusts contrast and so on to have a better generated surface
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_LSS (self, widget):
      self.vmtkinterface.LevSetSeg(self.vmtkinterface.output_filename)

```

```
  ## This is the function to create a surface from the DICOM file , in order to be able to compute it easily
  # @param self The object pointer
  # @param widget gdk widget
  def call_SurfGen (self , widget):
    self.vmtkinterface.SurfGen(self.vmtkinterface.output_filename)

  ## This is the command we need in order to smooth the surface.
  # @param self The object pointer
  # @param widget gdk widget
  def call_SmoothSurf (self , widget):
    self.vmtkinterface.SmoothSurf(self.vmtkinterface.output_filename)

  ## This is the command we need in order to smooth the surface.
  # @param self The object pointer
  # @param widget gdk widget
  def call_GenSmoothSurf (self , widget):
    self.vmtkinterface.GenSmoothSurf(self.vmtkinterface.output_filename)

  ## This command allows us to clip the volume. This is used to have a better top/bottom surface and to make it easier to compute the normal
  # @param self The object pointer
  # @param widget gdk widget
  def call_SurfClipp (self , widget):
    self.vmtkinterface.SurfClipp(self.vmtkinterface.output_filename)

  ## Function used to compute the centerlines
  # @param self The object pointer
  # @param widget gdk widget
  def call_CenterLines (self , widget):
    self.vmtkinterface.CenterLines(self.vmtkinterface.output_filename)

  ## Function needed to generate a mesh (if needed :-) )
  # @param self The object pointer
  # @param widget gdk widget
  def call_MeshGen (self , widget):
    self.vmtkinterface.MeshGen(self.vmtkinterface.output_filename)


  ## Function used to load a DICOM file
  # @param self The object pointer
  # @param widget gdk widget
  def call_dicom_path (self , widget):
    print "Trying to load the DICOM file"
    # planeWidgetX.InteractionOff()
    # planeWidgetY.InteractionOff()
    # planeWidgetZ.InteractionOff()
    # interactorStyle.EnabledOn()

  ## This is the command we need in order to smooth the surface.
  # @param self The object pointer
  # @param widget gdk widget
  def call_viewimg (self , widget):
    self.vmtkinterface.imageviewer()

  ## This is the command we need in order to smooth the surface.
  # @param self The object pointer
```

```
    #  @param widget gdk widget
    def call_viewsurf (self, widget):
      self.vmtkinterface.surfaceviewer()

    ## This is the command we need in order to smooth the surface.
    #  @param self The object pointer
    #  @param widget gdk widget
    def call_viewmesh (self, widget):
        print "Il␣bottone␣funziona!"
        self.vmtkinterface.meshviewer()

## This is tha calling to the main. We need this calling to launch and execute the code.
if __name__ == "__main__":

    hwg = VmtkGui()
    gtk.main()
```

B.2 *vmtkinterface.py*

In this file, as the name could suggest, we create and set all the stuff that deal with the *GUI*.

```
## @package vmtkinterface
#  This is the module that recall and uses the vmtk functions (see online documentation @ http://www.vmtk.org) that we want to "perform" when we click on a certain button.
#!/usr/bin/python

try:
        import gtk
        import gtk.glade
except:
        sys.exit(1)

import vtk
from vtk import *

import vmtk
from vmtk import *

import vtkcones
import vmtkgui
import vmtkcenterlineswithrenderer

from gtk import gdk
import math


## This class contains the vmtk functions needed to load a DICOM, to make it readable from vmtk, to act on the image itself and to do what we need for our purpose.
class VmtkInterface:

    ## The constructor.
    #  @param self The object pointer.
    #  @param img the folder where the DICOM serie is contained.
    #  @param ren It is the rendere I am going to use both for the valvestent and for the image.
    #  @param vtkda It is the canvas we want to use for all our objects.
    def __init__(self, img, ren, vtkda):
        # Preliminaries: initialization of some global variables
        self.subdir         = "/home/samiam/Documents/TESI/gui_python/"
        # self.subdir         = "/usr/scratch/zampini/vmtk/working_space/patients/"
        self.vtkda          = vtkda
        self.ren            = ren
        self.ImgVOI         = None
        self.Surface        = None
        self.Image          = None
        self.Mesh           = None
        self.init_type      = 0
        self.thres_lower    = 100
        self.thres_upper    = 600
```

```
        self.number_iteration = 300
        self.prop_scaling     = 0.0
        self.curve_scaling    = 0.0
        self.adv_scaling      = 1.0
        self.vmtkRenderer     = vmtkrenderer.vmtkRenderer()
        self.flag             = 0
        self.first_time       = 1
        # self.test             = 1
        # Comment or uncomment (it depends on which renderer
            we want to use - usually the following line must
            be commented)
        # self.vmtkRenderer.Initialize()
        # setting RenderWindow to the already existing one
            in vtkda (it enteres in the following 'if', only
            when I have not defined the renderer yet)
        if not self.vmtkRenderer.Renderer:
                self.vmtkRenderer.Renderer = self.ren
                self.vmtkRenderer.RenderWindow =
                    self.vtkda.GetRenderWindow()
                self.vmtkRenderer.RenderWindow.AddRenderer(self.vmtkRenderer.Ren
                self.vmtkRenderer.RenderWindow.SetSize(self.vmtkRenderer.Window
                        self.vmtkRenderer.WindowSize[1])
                self.vmtkRenderer.RenderWindow.SetPointSmoothing\
                (self.vmtkRenderer.PointSmoothing)
                self.vmtkRenderer.RenderWindow.SetLineSmoothing\
                (self.vmtkRenderer.LineSmoothing)
                self.vmtkRenderer.RenderWindow.SetPolygonSmoothing\
                (self.vmtkRenderer.PolygonSmoothing)
                # Note that if I use the
                    vtkGenericRenderWindowInteractor() I get
                    image i the right window but I can not
                    interact with it, while if I use the
                    vtkRenderWindowInteractor() I get an
                    image I can "elaborate" as I like, but
                    it does not appear in the right window.
                self.vmtkRenderer.RenderWindowInteractor =
                    self.vtkda.GetRenderWindow().GetInteractor()
                self.vmtkRenderer.RenderWindowInteractor =
                    vtkRenderWindowInteractor()
                # self.vmtkRenderer.RenderWindowInteractor =
                    vtkGenericRenderWindowInteractor()
                self.vmtkRenderer.RenderWindowInteractor.SetRenderWindow\
                (self.vtkda.GetRenderWindow())
                self.vmtkRenderer.RenderWindowInteractor.SetInteractorStyle\
                (vtkInteractorStyleTrackballCamera())
                self.vmtkRenderer.RenderWindow.SetInteractor\
                (self.vmtkRenderer.RenderWindowInteractor)
                self.vmtkRenderer.RenderWindowInteractor.Initialize()

        self.vmtkImageViewer   =
            vmtkimageviewer.vmtkImageViewer()
        self.vmtkSurfaceViewer =
            vmtksurfaceviewer.vmtkSurfaceViewer()
        self.vmtkMeshViewer    =
            vmtkmeshviewer.vmtkMeshViewer()

        self.vmtkImageViewer.vmtkRenderer   =
            self.vmtkRenderer
        self.vmtkSurfaceViewer.vmtkRenderer =
            self.vmtkRenderer
        self.vmtkMeshViewer.vmtkRenderer    =
            self.vmtkRenderer
```

```

        self.vmtkSurfaceWriter = vmtksurfacewriter.vmtkSurfaceWriter()
        self.vmtkMeshWriter    = vmtkmeshwriter.vmtkMeshWriter()

        self.vmtkImageVOISelector = None

        self.input_filename  = "aorta"
        self.output_filename = "aorta"
        self.patient         = "bianchi"


    ## Function to read the name of the input file
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def read_patient(self, wTree):
        self.wTree = wTree
        self.patient_name = self.wTree.get_widget("patient")
        self.patient = self.patient_name.get_text()
        print "␣"
        print "The␣patient␣is␣Mr(s)␣" + self.patient
        print "If␣this␣is␣the␣right␣patient,␣go␣ahead.␣Otherwise␣re-type␣the␣name!"
        print "␣"

    ## Function to read the name of the input file
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def read_in_name(self, wTree):
        self.wTree = wTree
        self.in_file_name = self.wTree.get_widget("in_file_name")
        self.input_filename = self.in_file_name.get_text()
        self.output_filename = self.input_filename
        print "␣"
        print "You␣have␣selected␣the␣file␣" + self.input_filename + ".vti"
        print "If␣this␣is␣the␣right␣choice,␣go␣ahead.␣Otherwise␣re-type␣the␣name␣of␣the␣file␣that␣you␣want␣to␣load"

    ## Function to read the name of the input file
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def read_out_name(self, wTree):
        """This␣function␣will␣read␣the␣name␣of␣the␣file␣we␣want␣to␣save"""
        self.wTree = wTree
        self.out_file_name = self.wTree.get_widget("out_file_name")
        self.output_filename = self.out_file_name.get_text()
        print "␣"
        print "The␣output␣file␣will␣be␣named␣" + self.output_filename + ".vti"
        print "Is␣this␣name␣ok?"

