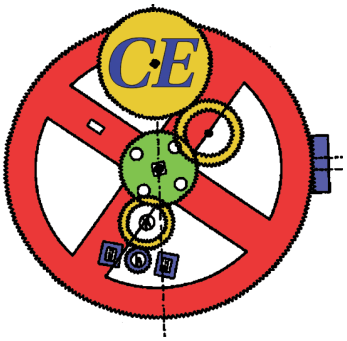


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

Acceleration of the Chan-Vese model for 3D segmentation of tumors in CT scans using GPUs

Matthijs Brobbel

CE-MS-2016-02

Abstract

Segmentation and annotation of tumors in CT scans of the brain is a cumbersome time-consuming task for medical experts. Carefully annotated data can be used to build training data sets for machine learning frameworks, with the ultimate goal to fully automate this process. This thesis focuses on acceleration of the annotation process by implementation of an interactive accelerated segmentation model rather than implementation or evaluation of the machine learning part.

The Chan-Vese model is an active contour model which can be used to detect objects for which the boundaries are not necessarily defined by gradient. An energy functional is minimized by evolvement of the contour. Evolvement of the contour in a numerical approximation, which uses finite differences and a level set formulation, is determined by solving a Partial differential equation (PDE) with an iterative solver. This thesis presents implementations for both 2D and 3D which use Successive over-relaxation (SOR) to solve the PDE.

This computationally intensive task benefits from acceleration to keep the feedback loop, in the process of tuning parameters and convergence to the searched segmentation, as short as possible. The effect of varying the different parameters of the model are visualized for different examples images to allow for educated guesses.

Accelerated implementations which leverage Compute Unified Device Architecture (CUDA) on a Graphics processing unit (GPU) are presented and compared to sequential and multithreaded OpenMP implementations. Evaluation of the CUDA implementations with single precision on a POWER8 platform with a K40 GPU shows a speedup of 56 and 107 over sequential implementations for 2D and 3D respectively.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Acceleration of the Chan-Vese model for 3D segmentation of tumors in CT scans using GPUs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Matthijs Brobbel
born in Rotterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Acceleration of the Chan-Vese model for 3D segmentation of tumors in CT scans using GPUs

by Matthijs Brobbel

Abstract

Segmentation and annotation of tumors in CT scans of the brain is a cumbersome time-consuming task for medical experts. Carefully annotated data can be used to build training data sets for machine learning frameworks, with the ultimate goal to fully automate this process. This thesis focuses on acceleration of the annotation process by implementation of an interactive accelerated segmentation model rather than implementation or evaluation of the machine learning part.

The Chan-Vese model is an active contour model which can be used to detect objects for which the boundaries are not necessarily defined by gradient. An energy functional is minimized by evolvement of the contour. Evolvement of the contour in a numerical approximation, which uses finite differences and a level set formulation, is determined by solving a PDE with an iterative solver. This thesis presents implementations for both 2D and 3D which use SOR to solve the PDE.

This computationally intensive task benefits from acceleration to keep the feedback loop, in the process of tuning parameters and convergence to the searched segmentation, as short as possible. The effect of varying the different parameters of the model are visualized for different examples images to allow for educated guesses.

Accelerated implementations which leverage CUDA on a GPU are presented and compared to sequential and multithreaded OpenMP implementations. Evaluation of the CUDA implementations with single precision on a POWER8 platform with a K40 GPU shows a speedup of 56 and 107 over sequential implementations for 2D and 3D respectively.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-02

Committee Members :

Advisor: Dr. Z. Al-Ars, CE, TU Delft

Chairperson: Prof. H.P. Hofstee, IBM Research and CE, TU Delft

Member: Dr. T.G.R.M. van Leuken, CAS, TU Delft

Dedicated to my parents
& Jantine

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ARL Austin Research Laboratory

CAPI Coherent Accelerator Processor Interface

CUDA Compute Unified Device Architecture

FAbRIC FPGA Research Infrastructure Cloud

FPGA Field-programmable gate array

GPU Graphics processing unit

GPGPU General-purpose computing on graphics processing units

IBM International Business Machines

NSF National Science Foundation

PDE Partial differential equation

PRAM Parallel random-access machine

SMT Simultaneous multithreading

SOR Successive over-relaxation

TACC Texas Advanced Computing Center

WT Work-Time

Acknowledgements

I gratefully acknowledge the support of everyone who has supported me on this journey.

