Medical Image Segmentation
using the RSFE Split-Bregman algorithm  Development
of a C++ Library

Marco FEDELE
Luca BARBAROTTA
Francesco CREMONESI
August 29, 2013

Alfio QUARTERONI
Simona PEROTTO

Luca FORMAGGIA
Elena FAGGIANO
Abstract

This project has the aim to implement a C++ library focused on 3d medical image processing and segmentation. Taking inspiration from the 2d implementation of Nicoletta Papucci in her graduation thesis [19], we implement the RSFE Split-Bregman Algorithm described by Yang et al. [26] because of its properties to work with inhomogeneous intensity images. We enrich the algorithm with some additional tools to make it work better with medical images. As usual in image processing, we consider an image as a discrete function in a uniform grid and we discretize the algorithm using the finite difference approach. In the first chapter, first we quickly describe how to derive the algorithm, then we provide a detailed description of our implementation choices and finally we talk about our customizations focused on medical application (like the building of a computational 3d domain to do, for instance, patient-specific hemodynamic numerical simulation). The second chapter is totally dedicated to the C++ implementation of the most important parts of the library and it can be integrated with the detailed doxygen html guide of the whole project. The third chapter shows results of the algorithm applied to some 3d medical images and it explains how to setup every parameters of the algorithm and how to do the correct preprocessing using some methods implemented in the library in order to adapt the algorithm at those images that don’t respect the necessary prerequisites.
Chapter 1

The RSFE Split-Bregman Algorithm

The objective of this work is to provide a sufficiently general algorithm for the segmentation of medical images and its implementation in C++.

In section 1.1 the concept of image segmentation will be introduced underlining what are the most important families of algorithm to segment an image and what are the general applications of 3d segmentation in medicine. Then we derive the RSFE Functional at the base of our algorithm in section 1.2, we explain the general idea behind the Split-Bregman algorithm for the minimization of some kind of functional in section 1.3 and we will detail its application to our functional in section 1.4. Finally, starting from the pseudocode of the algorithm, we describe in detail our numerical choices of implementation in section 1.5 and our customization in section 1.6.

1.1 Introduction to image segmentation

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse. More strictly, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain visual characteristics or computed properties (such as colour, intensity, or texture). Hence, the result of image segmentation is a set of subregions such that each region is significantly different with respect to the characteristics of interest. It is typically used to locate objects and boundaries inside images. In this project we are particularly interested in the applications to 3D medical imaging. The typical way of looking at 3D medical images like CAT or MRI is scrolling through the 2D slices in which they are divided. Generating contours inside a 3D image using segmentation allows 3D reconstructions of any human organ. This has lots of practical applications [12], like locating tumours and other pathologies. A particularly interesting use of segmentation in computational science and engineering is the 3D reconstruction of a blood vessel, for instance an artery, to generate a computational 3D domain on which a numerical simulation of the blood flow is possible [2, 7]. This simulation allows to study pathologies and subsequent treatments directly on the 'patient geometry'.

We now see briefly what are the two typical approaches in image segmentation [6]: edge-based methods and region-based methods. In the edge-based methods [8] the contour is described like a curve that evolves according to a law of evolution (PDE) involving the parametrizations of the curve itself and generally an edge detector function. Inside this class of methods we can cite: snakes, baloon and GAC. Edge-based methods have some difficulties: they are bounded by parametrization of the curve, often if parametrization is
changed the result will change; moreover dealing with topological changes of the contour is difficult with these methods; finally they can segment only well defined contours and only areas near the curve initialization.

Region-based methods have some advantages: they are global methods which consider all information of the whole image, do not depend on the parametrization and are more robust with respect to the choice of the initial contour, they are not very sensitive to noise and can segment also poorly defined contours. Among the defects of these methods one finds: an image cannot be divided in more than two regions, if the contour we want to segment is not the one with the maximum gradient some preprocessing operations on the image are needed, in images in which regions to segment are very close obtaining separate contours could be difficult. Despite these limits, region-based methods are considered a good compromise between quality of results, ease of implementation and computational cost.

In this report we develop a region based method that divides the image into two regions. The image to be segmented is in grey scale and it is represented with its intensity function \( u_0(x) \) assigning to each pixel \( x \in \Omega \) an intensity value. Our solution is stored in a level set function \( \phi \) taking values in the range \( [a_0, b_0] \). The rectangle/cube of the image represents the domain \( \Omega \) of the function:

\[
\phi(x) : \Omega \rightarrow [a_0, b_0].
\] (1.1.1)

The level set approach allows to find at the end of the algorithm the contour in the level \( \alpha = (a_0 + b_0)/2 \).

### 1.2 Derivation of Energy functional

Our implementation is based on the region-scalable fitting energy model (RSFE). RSFE is a new model born with the aim to overcome the limit of Chan-Vese model [4]. Piecewise constant (PC) model of Chan-Vese is conceived in order to segment images consisting of homogeneous regions. In fact it is demonstrated [13] that it fails to provide the correct segmentation in inhomogeneous images.

The idea behind this model is to divide the image in two regions and approximate the intensity of the image inside each region with a constant value: the average of the intensity inside the region. The regions are obtained minimizing a functional derived from Mumford-Shah functional [4].

The RSFE model can be seen as an improvement of this model. In fact the idea behind is similar but it substitutes the constant values with functions that approximate the image intensity in a local region through the use of a Gaussian kernel. The name comes from this, the method is considered scalable because through the use of the Gaussian kernel with a scale parameter it is possible to decide the sizes of the spots on which we calculate the average of the intensity around a pixel. There is a connection between the PC model and RSFE: indeed the RSFE model with a large scale parameter tends to the PC model, from here we can sense the importance of the scale parameter in RSFE for the segmentation of inhomogeneous images.

We now introduce the region scalable fitting energy \( \mathcal{F} \) to minimize in order to obtain the
regions and consequently the contour (the contour is the interface between the two regions):

\[
\mathcal{F}(\phi, f_1(x), f_2(x)) = \int_{\Omega} \left[ \sum_{i=1}^{2} \lambda_i \int_{\Omega} K_\sigma(x - y) |u_0(y) - f_i(y)|^2 M^e_i(y) dy \right] dx \\
+ \nu \int_{\Omega} |\nabla M^e_i(x)| dx \\
+ \mu \int_{\Omega} \frac{1}{2} (|\nabla \phi(x)| - 1)^2 dx,
\]  

(1.2.1)

The first term in 1.2.1 is the error we make by approximating the image intensity \(u_0(x)\) with the functions \(f_i(x)\), which represent the average of intensities in the neighbours of \(x\) defined by \(K_\sigma\), inside the region \(\Omega_i\).

The second term is a penalization on the length/area of the contour: the more the length/area of the contour grows, the more the energy rises.

The third term is introduced in [14] and has the purpose of regularizing the contour forcing it to be similar to a distance function.

In 1.2.1 \(u_0\) is the image intensity and \(\phi\) is the level set function that defines the contour. \(K_\sigma\) is the Gaussian kernel with scale parameter \(\sigma\):

\[
K_\sigma(x) = \frac{1}{(\sqrt{2\pi} \sigma)^d} \exp\left\{ -\frac{|x|^2}{2\sigma^2} \right\}, \quad x \in \mathbb{R}^d.
\]  

(1.2.2)

\(M^e_1\) and \(M^e_2\) are two functions that identify the regions based on the smoothed Heaviside function:

\[
H^e(\phi(x)) = \frac{1}{2} \left( 1 + \frac{2}{\pi} \arctan \left( \frac{\phi(x) - (a_0 + \alpha)}{\varepsilon |b_0 - a_0|} \right) \right)
\]  

(1.2.3)

\[
M^e_1(x) = H^e(\phi(x)), \quad M^e_2(x) = 1 - H^e(\phi(x)).
\]  

(1.2.4)

\(f_1\) and \(f_2\) are two functions that approximate image intensities:

\[
f_i(x) = \frac{K_\sigma * M^e_i u_0}{K_\sigma * M^e_i}.
\]  

(1.2.5)

\(\lambda_1, \lambda_2, \nu, \mu\) are positive scalar parameters of this model.

The minimization of this equation leads to the next flow equation [26]:

\[
\frac{\partial \phi}{\partial t} = -\delta_\varepsilon(\phi) (\lambda_1 e_1 - \lambda_2 e_2) \\
+ \nu \delta_\varepsilon(\phi) \nabla \cdot \left( \frac{\nabla \phi}{|\nabla \phi|} \right) \\
+ \mu \left( \Delta \phi - \nabla \cdot \left( \frac{\nabla \phi}{|\nabla \phi|} \right) \right),
\]  

(1.2.6)

where \(e_i(x) = \int_{\Omega} K_\sigma(x - y) |u_0(x) - f_i(y)|^2 dy\).

(1.2.7)

Functional (1.2.1) is non convex, so standard minimization methods like gradient descent can be easily trapped in local minima and thus the evolution of the contour would be stopped. Furthermore (1.2.6) is non-linear and its resolution can be very expensive computationally. Our purpose is to minimize (1.2.1) in a different way: we want to apply the GCS method to RSFE in order to make it convex and then minimize the resulting functional with the Split-Bregman algorithm. Before doing this in the next section we discuss in detail the Split-Bregman method.
1.3 Split-Bregman Method

The Split-Bregman method is a fast way to minimize functionals of the form:

$$\min_{v \in \mathbb{R}^n} |\Phi(v)| + H(v) \quad (1.3.1)$$

where $|\Phi(v)|$ and $H(v)$ are convex functions ($\Phi$ could be both scalar or vectorial) [10].

The strategy of the method is:

1. passing from the unconstrained minimization problem to a constrained one decoupling the two terms in the (1.3.1) through the introduction of a new variable;
2. reinterpreting the problem as an unconstrained problem by adding a penalization term which replaces the constraint;
3. applying the Bregman iteration to the unconstrained problem;
4. splitting the minimization in the two variables.

As shown in [26] this approach can be applied to functionals defined on a Hilbert space, like the $L^2$ space which provides the setting for this paragraph because it is the functional space of our application:

$$\min_{u \in L^2(\Omega)} \|\Phi(u)\|_{L^1(\Omega)} + H(u) \quad (1.3.2)$$

where $H$ is a functional defined over $L^2(\Omega)$.

From unconstrained to constrained problem: we decouple the problem (1.3.2) introducing the variable $d$ and enforcing it to be equal to $\Phi(u)$.

$$\begin{cases}
\min_{u,d \in L^2(\Omega)} \|d\|_{L^1(\Omega)} + H(u) \\
\quad d = \Phi(u)
\end{cases} \quad (1.3.3)$$

Return to the unconstrained problem: now we impose the constraint using a quadratic penalization function:

$$\min_{u,d \in L^2(\Omega)} \|d\|_{L^1(\Omega)} + H(u) + \frac{\lambda}{2} \|d - \Phi(u)\|_{L^2(\Omega)}^2 \quad (1.3.4)$$

the functional is now in a form on which we can use the Bregman iteration.

The Bregman iteration: First of all we now introduce a general formulation of the Bregman iteration. Given a problem of the type:

$$\min_{u \in L^2(\Omega)} E(u) + \lambda K(u) \quad (1.3.5)$$

where $E(u)$ and $K(u)$ are convex functional defined over $L^2(\Omega)$ and $\min_{u \in L^2(\Omega)} K(u) = 0$.

We define the concept of "Bregman Distance":

$$D^E_E(u, v) = E(u) - E(v) - \langle p, u - v \rangle_{L^2(\Omega)} \quad (1.3.6)$$

where $p$ is the subgradient of $E$ at $v$:

$p \in \partial E(v) \subset L^2(\Omega)$, if $E(w) \geq E(v) - \langle p, w - v \rangle_{L^2(\Omega)} \quad \forall w \in L^2(\Omega) \quad (1.3.7)$
The speed of this approach depends on how fast we can perform the two minimization techniques to solve the problem, depending on the properties of $H$. All the terms in (1.3.11) are differentiable so we can use a wide variety of optimization techniques to solve the problem, depending on the properties of $H(u)$. We assume for simplicity that $K(u)$ is differentiable. Problem (1.3.5) can be solved by iteratively computing:

$$u^{k+1} = \arg \min_{u \in L^2(\Omega)} D_E^p(u, u^k) + \lambda K(u)$$

$$= \arg \min_{u \in L^2(\Omega)} E(u) - \langle p^k, u - u^k \rangle_{L^2(\Omega)} + \lambda K(u) \quad (1.3.8)$$

$$p^{k+1} = p^k - \nabla K(u^{k+1})$$

Under the conditions set out above, if the problem (1.3.5) has a solution then the sequence written in (1.3.8) converges to the same solution. Further result about the convergence properties can be found in [18].

**Split the minimization in the two variables:** We apply the Bregman iteration to the (1.3.4) with $E(u, d) = \|d\|_{L^1(\Omega)} + H(u)$ and $K(u, d) = \frac{1}{2}\|d - \Phi(u)\|_{L^2(\Omega)}^2$ and we obtain:

$$(u^{k+1}, d^{k+1}) = \arg \min_{u,d \in L^2(\Omega)} D_E^p(u, u^k, d, d^k) + \frac{\lambda}{2}\|d - \Phi(u)\|_{L^2(\Omega)}^2$$

$$= \arg \min_{u,d \in L^2(\Omega)} E(u, d) - \langle p^k, u - u^k \rangle_{L^2(\Omega)} - \langle p^k_d, d - d^k \rangle_{L^2(\Omega)} + \frac{\lambda}{2}\|d - \Phi(u)\|_{L^2(\Omega)}^2$$

$$p^{k+1}_u = p^k_u - \lambda(\nabla \Phi)^T(\Phi(u^{k+1}) - d^{k+1})$$

$$p^{k+1}_d = p^k_d - \lambda(d^{k+1} - \Phi(u^{k+1})) \quad (1.3.9)$$

In [27] it is shown how equations (1.3.9) can be put in the following equivalent form, if the operator $\Phi$ is linear:

$$(u^{k+1}, d^{k+1}) = \arg \min_{u,d \in L^2(\Omega)} \|d\|_{L^1(\Omega)} + H(u) + \frac{\lambda}{2}\|d - \Phi(u) - b^k\|_{L^2(\Omega)}^2$$

$$b^{k+1} = b^k + (\Phi(u^{k+1}) - d^{k+1}) \quad (1.3.10)$$

Now we have to solve the problem of minimizing the first equation in (1.3.10) but thanks to the splitting the $L_1$ and the $L^2$ portions are now decoupled, so the minimization can be performed iteratively: minimizing first with respect to $u$ and then with respect to $d$:

$$u^{k+1} = \arg \min_{u \in L^2(\Omega)} H(u) + \frac{\lambda}{2}\|d^k - \Phi(u) - b^k\|_{L^2(\Omega)}^2 \quad (1.3.11)$$

$$d^{k+1} = \arg \min_{d \in L^2(\Omega)} \|d\|_{L^1(\Omega)} + \frac{\lambda}{2}\|d - \Phi(u^{k+1}) - b^k\|_{L^2(\Omega)}^2 \quad (1.3.12)$$

All the terms in (1.3.11) are differentiable so we can use a wide variety of optimization techniques to solve the problem, depending on the properties of $H(u)$. The speed of this approach depends on how fast we can perform the two minimization
subproblems. While the speed of the first subproblem depends on the choice of the solver, for the second subproblem we can use shrinkage operator which is extremely fast:

\[ d^{k+1} = \text{shrink}(\Phi(u^{k+1}) + b^k, 1/\lambda) \]  (1.3.13)

where \( \text{shrink}(x, \gamma) \) is defined as:

\[ \text{shrink}(x, \gamma) = \frac{x}{|x|} \max(|x| - \gamma, 0) \]  (1.3.14)

In [10] it is shown how it is useless performing more steps of minimization of \( u \) and \( d \) before updating the Split-Bregman parameter \( b \). Indeed, the further accuracy we would obtain will be wasted when we update \( b \). Furthermore it is not necessary to solve \( u^{k+1} \) until convergence, few steps of the iterative method chosen are enough.

1.4 Minimization of RSFE with Split-Bregman

We now apply the method described above to the functional (1.2.1), but before doing that we have to make some changes in order to make the functional convex. In [3] is presented the Global Convex Segmentation method (GCS), which is used to modify the functional in order to become convex. The first step we need to apply GCS is to drop the last term in equation (1.2.6):

\[ \frac{\partial \phi}{\partial t} = \delta_\epsilon(\phi) \left[ -\lambda_1 e_1 + \lambda_2 e_2 + \nu \text{div} \left( \frac{\nabla \phi}{|\nabla \phi|} \right) \right] \]  (1.4.1)

It can be proven [3] that the equilibrium of (1.4.1) is equal to the equilibrium of the same equation omitting \( \delta_\epsilon(\phi) \):

\[ \frac{\partial \phi}{\partial t} = -\lambda_1 e_1 + \lambda_2 e_2 + \nu \text{div} \left( \frac{\nabla \phi}{|\nabla \phi|} \right) \]  (1.4.2)

Solving (1.4.2) corresponds to the minimization of the following energy [26]:

\[ E(\phi) = \nu \| \nabla \phi \|_{L^1(\Omega)} + \langle \phi, r \rangle_{L^2(\Omega)} \]  (1.4.3)

where \( r = \lambda_1 e_1 - \lambda_2 e_2 \)  (1.4.4)

Note that the energy functional (1.4.3) is now convex but not strictly because of the linear term, this ensures existence of the minimum but not its uniqueness if \( \phi \in V \) where \( V \) is a generic functional space. To ensure also the uniqueness we have to optimize the functional forcing the solution to lie in a finite interval [26], hence we introduce the subspace \( U_{ad} \subset V \) defined as follow:

\[ U_{ad} := \{ \phi \in V : a_0 \leq \phi(x) \leq b_0, \quad \text{q.o.} \quad x \in \Omega \} \]  (1.4.5)

We are now ready to apply the Split-Bregman method. As seen in the previous section we have to decouple the \( L^1 \) term from the other term. This step is done by adding an auxiliary variable and a constraint on it:

\[ \begin{align*}
\min_{\phi, \tilde{d}} & \left\{ \nu \| \tilde{d} \|_{L^1(\Omega)} + \langle \phi, r \rangle_{L^2(\Omega)} \right\} \\
\tilde{d} & = \nabla \phi
\end{align*} \]  (1.4.6)
then imposing the constraint through a quadratic penalization:

\[
\min_{\phi, \bar{d}} \left\{ \nu \| \bar{d} \|_{L^1(\Omega)} + \langle \phi, r \rangle_{L^2(\Omega)} + \| \bar{d} - \nabla \phi \|_{L^2(\Omega)}^2 \right\}
\]  

(1.4.7)

and finally applying the Bregman iteration like in (1.3.10):

\[
(\phi^{k+1}, \bar{d}^{k+1}) = \arg \min_{\phi \in U_{ad}, \bar{d}} \left\{ \nu \| \bar{d} \|_{L^1(\Omega)} + \langle \phi, r^k \rangle_{L^2(\Omega)} + \frac{\lambda}{2} \| \bar{d} - \nabla \phi - \bar{b}^k \|_{L^2(\Omega)}^2 \right\}
\]  

(1.4.8)

\[
\bar{b}^{k+1} = \bar{b}^k + \nabla \phi^{k+1} - \bar{d}^{k+1}
\]  

(1.4.9)

We now introduce a function into the \( L_1 \) position of the edges. This function is defined as follows:

\[
g(u_0(x)) = \frac{1}{1 + \beta |\xi(u_0(x))|^2}, \quad \beta \in \mathbb{R}^+
\]  

(1.4.10)

\( \xi(u_0(x)) \) is a function that looks for the edges of the image \( u_0(x) \), it assumes values close to zero if the pixels of the image is far from an edge and big values when these pixels are edges of the image. A typical choice is setting \( \xi \) as the gradient of the intensity function of the image: \( \xi(u_0(x)) = \nabla (u_0(x)) \). Hence, in order to make less costly for the contour to move in areas where the image has edges we substitute the \( L_1 \) norm in (1.4.8) with the \( g \)-norm:

\[
\| \hat{v} \|_g = \int_{\Omega} g(u_0(x)) |\hat{v}(x)| \, dx
\]  

(1.4.11)

The next step in the Split-Bregman algorithm is the splitting of the equation (1.4.8):

\[
\phi^{k+1} = \arg \min_{\phi \in U_{ad}} \left\{ \langle \phi, r^k \rangle_{L^2(\Omega)} + \frac{\lambda}{2} \| \bar{d}^k - \nabla \phi - \bar{b}^k \|_{L^2(\Omega)}^2 \right\}
\]  

(1.4.12)

\[
\bar{d}^{k+1} = \arg \min_{\bar{d}} \left\{ \nu \| \bar{d} \|_g + \frac{\lambda}{2} \| \bar{d} - \nabla \phi - \bar{b}^k \|_{L^2(\Omega)}^2 \right\}
\]  

(1.4.13)

\[
\bar{b}^{k+1} = \bar{b}^k + \nabla \phi^{k+1} - \bar{d}^{k+1}
\]  

(1.4.14)

As seen in (1.3.13) minimization of the second equation (1.4.13) can be done using the shrinkage operator. We focus now our attention on the minimization of (1.4.12) showing the equivalence with the resolution of a Poisson problem.