    ## Function to read the name of the file that you want to do
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def file_to_view(self, wTree):
```

```
        self.wTree = wTree
        self.file_view = self.wTree.get_widget("file_view")
        self.filetoview = self.file_view.get_text()
        print "␣"
        print "You␣chose␣to␣visualize␣the␣file␣" + self.filetoview + ".␣If␣it␣is␣correct,␣continue.␣If␣not,␣re-type"
        self.test = 0

    ## Function to read the name of the file that you want to do
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def read_view(self):

        if self.flag == 0:
            # print "Read␣&␣View␣an␣image"
            self.imagereader(self.filetoview[:-4])
            self.imageviewer()
            print "Well␣done.␣Select␣a␣new␣action!"
        elif self.flag == 1:
            # print "Leggi/Vedi␣una␣superficie"
            self.surfacereader(self.filetoview[:-4])
            self.surfaceviewer()
            print "Well␣done.␣Select␣a␣new␣action!"
        elif self.flag == 2:
            # print "Leggi/Vedi␣una␣mesh"
            self.meshreader(self.filetoview[:-4])
            self.meshviewer()
            print "Well␣done.␣Select␣a␣new␣action!"
        else:
            print "File␣not␣supported.␣Check␣the␣name␣of␣the␣file␣please!"




    ## Function to assign certain values (chosen by the user)
    #  @param self The object pointer
    #  @param wTree The dictionary where I get my information
    def setvalues(self, wTree):
        """This␣function␣will␣read␣the␣name␣of␣the␣file␣we␣want␣to␣save"""
        self.wTree = wTree
        self.num_iter = self.wTree.get_widget("num_iter")
        self.number_iteration_str = self.num_iter.get_text()
        self.number_iteration = int(self.number_iteration_str)
        self.prop_sc = self.wTree.get_widget("prop_sc")
        self.prop_scaling_str = self.prop_sc.get_text()
        self.prop_scaling = float(self.prop_scaling_str)
        self.curve_sc = self.wTree.get_widget("curve_sc")
        self.curve_scaling_str = self.curve_sc.get_text()
        self.curve_scaling = float(self.curve_scaling_str)
        self.adv_sc = self.wTree.get_widget("adv_sc")
        self.adv_scaling_str = self.adv_sc.get_text()
        self.adv_scaling = float(self.adv_scaling_str)
        print ""
        print "The␣values␣has␣been␣assigned"
        print ""

    ## function used to clean the renderer
```

```
    #  @param self The object pointer
    def clearActors(self):
        # clearing renderer
        if self.vmtkSurfaceViewer.Actor:
            self.vmtkRenderer.Renderer.RemoveActor\
                (self.vmtkSurfaceViewer.Actor)
        if self.vmtkMeshViewer.Actor:
            self.vmtkRenderer.Renderer.RemoveActor\
                (self.vmtkMeshViewer.Actor)
        if self.vmtkImageViewer.PlaneWidgetX:
            self.vmtkImageViewer.PlaneWidgetX.Off()
        if self.vmtkImageViewer.PlaneWidgetY:
            self.vmtkImageViewer.PlaneWidgetY.Off()
        if self.vmtkImageViewer.PlaneWidgetZ:
            self.vmtkImageViewer.PlaneWidgetZ.Off()
        if self.vmtkImageVOISelector:
            self.vmtkImageVOISelector.PlaneWidgetX.Off()
            self.vmtkImageVOISelector.PlaneWidgetY.Off()
            self.vmtkImageVOISelector.PlaneWidgetZ.Off()

    ## Function that allows us to load a DICOM and to write a file in the right format
    #  @param self The object pointer
    #  @param input_filename This is the name of the file that we want to load
    #  @param input_filename This is the name of the file that we want to create
    def loadDICOM(self, input_filename, output_filename):
        print "E' la prima volta? [1 - si; 0 - no.]"
        print self.first_time
        if self.first_time == 0:
            self.ren.RemoveAllViewProps()
        self.first_time = 0
        # dicom_dir = "/usr/scratch/zampini/vmtk/working_space/coronaries/"
        vmtkImageReader = vmtkimagereader.vmtkImageReader()
        vmtkImageReader.Format = "dicom"
        # vmtkImageReader.InputDirectoryName = dicom_dir
        vmtkImageReader.InputDirectoryName = self.subdir + self.patient + "/DICOM/"
        vmtkImageReader.Execute()
        print " "
        print "DICOM loaded successfully!"
        print " "
        self.Image =  vmtkImageReader.Image
        self.imagewriter(output_filename + ".vti")
        print " "
        print "File " + output_filename + ".vti has been written. Please, go ahead."
        print " "
        self.test = 1

    ## function that allows us to view the .vti image just created
    #  @param self The object pointer
    #  @param input_filename The name of the .vti file we want to view (inserted from the user)
    def imagereader(self, output_filename):
        print "self.test"
        print self.test
        print "output_filename" + output_filename
        # if self.test == 0:
```

```
        #     output_filename =  self.filetoview[:-4]
        vmtkImageReader = vmtkimagereader.vmtkImageReader()
        vmtkImageReader.InputFileName = self.subdir + self.patient + "/img/" + output_filename + ".vti"
        vmtkImageReader.Execute()
        self.Image = vmtkImageReader.Image
        # self.imageviewer()

    ## function that execute the function 'imagereader'
    # @param self The object pointer
    def imageviewer(self):
        self.clearActors()
        # print self.flag
        # if self.flag == 0:
        self.vmtkImageViewer.Image = self.Image
        print "␣"
        print "Press␣'q'␣to␣continue"
        print "␣"
        self.vmtkImageViewer.Execute()
        # elif self.flag == 1:
        #     self.vmtkImageViewer.ArrayName = self.filetoview
        #     print "␣"
        #     print "Press␣'q'␣to␣continue"
        #     print "␣"
        #     self.vmtkImageViewer.Execute()


    ## function that execute write a vtk-xml file
    # @param self The object pointer
    # @param output_filename This is the name of the file we have created
    def imagewriter(self, output_filename):
        vmtkImageWriter = vmtkimagewriter.vmtkImageWriter()
        vmtkImageWriter.Image = self.Image
        vmtkImageWriter.OutputFileName = self.subdir + self.patient + "/img/" + output_filename
        self.Image = vmtkImageWriter.Image
        vmtkImageWriter.Execute()

    ## function that allows us to view the .vti image just created
    # @param self The object pointer
    # @param input_filename The name of the .vti file we want to view (inserted from the user)
    def surfacereader(self, output_filename):
        print "self.test"
        print self.test
        print "output_filename" + output_filename
        # if  self.test == 0:
        #    output_filename =  self.filetoview[:-4]
        vmtkSurfaceReader = vmtksurfacereader.vmtkSurfaceReader()
        vmtkSurfaceReader.InputFileName = self.subdir + self.patient + "/img/" + output_filename + ".vtp"
        vmtkSurfaceReader.Execute()
        self.Surface = vmtkSurfaceReader.Surface

    ## function that execute the function 'imagereader'
    # @param self The object pointer
    def surfaceviewer(self):
        self.clearActors()
```

```
        self.vmtkSurfaceViewer.Surface = self.Surface
        print "␣"
        print "Press␣'q'␣to␣continue"
        print "␣"
        self.vmtkSurfaceViewer.Execute()


    ## function that allow to view a vtk-xml file
    #  @param self The object pointer
    #  @param output_filename This is the name of the file we have created
    def surfacewriter(self, output_filename):
        vmtkSurfaceWriter = vmtksurfacewriter.vmtkSurfaceWriter()
        vmtkSurfaceWriter.Surface = self.Surface
        vmtkSurfaceWriter.OutputFileName = self.subdir + self.patient + "/img/" + output_filename
        self.Surface = vmtkSurfaceWriter.Surface
        vmtkSurfaceWriter.Execute()

    ## function that allows us to view the .vti image just created
    #  @param self The object pointer
    #  @param input_filename The name of the .vti file we want to view (inserted from the user)
    def meshreader(self, output_filename):
        print "self.test"
        print self.test
        print "output_filename" + output_filename
        # if  self.test == 0:
        #      output_filename =  self.filetoview[:-4]
        vmtkMeshReader = vmtkmeshreader.vmtkMeshReader()
        # print self.subdir + self.patient + "/img/" + output_filename + ".vtu"
        vmtkMeshReader.InputFileName = self.subdir + self.patient + "/img/" + output_filename + ".vtu"
        vmtkMeshReader.Execute()
        self.Mesh = vmtkMeshReader.Mesh
        # self.imageviewer()