First and foremost I want to express my sincere gratitude towards Peter Hofstee. Not only has his daily supervision and insights been invaluable, his immense knowledge, encouragement, and patience, all have been of great help and motivation. I would also like to thank him for offering me to conduct my thesis research at the IBM Austin Research Laboratory in Austin, Texas.

A very special thanks goes out to all the people at IBM who have helped me in any way, with either my thesis or any of the other projects I was working on. In no particular order, I would especially like to thank Christopher Durham, Teguh Hofstee, Jeffrey Kellington and Paul Lecocq.

I must also acknowledge Frank Liu from IBM, who's expertise on differential equations has been of tremendous help in helping me to understand mathematics I was unfamiliar with, including the numerous ways to solve a PDE.

I would also like to thank Billy Braithwaite for his help during the early stages of my research project.

I would also like to thank Zaid Al-Ars and René van Leuken for agreeing to be part of the thesis committee. Another words of thanks to the people from Texas Advanced Computing Center (TACC) involved with the FPGA Research Infrastructure Cloud (FAbRIC) for allowing me early access to their systems.

I would also like to thank my parents, brothers and sister, and other family and friends for their support.

And last but not least, I would like to thank my lovely fiancée, Jantine, for always supporting me.

Matthijs Brobbel
Delft, The Netherlands
March 7, 2016

1 Introduction

In this chapter the outline of the thesis is presented together with some background information on the thesis topic, a set of research questions, and a description of the contributions made.

1.1 Thesis outline

In this thesis the details of the work done towards accelerating the Chan-Vese model are discussed. This is done in a top-down fashion. In Chapter 1, the problem is stated together with some questions and details on the contributions made. Chapter 1 also gives some background information on the thesis topic. In Chapter 2 the Chan-Vese model is introduced, by first describing some methods, which the model uses and is based on. Chapter 2 also goes in detail on how to guide the Chan-Ve model by varying the parameters. Chapter 3 discusses all the details regarding the discretization and implementation of the model for both 2D and 3D. Chapter 3 also investigates parallelization opportunities and discusses acceleration of the given implementations for the Chan-Vese model. In Chapter 4 some results of tests run with the implementation are presented. And finally in Chapter 5 some conclusions are drawn and a list of suggestions for future work is given.

1.2 Background information

This section introduces some concepts which are key to the thesis topic.

1.2.1 Medical imaging for cancer diagnostics

Cancer is a group of diseases which involve abnormal growth of cells. This abnormal growth is also referred to as neoplasm, or tumor, in the case where these cells start to form a mass. Not all tumors are malignant i.e. cancerous. There are also noncancerous or benign tumors. Cancer can be detected when certain signs and symbols start to appear or through screening tests, the latter becoming more and more common. Further investigation and precise diagnostics are committed through all kinds of medical tests which often include medical imaging. The diagnostics are commonly confirmed by means of a biopsy and examination of the tissue sample by a pathologist.

In the early stages, when cancer starts to develop, there may be little to none observable signs and symptoms. Only when the mass continues to grow, some signs and symptoms might appear. This also strongly depends on the type and position of the cancer. Screening tests can help early detection and diagnostics of cancer, but this does not hold for all cancers.

Recent advances in medical imaging technology have caused a drastic change in the way people are diagnosed and treated for all kinds of different diseases. Not only has it become easier and less intrusive, the resulting images contain more detail and allow medical experts to do quicker and better diagnosis.

Several medical imaging technologies are widely used in cancer diagnostics and treatment. These resulting images allow medical experts to learn more about a tumor, plan cancer treatments, monitor the effectiveness of a treatment or learn more about the stage of cancer. Different medical imaging techniques can be used for different cancers.

With these medical imaging techniques becoming more and more advanced, there is a growing need for more advanced image processing algorithms and computation methods.

1.2.1.1 Machine learning

There is an ongoing trend to screen large populations for different types of cancer. As a result, medical researchers are getting access to much larger data sets with images and volumes. By annotating the data, a training data set can be built to be used within a machine learning framework. The task of annotating data, to prepare for usage in a training data set, can be cumbersome due to, for example, the lack of responsive, efficient and effective image processing algorithms which could prevent medical experts from having to annotate the data fully by hand. This thesis does not go into detail on how machine learning methods can be used to improve the accuracy of diagnosis. This thesis rather focuses on speeding up the annotation process to speed up the process of building the training data sets.

1.2.2 Image processing

Digital image processing can be used to do all kinds of analysis on image data. Image data here can be a two dimensional image resulting from, for example, a radiograph. It can also be a stack of two dimensional images combined to a three dimensional volume resulting, for example, from an X-ray computed tomography.