We first rearrange the terms as follow (omitting the \( k \) to simplify notation):

\[
\langle \phi, r \rangle_{L^2(\Omega)} + \frac{\lambda}{2} \| \bar{d} - \nabla \phi - \bar{b} \|_{L^2(\Omega)}^2 = \langle \phi, r \rangle_{L^2(\Omega)} + \frac{\lambda}{2} \left( \langle \bar{d} - \nabla \phi - \bar{b}, \bar{d} - \nabla \phi - \bar{b} \rangle_{L^2(\Omega)} \right)
\]  

\[
= \frac{\lambda}{2} \langle \nabla \phi, \nabla \phi \rangle_{L^2(\Omega)} + \langle \bar{d}, \nabla \phi \rangle_{L^2(\Omega)} - \lambda \langle \bar{b}, \nabla \phi \rangle_{L^2(\Omega)} + \frac{\lambda}{2} \left( \langle \bar{d}, \bar{d} \rangle_{L^2(\Omega)} + \langle \bar{b}, \bar{b} \rangle_{L^2(\Omega)} - 2 \langle \bar{d}, \bar{b} \rangle_{L^2(\Omega)} \right)
\]  

\[
= a(\phi, \phi) - L \phi + \text{cost},
\]  

(1.4.15)

where we have defined the bilinear form \( a(\phi, v) \) and the linear functional \( L \phi \) as follow:

\[
a(\phi, v) = \frac{\lambda}{2} \langle \nabla \phi, \nabla v \rangle_{L^2(\Omega)}
\]  

\[
L \phi = \lambda \langle \bar{d} - \bar{b}, \nabla \phi \rangle_{L^2(\Omega)} - \langle r, v \rangle_{L^2(\Omega)}
\]
Since in (1.4.15) the gradient of $\phi$ appears, a natural choice is to set the functional space $V$ equal to the Sobolev space $H^1(\Omega)$. Nevertheless we first set $V = H^1_0(\Omega)$ to simplify computations, generalizing later to the more general space $H^1(\Omega)$.

In this case $a(\phi, v)$ is a continuous and coercive bilinear form and $L v$ is a continuous linear functional, hence all terms in (1.4.15) are Fréchet-differentiable [20].

Since the functional is convex, we can differentiate the equation and put it equal to zero in order to find the minimum [20]:

$$D [a(\phi, \phi) - L \phi + \text{cost}] [v] = a(\phi, v) - L v = 0.$$  

(1.4.16)

Hence, we have just shown that the minimization problem (1.4.12) is equivalent to the resolution of the following weak formulation:

$$\forall \phi \in V : a(\phi, v) - L v = 0,$$  

(1.4.17)

The existence and the uniqueness of the solution of (1.4.17) is guaranteed by Lax-Milgram theorem [20].

Since we will solve this problem numerically using finite difference method, we are interested in the equivalent strong formulation, obtained as follows using Gauss-Green theorem and the fact that all functions in $H^1_0(\Omega)$ have nil trace [20]:

$$a(\phi, v) - L v = \frac{\lambda}{2} \langle \nabla \phi, \nabla v \rangle_{L^2(\Omega)} + \lambda \langle \vec{b} - \vec{d}, \nabla v \rangle_{L^2(\Omega)} + \langle r, v \rangle_{L^2(\Omega)}$$

$$= - \frac{\lambda}{2} \langle \Delta \phi, v \rangle_{L^2(\Omega)} + \langle \partial_\nu \phi, v \rangle_{L^2(\partial\Omega)} + \lambda \langle \nabla (\vec{b} - \vec{d}) \cdot \vec{n}, v \rangle_{L^2(\partial\Omega)} + \langle r, v \rangle_{L^2(\Omega)}$$

$$= - \frac{\lambda}{2} \langle \Delta \phi, v \rangle_{L^2(\Omega)} + \lambda \langle \nabla (\vec{b} - \vec{d}), v \rangle_{L^2(\Omega)} + \langle r, v \rangle_{L^2(\Omega)},$$

that is equal to the following Dirichlet Poisson Problem:

$$\begin{cases}
- \Delta \phi &= \text{div} \left( \vec{b} - \vec{d} \right) - \frac{r^k}{\lambda} \quad \text{in } \Omega \\
\phi &= 0 \quad \text{on } \partial \Omega
\end{cases}$$

(1.4.18)

To extend this result to the more general case of $\phi \in H^1(\Omega)$ we can simply define the solution of the minimization problem (1.4.12) as $\phi = \tilde{\phi} + R_g$ where $\tilde{\phi} \in H^1_0(\Omega)$ (hence is nil on the boundary) and $R_g$ is a selected function in $H^1(\Omega)$ such that is equal to a function $g$ on the boundary of the domain $\Omega$ ($g : g = \phi$ on $\partial \Omega$). Proceeding again as before we obtain the same equation (1.4.18) with $\tilde{\phi}$ in place of $\phi$ and with $\Delta R_g$ as a new addend in the forcing function. Substituting $\tilde{\phi} = \phi - R_g$ we arrive at the following Dirichlet Poisson Problem:

$$\begin{cases}
- \Delta \phi &= \text{div} \left( \vec{b} - \vec{d} \right) - \frac{r^k}{\lambda} \quad \text{in } \Omega \\
\phi &= g \quad \text{on } \partial \Omega
\end{cases}$$

(1.4.19)

Projecting the solution in the space $U_{ad} \subset V$ defined in (1.4.5) (with $V = H^1(\Omega)$) we obtain the solution of the problem (1.4.12).

Hence, we can finally summarise as follow the procedure to minimise functional (1.4.3):

$$\begin{cases}
\phi^{k+1} &= \text{solution of (1.4.19)} \\
\text{project} \quad \phi^{k+1} \text{ on } U_{ad} \\
d^{k+1} &= \text{shrink} \left( \vec{b}^k + \nabla \phi^{k+1}, \frac{\nu}{\lambda} g \right) \\
\vec{b}^{k+1} &= \vec{b}^k + \nabla \phi^{k+1} - \vec{d}^{k+1}
\end{cases}$$

(1.4.20)
Since the rhs of (1.4.19) $\in L^2(\Omega)$ (to ensure this it is sufficient to initialize $\vec{b}^0 = \vec{d}^0 = 0$) and the domain is convex, we can apply the global elliptic regularity theorem [20] stating that $\phi \in H^2(\Omega)$. Thanks to the Sobolev embedding theorem [20], this implies that $\phi$ is a continuous function either in the 2D or in the 3D case. Hence, it makes sense to speak of point values of $\phi$ and, in order to solve (1.4.19) with a numerical method, also the finite difference method is allowed (implementation details will be discussed in the next section). Note finally that, as suggested in [26], in (1.4.20) $\nu$ can be set equal to one without loss of generality.

As we will see in the last chapter, it could be useful to see the solution of (1.4.19) as the equilibrium $\phi^*$ of the corresponding unsteady parabolic problem (1.4.21):

$$\begin{cases}
\frac{\partial \phi}{\partial t} - \Delta \phi = \text{div} \left( \vec{b}^k - \vec{d}^k \right) - \frac{r^k}{\lambda} & \text{in } \Omega \times (0, T) \\
\phi(t, x) = g & \text{on } \partial\Omega \times (0, T) \\
\phi(0, x) = \phi_0 & \text{in } \Omega
\end{cases} \quad (1.4.21)$$

This has been done because the robustness of the Split-Bregman method allows us not to solve (1.4.21) until equilibrium, but few temporal steps are sufficient [18]. Implementation details will be shown in section 1.5.

We conclude this section with a pseudocode of the whole algorithm.

**Algorithm 1** Minimization of RSFE with Split-Bregman

1: $\phi^0 \leftarrow \text{initial contour}$  
2: $\vec{b}^0 \leftarrow 0$  
3: $\vec{d}^0 \leftarrow 0$
4: \textbf{while } $\|\phi^{k+1} - \phi^k\| > \text{tol}$ \textbf{do}
5: \hspace{1em} update $M^e_{i}(x)$ \hspace{1em} \triangleright update domains $\Omega_i$ computing $\mathcal{H}^e(\phi^k(x))$ (eq. (1.2.4))
6: \hspace{1em} update $e^k_{i}(x)$ \hspace{1em} \triangleright compute convolutions to update $e^k_{i}$ (eq. (1.2.7))
7: \hspace{1em} $r^k \leftarrow \lambda_1 e^k_{1} - \lambda_2 e^k_{2}$
8: \hspace{1em} $\phi^{k+1} \leftarrow \text{PDE solver} \left( \phi^k, r^k, \vec{b}^k, \vec{d}^k, \lambda \right)$ \hspace{1em} \triangleright compute new levelset (eq. (1.4.21))
9: \hspace{1em} $a_0 \leq \phi^{k+1} \leq b_0$ \hspace{1em} \triangleright take back $\phi^{k+1}$ in the range $[a_0, b_0]$
10: \hspace{1em} $\vec{d}^{k+1} \leftarrow \text{shrink} \left( \vec{b}^k + \nabla \phi^{k+1}, \frac{g}{\lambda} \right)$ \hspace{1em} \triangleright update Split-Bregman variables
11: \hspace{1em} $\vec{b}^{k+1} \leftarrow \vec{b}^k + \nabla \phi^{k+1} - \vec{d}^{k+1}$
12: \textbf{end while}

### 1.5 Implementation Details

In our implementation we consider all functions introduced in previous sections as discrete functions defined in a uniform grid. This is a common choice in image processing because pixels naturally define a uniform grid and it is not possible to go over pixels definition without 'inventing information'.

We have implemented the algorithm both for 2d and 3d images. The main difference is that usually two-dimensional images are characterized from square pixels and therefore they generate uniform grid easier to handle as a discrete signal, on the contrary three-dimensional ones have got pixels with different spacing in each direction called voxels and we have to
consider this especially in convolution computation. Hence, to implement the algorithm in our discrete space we follow these steps:

1. we simply implement all algebraic operations as pixels-by-pixels operations.

2. we approximate continuous convolution products with discrete convolution evaluating continuous functions in each pixel and solving convolution using Discrete Fourier Transform (DFT) with FFT algorithm to improve performance.

3. we use finite difference to discretize PDE (1.4.21) and every differential operator.

4. we solve the linear system generated by the finite difference discretization of (1.4.21) using Gauss-Seidel iterative method without saving the matrix to prevent the excessive use of memory.

**Convolution:** solving convolution products is the most expensive thing we have to do at each iteration. In fact, first we have to solve four convolution products to compute \( f_i \) as seen in equation (1.2.5) and then we can compute \( e_i \) decomposing equation (1.2.7) in other three convolution products as follows:

\[
e_i(x) = \int_{\Omega} K_\sigma(x - y) |u_0(x) - f_i(y)|^2 \, dy
= |u_0|^2 (1_\Omega * K_\sigma) + (|f_i|^2 * K_\sigma) - 2u_0 (f_i * K_\sigma)
\]

Hence we have a total of ten convolution products for each iteration.

In this paragraph at first we will see how to minimize the number of this products and secondly we will show how to solve each convolution in the fastest possible way using Fast Fourier Transform algorithm.

We first define two functions that don’t depend on variables changing during the algorithm, so we can compute them before the while cycle starts:

\[
K_{1_\Omega} = 1_\Omega * K_\sigma \quad K_{u_0} = u_0 * K_\sigma
\]

We can now halve the convolution products needed to compute \( f_i \) at each iteration:

\[
um = M_1^* u_0 * K_\sigma, \quad \den = M_2^* * K_\sigma,
\]

\[
f_1 = \frac{\num}{\den}, \quad f_2 = \frac{K_{u_0} - \num}{K_{1_\Omega} - \den},
\]

where in (1.5.4) we have obtained the expression of \( f_2 \) exploiting linearity property of the convolution in equation (1.2.5):

\[
f_2 = \frac{M_2^* u_0 * K_\sigma}{M_2^* * K_\sigma} = \frac{[(1 - M_1^*)u_0] * K_\sigma}{(1 - M_1^*) * K_\sigma} = \frac{u_0 * K_\sigma}{1_\Omega * K_\sigma} - M_1^* u_0 * K_\sigma.
\]

If we proceed as follow we can reduce the total number of convolution products in each iteration at only four:

\[
r_1 = \lambda_1 |f_1|^2 - \lambda_2 |f_2|^2, \quad r_2 = \lambda_1 f_1 - \lambda_2 f_2,
\]

\[
K_{r_1} = r_1 * K_\sigma, \quad K_{r_2} = r_2 * K_\sigma,
\]
where to derive equation (1.5.7) it is sufficient doing simple algebraic calculations:

\[
r = (\lambda_1 - \lambda_2)|u_0|^2\mathcal{K}_1 + \mathcal{K}_{r_1} - 2u_0\mathcal{K}_{r_2}, \tag{1.5.7}
\]

Hence, as it is outlined in algorithm 2, it is sufficient to compute before starting the cycle \(\mathcal{K}_{1\Omega}\) and \(\mathcal{K}_{u_0}\) using (1.5.2) and at each iteration \(num\) and \(den\) using (1.5.3) and \(\mathcal{K}_{r_1}\) and \(\mathcal{K}_{r_1}\) using (1.5.6). All the other calculations are algebraic operations.

**Algorithm 2** RSFE Split-Bregman: convolution products

1: 

2: \(\mathcal{K}_{1\Omega} \leftarrow \mathbb{1}_{\Omega} * K_{\sigma}\) \hspace{1cm} \triangleright \text{ initialize constant convolution}

3: \(\mathcal{K}_{u_0} \leftarrow u_0 * K_{\sigma}\)

4: while \(||\phi^{k+1} - \phi^k|| > tol\) do

5: update \(M^{k}_{\mathcal{E}}\) \hspace{1cm} \triangleright \text{ convolution using (1.5.3)}

6: update \(num^k, den^k\) \hspace{1cm} \triangleright \text{ eq. (1.5.4)}

7: update \(f^k_i\) \hspace{1cm} \triangleright \text{ eq. (1.5.4)}

8: update \(\mathcal{K}^k_i\) \hspace{1cm} \triangleright \text{ convolution using (1.5.6)}

9: update \(r^k\) \hspace{1cm} \triangleright \text{ eq. (1.5.7)}

10: 

11: end while

Let’s see now how we can discretize the convolution integral. First of all note that in our convolution products one of the factor is always the Gaussian Kernel \(K_{\sigma}\) defined in (1.2.2). Therefore we have to solve in a discrete space this equation:

\[
g(x) = K_{\sigma} * f = \int_{\mathbb{R}^n} K_{\sigma}(x - y)f(y) \, dy, \quad n = 2, 3. \tag{1.5.8}
\]

The only way to use DFT to solve (1.5.8) is to approximate it with the midpoint/rectangle rule in order to transform it in a Discrete Convolution. We do it explicitly in the three-dimensional case (2d is easier):

\[
g(x_n) = \sum_{i,j,k=-\infty}^{+\infty} h_x h_y h_z \, K_{\sigma}(x_n - y_{i,j,k}) \, f(y_{i,j,k}) \approx \sum_{i=-M_x}^{M_x} \sum_{j=-M_y}^{M_y} \sum_{k=-M_z}^{M_z} h_x h_y h_z \, K_{\sigma}(x_n - y_{i,j,k}) \, f(y_{i,j,k}) \tag{1.5.9}
\]

where \(M_\ast = \lceil 3 \, h_\ast \sigma \rceil \) \tag{1.5.10}
and $h_s$ are the dimension in each direction of the pixel/voxel. In (1.5.9) we assume the Gaussian kernel null if we are at a distance greater than $3\sigma$ from the central value.

We can now apply the Convolution Theorem. Let’s see before a complete version of it where we indicate with $\mathcal{F}$ the Fourier Transform, with DTFT the Discrete Time Fourier Transform and with DFT the Discrete Fourier Transform:

**Theorem 1.** The Fourier transform of a convolution is the point-wise product of Fourier transforms. In other words, convolution in time domain equals point-wise multiplication in the frequency domain as can be seen in following formulas:

\[
\begin{align*}
  f(x) \ast g(x) &= \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\} \\
  f(x_n) \ast g(x_n) &= DTFT^{-1}\{DTFT\{f(x_n)\} \cdot DTFT\{g(x_n)\}\} \\
  f_T(x_n) \ast g(x_n) &= DFT^{-1}\{DFT\{f(x_n)\} \cdot DFT\{g(x_n)\}\}
\end{align*}
\]  

(1.5.11) (1.5.12) (1.5.13)

In the last equation we denote with $f_T$ the periodic expansion of the function $f$ defined on a finite and discrete interval (e.g. in 1d this means: $f_T(x_n + mT) = f(x_n), \forall m \in \mathbb{Z}$). The resulting convolution (1.5.13) is called Circular Convolution, in fact in this product boundary values of opposite edges of original factor $f$ interact each other because of periodization.

Now to use FFT algorithm we need to use DFT, hence we would like to transform equation (1.5.9) in order to apply (1.5.13). We are dealing with two discrete and finite factors, but we don’t need to perform the Circular Convolution but the Discrete one. The solution is the zero padding method. To simplify the procedure, we first assume the two factors $f$ and $K_\sigma$ as one-dimensional denoting respectively with $N$ and $M$ their number of elements. We can proceed as follow:

1. add $M - 1$ zeros at the end of factor $f$ (zero-padding $f$).
2. add $N - 1$ zeros at the end of factor $K_\sigma$ (zero-padding $K_\sigma$), note that both the two factors have got now $N + M - 1$ elements.
3. compute DFT of zero-padded $f$ and $K_\sigma$.
4. multiply the DFT of the two zero-padded factors as in equation (1.5.13).
5. compute DFT$^{-1}$ of the product just computed.

Adding the correct number of zeros ensures that there are no interactions between values of different bounds. Hence, zero padding the two factors allows to compute Discrete Convolution using DFT.

It is easy to generalize this procedure in the multi-dimensional case. For instance, in case of equation (1.5.9) we have to add $[(M_x - 1) \times (M_y - 1) \times (M_z - 1)]$ zeros to the factor $f$ and $[(N_x - 1) \times (N_y - 1) \times (N_z - 1)]$ zeros to the factor $K_\sigma$ in order to transform both in factors of dimensions $[(N_x + M_x - 1) \times (N_y + M_y - 1) \times (N_z + M_z - 1)]$. Then it is sufficient to perform a 3d DFT instead of the one-dimensional.