    ## function that allows to view the vtk-xml file
    #  @param self The object pointer
    def meshviewer(self):
        self.clearActors()
        self.vmtkMeshViewer.Mesh = self.Mesh
        print "␣"
        print "Press␣'q'␣to␣continue"
        print "␣"
        # self.vmtkMeshViewer.Renderer = self.vmtkRenderer
        self.vmtkMeshViewer.Execute()

    ## function that allow to view a vtk-xml file
    #  @param self The object pointer
    #  @param output_filename This is the name of the file we have created
    def meshwriter(self, output_filename):
        vmtkMeshWriter = vmtkmeshwriter.vmtkMeshWriter()
        vmtkMeshWriter.Mesh = self.Mesh
        vmtkMeshWriter.OutputFileName = self.subdir + self.patient + "/img/"  + output_filename + "_mesh.vtu"
        vmtkMeshWriter.Execute()
```

```

    ## function to check if the file has been loaded
    # @param self The object pointer
    def DICOMinteract(self):
        if vmtkImageViewer:
                vmtkImageViewer.Execute()
        else:
                print "Image␣not␣loaded␣yet"

    ## function that allows us to select a volume of interest
    # @param self The object pointer
    # @param output_filename This is the name of the file we have created
    def VOI(self, output_filename):
        self.clearActors()
        self.vmtkImageVOISelector = vmtkimagevoiselector.vmtkImageVOISelector()
        self.vmtkImageVOISelector.Image = self.Image
        self.vmtkImageVOISelector.vmtkRenderer = self.vmtkRenderer
        self.vmtkImageVOISelector.Interactive = 1
        print "Press␣i␣to␣activate␣the␣volume␣selector␣and␣q␣(twice)␣when␣you␣have␣done"
        print "␣"
        self.vmtkImageVOISelector.Execute()
        self.Image = self.vmtkImageVOISelector.Image
        self.imagewriter(output_filename + "_VoI.vti")
        print "␣"
        print "File␣" + output_filename + "_VoI.vti␣has␣been␣written.␣Please␣select␣a␣new␣action␣or␣exit␣the␣program"
        print "␣"

    ## function that allows us to do a level set segmentation: it is important to underline that if we set a Number of Iteration higher than 0, then it will perform only one loop for the levelset (i.e. you can add only a branch). And we want to point out that it's␣not␣a␣limitation␣if␣you␣think␣about␣the␣final␣target␣of␣the␣whole␣project,␣since␣you␣do␣not␣need␣to␣use␣bifurcation␣in␣order␣to␣reconstruct␣the␣area␣around␣te␣mitral-valve.
␣␣␣␣#␣␣@param␣self␣The␣object␣pointer
␣␣␣␣#␣␣@param␣output_filename␣This␣is␣the␣name␣of␣the␣file␣we␣have␣created
␣␣␣␣def␣LevSetSeg(self,␣output_filename):
␣␣␣␣␣␣␣␣self.clearActors()
␣␣␣␣␣␣␣␣if␣not␣self.Image:
␣␣␣␣␣␣␣␣␣␣␣␣self.imagereader(output_filename␣+␣"_VoI")
␣␣␣␣␣␣␣␣vmtkLevelSetSegmentation␣=␣vmtklevelsetsegmentation.vmtkLevelSetSegmentation()

␣␣␣␣␣␣␣␣vmtkLevelSetSegmentation.PropagationScaling␣=␣self.prop_scaling
␣␣␣␣␣␣␣␣vmtkLevelSetSegmentation.CurvatureScaling␣=␣self.curve_scaling
␣␣␣␣␣␣␣␣vmtkLevelSetSegmentation.AdvectionScaling␣=␣self.adv_scaling

␣␣␣␣␣␣␣␣##␣if␣self.number_iteration␣>␣0␣we␣only␣do␣one␣loop␣for␣levelset
␣␣␣␣␣␣␣␣if␣self.number_iteration␣>␣0:
```

```
            ImageSeeder = vmtkscripts.vmtkImageSeeder()
            ImageSeeder.vmtkRenderer = self.vmtkRenderer
            ImageSeeder.Image = self.Image
            ImageSeeder.Display = 0
            ImageSeeder.Execute()
            ImageSeeder.Display = 1
            ImageSeeder.BuildView()

            vmtkImageInitialization = vmtkscripts.vmtkImageInitialization()
            vmtkImageInitialization.Image         = self.Image
            vmtkImageInitialization.vmtkRenderer  = self.vmtkRenderer

            vmtkImageInitialization.ImageSeeder   = ImageSeeder
            vmtkImageInitialization.SurfaceViewer = self.vmtkSurfaceViewer
            vmtkImageInitialization.OwnRenderer   = 0

            vmtkImageInitialization.Execute()

            vmtkLevelSetSegmentation.InitialLevelSets   = vmtkImageInitialization.InitialLevelSets
            vmtkLevelSetSegmentation.NumberOfIterations = self.number_iteration
        ##


        vmtkLevelSetSegmentation.Image = self.Image
        vmtkLevelSetSegmentation.vmtkRenderer = self.vmtkRenderer
        vmtkLevelSetSegmentation.Execute()
        self.Image = vmtkLevelSetSegmentation.LevelSets
        self.imagewriter(output_filename + "_VoI_LSS.vti")
        print " "
        print "File " + output_filename  + "_VoI_LSS.vti has been successfully written. Please go ahead."
        print " "

    ## function that allows us to generate a surface from the file we have just written
    #  @param self The object pointer
    #  @param output_filename This is the name of the file we have created
    def SurfGen(self, output_filename):
        self.clearActors()
        #if not self.Image:
        self.imagereader(output_filename + "_VoI_LSS")
        vmtkMarchingCubes = vmtkmarchingcubes.vmtkMarchingCubes()
        vmtkMarchingCubes.Image = self.Image
        vmtkMarchingCubes.Execute()
        self.Surface = vmtkMarchingCubes.Surface
        self.surfacewriter(output_filename + "_VoI_LSS_mc.vtp")
        print " "
        print "File " + output_filename  + "_VoI_LSS_mc.vtp has been successfully written. (Now press ""q"" to continue)"
        print " "
```

```
        self.surfaceviewer()
        print " "
        print "The surface is now displayed. Please go ahead (select a new action or exit the program)."
        print " "
        print "The file has been written! Choose a new action!"

    ## This function is used to make the surface we have generated smoother
    #  @param self The object pointer
    #  @param output_filename This is the name of the file we have created
    def SmoothSurf(self, output_filename):
        vmtkSurfaceSmoothing = vmtksurfacesmoothing.vmtkSurfaceSmoothing()
        vmtkSurfaceSmoothing.NumberOfIterations = 100
        vmtkSurfaceSmoothing.PassBand = 0.01
        vmtkSurfaceSmoothing.Surface = self.Surface
        vmtkSurfaceSmoothing.vmtkRenderer = self.vmtkRenderer
        vmtkSurfaceSmoothing.Execute()
        self.surfacewriter(output_filename + "_VoI_LSS_mc_sm.vtp")
        print " "
        print "File " + output_filename  + "_VoI_LSS_mc_sm.vtp has been successfully written."
        print " "
        self.Surface = vmtkSurfaceSmoothing.Surface
        self.surfaceviewer()
        print " "
        print "The surface is now displayed. Please go ahead (select a new action or exit the program)."
        print " "

    ## This function is used to make the surface we have generated smoother
    #  @param self The object pointer
    #  @param output_filename This is the name of the file we have created
    def GenSmoothSurf(self, output_filename):
        self.SurfGen(output_filename)
        self.SmoothSurf(output_filename)

    ## This function is used to make the surface we generater smoother
    #  @param self The object pointer
    #  @param input_filename This is the name of the file we have created
    def SurfClipp(self, output_filename):
        vmtkSurfaceClipper = vmtksurfaceclipper.vmtkSurfaceClipper()
        vmtkSurfaceClipper.Surface = self.Surface
        vmtkSurfaceClipper.vmtkRenderer = self.vmtkRenderer
        vmtkSurfaceClipper.Execute()
        print "Executed"
        self.Surface = vmtkSurfaceClipper.Surface
        self.surfacewriter(output_filename + "_VoI_LSS_mc_sm_cl.vtp")
        print " "
        print "File " + output_filename  + "_VoI_LSS_mc_sm_cl.vtp has been successfully written."
        print " "
```

```
␣␣␣␣␣␣␣␣self.Surface␣=␣vmtkSurfaceClipper.Surface
␣␣␣␣␣␣␣␣self.surfaceviewer()
␣␣␣␣␣␣␣␣print␣"␣"
␣␣␣␣␣␣␣␣print␣"The␣surface␣is␣now␣displayed.␣Please␣go␣ahead␣(select␣a␣new␣action␣or␣exit␣the␣program)."
␣␣␣␣␣␣␣␣print␣"␣"

␣␣␣␣##␣This␣function␣is␣used␣to␣compute␣the␣centerlines
␣␣␣␣#␣␣@param␣self␣The␣object␣pointer
␣␣␣␣#␣␣@param␣input_filename␣This␣is␣the␣name␣of␣the␣file␣we␣have␣created
␣␣␣␣def␣CenterLines(self,␣output_filename):
␣␣␣␣␣␣␣␣vmtkCenterlines␣=␣vmtkcenterlineswithrenderer.vmtkCenterlinesWithRenderer()
␣␣␣␣␣␣␣␣vmtkCenterlines.Surface␣=␣self.Surface
␣␣␣␣␣␣␣␣vmtkCenterlines.vmtkRenderer␣=␣self.vmtkRenderer
␣␣␣␣␣␣␣␣vmtkCenterlines.SeedSelectorName␣=␣'openprofiles'
␣␣␣␣␣␣␣␣vmtkCenterlines.AppendEndPoints␣=␣1
␣␣␣␣␣␣␣␣vmtkCenterlines.Execute()
␣␣␣␣␣␣␣␣print␣"The␣centerline␣has␣been␣computed"
␣␣␣␣␣␣␣␣self.Surface␣=␣vmtkCenterlines.Surface
␣␣␣␣␣␣␣␣self.surfacewriter(output_filename␣+␣"_VoI_LSS_mc_sm_cl_clns.vtp")
␣␣␣␣␣␣␣␣print␣"␣"
␣␣␣␣␣␣␣␣print␣"File␣"␣+␣output_filename␣␣+␣"_VoI_LSS_mc_sm_cl_clns.vtp␣has␣been␣successfully␣written.␣Please␣go␣ahead."
␣␣␣␣␣␣␣␣print␣"␣"

␣␣␣␣##␣function␣that␣allows␣us␣to␣do␣a␣level␣set␣segmentationgenerate␣a␣surface␣from␣the␣file
␣␣␣␣#␣␣@param␣self␣The␣object␣pointer
␣␣␣␣#␣␣@param␣input_filename␣This␣is␣the␣name␣of␣the␣file␣we␣have␣created
␣␣␣␣def␣MeshGen(self,␣output_filename):
␣␣␣␣␣␣␣␣self.clearActors()
␣␣␣␣␣␣␣␣vmtkMeshGenerator␣=␣vmtkmeshgenerator.vmtkMeshGenerator()
␣␣␣␣␣␣␣␣vmtkMeshGenerator.Surface␣=␣self.Surface
␣␣␣␣␣␣␣␣vmtkMeshGenerator.Mesh␣=␣self.Mesh
␣␣␣␣␣␣␣␣vmtkMeshGenerator.edgelength␣=␣3
␣␣␣␣␣␣␣␣vmtkMeshGenerator.vmtkRenderer␣=␣self.vmtkRenderer
␣␣␣␣␣␣␣␣vmtkMeshGenerator.Execute()
␣␣␣␣␣␣␣␣self.Mesh␣=␣vmtkMeshGenerator.Mesh
␣␣␣␣␣␣␣␣self.meshwriter(output_filename)
␣␣␣␣␣␣␣␣print␣"Well␣done,␣the␣mesh␣has␣been␣written!"
␣␣␣␣␣␣␣␣self.meshviewer()
␣␣␣␣␣␣␣␣print␣"The␣mesh␣is␣now␣displyed!"
```

B.3 *canvas3D.py*

This is another "graphical" file that helps to prepare and set the canvas where we represent the scene we are studying.

```
## @package canvas3D
#In this package we have 2 classes that create a drawing area where we want to represent the stent(s) and the DICOM image. Also, there are a serie of events we can use to enjoy the mouse interaction with what we represent in the drawing area.

try:
    import gtk
    import gtk.glade
except:
    sys.exit(1)

from gtk import gdk
import vtk

import vtkcones

import math

## This class contains the basic functions needed to create a suitable drawing area able to host our images.
class Canvas3DBase(gtk.DrawingArea):

    ## The constructor
    # @param self The object pointer
    # @param *args The special syntax, *args in function definitions is used to pass a variable number of arguments to a function. The single asterisk form (*args) is used to pass a non-keyworded, variable-length argument list.
    def __init__(self, *args):

        gtk.DrawingArea.__init__(self)
        self._RenderWindow = vtk.vtkRenderWindow()

        # private attributes
        self.__Created = 0
        # used by the LOD actors
        self._DesiredUpdateRate = 15
        self._StillUpdateRate = 0.0001
    self.ConnectSignals0()
        # need this to be able to handle key_press events.
        self.set_flags(gtk.CAN_FOCUS)
        # default size
        self.set_size_request(300, 300)

    ## The connections beetween an action and its consequences
    # @param self The object pointer
    def ConnectSignals0(self):

```

```
        self.MouseMoveConnected = None

                self.connect("realize", self.OnRealize)
                self.connect("expose_event", self.OnExpose)
                self.connect("configure_event", self.OnConfigure)
                self.connect("button_press_event", self.OnButtonDown)
                self.connect("button_release_event", self.OnButtonUp)

                self.connect("enter_notify_event", self.OnEnter)
                self.connect("leave_notify_event", self.OnLeave)
                self.connect("key_press_event", self.OnKeyPress)
                self.connect("delete_event", self.OnDestroy)
                self.add_events(gdk.EXPOSURE_MASK |
                                               gdk.BUTTON_PRESS_MASK |
                                               gdk.BUTTON_RELEASE_MASK |
                                               gdk.KEY_PRESS_MASK |
                                               gdk.POINTER_MOTION_MASK |
                                               gdk.POINTER_MOTION_HINT_MASK |
                                               gdk.ENTER_NOTIFY_MASK |
                                               gdk.LEAVE_NOTIFY_MASK)


        ## The connections beetween an action and its consequences
        #  @param self The object pointer
        def ConnectSignals(self):
                if self.MouseMoveConnected:
              self.disconnect(self.MouseMoveConnected)
        self.MouseMoveConnected = self.connect("motion_notify_event", self.OnMouseMove)

        ## The connections beetween an action and its consequences
        #  @param self The object pointer
        def ConnectSignals2(self):
                if self.MouseMoveConnected:
              self.disconnect(self.MouseMoveConnected)
        self.MouseMoveConnected = self.connect("motion_notify_event", self.OnMouseMove2)

        ## The function to get a Render Window
        #  @param self The object pointer
        def GetRenderWindow(self):
                return self._RenderWindow

        ## The function to begin a new Renderer
        #  @param self The object pointer
        def GetRenderer(self):
```

```
                self._RenderWindow.GetRenderers().InitTraversal()
                return
                    self._RenderWindow.GetRenderers().GetNextItem()

        ## Mirrors the method with the same name in
            vtkRenderWindowInteractor.
        # @param self The object pointer
        # @param rate ...
        def SetDesiredUpdateRate(self, rate):
                self._DesiredUpdateRate = rate

        ## Mirrors the method with the same name in
            vtkRenderWindowInteractor.
        # @param self The object pointer
        def GetDesiredUpdateRate(self):
                return self._DesiredUpdateRate

        ## Mirrors the method with the same name in
            vtkRenderWindowInteractor.
        # @param self The object pointer
        # @param rate ...
        def SetStillUpdateRate(self, rate):
                self._StillUpdateRate = rate

        ## Mirrors the method with the same name in
            vtkRenderWindowInteractor.
        # @param self The object pointer
        def GetStillUpdateRate(self):
                return self._StillUpdateRate

        ## Sets the renderer
        # @ param self The object pointer
        def Render(self):
                if self.___Created:
                        self._RenderWindow.Render()

        ##  Creates the window
        ##  @ param self The object pointer
        ##  @ param *args The special syntax, *args
            in function definitions is used to pass a
             variable number of arguments to a
            function. The single asterisk form (*args)
            is used to pass a non-keyworded,
            variable-length argument list.
        def OnRealize(self, *args):
                if self.___Created == 0:
                        # you can't␣␣get␣␣the␣␣xid␣␣without␣␣
                            the␣␣window␣␣being␣␣realized.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.realize()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣win_id␣␣=␣␣str(self.widget.window.xid)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    self._RenderWindow.SetWindowInfo(win_id)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.___Created␣␣=␣␣1
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣Created(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣self.___Created