Last thing to underline is that the result of Discrete Convolution is obviously a discrete function with $[(N_x + M_x - 1) \times (N_y + M_y - 1) \times (N_z + M_z - 1)]$ elements. However we are not interested to go outside the range $[N_x \times N_y \times N_z]$ because the rest of function in the algorithm has got these dimensions. Hence, at the end, we crop the external frame of the resulting image to reduce dimensions to the desired ones.

Let’s conclude the paragraph analysing performance of the approach just described. Because $K_\sigma$ does not change during the algorithm, we can compute its DFT at the beginning,
hence every convolution costs only a Forward Fourier Transform (of the factor \( f \)), a multiplication \( \text{DFT}\{f\} \times \text{DFT}\{K_\sigma\} \) and the final Inverse Fourier Transform. Considering that the FFT algorithm to compute a single DFT has a complexity of \( \Theta((N+M) \times \log(N+M)) \), we obtain a really good improvement of performance using this approach. In fact computing direct convolution as in equation (1.5.8) costs \( \Theta(N \times M) \) and this means a great loss of time considering that in our application we usually have \( M \simeq N \). Moreover a more precise quadrature formula in (1.5.8) would increase complexity even more.

Despite DFT approach leads to the midpoint rule (that is the most approximate quadrature rule), it is the only way to solve convolution quickly if you are dealing with a quite big three-dimensional image like in each medical application.

**PDE resolution with Finite Difference and Gauss-Seidel:** in this paragraph we discuss how to solve in a discrete space the partial differential equation (1.4.21). We first have a look at the finite difference scheme adopted to approximate differential operator and then we discuss the resolution of the linear system with the Gauss-Seidel method.

The use of finite difference method to discretize differential operator is a very natural choice in image processing because, as described at the beginning of this section, images naturally defined a uniform grid perfect to compute finite difference.

We start describing the approximation of the spatial differential operator. To approximate the derivatives of the first order we use a second order finite difference scheme. For non-border values, central differences (1.5.14) have been used whereas for border values a second order preserving scheme with three points (1.5.15) has been implemented:

\[
\frac{\partial f(x_i)}{\partial x} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h_x} + o(h_x^2), \tag{1.5.14}
\]

\[
\frac{\partial f(x_i)}{\partial x} = -\frac{3f(x_i)}{2} + 4f(x_{i+1}) - f(x_{i+2}) + o(h_x^2). \tag{1.5.15}
\]

Previous equations refer to the discretization in \( x \) direction and we omit indexes \( j \) and \( k \) because they are constant in each derivative computation. It is sufficient to replace \( x \) with \( y \) or \( z \) and \( i \) with \( j \) or \( k \) to have the correct formulation of the other two directions.

We now group all explicit terms of (1.4.21) in a unique forcing function:

\[
\frac{\partial \phi(x,t)}{\partial t} = \Delta \phi(x,t) + F^k(x) \tag{1.5.16}
\]

where

\[
F^k(x) = \nabla \cdot \left( \hat{b}^k(x) - \bar{d}^k(x) \right) - \frac{r^k(x)}{\lambda}. \tag{1.5.17}
\]

Hence, to compute \( F^k \) we can simply apply (1.5.14) and (1.5.15) to approximate the divergence operator.

For the laplacian operator in the three-dimensional case (2d is analog) we use the following second order scheme:

\[
\Delta f(x_{i,j,k}) = \frac{f(x_{i+1,j,k}) + f(x_{i-1,j,k})}{h_x^2} + \frac{f(x_{i,j+1,k}) + f(x_{i,j-1,k})}{h_y^2} + \frac{f(x_{i,j,k+1}) + f(x_{i,j,k-1})}{h_z^2} - \left( \frac{2}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2} \right) f(x_{i,j,k}) + o(h_{x,y,z}^2). \tag{1.5.18}
\]
We can now proceed with the temporal discretization as follows:

\[
\frac{\phi(x_{i,j,k}, t + 1) - \phi(x_{i,j,k}, t)}{dt} = \Delta \phi(x_{i,j,k}, t^*) + F(x_{i,j,k}) \tag{1.5.19}
\]

where we omit the Split-Bregman index \( k \) to simplify the notation and we consider now \( t \) as the discrete time and \( dt \) the time step. Now we focus our attention on the discretization of the laplacian operator. Choosing \( t^* = t \) we find an explicit method, choosing \( t^* = t + 1 \) we find an implicit method. Instead Gauss-Seidel is a semi-implicit method, that generates a lower triangular matrix, also sparse for our equation. This is possible by evaluating in equation (1.5.18) all pixels with a +1 index \((i + 1 \text{ or } j + 1 \text{ or } z + 1)\) in an explicit way \((t^* = t)\) and all pixels with a −1 index \((i − 1 \text{ or } j − 1 \text{ or } z − 1)\) in a implicit way \((t^* = t + 1)\) as shown in (1.5.20):

\[
\tilde{H} \phi_{i,j,k}^{t+1} = \frac{\phi_{i+1,j,k}^{t+1}}{h_x^2} + \frac{\phi_{i-1,j,k}^{t+1}}{h_x^2} + \frac{\phi_{i,j+1,k}^{t+1}}{h_y^2} + \frac{\phi_{i,j-1,k}^{t+1}}{h_y^2} + \frac{\phi_{i,j,k+1}^{t+1}}{h_z^2} + \frac{\phi_{i,j,k-1}^{t+1}}{h_z^2} + \frac{\phi_{i,j,k}^t}{dt} + F_{i,j,k} \tag{1.5.20}
\]

where \( \tilde{H} = \frac{1}{dt} + \frac{2}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2} \) (1.5.21)

Leaving only the explicit terms on the right of the equal we can easily see the discretization as a linear system \( Ax = b \):

\[
\tilde{H} \phi_{i,j,k}^{t+1} - \frac{\phi_{i-1,j,k}^{t+1}}{h_x^2} - \frac{\phi_{i,j-1,k}^{t+1}}{h_y^2} - \frac{\phi_{i,j,k-1}^{t+1}}{h_z^2} = \frac{\phi_{i+1,j,k}^{t+1}}{h_x^2} + \frac{\phi_{i,j+1,k}^{t+1}}{h_y^2} + \frac{\phi_{i,j,k+1}^{t+1}}{h_z^2} + \frac{\phi_{i,j,k}^t}{dt} + F_{i,j,k} \tag{1.5.22}
\]

On the right side of the equation we have all known terms, instead on the left one we have different coefficients multiplying the unknown \( \phi_i^{t+1} \) evaluated in different pixels in order to form a lower triangular matrix.

Note that if we choose the homogeneous Dirichlet condition the boundary values are simply strongly imposed to zero. Now, to avoid the storage of the matrix in memory, it is more interesting going back to the formulation (1.5.20) looking at it ‘deleting’ all indexes \( t \) and \( t + 1 \). In fact if we compute values of all the pixels respecting (1.5.20) starting from the first internal pixel (coordinates \( i = 1, j = 1, k = 1 \)) and proceeding in an iterative ordered way (like in three nested for-cycles), at the end of all iterations the updated levelset will be stored in \( \phi \). In chapter 2 we will show the C++ code with the three nested for-cycles to clarify even better this procedure.

We conclude this paragraph underlining again that, despite to obtain the equilibrium of this equation we need usually lots of steps of Gauss-Seidel method, considering that this problem is nested in the other iterative problem of Split-Bregman, thanks to the robustness of the last one it is sufficient to do only few steps in order to ensure the global convergence of the whole algorithm [18]. Furthermore, as shown in [24], no stability condition are needed for the time step \( dt \) because the matrix generated from the discretization of the laplacian operator in a uniform grid is positive-definite and in this case the Gauss-Seidel is unconditionally stable.
1.6 Customization

In this section we discuss the additional tools that we decide to add to the algorithm in order to render it more suitable to the typical application in medical image segmentation described in section 1.1. In fact the use of this algorithm leads to two typical problems:

1. We are usually interested in segmenting only an object inside an image (typical a human organ in medical images) and not all the objects inside of it. However the algorithm 'thinks globally' and it segments all the objects it founds in the analysed image. This can not be a problem if objects are far from each other (unusual in medical images), instead, if they are near, with the progress of the iterations of the algorithm they tend to stick to each other.

2. The algorithm divides the image in only two regions \( \phi = a_0 \) and \( \phi = b_0 \) and to do that it analyses the intensity of the image \( u_0 \). Hence, if the object we would like to segment does not have a good contrast with the background, the algorithm typically considers the object joined to the background without segmenting it. For instance this happens in medical images if the organ is grey on a slightly darker background and in the image there are also parts very clear (white) and parts very dark (black).

To solve the first problem we implemented an algorithm to extract a connected component and we allowed its application to the levelset, to solve the second problem we implemented a set of pre-processing tools.

**Connected Component:** the extraction of a connected component of an image is a morphological algorithm that works in black and white images as follow:

1. let the user choose a pixel as a starting point.
2. starting from this pixel, it looks at its neighbours and only if they are of the same colour it marks them as pixels of the connected component.
3. algorithm proceed iteratively and it ends only if current iteration has not made any changes to the previous connected component.

To apply the algorithm to the levelset \( \phi \) we first transform it in a black and white image using the threshold \( \alpha \). This means that if a pixel \( x \) is such that \( \phi(x) \in [\alpha, b_0] \) we mark it as white, otherwise we mark it as black \( \phi(x) \in [a_0, \alpha] \).

We can now allow the user (after choosing the initial pixel) to extract a connected component from the current levelset at each iteration he wants. After showing the result of the extraction, the user can also decide if he wants to reinitialize the algorithm setting the connected component just shown as the initial levelset. This makes the algorithm more interactive but also slower because the user, each time he wants to perform an extraction, has to set the initial pixel and choose if he wants to reinitialize the algorithm or not. To avoid these slowdowns we add also a fully automatic mode of extraction. This means that the user has only to set before the beginning of the algorithm the initial pixel \( x \) inside the object he would like to segment and the frequency \( n \) with which he would like to perform the connected component extraction during execution. When the algorithm starts, it evaluates at each iteration the value of \( \phi(x) \), if it is white \( (\phi(x) \in [\alpha, b_0]) \) it proceeds with the automatic extraction, it reinitializes the algorithm setting the connected component just extracted as the initial levelset and it restarts the algorithm from here executing other \( n \)
iterations before performing a connected component extraction again. As we will show in chapter 3 this procedure really improves the results of the algorithm and allows it 'to think locally' instead of 'thinking globally' segmenting only a selected elements of an image.

**Additional Tools:** the additional tools we have implemented have the aim of doing a pre-processing on the image to avoid the second problem described at the beginning of this section or, more in general, to improve the quality of the image we want to segment. In this paragraph we just describe quickly the most important ones, have a look to the html guide of our library for more details:

1. **crop** allows user to quickly crop an image in a smaller one. This is the first step to do before starting the segmentation algorithm because the more an image is big, the more the algorithm is slow and it needs more memory. Hence, first of all it is really suggested to crop the image you would like to segment in order to obtain the smallest image as possible, containing the object of interest.

2. **change resolution** allows to decrease resolution of an image or to increase it linearly interpolating the original values. This can be useful if, also after a crop operation, the image is too big and accordingly the algorithm is too slow. In this case a good strategy can consist in the execution of the algorithm on the image with a lower resolution obtaining a less precise solution, following by the rerunning of it on the original image initializing the levelset with the interpolated solution of the step with lower resolution.

3. **select range of intensity** allows the user to select only those pixels \( x \) such that \( u_\theta(x) \in [c, d] \) where \( c, d \) is the desired range. Pixels not satisfying this condition are set to values \( c \) if they are smaller or \( d \) if they are greater. Optionally they can also all be set only to the value \( c \) or \( d \). This is the simplest way to try to solve the problem of low contrast. For instance if you are dealing with a medical image with values in the range \([0, 1000]\) and you want to segment an object with values in a range of \([300, 350]\) in a background characterized from a range \([250, 300]\) it is quite sure that our algorithm does not work. Instead if you select from the original image the range of intensity \([250, 350]\) killing all others pixels algorithm will surely work better in order to obtain your aim.

4. **histogram equalization** is a more advanced algorithm to improve the contrast of an image, useful to be applied sometimes after the selection of a range of intensity.

5. **median filter** is instead a non linear filter that helps to cut outliers values from an image. This allows to remove noise like salt and pepper and it is suggested to apply it before the algorithm starts if the original image is pimpled or with a clear presence of noise.
Chapter 2

C++ Implementation

2.1 Structure of the implementation

The development of the code has been carried out following two (see [23]) cornerstone philosophies: modularity and flexibility. Modularity means that the code is subdivided in logical blocks that are independent one from the other and flexibility means that each block may be easily modified without too much hassle.

A visual representation of the implementation is given in fig. 2.1.

Figure 2.1: Flowchart of the implementation.
The logical areas of our code are:

1. **Interface** to import/export data and to visualize images.

2. **Linear Filtering** to perform convolution between a filter and a function.

3. **Poisson Problem** to solve the unsteady Poisson problem.

4. **Image Segmentation** to apply an algorithm like the RSFE Split-Bregman one (algorithm 1).

5. **Image Processing** to apply the additional tools dealt in section 1.6.

As explained in section 1.5, each logical area has to deal with discrete functions defined on uniform grids of pixels. Hence, first of all we develop a basic class called `image3d` (used by all the others) in which all basic algebraic and differential operators are implemented. Then we develop a class called `interface` corresponding to the first logical block and two functors called `filtering` and `unsteady_poisson_functor` corresponding respectively to the second and the third block. Finally we implement a class called `rsfe_splitbregman` to apply algorithm 1 using all others. Each operation dealing with the fifth block is instead implemented as a method of the basic class `image3d` to be available to any other classes.

We also group each class in some namespaces to emphasize the logical block they belong to:

<table>
<thead>
<tr>
<th>namespace</th>
<th>conv</th>
<th>im3d</th>
<th>lapl</th>
<th>segm</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>abstract_interface</td>
<td>interface</td>
<td>unsteady_poisson_functor</td>
<td>rsfe_splitbregman</td>
</tr>
<tr>
<td>filtering</td>
<td>image3d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We will discuss about pure virtual classes `abstract_interface` and `segmentation` later.

Since the full library is a template library with the template parameter corresponding to the precision of the floating point number chosen to represent the elements of an `image3d` (`float`, `double`, `long double`...), we write each class in a corresponding header file with a similar name. Looking at the include dependency graph in figure 2.2, it appears more clear how logical blocks interact each others. Everyone needs class `image3d`, but while classes `interface`, `filtering` and `unsteady_poisson_functor` are mutually independent, class `rsfe_splitbregman` depends on all the others.

We choose to divide our code in so many parts in order to allow any future developer to easily change some of them maintaining the general structure and the rsfe segmentation algorithm intact. In this perspective the two functors are particularly important. As we discuss in section 1.5, from the point of view of computational complexity, convolution and PDE resolution are the two most expensive things to do at each iteration of the algorithm. Using functors not only developer, but also a simple user of the library can implement himself a way to solve these parts writing his own function without changing the code of the library. Instead the separation between class `interface` and class `image3d` has two main advantages: on the one hand allows not to make too heavy each `image3d` objects (very often used in the code) storing the part linked to the visualization in separated `interface` objects, on the other hand allows developers to easily redesign the part of the library linked to the visualization without altering the rest of the code. In fact inside the class `interface` we made regular use of `vtk` libraries [1] to allow 3D visualization, but developers that don’t want to
use these libraries could implement class *interface* again using another approach. The pure virtual version of this class (*abstractinterface*) should be considered as a template guide for them and therefore contains the strictly necessary members for class *interface* to make the rest of the code work correctly. Finally the pure virtual class *segmentation* is conceived instead as a base class for all the segmentation algorithm, hence every new algorithm should be implemented as a class derived from it.

Before describing in detail each class we conclude this section underlining that in order to improve performance we have parallelized our code using OpenMP [5] whenever it was possible. Since we are dealing at the same time with objects that may be considered images or generic functions, the OpenMP parallelization provides a means to improve computational time whilst maintaining a readable, simple code. We have discarded the choice of CUDA because in our algorithm a lot of input-output read and write of images is carried out, something that in CUDA would have generated too much overhead.

### 2.2 Class: *image3d*

Underneath the fabric of interconnections represented in fig. 2.1 is a common denominator, a class called *image3d*, used to represent 3D images for the computations in our code. This class is the backbone of the implementation as it is passed to the various other classes in charge of each of the logical blocks mentioned before, and can at the same time be used to represent the image under consideration and all the necessary structures to carry out the algorithm, such as the level set function for example.

Other than providing a standard way through which all the other elements of the code can communicate and interact, the main advantage this class provides is in the optimization of the code: by giving the developer complete freedom in low level functions such as computing sums and derivatives, he has the power to perform some changes that can be really effective in terms of computational costs and accuracy. In this frame of mind, it has been developed essentially as a container for a *std::vector* holding the pixel-wise values of the image and equipped with various functions to allow easy manipulation both from an image treatment...
and computational point of view. We write here the header file of image3d in order to understand better the structure of the class:

```cpp
#include <vector>
#include <cstdint>
#include <cstdlib>
#include <iostream>
#include <assert.h>

#define REAL_DOUBLE

namespace image3d
{
    class image3d
    {
        protected:
            std::vector<T> rawimage;
            uint dimx, dimy, dimz;
            real hx, hy, hz;
        public:
            // CONSTRUCTORS
            image3d();
            image3d(uint const & x, uint const & y, uint const & z,
                     real const & hx=1., real const & hy=1., real const & hz=1.);
            image3d(image3d const & tocopy);
            virtual ~image3d();
            // MEMBERS TO GET AND SET PRIVATE PARAMETERS
            inline uint getdimx() const
            {
                return this->dimx;
            }
            // [ getdimy, getdimz, setdimx, setdimy, setdimz ] ...
            inline real gethx() const
            {
                return this->hx;
            }
            // [ gethy, gethz, sethx, sethy, sethz ] ...
            inline void seth(real const & hx, real const & hy, real const & hz);
            // OVERLOADING OF VARIOUS OPERATORS
            virtual inline T operator()(uint const &i, uint const & j, uint const & k) const;
            virtual inline T& operator()(uint const &i, uint const & j, uint const & k);
            template <typename S>
            image3d<T> & operator=(S const & toassign);
            template <typename S>
            image3d<T> & operator=(image3d<S> const & toassign);
            // [ +=, -=, *=, /= ] ...
            template <typename S, typename R>
            friend image3d<S> const operator+(image3d<S> const & addend1, R const & addend2);
            // [ -, *, / ] ...
            template <typename S, typename R>
            friend void vector_abs(image3d<S> & res, std::vector<image3d<R>> const & fun);
            template <typename S, typename R>
            friend void div(image3d<S> & res, std::vector<image3d<R>> const & fun);
            T const norm1() const;
            // [ normL1, norm2, normL2, norminf ] ...
            T const max() const;
            T const min() const;
            // PROCESSING
            void crop(image3d<T> & res,
                      uint const & xstart, uint const & ystart, uint const & zstart,
                      uint const & xend, uint const & yend, uint const & zend) const;
            void crop(image3d<T> & res,
```

22
real const & xstart, real const & ystart, real const & zstart, 
real const & xend, real const & yend, real const & zend) const;

void change_resolution (image3d<T> & res, uint ratio=2, bool increase=false) const;

void histogram_equalization(image3d<T> & res, uint const & quantization) const;
void histogram_equalization(image3d<T> & res) const;

template<typename S>
void im_to_black_and_white (image3d<S> & res,
    real threshold = 0.5, bool negative = false) const;

void change_range_of_intensity(T const & max, T const & min = 0);

void select_range_of_intensity (image3d<T> & res, T const & lowerbound, 
    T const & upperbound, int type=0) const;

template <typename S>
void connected_component (image3d<S> & res, image3d<S> & bw, 
    bool const & full_connected) const

template <typename S>
void connected_component (image3d<S> & res, 
    uint const & i, uint const & j, uint const & k, 
    real threshold = 0.5, bool full_connected = false) const;

void median_filter(image3d<T> & res, int const & radius=1) const;

An image3d is a rectangular parallelepiped structured grid whose values are stored in a vector. The size and the spacing of the grid are stored in attributes such as dimx, dimy, dimz and hx, hy, hz. We can think the grid nodes as the centres of the voxels and the distances between adjacent nodes as the resolution of the image (dimension of a voxel).