␣␣␣␣␣␣␣␣def␣␣OnConfigure(self,␣␣wid,␣␣event=None):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.widget=wid
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sz␣␣=␣␣self._RenderWindow.GetSize()

```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(event.width␣␣!=␣␣sz[0])␣␣or␣␣
    (event.height␣␣!=␣␣sz[1]):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    self._RenderWindow.SetSize(event.width,␣␣event.height)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnExpose(self,␣␣*args):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnDestroy(self,␣␣*args):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.hide()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣del␣␣self._RenderWindow
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.destroy()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnButtonDown(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Mouse␣␣button␣␣pressed."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    self._RenderWindow.SetDesiredUpdateRate(self._DesiredUpdateRate)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnButtonUp(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Mouse␣␣button␣␣released."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    self._RenderWindow.SetDesiredUpdateRate(self._StillUpdateRate)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnMouseMove(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Mouse␣␣has␣␣moved."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnMouseMove2(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Mouse␣␣has␣␣moved."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnEnter(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Entering␣␣the␣␣vtkRenderWindow."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnLeave(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Leaving␣␣the␣␣vtkRenderWindow."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnKeyPress(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""Key␣␣pressed."""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnKeyRelease(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"Key␣␣released."
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True


##␣␣An␣␣example␣␣of␣␣a␣␣fully␣␣functional␣␣
    GtkGLExtVTKRenderWindow␣␣that␣␣is␣␣based␣␣on␣␣the␣␣
    vtkRenderWidget.py␣␣provided␣␣with␣␣the␣␣VTK␣␣sources.
class␣␣Canvas3D(Canvas3DBase):

␣␣␣␣␣␣␣␣##␣␣The␣␣constructor
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣self␣␣The␣␣object␣␣pointer
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣demowindow
```

```
        #  @param *args The special syntax, *args in function definitions is used to pass a variable number of arguments to a function. The single asterisk form (*args) is used to pass a non-keyworded, variable-length argument list.
        def __init__(self, demowindow, *args):

                Canvas3DBase.__init__(self)

                self._CurrentRenderer = None
                self._CurrentCamera = None
                self._CurrentDolly = 1.0
                self._CurrentLight = None

                self._ViewportCenterX = 0
                self._ViewportCenterY = 0

                self._ClippingRange = (0,0)

                self._OldFocus = None

                # these record the previous mouse position
                self._LastX = 0
                self._LastY = 0

                # keeps reference to window - a hack to let this class manipulate the greater gui
                self.demowindow = demowindow


        def OnButtonDown(self, wid, event):
                self._RenderWindow.SetDesiredUpdateRate(self._DesiredUpdateRate)
                return self.StartMotion(wid, event)


        def OnButtonUp(self, wid, event):
                self._RenderWindow.SetDesiredUpdateRate(self._StillUpdateRate)
                if ((event.state & gdk.SHIFT_MASK) == gdk.SHIFT_MASK):
                        m = self.get_pointer()
                        self.VoxelInfo(m[0],m[1])
                return self.EndMotion(wid, event)

        ## Function that deactivate the mouse-interaction if the shift key is pressed.
        #  @param self The object pointer
        #  @param wid ...
        #  @param event=None ...
        def OnMouseMove(self, wid, event=None):
                # don't do anything if the shift key is pressed
                if ((event.state & gdk.SHIFT_MASK) == gdk.SHIFT_MASK):
                        pass
                elif ((event.state & gdk.BUTTON2_MASK) == gdk.BUTTON2_MASK):
                                m = self.get_pointer()
                                self.MouseTumble(m[0], m[1])
                                return True
```

```
                elif ((event.state & gdk.BUTTON3_MASK)
                    == gdk.BUTTON3_MASK):
                        m = self.get_pointer()
                        self.MouseTrack(m[0], m[1])
                        return True
                elif ((event.state & gdk.BUTTON1_MASK)
                    == gdk.BUTTON1_MASK):
                        m = self.get_pointer()
                        self.MouseDolly(m[0], m[1])
                        return True
                else:
                        return False


        ## Function that deactivate the
            mouse-interaction if the shift key is
            pressed.
        #   @param self The object pointer
        #   @param wid ...
        #   @param event=None ...
        def OnMouseMove2(self, wid, event=None):
                # don't␣␣do␣␣anything␣␣if␣␣the␣␣shift␣␣key␣␣
                    is␣␣pressed
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣((event.state␣␣&␣␣gdk.SHIFT_MASK)␣␣==␣␣
    gdk.SHIFT_MASK):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣pass
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣elif␣␣((event.state␣␣&␣␣gdk.BUTTON1_MASK)␣␣
    ==␣␣gdk.BUTTON1_MASK):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣m␣␣=␣␣self.get_pointer()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.MouseTumble(m[0],␣␣m[1])
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣elif␣␣((event.state␣␣&␣␣gdk.BUTTON2_MASK)␣␣
    ==␣␣gdk.BUTTON2_MASK):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣m␣␣=␣␣self.get_pointer()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.MouseTrack(m[0],␣␣m[1])
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣elif␣␣((event.state␣␣&␣␣gdk.BUTTON3_MASK)␣␣
    ==␣␣gdk.BUTTON3_MASK):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣m␣␣=␣␣self.get_pointer()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.MouseDolly(m[0],␣␣m[1])
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣False

␣␣␣␣␣␣␣␣def␣␣OnEnter(self,␣␣wid,␣␣event=None):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.grab_focus()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣w␣␣=␣␣self.get_pointer()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.UpdateRenderer(w[0],␣␣w[1])
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣OnLeave(self,␣␣wid,␣␣event):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True


␣␣␣␣␣␣␣␣def␣␣Render(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(self._CurrentLight):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣light␣␣=␣␣self._CurrentLight
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    light.SetPosition(self._CurrentCamera.GetPosition())
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    light.SetFocalPoint(self._CurrentCamera.GetFocalPoint())

```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Canvas3DBase.Render(self)

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#if␣␣self.GetCurrentRenderer()␣␣is␣␣not␣␣None:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣␣␣␣␣␣␣␣self.demowindow.updateCameraInfo()


␣␣␣␣␣␣␣␣##␣␣UpdateRenderer␣␣will␣␣identify␣␣the␣␣renderer␣␣under␣␣the␣␣mouse␣␣and␣␣set␣␣up␣␣_CurrentRenderer,␣␣_CurrentCamera,␣␣and␣␣_CurrentLight.
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣self␣␣The␣␣object␣␣pointer
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣x,␣␣y␣␣The␣␣2␣␣coordinates
␣␣␣␣␣␣␣␣def␣␣UpdateRenderer(self,x,y):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣windowX,windowY␣␣␣␣=␣␣self.widget.window.get_size()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderers␣␣=␣␣self._RenderWindow.GetRenderers()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣numRenderers␣␣=␣␣renderers.GetNumberOfItems()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentRenderer␣␣=␣␣None
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderers.InitTraversal()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣␣i␣␣in␣␣range(0,numRenderers):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer␣␣=␣␣renderers.GetNextItem()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vx,vy␣␣=␣␣(0,0)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(windowX␣␣>␣␣1):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vx␣␣=␣␣float(x)/(windowX-1)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(windowY␣␣>␣␣1):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vy␣␣=␣␣(windowY-float(y)-1)/(windowY-1)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(vpxmin,vpymin,vpxmax,vpymax)␣␣=␣␣renderer.GetViewport()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(vx␣␣>=␣␣vpxmin␣␣and␣␣vx␣␣<=␣␣vpxmax␣␣and
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vy␣␣>=␣␣vpymin␣␣and␣␣vy␣␣<=␣␣vpymax):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentRenderer␣␣=␣␣renderer
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._ViewportCenterX␣␣=␣␣float(windowX)*(vpxmax-vpxmin)/2.0\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+vpxmin
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._ViewportCenterY␣␣=␣␣float(windowY)*(vpymax-vpymin)/2.0\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+vpymin
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentCamera␣␣=␣␣self._CurrentRenderer.GetActiveCamera()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣lights␣␣=␣␣self._CurrentRenderer.GetLights()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣lights.InitTraversal()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentLight␣␣=␣␣lights.GetNextItem()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣break

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastX␣␣=␣␣x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastY␣␣=␣␣y

␣␣␣␣␣␣␣␣def␣␣GetCurrentRenderer(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣self._CurrentRenderer

```

```
␣␣␣␣␣␣␣␣def␣␣StartMotion(self,␣␣wid,␣␣event=None):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x␣␣=␣␣event.x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y␣␣=␣␣event.y
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.UpdateRenderer(x,y)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True

␣␣␣␣␣␣␣␣def␣␣EndMotion(self,␣␣wid,␣␣event=None):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣self._CurrentRenderer:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣True


␣␣␣␣␣␣␣␣##␣␣By␣␣manipulating␣␣the␣␣camera␣␣position,␣␣create␣␣the␣␣appearance␣␣of␣␣tumbling␣␣the␣␣microstructure␣␣using␣␣the␣␣mouse.
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣self␣␣The␣␣object␣␣pointer
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣x,␣␣y␣␣The␣␣2␣␣coordinates
␣␣␣␣␣␣␣␣def␣␣MouseTumble(self,x,y):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣self._CurrentRenderer:

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentCamera.Azimuth(self._LastX␣␣-␣␣x)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentCamera.Elevation(y␣␣-␣␣self._LastY)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentCamera.OrthogonalizeViewUp()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastX␣␣=␣␣x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastY␣␣=␣␣y