Image3d is a default and copy constructable class. The default constructor creates an empty image whereas the copy constructor initialises a new image copying the values and the attributes of the one passed as arguments. Another constructor has been implemented in order to permit to set the sizes and the spacings of the image. This constructor allocates the right amount of memory to the vector then the user has to fill the vector with the desired values (he could use assignment operator or the ()-bracket operator to do that).

After the constructors, in the header file, there are some methods which are used to read and write the class attributes. Among them there is the setdim method. It has been implemented in such a way that after the change of size a value, passed as argument, is set to the elements of the image. This kind of implementation has proven to be computationally efficient, indeed in our algorithm it is often used for the initialisation of variables allowing us to save some CPU time. The operators form the core of this class. They allow to access and to assign data and to compute some operations on the class itself.
In the following pages we show in details the implementations of some of these operators.

Operators: for more readable access and manipulation, this class is equipped with some useful overloading of mathematical operators, as shown below. They have been made virtual so that if any future user wishes to create a class that inherits from image3d but with data organised in a different fashion he is allowed to overwrite the () operator.
The brackets () operator has been implemented in two ways: the constant and non constant version. They have been implemented in order to account for the two possible uses of this operator, one for assignment and the other for value retrieval. This operator determines the
scrolling order of the indexes, in our implementation the fastest scrolling order is obtained by first scrolling the last index until its size then passing to the second index and at last scroll the first index. This is due to the fact that in this way we scroll in a coalescent way the vector containing the data. From our test the coalescent scrolling is up to twice as fast as the non-coalescent one. The ”unnatural” choice of the scrolling order is due to compatibility with the FFTW3 library, it is not necessary for its integration but allows a saving in term of computational costs in one of the bottleneck of our code. The last note regarding the indexes: in all methods, the indexes are treated as the Cartesian index of the image. The origin is the farther bottom left corner, the first index represents the horizontal direction and the others follow the right-hand rule. This choice has been influenced by the use of the VTK library, in fact VTK uses this indexes system in its visualisation so we decided to follow their convention instead of the most common matrix ones.

```cpp
template <typename T>
T im3d::image3d<T>::operator() (uint const &i, uint const & j, uint const & k) const
{
    return rawimage[ this->dimy * this->dimz * i + this->dimz * j + k ];
}

template <typename T>
T& im3d::image3d<T>::operator() (uint const &i, uint const & j, uint const & k)
{
    return rawimage[ this->dimy * this->dimz * i + this->dimz * j + k ];
}
```

The other operators have a parallel implementation using OpenMP [5] whenever possible in order to reduce computational time. Self run tests have shown that the improvement can be significant, with a ratio of up to 3 between serial and parallel time in case of a quad-core CPU. Moreover, the operators that have access to private members (including friend functions) are implemented ()-free, that is to say in a single loop rather than preserving the three-dimensional interpretation of the data, which also provides an improvement in computational time.

Here we show the code of some algebraic operators: operator+ = and operator+. For all other operators the scheme is the same. The general rule was: to implement the operator\(\Delta =\) (where \(\Delta\) is an algebraic binary operation) in such a way that no temporary variables are allocated and then reuse it for the implementation of the operator\(\Delta\). All \(\Delta =\) operators have two implementations, the first takes as arguments a scalar value while the second takes an image3d. As consequence, the operator\(\Delta\) calls the proper operator\(\Delta =\) provided that the scalar value is its second argument.

```cpp
template <typename T>
template <typename S>
im3d::image3d<T> & im3d::image3d<T>::operator += (S const & addend)
{
#pragma omp parallel for
    for(uint i = 0; i < this->dimx*this->dimy*this->dimz; ++i)
        this->rawimage[i] += static_cast<T>(addend);
    return *this;
}
```

24
this->dimz == addend.getdimz() )
{
    #pragma omp parallel for
    for(uint i = 0; i < this->dimx; ++i)
        for(uint j = 0; j < this->dimy; ++j)
            for(uint k = 0; k < this->dimz; ++k)
                (*this)(i,j,k) += static_cast<T>(addend(i,j,k));
}
else
    std::cout << "WARNING::image3d::operator+= : dimensions must agree" << std::endl;
    return *this;
}

template <typename S, typename R>
im3d::image3d<S> const im3d::operator + (image3d<S> const & addend1, R const & addend2)
{
    image3d<S> sum(addend1);
    return sum += addend2;
}

Because of its double nature, this class also possesses both functions that belong to the image treatment and to the scientific computation families.

**Numerical Functions:** these functions have been implemented to account for the numerical manipulations that need to be done by the algorithm. Just as the members mentioned in the previous paragraph consider the array of values as a digital image, that is to say a map from the three(two)-dimensional physical space to the set of possible pixel intensities, the members contained under the *numerical functions* category generalize this view to arbitrary (though fairly regular) functions from $\mathbb{R}^3$ to some result space $\Omega \subset \mathbb{R}^n$. This means that we will need routines to compute useful mathematical quantities such as the gradient, the divergence, the scalar product between two images and so forth. In particular five kinds of norms have been implemented: norm1 and norm2 correspond to the definitions (2.2.1) where the data is considered as a sequence of real values, norm $L^1$ and norm $L^2$ are defined in (2.2.2) and consider the image as embedded in a geometrical space.

$$
\|a\|_1 = \sum_{i=0}^{N} |a_i|, \quad \|a\|_2 = \sqrt{\sum_{i=0}^{N} (a_i)^2}, \quad \tag{2.2.1}
$$

$$
\|a\|_{L^1(\Omega)} = \int_{\Omega} |a|, \quad \|a\|_{L^2(\Omega)} = \sqrt{\int_{\Omega} (a)^2}, \quad \|a\|_{\infty} = \max_{i} |a_i|. \quad \tag{2.2.2}
$$

Note that, given the nature of a digital image, the integrals in (2.2.2) have been approximated thinking the integrand function piecewise constant: in fact we thought that quadrature rule of higher order would have added wrong information since the datum is really constant on a voxel. Moreover, as seen in section 1.5, also convolution through fft needs this kind of quadrature rule.

Differential operators have been implemented using second order finite difference schemes reproducing (1.5.14) and (1.5.15). An example of the implementation may be found below for the gradient operator. Note that the parallelization is only in the loop that scans the internal values, and not the boundary ones. This is due to the overhead of the fork/join
model, which proved to be too important in the boundary loops where the number of
iterations is considerably lower than the internal loop.

... internal nodes of the first two components ...

#pragma omp for
for(uint i = 0; i < this->dimx; ++i)
  for(uint j = 0; j < this->dimy; ++j)
    for(uint k = 1; k < this->dimz-1; ++k)
      res[2](i,j,k) =
        ( static_cast<S>( (*this)(i,j,k+1) ) -
        static_cast<S>( (*this)(i,j,k-1) ) ) / (2.*hz) ;

} // ending parallel section

// boundary values
for(uint j = 0; j < this->dimy; ++j)
  for(uint k = 0; k < this->dimz; ++k)
  {
    res[0](0,j,k) = ( 4*static_cast<S>( (*this)(1,j,k) ) -
        3*static_cast<S>( (*this)(0,j,k) ) -
        static_cast<S>( (*this)(2,j,k) ) )/(2.*hx) ;
    res[0](this->dimx-1,j,k) = (3*static_cast<S>( (*this)(dimx-1,j,k) ) -
        4*static_cast<S>( (*this)(dimx-2,j,k) ) +
        static_cast<S>( (*this)(dimx-3,j,k) ))/(2.*hx);
  }

... other boundary nodes ...

Image Treatment: these members have been implemented to allow the user to perform
simple manipulations on the values of the image, namely those that would not require the
use of filters but only access to the image values. At the end of chapter 1 the image pro-
cessing tools we have implemented and the concepts behind them are listed and briefly
explained, here we will see some implementative details about some of them.

We show some examples of such methods: crop, connected component and median filter. We
choose these methods because the first two highlight one of the criteria we use in the imple-
mentation of our code: reusing pieces of code. Both the methods have two implementations:
one containing the core of the algorithm, and the other based on the former producing a
new method with different signature. This is done mainly for two reason: first, by adopting
this paradigm the code will be conciser and hence more readable for other users; second,
if someone wants to change something he only has to modify the core algorithm and the
modification will be spread away through other methods. Median filter is chosen because
of its importance in the pre-processing operations.

Crop is a method that allows user to create a new image copying a selected area from
another one. It is a useful pre-processing tool in particular in segmentation of medical image,
in fact, often times it could be used in order to reduce the computational cost and to select
the area with the right properties within the image. Two methods have been implemented:
one taking the indexes of the original image area, the other takes the coordinates of the
image (it considers the spacings between elements).

The indexes passed to the method are three couples of integer values: for each directions
the first voxel from which the copy starts and the last voxel in which the copy ends. Here
is the code of the former version.

```cpp
template <typename T>
void im3d::image3d<T>::crop(image3d<T> & res,
    uint const & XSTART, uint const & YSTART, uint const & ZSTART,
    uint const & XEND, uint const & YEND, uint const & ZEND) const
{  
```
// verify consistency of the indexes
{...}
// compute size of the result
{...}
// set the proper size to the result
res.setdim( newdimx, newdimy, newdimz);
res.seth( this->hx, this->hy, this->hz);
// copy the region selected from the original image
#pragma omp parallel for
  for (uint i= 0; i< res.getdimx(); ++i)
    for (uint j= 0; j< res.getdimy(); ++j)
      for (uint k= 0; k< res.getdimz(); ++k)
        res(i,j,k) = (*this)(i+xstart, j+ystart, k+zstart);
return;
}

The latter implementation of crop takes real coordinates, convert it into integer indexes and then call the other crop method:

```cpp
template <typename T>
void im3d::image3d<T>::crop(image3d<T> & res,
  real const & XSTART, real const & YSTART, real const & ZSTART,
  real const & XEND, real const & YEND, real const & ZEND) const
{
  // converting coordinates to indexes
  {...}
  // in crop will be checked the consistency of the indexes
  this->crop(res, xs, ys, zs, xe, ye, ze);
  return;
}
```

Connected component is based on a morphological algorithm, indeed it works on binary images. This is the core algorithm for the connected component extraction:

```cpp
template <typename T>
template <typename S>
void im3d::image3d<T>::connected_component (image3d<S> & res, image3d<S> & bw,
  bool const & full_connected) const
{
  image3d<S> resold(this->dimx,this->dimy,this->dimz);
  uint norm1=1;
  // 2D CASE
  {...}
  // 3D CASE
  else
  {
    while( norm1 != 0 )
    {
      // 8 DIRECTIONS: (UP k+1, NORTH j+1, EAST i+1)
      // up-north-east, up-north-west, up-south-east, up-south-west
      // down-north-east, down-north-west, down-south-east, down-south-west
      // UP-NORTH-EAST
      for(uint i=1; i < dimx-1; ++i)
        for (uint j=1; j < dimy-1; ++j)
          for (uint k=1; k< dimz-1; ++k)
            if ( res(i,j,k) == 1 )
            {
              // east
```
res(i+1,j,k) = 1 * bw(i+1,j,k);
// north
res(i,j+1,k) = 1 * bw(i,j+1,k);
// up
res(i,j,k+1) = 1 * bw(i,j,k+1);
if(full_connected)
{
    // north-east
    res(i+1,j+1,k) = 1 * bw(i+1,j+1,k);
    // up-east
    res(i+1,j,k+1) = 1 * bw(i+1,j,k+1);
    // up-north
    res(i,j+1,k+1) = 1 * bw(i,j+1,k+1);
    // up-north-east
    res(i+1,j+1,k+1) = 1 * bw(i+1,j+1,k+1);
}
... all other directions ...

norm1 = (res-resold).norm1();
resold = res;
} //end while
}// end 3d case
return;
}

The core algorithm takes three parameters: res is the object in which the result is written, bw is the input binary image and full_connected is a boolean flag indicating which kind of connection between voxels we are looking for; if full_connected is set to false will be considered connected only voxels which share a face, otherwise will be considered connected voxels which share an edge or a corner.

This is a complete algorithm which starts from a pixel and extracts the component connected to it:

```cpp
template <typename T>
template <typename S>
void im3d::image3d<T>::connected_component (image3d<S> & res,
    uint const & i, uint const & j, uint const & k,
    real threshold, bool full_connected) const
{
    image3d<S> bw;
    uint I(i), J(j), K(k);
    // initialize res and resold as a black image and norm1!=0
    res.setdim(this->dimx,this->dimy,this->dimz);
    res.seth(this->hx,this->hy,this->hz);
    if(I==0) ++I;
    if(J==0) ++J;
    if(I>=this->dimx-1) I = this->dimx-2;
    if(J>=this->dimy-1) J = this->dimy-2;
    if(this->dimz==1)
        K=0;
    else
    {
        if(K==0) ++K;
        if(K>=this->dimz-1) K = this->dimz-2;
    }
    this->im_to_black_and_white(bw,threshold);
    if( bw(I,J,K)==0 )
```
It takes some more parameters: three integer indexes indicating the starting point of the extraction process and a real threshold used in the binary conversion of the input image.

At last, median filter. It is a powerful non linear filter very useful when dealing with the characteristic noise originated by image acquisition. Precisely because of its non-linearity we choose to put it into image3d rather than in filtering (where we implement linear filters through the usage of convolution and Fourier transform, see section 2.5). Here is the code:

```cpp
template <typename T>
void im3d::image3d<T>::median_filter(image3d<T> & res, int const & radius) const
{
    // check consistency of dimensions
    // computing dimension of the mask
    int dimmask = (2*radius+1)*(2*radius+1);
    // 2D CASE
    // 3d case
    else
    {
        dimmask *= (2*radius+1);
        // initialize mask
        std::vector<T> mask;
        mask.reserve(dimmask);
        // internal nodes
        #pragma omp parallel for private (mask)
        for (int I = radius; I < dimx-radius; ++I)
            for (int J = radius; J < dimy-radius; ++J)
                for (int K = radius; K < dimz-radius; ++K)
                {
                    for (int i = -radius; i < radius+1; ++i)
                        for (int j = -radius; j < radius+1; ++j)
                            for (int k = -radius; k < radius+1; ++k)
                                mask.push_back( (*this)(I+i,J+j,K+k) );
                    sort( mask.begin() , mask.end() );
                    res(I,J,K) = mask[dimmask/2];
                    mask.resize(0);
                }
    // special treatment such as flipping the image for faces, edges and corners
    {...
    }
    // end 3D version
    return;
}
```

## 2.3 Class: abstractinterface and interface

These two classes have been designed to handle all the input/output flow between the environment defined by our namespace *im3d* and a visual toolkit library, whose choice is left to the developer. In our case, we have decided to refer to the VTK library [1], [22].
**Abstractinterface:** is a purely virtual abstract class serving the role of template guide for the developer who wishes to migrate from VTK to another library. In this sense, it defines some members that any image handling class should have in order to function properly within our implementation. In particular it should contain some pointer to an image, as well as some methods that can be divided in three categories: conversion to and from the im3d class, showing of images (and in particular of contours, given that the purpose of the algorithm is to perform segmentation) and finally methods to write out results that may be opened in other software. Here is shown briefly the structure of the class:

```cpp
namespace im3d
{
    template <typename T, typename S>
    class abstractinterface
    {
    protected:
        T imageptr;
    public:
        abstractinterface () {};
        abstractinterface (std::string const & imagename) {};
        abstractinterface (image3d<S> const & myim) {};
        virtual ~abstractinterface () {};
        T const & getimageptr() { return this->imageptr; }
        virtual void convertfromimage3d (image3d<S> const & myim) =0;
        virtual void convert2image3d (image3d<S> & myim) const =0;
        virtual void show_contour (real const & level=0) =0;
        virtual void show_image () =0;
        virtual void show_image_and_contour (real const & level=0) =0;
        virtual void show_contour_with_background_image
        (abstractinterface & backgroundimage, real const & level=0) {};
        virtual void get_coordinates (uint & i, uint & j, uint & k) =0;
        virtual void get_coordinates (uint & i1, uint & j1, uint & k1,
        uint & i2, uint & j2, uint & k2) =0;
        virtual void write (std::string const & imagename, std::string const & extension) const =0;
    };
}
```

**Interface:** is merely the concretization of what has been stated above in our case. It has been developed in a flexible way with three constructors: a default that only instantiates the necessary memory, one from an `imagename` that can handle various image formats (mhd, bmp, dcm, jpeg, dicom) based on the extension given by the user in an automatic way and one from an `im3d` to visualize, for example, the results of the computations. Also the class contains the necessary members to visualize images implemented using the native VTK pipeline.

```cpp
... include other files ...
#include <vtkImageData.h>
#include <vtkMetaImageWriter.h>
... other VTK include files ...
namespace im3d
{
    // STATIC VARIABLES
    static vtkSmartPointer<vtkRenderer> RENDERER=NULL;
    static vtkSmartPointer<vtkRenderWindowInteractor> RENWININT=NULL;
    template <typename S>
    class interface : public abstractinterface < vtkSmartPointer<vtkImageData>, S >
```
As we can see, in the first part of the header file of interface we can find the declarations (and some inline definitions) of the methods inherited from abstract interface. The visualization methods in this part such as show_image, show_contour, etc. are conceived in order to be...
easy to handle even for users who do not know the VTK library. In the second part there are methods that are used in order to reproduce the visualization pipeline of VTK, these methods are useful when the user wants to implement an alternative view from those already implemented.

The class has two static members, they are two VTK objects used in the visualization pipeline. This has been done in order to allow a VTK expert user, who wants to implement manually an alternative visualization pipeline of VTK, to represent the image data through the static members of the class interface.

This generality in terms of visualization possibility is at expense of simplicity, in fact we could have implemented only the members of the base class abstract interface but since interface was one of the first class we have implemented and deals with visualization of image we designed this class without knowing what kind of visualization we would need; so we kept more options available.

Here we report the method show_contour_with_background_image, it is one of the most complete methods and it serves to understand the visualization pipeline of VTK:

```cpp
template <typename S>
void im3d::interface<S>::show_contour_with_background_image (interface & backgroundimage, 
                           real const & level)
{
    int dims[3], dimz;
    this->imageptr->GetDimensions(dims);
    dimz = dims[2];
    vtkSmartPointer<vtkRenderer> renderer=NULL;
    vtkSmartPointer<vtkRenderWindowInteractor> renWinInt=NULL;

    interface::newrendering(renderer,renWinInt);
    this->setopacity();
    this->setcolour();
    this->addcontour2renderer(renderer, level);
    vtkSmartPointer<vtkRenderWindow> renWin = vtkRenderWindow::New();
    renWin->SetInteractor(renWinInt);
    renWin->AddRenderer(renderer);

    vtkSmartPointer<vtkImagePlaneWidget> imagePlaneWidgetZ = vtkImagePlaneWidget::New();
    imagePlaneWidgetZ->SetInput(backgroundimage.getimageptr());
    imagePlaneWidgetZ->SetInteractor(renWinInt);
    imagePlaneWidgetZ->SetPlaneOrientationToZAxes();
    imagePlaneWidgetZ->DisplayTextOn();
    imagePlaneWidgetZ->UseContinuousCursorOn();
    imagePlaneWidgetZ->On();

    if (dimz!=1)
    {
        vtkSmartPointer<vtkImagePlaneWidget> imagePlaneWidgetX = vtkImagePlaneWidget::New();
        imagePlaneWidgetX->SetInput(backgroundimage.getimageptr());
        imagePlaneWidgetX->SetInteractor(renWinInt);
        imagePlaneWidgetX->SetPlaneOrientationToXAxes();
        imagePlaneWidgetX->DisplayTextOn();
        imagePlaneWidgetX->UseContinuousCursorOn();
        imagePlaneWidgetX->On();

        vtkSmartPointer<vtkImagePlaneWidget> imagePlaneWidgetY = vtkImagePlaneWidget::New();
        imagePlaneWidgetY->SetInput(backgroundimage.getimageptr());
        imagePlaneWidgetY->SetInteractor(renWinInt);
        imagePlaneWidgetY->SetPlaneOrientationToYAxes();
    }
}
```
imagePlaneWidgetY->DisplayTextOn();
imagePlaneWidgetY->UseContinuousCursorOn();
imagePlaneWidgetY->On();
}

interface::startshowing(renderer, renWinInt);
this->setopacity();
this->setcolour();
interface::deleterendering(renderer, renWinInt);
return;
}

First, the method newrendering assigns the vtkRenderer and the vtkRenderWindowInteractor. Then a vtkRenderWindow, which manages the renderer and the interactor, is created. Second, three planes for the visualization of a 3D image are created and the background image is assigned to them. Finally the startshowing method initiates the visualization.

2.4 Class: unsteady_poisson_functor

The aim of this class is to solve the linear system that arises from the finite difference approximation of the unsteady Poisson problem (1.5.20). We have implemented this class as a functor in the separate namespace lapl. Each call to the functor has to deal with solving a single time step of the Poisson problem, hence to reach equilibrium user has to call it iteratively. The choice of boundary conditions to close the problem depends on the function associated with the functor and it is left to the user. The full Dirichlet and full Neumann ones are already implemented in namespace lapl using Gauss Seidel discretization (1.5.20), but user can implement himself any other function with different kind of boundary condition or with a different solver. Hence, flexibility is the main reason of this implementation choice. First of all have a look to the header file poisson.hxx.

...  
namespace lapl
{
    template <typename T>
    class unsteady_poisson_functor
    {
        private:
            void (*f) (im3d::image3d<T> & res, im3d::image3d<T> const & b, T const & dt,
                    T const & bc, im3d::image3d<T> const & input);
        public:
            unsteady_poisson_functor (void (*f) (im3d::image3d<T> & res,
                                                  im3d::image3d<T> const & b,
                                                  T const & dt, T const & bc,
                                                  im3d::image3d<T> const & input) ):f(f) {};
            void operator () (im3d::image3d<T> & res, im3d::image3d<T> const & b,
                              T const & dt=1., T const & bc=0.,
                              im3d::image3d<T> const & input = im3d::image3d<T>() )
            {
                this->f (res,b,dt,bc,input);
                return;
            };
            void changef (void (*f) (im3d::image3d<T> & res, im3d::image3d<T> const & b,
                                      T const & dt, T const & bc, im3d::image3d<T> const & input))
            {
                this->f = f;
            };
};

33
This is a classic implementation of a functor. Note that \texttt{operator()} (and accordingly the associated function) has to take as parameters the following:

1. \texttt{res} where resulting function is written.

2. \texttt{b} where the rhs of the equation is stored, in the case of our algorithm the forcing function (1.5.17).

3. \texttt{dt} and \texttt{bc} to store the time step and the value of the constant boundary condition (default respectively equal to 1 and 0)

4. \texttt{input}, the input function to be update at the next step. Default is an empty image because if you use an 'in-place' method to solve linear system (like Gauss Seidel) you would prefer to save input image directly in parameter \texttt{res} to limit memory usage.

Let’s see now the implementation of the \texttt{DirGaussSeidel} function.

```cpp
template <typename T>
void lapl::DirGaussSeidel (im3d::image3d<T> & res, im3d::image3d<T> const & b,
T const & dt, T const & bc, im3d::image3d<T> const & input){

uint const X=res.getdimx(), Y=res.getdimy(), Z=res.getdimz();
real const hx=res.gethx(), hy=res.gethy(), hz=res.gethz();
// BC ASSIGNMENT
// 2D case
if (Z==1){ ... }
// 3D case
else{
  // face i=0, i=X-1
  for(uint k = 0; k < Z; ++k)
    for(uint j = 0; j< Y; ++j)
      res(0,j,k) = res(X-1,j,k) = bc;
  ... other faces ...
}
```

The first part it is simply dedicated to the assignment of the boundary values. Note also that there is not any checking on the coherence of the dimensions of the result and the forcing function. This is an implementative choice because despite this function solves one step of the unsteady Poisson problem, most of time it could be used in a loop to reach the equilibrium, in this case introducing an if statement there would be a loss of performances.

```cpp
} //end namespace lapl
```
// 2D case
if (Z==1)
  {
    real const htilde = 1./dt + 2./(hx*hx) + 2./(hy*hy);
    ...  
  }
// 3D case
else{
  real const htilde = 1./dt + 2./(hx*hx) + 2./(hy*hy) + 2./(hz*hz);
  for(uint i = 1; i<X-1; ++i)
    for(uint j = 1; j<Y-1; ++j)
      for(uint k = 1; k<Z-1; ++k)
        res(i,j,k) =
        ( (res(i+1,j,k) + res(i-1,j,k))/(hx*hx) + 
          (res(i,j+1,k) + res(i,j-1,k))/(hy*hy) + 
          (res(i,j,k+1) + res(i,j,k-1))/(hz*hz) +
          b(i,j,k) + res(i,j,k)/dt ) / htilde;
  return;
}

Computation of the internal pixels is performed without saving any matrix in the memory as explained in section 1.5 and following equation (1.5.22). Looking at the code it becomes clearer the reason why scrolling in an orderly way the coordinates of the pixels only the ones with a lower value (with a -1 in their coordinates) are considered in an implicit way (at iteration \((i,j,k)\) they have already been updated previously). The Neumann implementation instead is more complicated because the boundary pixels are part of the unknown, hence you have to discretize the problem also in the boundary computing a different normal in each face, edge or angle of the domain. Moreover you have to do this operation respecting the growing order of the coordinates and this leads in a large number of subsequent for-loops. See the code or the doxygen guide for more details.
Note that inside the method there is not any computation of the error to make the user know how far he is from the equilibrium, because this calculation would cost almost as much as the rest of the function. Hence the user can choose whether to calculate the residual or not outside of the function.

### 2.5 Class: filtering

The *filtering* class is conceived to perform convolution between a pre-built real and even filter and a general real function. Once again, in order to achieve modularity and leave maximal space for code optimization and customization, this class belongs to a different namespace and is entirely detached from the rest of the code. Throughout the image segmentation algorithm (but this may be extended to a large class of image treatment algorithms) many convolution operations need to be performed. Needless to say, an efficient implementation will yield very significant improvement in terms of computational costs, which is why also a parallel implementation has been devised.
This class is implemented as a functor like class *unsteady_poisson_functer*, but it has a more complex structure exploiting filters as static members.

```cpp
namespace conv
{
  template<typename T>
  class filtering
  {
```
The filter representing the first factor of the convolution is stored in the static member \textit{time\_filter}. To perform a convolution with a filter user has to perform the following steps:

1. He has to decide what kind of filter he’d like to build choosing from those available (using the related static function) or building it by himself exploiting the fact that it is a public member. All filters must be real and even in order to ensure that its DFT is also real and even.

2. He has to call the static function \textit{build\_freq\_filters}. In this way the DFT of the time filter will be stored in the static members \textit{cycl\_freq\_filter} and \textit{acycl\_freq\_filter}. The difference between this two members is only their dimensions: the former is smaller and is used to perform a cyclic convolution, the latter has the zero-padded dimensions to perform a Discrete Convolution exploiting theorem 1 as explained in section 1.5.

3. He can construct a functor choosing the kind of convolution desired between those available at the end of the previous header file or building itself a function to pass
to the functor. In our algorithm, in order to perform a fast convolution, we use the function \textit{acyclic\_fftw\_convolution} that exploit \texttt{fftw} library \cite{9} to compute DFT using Fast Fourier Transform algorithm.

Let’s see more in details the implementations of the main functions.

**Build Filters:** The already implemented \textit{build\_\_filter} simply initializes the time filter, for instance function \textit{build\_gaussian\_filter} stores in the related static member a Gaussian kernel using convention that the pixel in the middle position in each direction corresponds to the origin of the Cartesian axis.

Let’s see the implementation of function \textit{build\_freq\_filters} noting at the beginning that it needs factor \( f \) as parameter only to know its dimensions.

The first steps are preparing \texttt{fftw} library to use \texttt{openmp} (in order to improve performance exploiting all processors available in the computer) and computing the correct dimensions of the frequency filters. Note that, since \texttt{fftw} is a C library, input and output vectors are initialized as C arrays of complex numbers.

```cpp
template <typename T>
void conv::filtering<T>::build_freq_filters (im3d::image3d<T> const & f)
{
    // allowing usage of fftw3\_omp library to improve performances
    if (omp_get_max_threads()>1)
    {
        int good = fftw_init_threads();
        if(good)
            fftw_plan_with_nthreads(omp_get_max_threads());
        else
            std::clog << "WARNING::build_freq_filters: thread creation error" << std::endl;
    }

    // getting time filter dimensions
    int const Nhx = filtering<T>::time_filter.getdimx(), Nhy = ..., Nhz = ...;

    // SET FREQUENCY FILTERS DIMENSIONS
    // Set correct acyclic\_freq\_filter dimensions.
    int const Nx = f.getdimx() + Nhx - 1;
    int const Ny = f.getdimy() + Nhy - 1;
    int const Nz = f.getdimz() + Nhz - 1;
    // Set correct cyclic\_freq\_filter dimensions.
    ... [Ncx, Ncy, Ncz] ...
    // assigning dimensions to frequency filters
    filtering<T>::acycl_freq_filter.setdim(Nx,Ny,Nz);
    filtering<T>::cycl_freq_filter.setdim(Ncx,Ncy,Ncz);

    // ***** acyclic\_freq\_filter *****
    fftw_complex *in, *out;
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*((Nx*Ny*Nz)));
    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*Nx*Ny*Nz);
}
```

To build frequency filters using \texttt{fftw} library, the origin needs to be in the first pixel, hence the time filter has to be copied in the correct way into the C input array of the DFT reflecting values with negative coordinates in each corner of the volume and adding in the middle the correct number of zeros in respect with the zero padding procedure explained in section 1.5.

```cpp
// VALUES COMPUTATION
// compute position of the central value of the time_filter in every direction
int const shiftx = floor(static_cast<real>(Nhx)/2.), shifty = ..., shiftz = ...;
```
// copying time_filter values in the correct way
// to obtain a coherent Fourier transform using fftw
#pragma omp parallel
{
    // starting parallel section
    #pragma omp for
    {
        // initializing the whole vector to nil values
        for(int i = 0; i < Nx; ++i)
            for(int j = 0; j < Ny; ++j)
                for(int k = 0; k < Nz; ++k){
                    in[Ny*Nz*i + Nz*j + k][0] = 0.;
                    in[Ny*Nz*i + Nz*j + k][1] = 0.;
                }
    }
    #pragma omp for
    {
        // block 1: x=0...X/2, y=0...Y/2, z=0...Z/2
        for(int i = 0; i < shiftx+1; ++i)
            for(int j = 0; j < shifty+1; ++j)
                for(int k = 0; k < shiftz+1; ++k){
                    in[Ny*Nz*i + Nz*j + k][0] =
                        filtering<T>::time_filter(shiftx+i,shifty+j,shiftz+k);
                }
    }
    #pragma omp for
    {
        // block 2: x=0...X/2, y=0...Y/2, z=Z/2...Z
        for(int i = 0; i < shiftx+1; ++i)
            for(int j = 0; j < shifty+1; ++j)
                for(int k = Nz-shiftz; k < Nz ; ++k){
                    in[Ny*Nz*i + Nz*j + k][0] =
                        filtering<T>::time_filter(shiftx+i,shifty+j,k-Nz+shiftz);
                }
    }
    ... blocks 3-->8 ...
}

// ending parallel section

Once the input has been correctly initialized, the fftw3 library specs require to create a
fftw Plan before calling the execution command in which user has to specify the kind of
desired DFT (forward or backward). Without significant overhead, the fftw3 plan decides
automatically which is the best way to perform that particular DFT.

// Define fftw Plan
fftw_plan forw_p;
forw_p = fftw_plan_dft_3d(Nx,Ny,Nz, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
// compute fftw
fftw_execute(forw_p);

Finally the result is converted from the C array to the image3d static member. Note that the
former is an array of complex numbers and the latter is a vector of real numbers, however
we are sure that the imaginary part of the output vector is nil thanks to the property that
a DFT of a real and even function is still real and even. Hence, working with real and even
time_filter is strictly necessary to the good working of this class.

// copying out values in acycl_freq_filter
#pragma omp parallel for
for(int i = 0; i < Nx; ++i)
    for(int j = 0; j < Ny; ++j)
        for(int k = 0; k < Nz; ++k)
            filtering<T>::acycl_freq_filter(i,j,k) = out[Ny*Nz*i + Nz*j + k][0];

// Delete data

38
We limit our treatment to the computation of the \textit{acycl freq filter}. The cyclic one has a similar procedure but with different number of zeros during the zero-padding and, depending on the dimensions of the factor, with the addition of the procedure to periodize the filter. We omit it because it is useless for the RSFE algorithm. Note finally that the function just described is automatically called at the end of each \textit{time filter} builder (hence they build also the frequency filters), but needs to be re-called explicitly every time you want to perform a convolution with a factor of different dimensions because frequency filters have to be accordingly update.

Perform Convolution: We now see the code of the function \textit{acyclic\_fftw\_convolution} that allows the functor to perform a Discrete Convolution as described in section 1.5. Once again, the first part of the function is dedicated to prepare \textit{fftw} library to use \textit{openmp}.

```cpp
template <typename T>
void conv::acyclic_fftw_convolution(im3d::image3d<T>& result, im3d::image3d<T> const& factor)
{
    // allowing usage of fftw3\_omp library to improve performances
    if (omp_get_max_threads()>1){
        ... same of build_freq_filters ...
    }
    // Saving useful dimensions in const variable
    int const dimx=factor.getdimx(), dimy=factor.getdimy(), dimz=factor.getdimz();
    int const Nx=(filtering<T>::acycl_freq_filter.getdimx()), Ny=..., Nz=...;

    // dimensioning output only if it is of wrong dims
    if (static_cast<int>(result.getdimx())!=dimx ||
        static_cast<int>(result.getdimy())!=dimy ||
        static_cast<int>(result.getdimz())!=dimz)
    {
        result.setdim(factor.getdimx(),factor.getdimy(),factor.getdimz());
        result.seth(factor.gethx(),factor.gethy(),factor.gethz());
    }

    //Input and Output arrays
    fftw_complex * FTout, *in, *out;
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(Nx*Ny*Nz));

    #pragma omp parallel
    { // starting parallel section
        #pragma omp for
        ...
    }
}
```

Then we proceed to the actual preparation of the convolution. Integration of \textit{fftw3} in a C++ code is somewhat involved, since this library was written in C. This means in particular that old constructs such as \textit{malloc} should be used to allocate memory, and that STL containers such as vectors should be replaced by classical arrays. This obviously results in a decrease in performance, since we are wasting time to simply copy data. However we have found that, being able to do it in a parallel block, the gain in performance from the use of the \textit{fftw3} library was more important than this loss. Finally, zero padding is performed for the reasons stated in section 1.5.
// zero padd values in input array
for(int i = 0; i < Nx; ++i)
    for(int j = 0; j < Ny; ++j)
        for(int k = 0; k < Nz; ++k){
            in[Ny*Nz*i + Nz*j + k][0] = 0.;
            in[Ny*Nz*i + Nz*j + k][1] = 0.;
        }

#pragma omp for
// copying factor values in input array
for(int i = 0; i < dimx; ++i)
    for(int j = 0; j < dimy; ++j)
        for(int k = 0; k < dimz; ++k){
            in[Ny*Nz*i + Nz*j + k][0] = factor(i,j,k);
            in[Ny*Nz*i + Nz*j + k][1] = 0.;
        }
} // ending parallel section

DFT can now be computed using fftw functions like in build_freq_filters. Once this is done, in order to apply theorem 1, we must multiply the two DFTs and inverse them. This is done in parallel and in the same way as above, by defining an inverse fftw plan and calling the execution program. Note that we have to perform a complex multiplication between a complex factor (\(FTout\)) and a real one (\(acycl_freq_filter\)).

// define forward fftw plan
FTout = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(Nx*Ny*Nz));
fftw_plan_forw.p;
forw.p = fftw_plan_dft_3d(Nx,Ny,Nz, in, FTout , FFTW_FORWARD, FFTW_ESTIMATE);
// compute fftw
fftw_execute(forw.p);
// delete useless data
fftw_free(in);
fftw_destroy_plan(forw.p);

// multiply
#pragma omp parallel for
for(int i = 0; i < Nx; ++i)
    for(int j = 0; j < Ny; ++j)
        for(int k = 0; k < Nz; ++k){
            FTout[Ny*Nz*i + Nz*j + k][0] *= filtering<T>::acycl_freq_filter(i,j,k);
            FTout[Ny*Nz*i + Nz*j + k][1] *= filtering<T>::acycl_freq_filter(i,j,k);
        }

// define inverse fftw plan
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*Nx*Ny*Nz);
fftw_plan_back.p;
back.p = fftw_plan_dft_3d(Nx,Ny,Nz, FTout, out, FFTW_BACKWARD, FFTW_ESTIMATE);
// compute inverse fft
fftw_execute(back.p);
// delete useless data
fftw_destroy_plan(back.p);
fftw_free(FTout);

Finally, the results need to be translated from a C array to our image3d class. Thanks to the theorem 1, we are once again sure that the result is a real function, hence we can ignore the complex part of the output. Since fftw library does not use any factor in front of both direct and inverse DFT, in order to obtain the correct result we have to divide it for the
number of elements. Furthermore, the library is obviously not able to understand spacing between pixel, hence we have also to multiply the result for the pixel measure to respect the quadrature rule (1.5.9).

```cpp
T const pixel_measure = result.gethx() * result.gethy() * result.gethz();
// * copying out values in result
// * multiplying it for pixel_measure to make discrete convolution a good approximation
// of the continuous one
// * dividing it for number of elements to compute a coherent inverse FFT
#pragma omp parallel for
for(int i = 0; i < dimx; ++i)
    for(int j = 0; j < dimy; ++j)
        for(int k = 0; k < dimz; ++k)
            result(i,j,k) = (out[Ny*Nz*i + Nz*j + k][0])*pixel_measure/(Nx*Ny*Nz);
// Delete data
fftw_free(out);
fftw_cleanup_threads();
return;
```

Note also that, as explained in section 1.5, we are not interested in the whole output vector, but only in the output in the domain of the image. This is the reason for which we copy only the first \((\text{dimx}, \text{dimy}, \text{dimz})\) elements in the result \text{image3d}.

We finally underline how unused variables are always deleted as soon as possible to avoid an excessive usage of RAM. In fact dealing with big images causes a very big dimension of the various fftw variables, hence it is really important to free the memory when is possible.