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.ResetClippingRange()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()

␣␣␣␣␣␣␣␣#␣␣convert␣␣x,y␣␣translation␣␣in␣␣Display␣␣coordinates␣␣to␣␣World␣␣Coordinates
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣self␣␣The␣␣object␣␣pointer
␣␣␣␣␣␣␣␣#␣␣␣␣@param␣␣x,␣␣y␣␣The␣␣2␣␣coordinates
␣␣␣␣␣␣␣␣def␣␣MouseTrack(self,x,y):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣self._CurrentRenderer:

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer␣␣=␣␣self._CurrentRenderer
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera␣␣=␣␣self._CurrentCamera
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(pPoint0,pPoint1,pPoint2)␣␣=␣␣camera.GetPosition()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(fPoint0,fPoint1,fPoint2)␣␣=␣␣camera.GetFocalPoint()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣␣Specify␣␣a␣␣point␣␣location␣␣in␣␣world␣␣coordinates
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer.SetWorldPoint(fPoint0,fPoint1,fPoint2,1.0)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer.WorldToDisplay()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣␣Convert␣␣world␣␣point␣␣coordinates␣␣to␣␣display␣␣coordinates
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dPoint␣␣=␣␣renderer.GetDisplayPoint()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣focalDepth␣␣=␣␣dPoint[2]

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aPoint0␣␣=␣␣self._ViewportCenterX␣␣+␣␣(x␣␣-␣␣self._LastX)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aPoint1␣␣=␣␣self._ViewportCenterY␣␣-␣␣(y␣␣-␣␣self._LastY)

```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    renderer.SetDisplayPoint(aPoint0,aPoint1,focalDepth)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer.DisplayToWorld()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(rPoint0,rPoint1,rPoint2,rPoint3)␣␣=␣
    ␣renderer.GetWorldPoint()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣(rPoint3␣␣!=␣␣0.0):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rPoint0␣␣=␣␣rPoint0/rPoint3
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rPoint1␣␣=␣␣rPoint1/rPoint3
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rPoint2␣␣=␣␣rPoint2/rPoint3

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastX␣␣=␣␣x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastY␣␣=␣␣y

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Track(fPoint0␣␣-␣␣
    rPoint0,fPoint1␣␣-␣␣rPoint1,fPoint2␣␣-␣␣rPoint2)


␣␣␣␣␣␣␣␣def␣␣Track(self,x,y,z):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera␣␣=␣␣self._CurrentCamera
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(pPoint0,pPoint1,pPoint2)␣␣=␣␣
    camera.GetPosition()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(fPoint0,fPoint1,fPoint2)␣␣=␣␣
    camera.GetFocalPoint()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera.SetFocalPoint(x␣␣+␣␣fPoint0,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y␣␣
    +␣␣fPoint1,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣z␣␣
    +␣␣fPoint2)

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera.SetPosition(x␣␣+␣␣pPoint0,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y␣␣+␣␣
    pPoint1,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣z␣␣+␣␣
    pPoint2)

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()


␣␣␣␣␣␣␣␣def␣␣MouseDolly(self,x,y):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dollyFactor␣␣=␣␣
    math.pow(1.02,(0.5*(self._LastY␣␣-␣␣y)))
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentDolly␣␣=␣␣
    self._CurrentDolly␣␣*␣␣dollyFactor
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Dolly(dollyFactor)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastX␣␣=␣␣x
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._LastY␣␣=␣␣y

␣␣␣␣␣␣␣␣def␣␣Dolly(self,dollyFactor):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣self._CurrentRenderer:

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣renderer␣␣=␣␣self._CurrentRenderer
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera␣␣=␣␣self._CurrentCamera

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣camera.GetParallelProjection():
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣parallelScale␣␣=␣␣
    camera.GetParallelScale()/dollyFactor
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    camera.SetParallelScale(parallelScale)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣camera.Dolly(dollyFactor)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.ResetClippingRange()
```

```

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()

␣␣␣␣␣␣␣␣def␣␣Reset(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣␣self._CurrentRenderer:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._CurrentRenderer.ResetCamera()

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.Render()


␣␣␣␣␣␣␣␣def␣␣ResetClippingRange(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
    self._CurrentRenderer.ResetCameraClippingRange()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self._ClippingRange␣␣=␣␣
    self._CurrentCamera.GetClippingRange()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#self.demowindow.clippingadj.set_value(100)

␣␣␣␣␣␣␣␣def␣␣GetClippingRange(self):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣␣self._ClippingRange
```