### 2.6 Class: rsfe_splitbregman

This class is devoted to the application of the RSFE Split-Bregman algorithm to an image. Because of modularity, an abstract \text{segmentation} class has been implemented that corresponds to the family of image segmentation algorithms in arbitrary dimension that use a level set method. It therefore contains whatever is needed for these to work, and little more. If one day a new developer wishes to implement another segmentation algorithm, be it simpler or more complex, that still uses the level set function, he will be able to do so by inheritance from this class, thus need in very little modification in the rest of the code except for the implementation of the method itself.

```cpp
namespace segm
{
    template <typename T>
    class segmentation
    {
    protected:
        T alpha;
        im3d::interface<T> image;
        im3d::interface<T> levelset;
        im3d::image3d<T> phi;
    public:
        //CONSTRUCTOR
        segmentation() {};
        virtual inline void set_alpha (T const & alpha) {this->alpha=alpha;};
        virtual void apply(im3d::image3d<T> const& myim, im3d::image3d<T> const& init) =0;
```
virtual void apply(im3d::image3d<T> const& myim) =0;
inline im3d::image3d<T> getoutput () const {return this->phi;}
virtual inline void show_contour ()
virtual inline void show_levelset ();
virtual inline void show_levelset_and_contour();
virtual inline void show_levelset_with_background_image();
}; // end class
} // end namespace segm

The rsfe_splitbregman class is instead dedicated at the implementation of the RSFE Split-Bregman algorithm following the steps described in the previous chapter. It makes the hypothesis that an object of class image3d has been instantiated and is somehow representative of an image. We would like to point out once again that how this is done is of no concern to the class itself, meaning that if the conversion from any type of image to image3d is carried out in a correct way, then the algorithm will work just fine regardless. This was a choice of implementation made to encourage future extensions and modifications of our code.

namespace segm
{
    template <typename T>
    class rsfe_splitbregman : public segmentation<T>
    {
        private:
            enum bc_type {NEU=1, DIR};
            enum ed_type {GRAD=1, GRAD_THRESHOLD, LBP};
            // GENERAL PARAMETERS
            std::string getpotfile, outputname, logfilename;
            bool verbosity, onthego, save_current, end_now;
            uint maxiter, showfrequency, dumpfrequency;
            // RSFE PARAMETERS
            T sigma, lambda, lambda_1, lambda_2, nu, beta, dt, epsilon, a0, b0;
            uint ls_steps;
            // CONNECTED COMPONENT
            bool auto_extract_conn_comp;
            real cc_init_pixel_x, cc_init_pixel_y, cc_init_pixel_z;
            uint auto_extract_conn_comp_freq;
            // EXPERT
            bc_type bc;
            ed_type edge_detector;
            T ls_tol, edge_detector_sigma, tol;
            lapl::unsteady_poisson_functor<T> onestep_poisson;
            conv::filtering<T> cv;
            bool gaussian_pixel_approach;
            // NOT DIRECTLY SETTABLE
            int ix, iy, iz, fx, fy, fz;
            uint dimx, dimy, dimz, space_dim;
            real hx, hy, hz;
            bool init_variables;
            // PRIVATE MEMBERS USED BY APPLY
            void set_heaviside (im3d::image3d<T> & Heps) const;
            void initialize_phi_with_cube ();
            void initialize_phi_with_init (im3d::image3d<T> const & init);
            void edge_detect(im3d::image3d<T> & res, im3d::image3d<T> const & f ) const;
            void cut_phi();
            void shrink (std::vector<im3d::image3d<T> > & res,
Looking at the header file it is clear that it is sufficient to call the apply method with an image as input to make the algorithm work. It could be necessary also a presetting of the parameters to obtain a good result. We will discuss this in the next chapter just underlining now that, to make the setting more comfortable, this can be done using a configuration file (fig. 2.3) automatically called by the apply method through GetPot [21] (hence, it is not necessary to call explicitly each set method).

Once the algorithm has been initiated, we offer the user the possibility to carry it out in an iterative way (as shown in fig. 2.4) by successively refining the choice of parameters and being able to restart the computations from any intermediate result. This is done through the implementation of two apply methods which differ only by their signature (as shown in the header file). The first one simply initializes the level set function $\phi$ as a cube, the second one instead allows user to set a selected image (for instance a previous result) as initial contour. The execution of the algorithm itself can be divided in four logical areas:

1. **initialization**: parameters setting, allocation of needed memory, contour initialization;

2. **construction of rhs**: computation and assembly of the right hand side of the Poisson problem, mainly through the use of convolution;

3. **linear system resolution**: perform some time steps using the Gauss Seidel method described in section 1.5;

4. **post-processing**: show partial results, update split-Bregman variables, connected component extraction, etc.

Let’s see now the code of the apply method (the second one, because the first simply calls the second after the initialization of the contour as a cube) dedicating a paragraph at each logical area. Note that we use frequently private methods, we remind to the doxygen guide for the whole code of them.

**Initialization.** At the beginning of every algorithm there is a phase where all the objects that play a role during the execution are instantiated, the memory is allocated if its evolution
Figure 2.3: example of a configuration file used by apply through GetPot

can be predicted, the variables may be assigned arbitrary starting values and so on. In our case, we first use this code snippet to redirect standard error output to a log file if its path has been specified by the user. In this way user can find in a text file all the execution history like all the changes of the parameters.

```cpp
void segm::rsfe_splitbregman<T>::apply (im3d::image3d<T> const & myim,
   im3d::image3d<T> const & init)
{
  // redirecting stderr if logfilename is set
  if (this->logfilename !="")
    stderr = freopen (this->logfilename.c_str(),"a",stderr);

  // log parameters
  std::clog << "- RSFE/Split-Bregman Algorithm for 3d image segmentation" << std::endl;
  if (this->getpotfile !="")
    {
      if (this->verbosity)
        {
          std::clog << "- Initializing parameters of algorithm from file " <<
                     this->getpotfile << " (section splitbregman/init/)");
        }
    }

  // actual initialization begins: first the parameters of the algorithm are loaded using
  // a configuration file like above if a getpotfile is set, then the dimensions of the image is guessed
  // and finally, if necessary, the level set function is initialized from a given function. This last
  // functionality has been implemented to allow the user to stop the algorithm whenever he
  // wants and be able to resume it later on. In particular, we have found that sometimes this
  // can help convergence as it allows to modify some parameters in the meantime.

  std::clog << "- RSPE/Split-Bregman Algorithm for 3d image segmentation" << std::endl;
  if (this->getpotfile !="")
    {
      if (this->verbosity)
        {
          std::clog << "- Initializing parameters of algorithm from file " <<
                     this->getpotfile << " (section splitbregman/init/)");
        }
    }
Instantiate an object of class interface.

Instantiate an object of class image3d.

Bind together.

RSFE object.

Obtain results.

Reinitialize algorithm.

Final output.

We have chosen to print almost all the “verbose” output to clog for the reason that it is not absolutely necessary for the execution of the algorithm, and may end up polluting the console.

After this phase, all the necessary memory is allocated. Most of the variables in the algorithm have fixed sizes (mainly equal to that of the input image), however only some of them are created using a constructor that automatically sets the size, that ones necessary only in some parts of the algorithm are created empty to limit memory usage and enlarged later. Some of the variables that are initialized below correspond to the steps 1, 2 and 3 of algorithm 1.

//construction of all need variables with correct dimensions and spacing

Figure 2.4: Flowchart of the use of the rsfe_splitbregman class.
Private member `image` of class `interface` is initialized to be exploited in the visualization of partial result. Variable `b` is set to a very small positive value, because setting it to 0 would cause a floating point exception. This does not however affect the final outcome of the algorithm.

In the final stage of the initialization, some more complex quantities are precomputed since they do not change in the course of the algorithm. This includes performing some convolution operations, which would risk to create a bottleneck in the execution, corresponding to steps 1 and 2 in algorithm 2. This is done using the private functor `cv` and the fft algorithm with `openmp` exploiting all the CPU power available.

```cpp
// showing initial contour exploiting class interface private members
if ( this->showfrequency > 0 )
{
    // Build filters of the correct dimensions for the algorithm
    conv::filtering<T>::build_gaussian_filter (this->phi,
        this->sigma,
        3, // radius of filter
        this->gaussian_pixel_approach);

    // compute kones and kmyim at the beginning of the algorithm
    // (they don't change during execution)
    aux.setdim(this->dimx,this->dimy,this->dimz,1.);  // aux=1
    this->cv(kones,aux);
    this->cv(kmyim,myim);
    aux.setdim(0,0,0);
    // compute gforshrink using edge_detect private member (coeff. nu/lamda is included)
    this->edge_detect( gforshrink, myim );
}
```

Note that the `setdim(0,0,0)` is often called throughout the algorithm to temporarily free some unnecessary memory and to avoid an excessive usage of RAM when dealing with heavy images.

**Compute rhs.** Once all the pre-initialization has been done, the main loop may start. Inside the loop, we can still identify a dynamic initialization phase at the very beginning with the objective of checking if the user has changed some parameters on-the-go using the configuration file (section `onthego`) and in this case doing some initialization again.

```cpp
// algorithm loop
```
for (t=1; tolerance>tol && t<=this->maxiter && end_now==false; ++t)
{
    if (this->verbosity)
        std::clog << "-- Iteration " << t << std::endl;
    // if user has modified sigma on the go
    if (old_sigma!=this->sigma)
    {
        old_sigma=this->sigma;
        // Build filters with new sigma and compute again kones and kmyim
        ... like above ... 
    }
}

The computation of the right hand side of equation (1.4.21) requires a few intermediary steps. These are carried out mainly using the convolution product, as shown in steps 6-9 of algorithm 2. The use of *=, -= and so forth operations is preferred as they are carried out in place, without the need for temporary variables that may be memory consuming.

    // compute new heaviside using previous iteration result
    this->set_heaviside(Heps);
    // compute auxiliary variable num and den using convolution
    this->cv(num,Heps*myim);
    this->cv(den,Heps);
    // set the dimensions of Heps to zero to limit memory usage
    Heps.setdim(0,0,0);
    // compute f1 and f2
    f1 = num/den;
    num-=kmyim; // note: -= limits memory usage
    den-=kones;
    num/=den;
    // set the dimensions of den to zero to limit memory usage
    den.setdim(0,0,0);
    f2 = num; // f2 = (kmyim-num)/(kones-den)
    // set the dimensions of num to zero to limit memory usage
    num.setdim(0,0,0);
    // compute kr1 and kr2
    this->cv(kr2, f1*this->lamda1 - f2*this->lamda2); // factor = f1*lamda1 - f2*lamda2
    f1*=f1; f1*=this->lamda1; // note: *= limits memory usage
    f2*=f2; f2*=this->lamda2;
    f1-=f2;
    // set the dimensions of f2 to zero to limit memory usage
    f2.setdim(0,0,0);
    this->cv(kr1, f1); // factor = f1*f1*lamda1 - f2*f2*lamda2
    // set the dimensions of f2 to zero to limit memory usage
    f1.setdim(0,0,0);
    // choose bc value depending on kind of boundary condition
    real chosen_bc;
    if (this->bc==NEU)
        chosen_bc = 0;
    else if (this->bc==DIR)
        chosen_bc = this->a0;
    // compute one iteration of laplace unsteady equation with forcing function
    // depending on results of convolutions (kones, kmyim, kr1, kr2), myim, private
    // parameters (lamda1, lamda2, lamda) and shrink results at the previous iteration.
    // forcing function stored in kr2
    kr2*=myim; kr2*=2;
    aux.setdim(this->dimx,this->dimy,this->dimz, this->lamda1-this->lamda2);
    aux*=myim; aux*=myim; aux*=kones;
    kr2-=aux; kr2-=kr1; kr2/=this->lamda; kr2-=divdb;
    // set the dimensions of kr1 to zero to limit memory usage
Linear System Resolution. The solution of the linear system has been detailed in section 1.5. Concerning its implementation in the algorithm, the call to the functor `onestep_poisson` in a loop of the preset time steps are the only things needed to run resolution method.

```cpp
// solve linear system
// forcing function = (2*kr2*myim-kones*myim*myim*(lamda1-lamda2)-kr1)/lamda-divdb
T ls_err;
this->ls_tol ? ls_err=1000. : ls_err=-1.;
uint s;
for(s=0; s < this->ls_steps || ls_err > this->ls_tol; ++s) {
    if(this->verbosity && s==this->ls_steps-1 || this->ls_tol!=0)
        aux = this->phi;
    onestep_poisson(this->phi,kr2,this->dt,chosen_bc);
    if(this->ls_tol!=0 && s>=this->ls_steps-1)
        ls_err = (this->phi-aux).normL2()/aux.normL2();
}
```

The time loop ends after a preset time steps or when a tolerance is reached depending on some parameters setting, the correct strategy in the various applications will be discussed in the next chapter.

Post Processing. The post processing phase may be divided in three parts according to their different function: result processing and preparation for the next iteration, result display and interactive modification. In the processing and preparation phase, the function $\phi$ output of the linear system resolution is cropped to respect the maximal and minimal value constraint. Then the new values for $b$ and $d$ are computed, corresponding to steps 9 and 10 of algorithm 1.

```cpp
// cut all values of phi outside the range [a0,b0]
this->cut_phi();
// compute normL2 of error
tollerance = (this->phi-phiold).normL2()/phiold.normL2();
// initialize splitbregman variable d:
// d = grad(phi)
this->phi.grad(d);
// compute various norms useful to understand evolution of phi
if (this->verbosity) {
    ...
}
```

```cpp
// set the dimensions of phiold to zero to limit memory usage
phiold.setdim(0,0,0);
```
// partial update splitbregman variable b:
// b = b_old + d = b_old + grad(phi)
for (uint i=0; i<this->space_dim; ++i)
    b[i] += d[i];
// Update gforshrink if user has just changed edge_detector_sigma
if( old_eds!=this->edge_detection_sigma)
    this->edge_detection( gforshrink , myim );
old_eds = this->edge_detection_sigma;
// update splitbregman variable d:
// d_new = shrink(b, gforshrink) = shrink(b_old+grad(phi), gforshrink)
this->shrink(d, b, gforshrink);
for(uint i=0; i < this->space_dim; ++i)
{
    // update splitbregman variable b:
    // b_new = b-d_new = b_old+grad(phi)-shrink(b_old+grad(phi), gforshrink)
    b[i] -= d[i];
    // d = d_new - b_new
    d[i] -= b[i];
}
// compute div ( d_new - b_new ), this result will be used in the
// forcing function of unsteady laplace member in next iteration
im3d::div(divdb,d);
// set the size of d to zero to limit memory usage
d.resize(0);

We now deal with the connected component extraction. We have implemented a private member extract_connected_component() that, depending on some private parameters settable via GetPot, performs the extraction in some different ways exploiting the member connected_component of class image3d:

1. auto_extract_conn_comp = false: user can choose to extract or not to extract a connected component only when partial results are shown (this happens every showfrequency iteration). In particular: after the show he has to answer a question via terminal; then, in case of positive answer, the connected component is shown; finally answering other two questions he can choose to set or not to set it as current level set φ and to reinitialize or not all the algorithm variables restarting it from this connected component as initial contour.

2. auto_extract_conn_comp = true: the extraction is totally automatic and does not need the show of partial result. User has to set the frequency of the extraction (auto_extract_conn_comp_freq) and the coordinates of the initial pixel (cc_init_pixel_x, ...) in the configuration file. At this point, every auto_extract_conn_comp_freq iteration the private member check if the pixel is "white" (this means that the level set φ has reached that pixel) or not and, in the positive case, it extracts the connected component and reinitializes the algorithm from it as initial contour.

In the next chapter we discuss the importance of this option when dealing with medical images.

In the code we first deal with the second case.

// case total automatic connected component extraction
if ( this->auto_extract_conn_comp && this->auto_extract_conn_comp_freq!=0)
    if ( t % this->auto_extract_conn_comp_freq == 0 )
    {
        this->extract_connected_component();
        if (this->init_variables)

49
The first case, as explained, is instead joint with the visualization of the partial results that exploits the class interface. If private member onthego is true (default), during the show, the user has also the possibility to change parameters on the go exploiting private method update_param_onthego where GetPot features are used to read the configuration file.

The partial result and allowing user interaction every showfrequency

```cpp
if ( this->showfrequency!=0 && t%this->showfrequency==0 && t!=this->maxiter )
{
    this->levelset.convertfromimage3d(this->phi);
    // showing
    this->internal_show();
    // if not case total automatic
    if ( !(this->auto_extract_conn_comp && this->auto_extract_conn_comp_freq!=0))
    {
        // allowing showing of a connected component of phi and
        // setting it as current levelset
        this->extract_connected_component();
        if (this->init_variables)
        {
            divdb=0.;
            for (uint i=0; i<this->space_dim; ++i)
                b[i]=1.e-5;
            this->init_variables=false;
        }
    }
    // update parameters of the algorithm on the go
    this->update_param_onthego();
}
```

Last thing before the end of the algorithm loop deals the saving of the current level set \( \phi \) in a \textit{.mhd} file. This happens each dumpfrequency iteration or when user set on the go the private member save_current as true. This is a very important feature because allows to restart algorithm from a previous partial result.

```cpp
// saving partial results depending on values of private parameters
// dumpfrequency and save_current
if ( (dumpfrequency!=0 && t%dumpfrequency==0 && t!=this->maxiter) ||
    save_current==true )
{
    this->save_current = false;
    this->levelset.convertfromimage3d(this->phi);
    std::string aux = this->outputname;
    std::ostringstream oss;
    oss << t;
    aux += ".";
    aux += oss.str();
    this->levelset.write(aux);
} //end algorithm loop
```
We finally underline that, despite we use the percentage variation of $L^2$ norm as stop criterion, in the practical applications (that usually exploit connected component extraction) it is better to manually stop the algorithm when user likes the result because it is too difficult to find an objective stop criterion when the level set is continuously modified not only by the standard algorithm but also by our additional tools.

```cpp
this->levelset.convertfromimage3d(this->phi);
// last show
if (showfrequency>0 && this->ontheago==false)
{ ... }
// saving final result
if (dumpfrequency>0)
{ ... }
// set coordinates of initial rectangle to default values again
this->ix=-1; this->iy=-1; this->iz=-1;
this->fx=-1; this->fy=-1; this->fz=-1;
// setting stderr again to default
if (this->logfilename!="")
  fclose (stderr);
return;
```

Last part of the code is simply dedicated to the last show, to the last saving, to the reinitialization of some variables and to the redirection of `stderr` to its default value.

## 2.7 Extra features

In this section we briefly discuss some extra packages and features used in our code like GetPot, code profiling and CMake compiling. Some of them are not present in the last version of the code but have had an important role in the development phase, hence they could be useful for future developers.

### 2.7.1 GetPot

The GetPot command line parser is a common tool for handling complex parameter specification. We have decided to use it in our project because of its simple implementation and integration (the whole library is contained in a single header file) and because of the powerful features it provides. We have already discussed the benefits provided from this tool at the class `rsfe_splitbregman` through the configuration file. We now quickly see a piece of code of the private member `set_param_from_getpot` to better understand how does it work.

```cpp
template <typename T>
void segm::rsfe_splitbregman<T>::set_param_from_getpot (std::string const & section)
{
    // preparing GetPot input
    std::string comment_start = "#";
    std::string comment_end = "\n";
    std::string input_file = this->getpotfile;
    if (input_file.compare(input_file.size()-4, 4, ".pot") != 0)
        input_file += ".pot";
    // GetPot initialization
    GetPot ifile(input_file.c_str(), comment_start.c_str(), comment_end.c_str());
    // temp variables of various type initialization
    real ttemp; int itemp; std::string stemp;
} 51
```
The code automatically checks if the file extension has been provided by the user or not, then proceeds to open the file. The new version of GetPot allows the creation of sections, so we have created a separate section for parameters that may be specified on the go, for improved readability of the configuration file (have a look at fig. 2.3). When a parameter is read, we check if it contains a default invalid value (which means that it has not been modified by the user, so it should not be overloaded) before loading it. Note also that invalid values of each parameter are not settable because every set method allows the setting only in the correct range giving a warning message otherwise.

### 2.7.2 Exception handling

A separated class has been implemented for floating point exceptions only. This is because other kinds of exceptions may have appeared in two places of the algorithm only: in the input/output phase and inside the execution. The latter would be very unusual since, other than mathematical computations, not many other manipulations are done at runtime, whereas the former is taken in charge by the VTK exception handling. Here it is a piece of code:

```cpp
struct BadDivision : public std::exception{
    virtual char const * what() const throw(){
        return "Error::Division by Zero. Program is not stopping. Compile with the option -DFPE_ABORT to automatically abort program at Floating Point Operation Exceptions";
    }
};
```

The methods of the class are not used explicitly in the final release of the code since we used them for the finding of floating point exception during development. However, if the program is compiled with the option -D FPE_ABORT it automatically aborts if a floating point exception appears during some computations and this could be useful for future developers.

### 2.7.3 Code profiling

In order to assess the performance of our code and identify bottlenecks we have used the gprof code profiler [17]. Compiling with the flag -pg activates the execution of the code profiler. If the program terminates normally, gprof produces an output file called gmon.out. This file is in a special format and needs to be processed to become readable to a human user. Calling from the command line gprof [options] will generate two important files: a call graph and a flat profile. The former specifies the relations that exist between the calls of each function: who called it and which other functions it calls, plus the number of times. The flat profile, on the other hand, shows how much time is spent in the execution of each function. For better readability, we have written a simple Python script (see appendix A.2)
to extract the most relevant information from these files. The use of the code profiler has allowed us, most notably, to identify an important bottleneck in a previous implementation of the mathematical operators, which did not use the fact that it could access directly the private data and, more generally, to easier understand what were the part of the code to be optimized to obtain better performance.

2.7.4 CMake

In our project we have used the CMake [15] cross-platform free software program to manage the build process. The main advantages of this approach have been the possibility to compile the same project both on Ubuntu Linux and MacOS machines, with different versions of the g++ compiler. Furthermore, providing our CMakeLists.txt file to the users, it becomes really easy also for not expert user to compile our library. CMake allows the programmer to set some predefined behaviour in terms of compiling (such as the linking of static or dynamic libraries, the creation of libraries, the setting of compiler flags and so on) without the need to recheck each option everytimes one wishes to compile on a different machine, or simply with a different compiler. In fact, CMake is a sort of smart make able to handle most of this tedious process by itself. CMake works by generating a makefile from a set of pre defined options which it reads from a text file called CMakeLists.txt. For example, our project comes “shipped” with our version of this file, and unless the user who wishes to compile it has installed his own libraries in an unusual place, they should be able to run CMake and then simply make. The program will automatically detect the operating system and provide the user with a terminal interface from which they are able to set various options to finally generate the appropriate makefile. The full text of our CMakeLists.txt may be found in appendix A.1 but a few parts will be commented in this section.

VTK and fftw3 libraries. As stated earlier, our software requires VTK in its current implementation to perform image input/output. For this, VTK needs to be installed on the system. When the makefile is being generated, CMake will automatically look for the VTK libraries thanks to this section in the code

```
#------------------------------------
# looking for VTK libraries
#------------------------------------
FIND_PACKAGE(VTK REQUIRED)
IF (NOT VTK
    USE
    RENDERING)
    MESSAGE(FATAL_ERROR "Example ${PROJECT_NAME} requires VTK_USE_RENDERING."
ENDIF(NOT VTK
    USE
    RENDERING)
INCLUDE( ${VTK_USE_FILE})
```

As can be seen, CMakeLists allows the use of conditional constructs to check the presence of certain packages or features. This allows to automatically generate tailored executables according to the software installed on the machine or to the machine’s capabilities.

Fast Fourier Transform libraries are found with the construct

```
find_path (FFTW_INCLUDES fftw3.h)
find_library (FFTW_LIBRARIES NAMES fftw3)
# find fftw_omp_library
find_library (FFTW_PARALLEL_LIB NAMES fftw3_omp)
```

which automatically finds the library if it is in a usual path (such as /usr/lib/) or looks for it in all the paths specified by the programmer. The difference between the use of FIND_PACKAGE and find_library is that VTK has a specific package for CMake, making its integration easier.
**Executables.** The creation of an executable is quite simple, and once again we are able to use conditionals to check whether the user actually wishes to build that particular executable or not. An interesting feature is also the CMAKE\_DEPENDENT\_OPTION that allows the display of some options depending on the value of some others.

```
#------------------------------------
# create EXECUTABLES
#------------------------------------
OPTION(BUILD_EXECUTABLES
    "Build all example executables, set OFF if you want to build only some executable"
ON)
CMAKE\_DEPENDENT\_OPTION ( BUILD\_IMAGE\_TOOLKIT
    "Build an executable useful to use all members of the library"
OFF
    "NOT\_BUILD\_EXECUTABLES"
ON )
# ... other dependent option ...
if(BUILD\_IMAGE\_TOOLKIT)
    ADD\_EXECUTABLE(image\_toolkit ./examples/image\_toolkit.cxx)
    # link EXECUTABLE to our libraries
    TARGET\_LINK\_LIBRARIES(image\_toolkit
        ${VTK\_LIBRARIES} ${FFTW\_PARALLEL\_LIB} ${FFTW\_LIBRARIES} )
endif()
# ... other executables ...
```

The syntax is quite simple, even for linking the libraries. In fig. 2.5 are shown two different outputs of the ccmake GUI depending on the values of variable BUILD\_EXECUTABLES.

**Doxygen Documentation.** In our project we have decided to use the Doxygen software (see [16, 25]) to automatically generate documentation. Fortunately, we have found that CMake and Doxygen are easily integrated. In particular the following lines define the documentation generation process.