B.4 *vmtkcenterlineswithrenderer.py*

This file is very important since it modifies the existing “vmtkcenterlines” module, allowing the user to use the same canvas created before and not a new one, for the centerline(s) computation.

```
## @package vmtkcenterlineswithrenderer
#This module has been used in order to reproduce what it
    happens in the already-existing function
    vmtkcenterlines. This changing allows the user to use
    his own renderer and not the renderer created by
    vmtkcenterlines itself, since the already-existing
    function creates - by default - its own renderer.
#!/usr/bin/env python


try:
        import gtk
        import gtk.glade
except:
        sys.exit(1)

import vtk
from vtk import *

import vmtk
from vmtk import *

import vtkcones
import vmtkgui

from gtk import gdk
import math

# import vmtkcenterlines
# from vmtkcenterlines import *

class
    vmtkCenterlinesWithRenderer(vmtkcenterlines.vmtkCenterlines):

    def __init__(self):

        vmtkcenterlines.vmtkCenterlines.__init__(self)

        self.vmtkRenderer = None
    def Execute(self):

        if self.Surface == None:
            self.PrintError('Error:␣No␣input␣surface.')

        self.PrintLog('NonManifold␣check.')
        vmtkcenterlines.nonManifoldChecker =
            vmtkcenterlines.vmtkNonManifoldSurfaceChecker()
        vmtkcenterlines.nonManifoldChecker.Surface =
            self.Surface
        vmtkcenterlines.nonManifoldChecker.PrintError =
            self.PrintError

        if (vmtkcenterlines.nonManifoldChecker.\
                NumberOfNonManifoldEdges > 0):
            self.PrintLog(vmtkcenterlines.nonManifoldChecker.\
```

```
                Report)
            return

        self.PrintLog('Cleaning␣surface.')
        vmtkcenterlines.surfaceCleaner =
            vtk.vtkCleanPolyData()
        vmtkcenterlines.surfaceCleaner.SetInput(self.Surface)
        vmtkcenterlines.surfaceCleaner.Update()

        self.PrintLog('Triangulating␣surface.')
        vmtkcenterlines.surfaceTriangulator =
            vtk.vtkTriangleFilter()
        vmtkcenterlines.surfaceTriangulator.SetInput(vmtkcenterlines\
                .surfaceCleaner.GetOutput())
        vmtkcenterlines.surfaceTriangulator.PassLinesOff()
        vmtkcenterlines.surfaceTriangulator.PassVertsOff()
        vmtkcenterlines.surfaceTriangulator.Update()

        vmtkcenterlines.centerlineInputSurface =
            vmtkcenterlines.surfaceTriangulator.GetOutput()

        vmtkcenterlines.capCenterIds = None

        if (self.SeedSelectorName == 'openprofiles') |
            (self.SeedSelectorName == 'carotidprofiles') |
            (self.SeedSelectorName == 'pickpoint'):
            self.PrintLog('Capping␣surface.')
            vmtkcenterlines.surfaceCapper =
                vtkvmtk.vtkvmtkCapPolyData()
            vmtkcenterlines.surfaceCapper.SetInput\
                (vmtkcenterlines.surfaceTriangulator.GetOutput())
            vmtkcenterlines.surfaceCapper.SetDisplacement\
                (self.CapDisplacement)
            vmtkcenterlines.surfaceCapper.SetInPlaneDisplacement\
                (self.CapDisplacement)
            vmtkcenterlines.surfaceCapper.Update()
            vmtkcenterlines.centerlineInputSurface =
                vmtkcenterlines.surfaceCapper.GetOutput()
            vmtkcenterlines.capCenterIds =
                vmtkcenterlines.surfaceCapper.GetCapCenterIds()

        if self.SeedSelector:
            pass
        elif self.SeedSelectorName:
            if self.SeedSelectorName == 'pickpoint':
                self.SeedSelector =
                    vmtkPickPointSeedSelector()
                self.SeedSelector.vmtkRenderer =
                    self.vmtkRenderer
            elif self.SeedSelectorName == 'openprofiles':
                self.SeedSelector =
                    vmtkcenterlines.vmtkOpenProfilesSeedSelector()
                self.SeedSelector.SetSeedIds(vmtkcenterlines.surfaceCapper.\
                GetCapCenterIds())
                self.SeedSelector.OutputText =
                    self.OutputText
                self.SeedSelector.InputText = self.InputText
                self.SeedSelector.vmtkRenderer =
                    self.vmtkRenderer
            elif self.SeedSelectorName == 'carotidprofiles':
                self.SeedSelector =
                    vmtkcenterlines.vmtkCarotidProfilesSeedSelector()
```

```
                self.SeedSelector.SetSeedIds(vmtkcenterlines.surfaceCapper.\
                GetCapCenterIds())
            elif (self.SeedSelectorName == 'idlist'):
                self.SeedSelector = vmtkcenterlines.vmtkIdListSeedSelector()
                self.SeedSelector.SourceIds = self.SourceIds
                self.SeedSelector.TargetIds = self.TargetIds
            elif (self.SeedSelectorName == 'pointlist'):
                self.SeedSelector = vmtkcenterlines.vmtkPointListSeedSelector()
                self.SeedSelector.SourcePoints = self.SourcePoints
                self.SeedSelector.TargetPoints = self.TargetPoints
            else:
                self.PrintError("SeedSelectorName unknown (available: pickpoint | openprofiles | carotidprofiles | idlist | pointlist)")
                return
        else:
            self.PrintError('vmtkCenterlines error: either SeedSelector or SeedSelectorName must be specified')
            return

        self.SeedSelector.SetSurface(vmtkcenterlines.\
                centerlineInputSurface)
        self.SeedSelector.InputText = self.InputText
        self.SeedSelector.OutputText = self.OutputText
        self.SeedSelector.PrintError = self.PrintError
        self.SeedSelector.PrintLog = self.PrintLog
        self.SeedSelector.Execute()

        vmtkcenterlines.inletSeedIds = self.SeedSelector.GetSourceSeedIds()
        vmtkcenterlines.outletSeedIds = self.SeedSelector.GetTargetSeedIds()

        self.PrintLog('Computing centerlines.')
        vmtkcenterlines.centerlineFilter = vtkvmtk.vtkvmtkPolyDataCenterlines()
        vmtkcenterlines.centerlineFilter.SetInput(vmtkcenterlines.\
                centerlineInputSurface)
        if (self.SeedSelectorName == 'openprofiles') | (self.SeedSelectorName == 'carotidprofiles'):
            vmtkcenterlines.centerlineFilter.SetCapCenterIds\
                (vmtkcenterlines.capCenterIds)
        vmtkcenterlines.centerlineFilter.SetSourceSeedIds\
                (vmtkcenterlines.inletSeedIds)
        vmtkcenterlines.centerlineFilter.SetTargetSeedIds\
                (vmtkcenterlines.outletSeedIds)
        vmtkcenterlines.centerlineFilter.SetRadiusArrayName\
                (self.RadiusArrayName)
        vmtkcenterlines.centerlineFilter.SetCostFunction\
                (self.CostFunction)
        vmtkcenterlines.centerlineFilter.SetFlipNormals\
                (self.FlipNormals)
        vmtkcenterlines.centerlineFilter.\
        SetAppendEndPointsToCenterlines(self.AppendEndPoints)
        vmtkcenterlines.centerlineFilter.SetSimplifyVoronoi\
                (self.SimplifyVoronoi)
        if self.DelaunayTessellation != None:
```

```
            vmtkcenterlines.centerlineFilter.\
        GenerateDelaunayTessellationOff()
            vmtkcenterlines.centerlineFilter.SetDelaunayTessellation\
        (self.DelaunayTessellation)
        if self.UseTetGen==1:
            self.PrintLog('Running␣TetGen.')
            import vmtkscripts
            vmtkcenterlines.surfaceToMesh = vmtkscripts.vmtkSurfaceToMesh()
            vmtkcenterlines.surfaceToMesh.Surface = vmtkcenterlines.\
        centerlineInputSurface
            vmtkcenterlines.surfaceToMesh.Execute()
            vmtkcenterlines.tetgen = vmtkscripts.vmtkTetGen()
            vmtkcenterlines.tetgen.Mesh = vmtkcenterlines.surfaceToMesh.Mesh
            vmtkcenterlines.tetgen.PLC = 1
            vmtkcenterlines.tetgen.NoMerge = 1
            vmtkcenterlines.tetgen.Quality = 0
            if self.TetGenDetectInter == 1:
                vmtkcenterlines.tetgen.DetectInter = 1
                vmtkcenterlines.tetgen.NoMerge = 0
            vmtkcenterlines.tetgen.OutputSurfaceElements = 0
            vmtkcenterlines.tetgen.Execute()
            vmtkcenterlines.centerlineFilter.GenerateDelaunayTessellationOff()
            vmtkcenterlines.centerlineFilter.SetDelaunayTessellation\
        (vmtkcenterlines.tetgen.Mesh)
        vmtkcenterlines.centerlineFilter.SetCenterlineResampling\
        (self.Resampling)
        vmtkcenterlines.centerlineFilter.SetResamplingStepLength\
        (self.ResamplingStepLength)
        vmtkcenterlines.centerlineFilter.Update()

        self.Centerlines = vmtkcenterlines.centerlineFilter.GetOutput()
        self.VoronoiDiagram = vmtkcenterlines.centerlineFilter.\
        GetVoronoiDiagram()
        self.DelaunayTessellation = vmtkcenterlines.centerlineFilter.\
        GetDelaunayTessellation()
        self.PoleIds = vmtkcenterlines.centerlineFilter.GetPoleIds()

        self.EikonalSolutionArrayName = vmtkcenterlines.centerlineFilter.\
        GetEikonalSolutionArrayName()
        self.EdgeArrayName = vmtkcenterlines.centerlineFilter.\
        GetEdgeArrayName()
        self.EdgePCoordArrayName = vmtkcenterlines.centerlineFilter.\
        GetEdgePCoordArrayName()
        self.CostFunctionArrayName = vmtkcenterlines.centerlineFilter.\
        GetCostFunctionArrayName()


if __name__=='__main__':

    main = pypes.pypeMain()
    main.Arguments = sys.argv
```

```
    main.Execute()
```

B.5 *vtkcones.py*

This file is not needed for the *VMTKGui* itself, but it paves the road for the *VirtualValveStent* project. That is why we think it is useful to insert this file, too

```
## @package vtkcones
#This package contains a single class in which we can find a way to build and represent the cone(s) in the right window. Also, this is the file used to set up the renderer. That's␣why␣it␣is␣needed␣also␣in␣the␣1st␣part␣of␣the␣project,␣where␣we␣do␣not␣still␣have␣the␣stent.

try:
        import gtk
        import gtk.glade
except:
        sys.exit(1)

import vtk


## "vtkWindow␣class" is the class that allows us to print/remove the cones. Choosing the desired button, we could print/remove either 1 or 2 cones.
class vtkWindow:

        ## The constructor.
        #  @param self The object pointer
        #  @param wTree ...
        # def __init__(self, wTree, ren):
        def __init__(self, ren):

                self.ren = ren

                self.cone1_actor = vtk.vtkActor()
                self.cone2_actor = vtk.vtkActor()
                self.cone1_actor = None
                self.cone2_actor = None

        ## Function that allows us to print 2 cones
        #  @param self The object pointer
        #  @param widget gdk widget
        def both_cones(self, widget):

                self.ren.AddActor(self.cone1_actor)
                self.ren.AddActor(self.cone2_actor)
                self.ren.ResetCamera()



        ## With this function we could print 1st cone
        #  @param self The object pointer
        #  @param widget gdk widget
        def cone1(self, widget):

            if not self.cone1_actor:

                cone1 = vtk.vtkConeSource()
                cone1.SetHeight(100)
                cone1.SetRadius(50)
```

```
            cone1.SetResolution(150)

            # Map to graphics library
            cone1_map = vtk.vtkPolyDataMapper()
            cone1_map.SetInputConnection( cone1.GetOutputPort() )

            # Actor coordinates geometry, properties, transformation
            self.cone1_actor = vtk.vtkActor()
            self.cone1_actor.SetMapper( cone1_map )
            self.cone1_actor.SetPosition(0,3,0)
            self.cone1_actor.GetProperty().SetColor(0,0,0)

            # Create a clipping plane to clip cone 1
            plane1 = vtk.vtkPlane()
            plane1.SetOrigin(0.05,0.0,0.0)# [*1]
            plane1.SetNormal(-1.0,0.0,0.0)
            cone1_map.AddClippingPlane( plane1 )

            self.ren.AddActor(self.cone1_actor)
            self.ren.ResetCamera() #(Original version)


    ## With this function we could print 2nd cone
    # @param self The object pointer
    # @param widget gdk widget
    def cone2(self, widget):

        if not self.cone2_actor:

            # Cone2
            cone2 = vtk.vtkConeSource()
            cone2.SetHeight(100)
            cone2.SetRadius(50)
            cone2.SetResolution(150)

            # Map to graphics library
            cone2_map = vtk.vtkPolyDataMapper()
            cone2_map.SetInput( cone2.GetOutput() )

            # Actor coordinates geometry, properties, transfvaormation
            self.cone2_actor = vtk.vtkActor()
            self.cone2_actor.SetMapper(cone2_map)
            self.cone2_actor.SetPosition(0,3,0)
            # Cone_actor.RotateY(90)
            self.cone2_actor.RotateZ(180)
            self.cone2_actor.GetProperty().SetColor(1,0,0)

            ## Create a clipping plane to clip cone 2
            plane2 = vtk.vtkPlane()
            plane2.SetOrigin(0.05,0.0,0.0)
            plane2.SetNormal(1.0,0.0,0.0)
            cone2_map.AddClippingPlane(plane2)

            self.ren.AddActor(self.cone2_actor)
            self.ren.ResetCamera() #(Original version)

    ## With this function we could remove 1st cone
    # @param self The object pointer
    # @param widget gdk widget
```

```
    def rem_cone1(self, widget):
        self.ren.RemoveActor(self.cone1_actor)
            #(Original version)
        self.cone1_actor = None

    ## With this function we could remove 2nd cone
    #  @param self The object pointer
    #  @param widget gdk widget
    def rem_cone2(self, widget):
        self.ren.RemoveActor(self.cone2_actor)#
            (Original version)
        self.cone2_actor = None
```

Bibliography

[1] L. Antiga. *Patient-Specific Modeling of Geometry and Blood Flow in Large Arteries*. PhD thesis, Politecnico di Milano, 2002. 6, 11, 12, 13, 15, 17, 18, 19

[2] Luca Antiga, Bogdan Ene-Iordache, Lionello Caverni, Gian Paolo Cornalba, and Andrea Remuzzi. Geometric reconstruction for computational mesh generation of arterial bifurcations from ct angiography. *Computerized Medical Imaging and Graphics*, 26(4):227–235, 2002. 22

[3] D. Attali and J. O. Lachaud. Delaunay conforming iso-surface, skeleton extraction and noise removal. *Computational Geometry*, 19(2-3):175 – 189, 2001. 12

[4] C. Baillard and Christian Barillot. Robust 3d segmentation of anatomical structures with level sets. In *Proceedings of the Third International Conference on Medical Image Computing and Computer-Assisted Intervention*, MICCAI '00, pages 236–245, London, UK, 2000. Springer-Verlag. 22

[5] J. Bonnemain. From medical images to numerical simulations, 2009. iii, 25

[6] NDT Resource Center. Brief history of ct. Available at `http://www.imaginis.com/ct-scan/brief-history-of-ct`. 2

[7] NDT Resource Center. Computed tomography. Available at `http://www.ndt-ed.org/.../computedtomography.htm`. 3

[8] Bhargava. Chinni, Keerthi. Valluru, and Navalgund. Rao. Photoacoustic Imaging: Opening New Frontiers in Medical Imaging. *Journal of Clinical Imaging Science*, 1(1):24, 2011. 2

[9] Hema. Choudur, Jaspal. Hunjun, and Zameer. Hirji. Imaging of the Bursae. *Journal of Clinical Imaging Science*, 1(1):22, 2011. 2

[10] A.W. Date. *Introduction to computational fluid dynamics*. Cambridge University Press, 2005. 52

[11] H. Delingette and J. Montagnat. Shape and topology constraints on parametric active contours. *Computer Vision and Image Understanding*, 83(2):140 – 171, 2001. 16

[12] T. Deschamps. *Curve and Shape Extraction with Minimal Path and Level-Sets techniques - Applications to 3D Medical Imaging*. PhD thesis, Université Paris-IX Dauphine, Place du maréchal de Lattre de Tassigny, 75775 Paris Cedex, December 2001. 22

[13] Vikram. Dogra. New Horizons. *Journal of Clinical Imaging Science*, 1(1):1, 2011. 2

[14] Vikram. Dogra. The Beginning. *Journal of Clinical Imaging Science*, 1(1):25, 2011. 2

[15] F Fellner, R Schmitt, J Trenkler, C Fellner, and H Böhm-Jurkovic. Turbo gradient-spin-echo (grase): first clinical experiences with a fast t2-weighted sequence in mri of the brain. *European Journal of Radiology*, 19(3):171–176, 1995. 2

[16] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, Berlin, 1999. 52

[17] Francesca. Fornasa. Diffusion-weighted Magnetic Resonance Imaging: What Makes Water Run Fast or Slow? *Journal of Clinical Imaging Science*, 1(1):27, 2011. 2

[18] Jr. *Computational fluid dynamics: the basics with applications*. Aeronautical and Aerospace Engineering. McGraw-Hill, New York, 1995. 52

[19] H. M. Ladak, J. S. Milner, and D. A. Steinman. Rapid three-dimensional segmentation of the carotid bifurcation from serial mr images. *Journal of Biomechanical Engineering*, 122(1):96–99, 2000. 16

[20] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, August 1987. 12

[21] R. Malladi and J. A. Sethian. A real-time algorithm for medical shape recovery. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 304–, Washington, DC, USA, 1998. IEEE Computer Society. 18

[22] T. McInerney and D. Terzopoulos. Topologically adaptable snakes. In *Proceedings of the Fifth International Conference on Computer Vision*, ICCV '95, pages 840–, Washington, DC, USA, 1995. IEEE Computer Society. 16

[23] NEMA. Dicom standard. Available at `http://medical.nema.org/`. 6

[24] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Texts in Applied Mathematics Series. Springer, 2010. 19, 20

[25] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science ... on Applied and Computational Mathematics).* Cambridge University Press, 2 edition, June 1999. 19

[26] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. PhD thesis, University of California, at Berckly, 1982. 11, 18, 19

[27] Eftichis Sifakis, Christophe Garcia, and Georgios Tziritas. Bayesian level sets for image segmentation. *Journal of Visual Communication and Image Representation*, 13(1-2):44 – 64, 2002. 22

[28] Jasjit S. Suri, Sameer Singh, Swamy Laxminarayan, Xiaolan Zeng, Kecheng Liu, and Laura Reden. Shape recovery algorithms using level sets in 2-d/3-d medical imagery: A state-of-the-art review, 2001. 22

[29] C.M. Bemmel van, L.J. Spreeuwers, M.A. Viergever, and W.J. Niessen. Level-set based carotid artery segmentation for stenosis grading. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2002*, volume 2489 of *Lecture Notes in Computer Science*, pages 36–43, Berlin, 2002. Springer Verlag. 22

[30] C.M. van Bemmel, L.J. Spreeuwers, B. Verdonck, M.A. Viergever, and W.J. Niessen. Blood pool agent contrast-enhanced mra: Level-set based artery-vein separation. In *Proceedings of SPIE Medical Imaging*, volume 4684 of *Proceedings of SPIE*, pages 1464–1475, Bellington, Washington, USA, 2002. SPIE - The International Society for Optical Engineering. 22

[31] K C Wang, R W Dutton, and C A Taylor. Improving geometric model construction for blood flow modeling. *Engineering in Medicine and Biology Magazine IEEE*, 18(6):33–39, 1999. 22

[32] P. Wesseling. *An Introduction to Multigrid Methods*. R.T. Edwards, Inc., January 2004. 52

[33] Ross T. Whitaker. A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision*, 29:203–231, September 1998. 20

[34] Wikipedia. Allan cormack. Available at http://en.wikipedia.org/wiki/Allan_Cormack. 2

[35] Wikipedia. Computational fluid dynamics. Available at http://en.wikipedia.org/wiki/Computational_fluid_dynamics. 9, 52

[36] Wikipedia. The free induction decay. Available at http://en.wikipedia.org/wiki/Free_induction_decay. 5

[37] Wikipedia. Godfrey hounsfield. Available at http://en.wikipedia.org/wiki/Godfrey_Hounsfield. 2

[38] Wikipedia. The larmor frequency. Available at `http://en.wikipedia.org/wiki/Larmor_precession`. 5

[39] Wikipedia. Magnetic resonance imaging. Available at `http://en.wikipedia.org/wiki/Magnetic_resonance_imaging`. 5

[40] Wikipedia. Medical imaging. Available at `http://en.wikipedia.org/wiki/Medical_imaging`. 2

[41] Wikipedia. Medical ultrasonography. Available at `http://en.wikipedia.org/wiki/Medical_ultrasonography`. 5

[42] Wikipedia. The radon transform. Available at `http://en.wikipedia.org/wiki/Radon_transform`. 4

[43] Wikipedia. X-ray computed tomography. Available at `http://en.wikipedia.org/wiki/X-ray_computed_tomography`. 2

[44] Zoë Wood, Mathieu Desbrun, Peter Schröder, and David Breen. Semi-regular mesh extraction from volumes. In *Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*, VISUALIZATION '00, pages –, Washington, DC, USA, 2000. IEEE Computer Society. 12

[45] Chenyang Xu and J. L. Prince. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3):359–369, March 1998. 15