```
#------------------------------------------------------------------
# make doc
#------------------------------------------------------------------
#-- Add an Option to toggle the generation of the API documentation
option(BUILD\_DOCUMENTATION
    "Use Doxygen to create the HTML based API documentation typing make doc on the shell"
OFF)
if(BUILD\_DOCUMENTATION)
    FIND\_PACKAGE(Doxygen)
    if (NOT DOXYGEN\_FOUND)
        message(FATAL\_ERROR
            "Doxygen is needed to build the documentation.
Please install it correctly")
    endif()
    #-- Configure the Template Doxyfile for our specific project
    configure\_file(Doxyfile ${PROJECT\_BINARY\_DIR}/Doxyfile @ONLY IMMEDIATE)
    #-- Add a custom target to run Doxygen when ever the project is built
    add\_custom\_target (doc
        COMMAND ${DOXYGEN\_EXECUTABLE} ${PROJECT\_BINARY\_DIR}/Doxyfile
        SOURCES ${PROJECT\_BINARY\_DIR}/Doxyfile)
```
Also, thanks to the existence of a CMake Doxygen package, the command FIND_PACKAGE will check if Doxygen is installed on the system, modifying the variable DOXYGEN_FOUND accordingly. Instead, the Doxygen configuration file may be specified through the command configure_file. If variable MAKE_DOCUMENTATION is ON, cmake automatically creates a make option to build the doxygen documentation, hence user can simply type on the command line make doc to generate both the latex and the html guide of the library.

Finally, we underline that typing t on the ccmake gui (fig. 2.5) the interface becomes richer of advanced option and allows user to set, for instance, the option he would like to give to the C++ compiler. The default values of the various advanced variables are accessible in the appendix A.1.

### 2.7.5 Executables

We have already explained that our library is a totally template library, hence it does not need compilation. However, we have implemented some .cxx files to create executables that could be useful to better understand how to use our classes or simply to quickly use it without writing more code. Two of them are very useful: rsfe_splitbregman and image_toolkit. The former gives the user the possibility of easily using our algorithm with some options that allow the setting of the initial contour, the getpotfile and the logfile (fig. 2.6-top), the latter instead provides an interactive menu useful to perform each tool of pre/post-processing or the algorithm itself (fig. 2.6-bottom).
Figure 2.6: rsfe_splitbregman helper (top) and image_toolkit main menu (bottom).
Chapter 3

Parameters Setting and Examples

In this chapter we try to explain as simply as possible how to set each parameter of the algorithm in order to obtain the desired result. We first have a look at an inhomogeneous test image describing the changes on the level set evolution varying some parameters. Then we explain what are the smartest strategies to adapt the algorithm to the medical images segmentation in order to obtain a good 3d reconstruction of a single organ.

We begin repeating the expressions of the functional and all the variables inside it to better underline where the parameters (in red) appear.

\[
\mathcal{F}(\phi(x)) = \nu \| \nabla \phi \|_g + \langle \phi, \lambda_1 e_1 - \lambda_2 e_2 \rangle_{L^2(\Omega)}, \tag{3.0.1}
\]

where \( e_i(x) = \int_\Omega K_\sigma(x-y) |u_0(x) - f_i(y)|^2 dy, \)

with \( K_\sigma(x) = \frac{1}{(\sqrt{2\pi} \sigma)^d} \exp\left\{ -\frac{|x|^2}{2\sigma^2} \right\}, \)

\[
f_i(x) = \frac{K_\sigma * \left[ M^\varepsilon_i(\phi(x)) u_0(x) \right]}{K_\sigma(x) * M^\varepsilon_i(\phi(x))},
\]

\[
M^\varepsilon_1(x) = \frac{1}{2} \left( 1 + \frac{2}{\pi} \arctan \left( \frac{\phi(x) - (a_0 + \alpha)}{\varepsilon |b_0 - a_0|} \right) \right), \quad M^\varepsilon_2(x) = 1 - M^\varepsilon_1,
\]

\[
\| f \|_g = \| g(u_0) f \|_{L^1(\Omega)}, \quad g(u_0(x)) = \frac{1}{1 + \beta |\nabla(u_0(x))|^2}.
\]

As shown in [26], without losing of generality we can set \( \nu = 1. \)

As discussed in the first chapter, also the minimization of (3.0.1) with the Split-Bregman method introduces the important parameter \( \lambda. \) Moreover the time discretization introduces other two parameters: the time step \( dt \) and the desired number of iterations of the Gauss-Seidel method \( ls\_steps. \)

3.1 Parameters Setting

In this section we focus on a simple 3D black and white image, consisting of a sphere placed in a box with a degrading background (fig. 3.1). The difficulty with this example is in the fact that because the background takes all the possible values of the image and there is a gradient of colours also inside the sphere, it is not directly possible to segment it just by choosing a constant intensity level or with other region-based or edge-based algorithms, hence it is a perfect test case for our algorithm.

First of all we would like to underline that the parameters of the algorithm are sensible to the range of intensity values taken from the image to segment, hence in our executable \( rsfe\_splitbregman \) we always change the range of intensity values (exploiting the method
change\_range\_of\_intensity of class image3d) in a the default interval \([0, 255]\) before applying the algorithm. In this way we are able to find a family of parameters as general as possible.

We now show the best result with its evolution in fig. 3.2, underlining the adopted parameters in table 3.1. The algorithm segments correctly almost all the sphere since the end of the first iteration. The leftmost part is the most difficult because the contrast with the background is lower, hence it needs five iterations to obtain a complete result.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial contour.</td>
<td>1st iteration.</td>
<td>2nd iteration.</td>
<td>3rd iteration.</td>
<td>5th iteration.</td>
<td>10th iteration.</td>
</tr>
</tbody>
</table>

Figure 3.2: Level set evolution using parameters in table 3.1.

Table 3.1: Parameters adopted to obtain result in fig. 3.2.

In the following paragraphs, starting from these parameters, we focus on the difference in the evolution of the contour caused by their modification one at a time. We just omit an analysis of the parameters \(a_0\) and \(b_0\) because, as seen in the first chapter, they simply indicate the lower and the upper bound of the level set \(\phi\). According to our experience, modifying their values has no advantages, but causes the necessity of an adequate rescaling of all the others. Hence, we fix their values respectively to 0 and 1 in all the examples of this chapter. Another general strategy adopted in the whole chapter consists of the setting of the initial contour inside the object of interest. This ensures that the level set \(\phi\) takes the value of \(b_0\) (white) inside the contour and of \(a_0\) (black) outside it and, furthermore, avoids the problem of aggregation between near objects in lots of practical cases.

The \(\lambda\) parameter. As seen in section 1.4, this parameter appears in the rhs of the Poisson problem and at the denominator of the second argument of the shrinkage operator. Hence,
\( \lambda \) controls the weight given to the projection of \( d \) on the space of \( \nabla \phi \). This in particular has an effect on the speed of movement of the contour. In fig. 3.3 it may be seen how, varying \( \lambda \), we are able to obtain essentially the same result but with a significantly different number of iterations: 50 for \( \lambda = 1 \) and 5 for \( \lambda = 0.1 \). Despite this seems to prove that a smaller value is always better, we will see in the next section that the property of slowing down the algorithm increasing \( \lambda \) can be really useful in lots of practical applications.

**The \( \sigma \) Parameter.** This parameter characterizes the dimensions of the Gaussian kernel \( K_\sigma \) that represents the first factor of all the convolution products in the algorithm. More in general, convolving an image with a Gaussian kernel produces a blurred image as output, an example with a medical image is shown in fig. 3.4. Looking at this image it appears clear that a too high \( \sigma \) will cause an aggregation between all objects inside the image. We have found that a quite general rule is to set \( \sigma \) an order of magnitude smaller than the diameter of the object of interest in the image. Hence, both for our test image (diameter equal to 30) and for the aorta in fig. 3.4 the correct value of \( \sigma \) is about 3.

Despite the higher \( \sigma \) is, the smoother the final contour becomes, when dealing with images made of near objects (typical in medical images) the side effect of aggregation discourages the strategy of a \( \sigma \) too much higher in respect with the above rule. However, we have also noted that a \( \sigma \) a bit bigger than the one of the rule speeds up the algorithm in the very first iterations and avoids the problem of the creation of other contours inside the one of interest. Hence, sometimes a good compromise could be to adopt an hybrid strategy starting from a bigger \( \sigma \) for a few iterations and after decreasing it on-the-go.

**The \( \lambda_1 \) and \( \lambda_2 \) Parameters.** These variables, like the first \( \lambda \), still have an effect on the evolution of the contour but more on the direction rather than the speed. They weigh the terms \( e_1 \) and \( e_2 \) one in relation to the other. These terms are related to the approximation.
of the image density with the functions \( f_i(x) \) in the two regions defined by the inside and outside of the contour. We have found that setting \( \lambda_2 > \lambda_1 \) (fig. 3.5-bottom) will encourage the contour to develop towards the outside of the object of interest overestimating it a bit also at the convergence, while the opposite strategy (fig. 3.5-top) seems to underestimate the contour especially during its evolution. The larger the difference between the two values is, the more this behaviour is evident.

Despite this seems to prove that the best strategy is to set \( \lambda_1 = \lambda_2 \) to avoid both overestimation and underestimation problems, sometimes setting these parameters in order to underestimate the contour during its evolution can help to avoid the problem of aggregation between near objects. However in general it is not suggested to set \( \lambda_1 \) and \( \lambda_2 \) with too different values because this could create other problems like the generation of artificial contours (fig. 3.5-bottom).

**The \( \varepsilon \) parameter.** \( \varepsilon \) is the shape parameter that determines how sharp is the approximation of the Heaviside function through the arctangent, which in turn influences the way the inside and outside regions are distinguished. Looking at the expression of function \( M^1_\varepsilon(x) \) in (3.0.1) and at fig. 3.6 it is evident that the smaller \( \varepsilon \) becomes, the better the arctangent approximates the Heaviside function.

Despite the effect of increasing its value is that of producing smoother contours, a larger value will create a larger area where we are neither inside or outside, thus deleting any fea-

![Figure 3.5: Evolution with \( \lambda_2 = 10^{-5} \) and \( \lambda_1 = 1.5 \times 10^{-5} \) (top) and vice versa (bottom).](image)

![Figure 3.6: Plot of function \( M^1_\varepsilon(x) \) varying \( \varepsilon \).](image)
ture significantly smaller than this area. The user must not be fooled by the apparent better quality of the results obtained in this way: one cannot be sure that the contour matches closely enough the desired geometry as we can be in presence of artificially enlarged or shrunk contours. This phenomenon is clear in fig. 3.7, where at the convergence we have reached a contour that is clearly too large for the targeted geometry.

Because it is a shape and size parameter, taking a value for $\varepsilon$ smaller than two orders of magnitude in respect of the image spacing (equal to 1 in our test image) produces no noticeable effect as shown in fig. 3.8. Since in our examples we always deal with images characterized by spacing of about 1, we will fix $\varepsilon = 10^{-3}$ in all the other examples of this chapter.

**The $\beta$ parameter.** As shown in (3.0.1), $\beta$ plays the role of weight for the edge detector at the denominator of the function $g(u_0(x))$. In our examples we set the edge detector in (1.4.10) equal to $|\nabla(u_0)|^2$, but in our library, setting some advanced parameters, user can choose as edge detector also the Gaussian blur of the gradient of the intensity function. The benefit of the edge detector is to enforce the contour to move in areas where the image has edges making this action less costly in terms of the functional energy. Hence, if the chosen edge detector is a good detector for the edges of the object of interest, increasing $\beta$ can speed the algorithm. However, since in most practical cases the edge detector is not able to find all the edges of interest, it is not suggested to increase it too excessively in respect to our default value of 100. User can exploit our executable `image_toolkit` to show the modulus of the gradient of the image to segment, if it is clearly a good edge detector he can increase the value of $\beta$, on the contrary it is better to decrease it. We last note that this is one of the most sensible parameters to the range of values of the function $u_0$ and to the parameters $a_0$ and $b_0$, thus changing them causes the necessity of a recalibration of $\beta$.

**The $dt$ and $ls$ steps parameters.** These parameters deal with the Gauss-Seidel discretization of the unsteady Poisson problem: $dt$ indicates the time step chosen from the user and $ls$ steps is the number of Gauss-Seidel iterations to do at each iteration of the algorithm. Hence, at fixed $ls$ steps, the decreasing of $dt$ slows down the algorithm. On the
contrary, at fixed \( dt \), the more \( ls \_steps \) user does at each iteration, the less iterations the algorithm needs to converge, but each iteration lasts more.

In section 1.5 we have shown that the Gauss-Seidel method for our problem is unconditionally stable, this means that there are not any stability conditions on the value of \( dt \). A good rule in general is to set its value of the same order of magnitude of the spacing of the spatial grid in order to maintain uniform the grid also in the time direction.

Regarding the number of iterations, it is possible to set the linear system tolerance \( ls \_tol \) among the advanced parameters. In this way the Gauss-Seidel method continues at each iteration till the residual is more than the fixed tolerance. This means slowing down each iteration resolving almost exactly the Poisson problem. Despite it seems a good strategy mainly because at the end algorithm needs less iterations to converge, for our experience in the medical application it is always better to perform only some \( ls \_steps \) at each iteration.

The reasons are mainly two: the first, as already explained in section 1.5, considers that the Poisson problem is nested in the other iterative problem of Split-Bregman and thanks to the robustness of the last one it is sufficient to do only few steps in order to ensure the global convergence of the whole algorithm; the second is more practical and, once again, has the goal to avoid aggregation between objects in an image. Indeed, as we will see in the examples of the next section, performing too many \( ls \_steps \) increases the risk to notice this problem. Hence both these parameters will be exploited using their property of slowing down the contour evolution decreasing their values.

### 3.2 Medical Images

In this section we are going to apply the \textit{Rsfe-SplitBregman} segmentation algorithm to real life medical images taking advantage of the study we have done before on the parameters of the algorithm. Here we treat two medical images containing the \textit{MRI} of two chests in which we want to segment the aorta. We start our simulations taking as parameters of the algorithm those we indicated to be the benchmarks in the previous section.

<table>
<thead>
<tr>
<th>iterations</th>
<th>( \sigma )</th>
<th>( \lambda )</th>
<th>( \lambda_1 )</th>
<th>( \lambda_2 )</th>
<th>( \varepsilon )</th>
<th>( \beta )</th>
<th>( dt )</th>
<th>( ls _steps )</th>
<th>( a_0 )</th>
<th>( b_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st → end</td>
<td>3</td>
<td>( 10^{-2} )</td>
<td>( 10^{-5} )</td>
<td>( 10^{-5} )</td>
<td>( 10^{-3} )</td>
<td>100</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Parameters adopted to obtain result in fig. 3.9.

These results have been obtained using the \textit{rsfe-splitbregman} executable. It allows to run the algorithm in batch mode if the user sets correctly the parameter \textit{show frequency} equal to zero. This test allows us to give an idea of the execution time of the algorithm. The test has been executed on a dual core intel core i5-2410M processor (clock frequency 2.30GHz) using two cores, the test image is a 3D meta image of \( 77 \times 196 \times 45 \) MET\_DOUBLE elements and the relative \textit{raw} file, in which data are stored, occupies 5.4MB. As we already said the parameters used in this simulation are the same as those listed in the beginning of the previous section and hence the fastest possible to reach a meaningful result. The computations took 14.552 seconds.

We now have to analyse the quality of the results obtained. The \textit{rsfe} algorithm has been developed in order to succeed in segmenting images presenting intensity inhomogeneity, indeed it was able to catch all the inhomogeneities of the image but this is exactly the reason why the results are not suitable for our purposes. Our aim is to be able to extract from this image only the whole ascending and descending aorta without all the vessels.
present in its neighbourhood, in order to use the level set of the segmented contour for creating a mesh for doing some numerical computations. Clearly, looking at pictures 3.9, the aim has not been reached because the algorithm “thinks globally” and hence it segments all the inhomogeneity inside the image and agglomerates vessels, aorta and heart in a unique contour.

We tried various parameters settings but none of them gave the desired result.

In order to overcome the global “vision” of the algorithm, we have developed and embedded an additional tool in our implementation of the Rsfe-SplitBregman segmentation method. It is a tool to extract a connected component from a given pixel allowing the user to decide the frequency of the extraction and its starting point in a completely automatic way during the execution of the segmentation. However in order to make effective the extraction tool we need to change the strategy we adopted in the choice of the parameters. In particular it will not be possible to use a batch approach, but user must interact with the execution of the algorithm by looking at the partial results and driving the execution of the program setting properly parameters on the go.

The following could be a general strategy:

1. starting the algorithm with a configuration that allows a fast evolution of the contour for few steps, it is important in this phase to choose a \( \sigma \) bigger than the one we would
choose with the previous strategy in order to make more robust the algorithm with respect to the choice of the initial contour;

2. check the partial results and decide whether or not to slow down the evolution speed of the contour, furthermore if the contour has reached the desired region decrease \( \sigma \);

3. once the contour has reached the object to be segmented, if there are risks that the contour could include other undesired objects in the nearby, then it is convenient to begin with the extraction of the component connected to a pixel inside the portion of the desired region that the contour has already reached. When we decide to initiate with the connected component extraction we must carefully choose the frequency of the extraction coherently with the evolution speed of the contour otherwise the extraction would be vane. A possible way to slow down the evolution is to increase \( \lambda \) or to decrease \( dt \) or \( ls\_steps \);

In the next two paragraphs we will show the strategy and the parameters we used in order to obtain the desired output from our algorithm.

**Aorta_P2A:** is the same image of the example above. In this test we changed the parameters and we approached the segmentation in a different way varying some parameters on the go, as can be seen in the table 3.3.

<table>
<thead>
<tr>
<th>iterations</th>
<th>( \sigma )</th>
<th>( \lambda )</th>
<th>( \lambda_1 )</th>
<th>( \lambda_2 )</th>
<th>( \varepsilon )</th>
<th>( \beta )</th>
<th>( dt )</th>
<th>( ls_steps )</th>
<th>( a_0 )</th>
<th>( b_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1\textsuperscript{st} \rightarrow 5\textsuperscript{th}</td>
<td>10</td>
<td>5</td>
<td>( 10^{-5} )</td>
<td>( 10^{-5} )</td>
<td>( 10^{-3} )</td>
<td>100</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6\textsuperscript{th} \rightarrow end</td>
<td>3.5</td>
<td>1</td>
<td>( 10^{-5} )</td>
<td>( 10^{-5} )</td>
<td>( 10^{-3} )</td>
<td>100</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Parameters adopted to obtain result in fig. 3.10.

Compared to the previous settings we greatly increased \( \lambda \) to slow down the evolution of the contour, which was too fast, and we increased \( \sigma \) to allow the initial contour to move faster in the region of interest. At iteration 3, the contour reached the pixel inside the aortic arch on which we set the beginning of the connected component extraction algorithm; this told us that the initial contour identified the object. So, at the first visualization stop of the algorithm (at iteration 5) we reduced \( \sigma \) to its “natural” value and then we decided to speed up the evolution by reducing \( \lambda \). Note that this is a delicate part of the algorithm execution: if we do not calibrate properly the speed, the pixel of the extraction and the visualization stop we must restart the algorithm, because, once the unwanted objects are connected to our contour it is worthwhile to restart the algorithm rather than to try to separate the contours. Looking at the figures 3.10 we see the initial contour, on which is highlighted the pixel of the extraction procedure (in the red circle), and its evolution. Note that after the fifth iteration even if we quickened the evolution because of the connected component extraction the growth of the contour will result slower. Indeed after the extraction pixel is reached, we set that the extraction procedure is preformed at every step allowing the evolution only in the region connected to the previous step contour: all other modifications will be discarded.

We stopped the algorithm at iteration 50 because we were satisfied with the result. After restarting again the algorithm from this last result we tried different parameters settings but the result was the same in each configuration: the ascending aorta was connected to the heart and the heart to the pulmonary artery.
The final result can be considered excellent. Indeed, we were able to segment correctly the descending and the ascending aorta and the aortic arch with its three vessels: the left subclavian artery, the left carotid artery and the brachiocephalic artery without connecting them with the pulmonary arteries and the superior vena cava which are very close to the contour we wanted to segment.

**Aorta3D:** This image still contains an MRI of a chest. Our purpose on it is the same as before: to try to segment only the aorta. It is a 3D meta image of $68 \times 156 \times 45$ voxels containing a MET\_DOUBLE data. Given the similarity with the previous image we tried to use an equal approach to segment it, indeed looking at the table 3.4 we find almost the same values of 3.3. The difference between the two configurations is the value of $\lambda$ after the third iteration. This difference is due to the major closeness of the superior vena cava to the aorta which makes this segmentation more complicated. Due to this, we tried various configurations in order to avoid the aggregation of the aorta with the superior vena cava but none of them succeeded. So, we selected the fastest configuration among them and hence that with the least $\lambda$.
As we can see in the first figure of 3.11 since the initial contour is closer to the extraction pixel the extraction procedure begins formerly compared to the previous paragraph example and hence we must change consequently the parameters.

![Initial contour](image1)

![10th iteration](image2)

![20th iteration](image3)

![40th iteration](image4)

**Figure 3.11:** Level set evolution using parameters in table 3.4.

We are satisfied of the result obtained in this segmentation even if it is not perfect. We succeeded in avoiding the connection of the pulmonary artery whilst we were not able to prevent the superior vena cava from connecting with the aorta. However it is possible through the use of 3D image editing tools doing some post processing in order to remove this defect from our result.

### 3.3 Example of Pre-Processing

In the previous section we have analysed two medical images obtained through MRI with a contrast agent injected in the aorta of the patient in order to improve the visibility. These kinds of images are perfect for our algorithm and don’t need any pre-processing phase since the object of interest (the aorta) is that one with more contrast with the background. Other kinds of medical images, especially if the aim is to segment a blood vessel, don’t respect this property. This happens, for instance, when dealing with X-ray computed axial tomography (CAT).

In this section, exploiting a CAT of a chest, we explain a quite standard procedure of pre-processing using the additional tools of the class `image3d` to modify these kind of images increasing the contrast of the object of interest.

Once again, we want to segment the aorta in fig. 3.12a. The original image has some evident problems: first the aorta has not an adequate contrast with the background, second the whole image is affected by an evident salt & pepper noise (typical in CAT). The pre-processing procedure for this image consists of the following steps:
1. **crop**: cropping the image nearby the region of interest (fig. 3.12b) is always the first step, this allows to improve performance of the algorithm avoiding an excessive use of memory and speeding up each iteration. Note that also the MRI images analysed in the previous section are the cropped versions of bigger ones.

2. **range of intensity**: in the original image we deal with intensity values in the range $[0, 255]$, the aorta takes values in the range $[60, 95]$ with a background of values not lower than 55, the darkest parts consist of lungs and all the organs containing air like bronchi, the clearest ones are the bones and a metallic artificial object present in this patient near the aorta. Using the `select_range_of_intensity` method with lower bound equal to 55, upper bound equal to 100 and the option of setting all the values outside this range equal to the lower bound we obtain the result in fig. 3.12c. The quality of the result immediately captures the attention, the aorta is now clearly the object with the highest contrast. Note that the metallic object is almost disappeared thanks to the option of setting the out of range values to the lower bound.

3. **median filter**: the last defect is that the image is clearly too pimpled, to remove this kind of noise it is suggested to use the median filter. The resulting image is shown in fig. 3.12d and highlights how this kind of filter deletes almost all the salt & pepper noise without side effects like blurring, for this reason it is preferred to a Gaussian blur filter. We have now obtained an image that satisfies the necessary prerequisites to apply our algorithm.

We lastly show in fig. 3.13 the result of the segmentation algorithm applied to the modified image in fig. 3.12d with the evolution of the level set $\phi$ at various iterations. In
Table 3.5 the used parameters are shown.

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$\lambda$</th>
<th>$\lambda_1$</th>
<th>$\lambda_2$</th>
<th>$\varepsilon$</th>
<th>$\beta$</th>
<th>$dt$</th>
<th>ls_steps</th>
<th>$a_0$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>0.08</td>
<td>$10^{-5}$</td>
<td>$10^{-3}$</td>
<td>100</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5: Parameters adopted to obtain result in fig. 3.13.

![Image](image.png)

(a) initial contour.  
(b) 5th iteration.  
(c) 10th iteration.  
(d) 15th iteration.  
(e) 25th iteration.  
(f) 40th iteration.  
(g) 70th iteration.  
(h) 100th iteration.

Figure 3.13: Level set evolution using parameters in table 3.5.

This picture is made of a grid of 73 × 150 × 126 voxels, each one with spacing of 0.77 × 0.77 × 1. Note that before the cropping we were dealing with an image of 228 × 245 × 144, hence with a five times heavier image. Like for the other medical images we used again the automatic connected component extraction option at each iteration, the initial pixel is shown in red in fig. 3.13a. On the contrary, this time there is no need for a change on-the-go of the parameters. We used a $\sigma$ a little larger in respect of the other cases because otherwise the creation of other contours inside some parts of the aorta disturbs the result. Thanks to the pre-processing procedure, we have obtained an excellent result despite we had begun from an image apparently not suitable for our algorithm. In fig. 3.14 it is evident how the aortic root has been correctly segmented, with the three sinus of Valsalva clearly visible. At the beginning of the ascending aorta there is also another blood vessel in input which is evident also in the original image (fig. 3.15). This is an artificial vessel, indeed in this patient a Ventricular Assist Device (VAD) has been installed. Cropping less the original image in the pre-processing phase, user can obtain a result embedding more details of the artificial vessel.

We finally underline that, continuing the algorithm for some other iterations, the aorta continues to grow downward inside the left ventricle. We stopped the algorithm because we want a result starting from the aortic valve. Hence, to use this result as a domain for a numerical simulation, the post-processing necessary steps consist of smoothing all the surfaces and creating all the boundary inlet and outlet surfaces cutting the aorta and the artificial vessel at the correct levels.
Figure 3.14: Final result seen from different perspectives.

Figure 3.15: Detail of the original image where the artificial vessel joins the aorta.
3.4 Conclusions and Possible Developments

In this project we implemented a library with the aim of segmenting medical images using the Rsfe-SplitBregman method.

As we saw in the section on medical images tests we succeeded in segmenting various kinds of images coming from different acquisition devices, such as CAT and MRI, also thanks to the pre/post-processing toolkit and the connected component extraction algorithm that we embedded in it. Moreover, one of the weaknesses of this method was the copiousness of parameters but we are convinced that, thanks to our study on their settings, this algorithm will be easier to use.

However, this method is not lacking of defects. Indeed, we noted that it becomes greedy of memory when dealing with big or high resolution images especially if the scalability parameter $\sigma$ assumes a big value. Again, the scalability parameter allows to segment also images presenting a strong inhomogeneity, but this feature can be a problem when dealing with contours very close to each other in which every value of $\sigma$ causes the contours to join together.

In our work we met these defects, consequently we cropped the images when the execution was too resource consuming but we were not able to do anything when contours join together (Aorta3D test). Hence, we think that a possible development of this work should be a post processor which allows to remove undesired contact between contours or, alternatively, the integration of the whole project with an already existing open source tool of image processing.

Another proposal could be the benchmarking of our implementation on other kinds of images providing a default parameters configuration for them or trying to make more automatic the choice of the parameters.
Appendices
Appendix A

Extra Code

A.1 CMakeLists.txt

#-------------------------------------------------------------
# set default values of some CMAKE variables on our project
#-------------------------------------------------------------

SET(BIN_PATH "./bin")
SET(3DIMSEG ./header/ )
set (CMAKE_BUILD_TYPE Release
CACHE STRING "Choose the type of build, options are:
None (CMAKE_CXX_FLAGS or CMAKE_C_FLAGS used)
Debug Release RelWithDebInfo MinSizeRel."
if(${CMAKE_SYSTEM_NAME} MATCHES "Darwin")
set (CMAKE_CXX_FLAGS "-Wno -deprecated -Wno-boole-conversions -fopenmp"
CACHE STRING "Flags used by the compiler during all build types."
else ()
set (CMAKE_CXX_FLAGS "-fopenmp -Wno-deprecated"
CACHE STRING "Flags used by the compiler during all build types."
endif()
set (CMAKE_CXX_FLAGS_DEBUG "-g3 -Wall -pedantic -DFPE_ABORT"
CACHE STRING "Flags used by the compiler during C++ debug builds."
set (EXECUTABLE_OUTPUT_PATH ${BIN_PATH}
CACHE PATH "Single output directory for building all executables."

#-------------------------------------------------------------
# starting project
#-------------------------------------------------------------

PROJECT (3dimagetoolkit CXX)
INCLUDE(CMakeDependentOption)
cmake_minimum_required(VERSION 2.6)

#-------------------------------------------------------------
# looking for STL Library
#-------------------------------------------------------------

find_library (LIB_DL NAMES dl)
find_library (LIB_M NAMES m)

link_directories (${LIB_DL//libdl.so/})
link_directories (${LIB_M//libm.so/})

#-------------------------------------------------------------
# looking for VTK libraries
#-------------------------------------------------------------

FIND_PACKAGE(VTK REQUIRED)
IF (NOT VTK_USE_RENDERING)
  MESSAGE(FATAL_ERROR "Example ${PROJECT_NAME} requires VTK_USE_RENDERING.")
ENDIF(NOT VTK_USE_RENDERING)
INCLUDE(${VTK_USE_FILE})

# looking for fftw libraries
#------------------------------------
# Find the native FFTW includes and library
#
#  # FFTW_INCLUDES   - where to find fftw3.h
#  # FFTW_LIBRARIES - List of libraries when using FFTW.
#  # FFTW_FOUND      - True if FFTW found.
#  # FFTW_PARALLEL   
if (FFTW_INCLUDES)
  # Already in cache, be silent
  set(FFTW_FIND_QUIETLY TRUE)
endif (FFTW_INCLUDES)
find_path (FFTW_INCLUDES fftw3.h)
find_library (FFTW_LIBRARIES NAMES fftw3)
# find fftw_omp_library
find_library (FFTW_PARALLEL_LIB NAMES fftw3_omp)

# handle the QUIETLY and REQUIRED arguments and set FFTW_FOUND to TRUE if
# all listed variables are TRUE
find_package_handle_standard_args
mark_as_advanced (FFTW_LIBRARIES FFTW_INCLUDES)

#------------------------------------
# create EXECUTABLES
#------------------------------------
OPTION (BUILD_EXECUTABLES "Build all example executables, set OFF if you want to build only some executable" ON)

CMAKE_DEPENDENT_OPTION (BUILD_SHOW_IMAGE "Build an executable useful to show a 3d image containing a level set function" OFF "NOT BUILD_EXECUTABLES" ON )
CMAKE_DEPENDENT_OPTION (BUILD_RSFE_SPLITBREGMAN "Build an executable useful to test rsfe algorithm on a 3d image" OFF "NOT BUILD_EXECUTABLES" ON )
CMAKE_DEPENDENT_OPTION (BUILD_IMAGE_TOOLKIT "Build an executable useful to use all members of the library" OFF "NOT BUILD_EXECUTABLES"
CMAKE_DEPENDENT_OPTION( BUILD_PARALLEL
  "Build an executable to test parallel"
  OFF
  "NOT BUILD_EXECUTABLE"
  ON)

# 1
if(BUILD_SHOW_IMAGE)
  ADD_EXECUTABLE(show_image ./examples/showing.cxx)
  TARGET_LINK_LIBRARIES(show_image ${VTK_LIBRARIES})
endif()

# 2
if(BUILD_RSFE_SPLITBREGMAN)
  ADD_EXECUTABLE(rsfe_splitbregman ./examples/rsfe_splitbregman.cxx)
  TARGET_LINK_LIBRARIES(rsfe_splitbregman
                         ${VTK_LIBRARIES} ${FFTW_PARALLEL_LIB} ${FFTW_LIBRARIES})
endif()

# 3
if(BUILD_IMAGE_TOOLKIT)
  ADD_EXECUTABLE(image_toolkit ./examples/image_toolkit.cxx)
  TARGET_LINK_LIBRARIES(image_toolkit
                         ${VTK_LIBRARIES} ${FFTW_PARALLEL_LIB} ${FFTW_LIBRARIES})
endif()

#-- Add an Option to toggle the generation of the API documentation
option(BUILD_DOCUMENTATION
  "Use Doxygen to create the HTML based API documentation
typing make doc on the shell"
  OFF)
if(BUILD_DOCUMENTATION)
  FIND_PACKAGE(Doxygen)
  if(NOT DOXYGEN_FOUND)
    message(FATAL_ERROR
             "Doxygen is needed to build the documentation. Please install it correctly")
  endif()
  #-- Configure the Template Doxyfile for our specific project
  configure_file(Doxyfile ${PROJECT_BINARY_DIR}/Doxyfile
                             %ONLY IMMEDIATE)
  #-- Add a custom target to run Doxygen when ever the project is built
  add_custom_target(doc
                   COMMAND ${DOXYGENEXECUTABLE} ${PROJECT_BINARY_DIR}/Doxyfile
                      SOURCES ${PROJECT_BINARY_DIR}/Doxyfile)
endif()

#-- make cmakereset
add_custom_target(cmakereset
                   COMMAND rm -R CMakeCache.txt CMakeFiles/ cmake_install.cmake)
A.2 Code profiling

```python
#!/usr/bin/env python
import re
gprof = open("./output.txt" , "r")
out = open("./GPROFoutput_CallGraph.txt" , "w")
i = 0
for line in gprof:
    i = i + 1
    if i == 6:
        towrite = str(line) + '\n'
        out.write(towrite)
    if line[0] == '\[':
        towrite = str(line) + '\n'
        out.write(towrite)
gprof.close()
out.close()
```
Bibliography