By applying some form of filtering or analysis, several tasks can be completed. For example, one could try to extract features, classify the image or objects within the image, project onto the image, recognize patterns or segment an object or objects.

The annotation task focused on in this thesis is the separation of healthy tissue from cancerous tissue within a three dimensional volume resulting from an X-ray compute tomography scan of an human brain, by means of a segmentation technique.

1.2.2.1 Segmentation

Segmentation techniques can be used in image processing to delineate an object, or objects, in an image. There exist several segmentation algorithms and techniques that each try to tackle this problem, where background and foreground items need to be separated, in a different way.

A possible and common way to segment objects within an image is, to find edges within the image, and then define these edges to be the boundaries of the objects. Finding the edges is not a trivial task. A commonly used edge detection technique is based on

the gradient[1]. This works for a large set of images, but this technique fails where the boundary of the object, for example, is defined by a smooth boundary or by texture, which can be the case in noisy images. Most of the time a smoothing filter is applied to remove noise before the edges are detected.

1.3 Problem statement

This thesis aims to solve the problem of the time-consuming task for a medical expert to segment a tumor by hand from a three dimensional X-ray compute tomography, by implementation of an accelerator for a user guided segmentation model, which does not depend on edges.

Segmentation models which allow for user guidance exist, however these models are compute intensive which results in a bad user experience. The feedback loop is stalled every time the model needs to be reevaluated with a new set of parameters. This makes it difficult to get good segmentation results for different images, and as a result, medical experts might be better off doing the segmentation by hand. This is a problem since it slows the process of building large data sets for the training and learning phase of machine learning frameworks, which in the end can be used to improve the rate and accuracy at which people can be diagnosed.

This is also visualized in Figure 1.1.

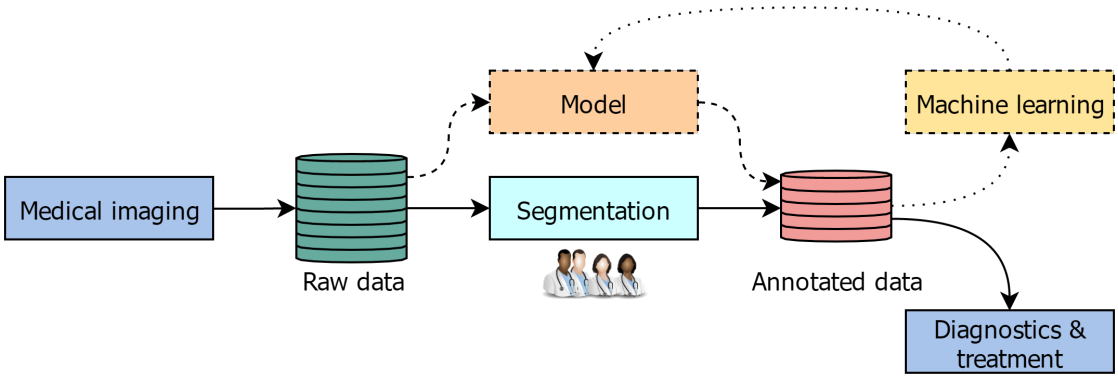


Figure 1.1: Overview of the different aspects related to the segmentation task.

1.3.1 Algorithmic challenges

The first challenge is to pick or design a segmentation technique or model, which gives the user good control over the segmentation process, and which does not depend on gradient to define the boundaries of objects.

The Chan-Vese model is a model for active contours, which is able to detect objects whose boundaries are not defined by gradient[2]. It is based on several techniques which all need to be understand in order to be able to understand the Chan-Vese model.

1.3.2 Computative challenges

Most segmentation techniques and models are mathematical models which need to be numerically approximated and discretized in order for them to be leveraged with digital images on computer hardware. For the Chan-Vese model, a numerical approximation of the model is described in the original paper. However, it still is important to have a good understanding of this in order to be able to map this approximation efficiently on hardware.

An iterative method to solve the PDE, which describes the evolvement of the contour in the model, needs to be evaluated and picked carefully, to make sure the PDE converges to its solution in reasonable time.

Since the input images, and volumes, will be of reasonable size, the number of computations required to solve the problem of segmentation for a single set of parameters, is large. It is a challenge to keep the feedback loop as short as possible in order to provide the best user experience, which allows the medical expert to converge quickly to the sought segmentation.

1.3.3 Implementative challenges

Acceleration of image processing algorithms require both a good understanding of all the different steps required to solve the problem, and a deep understanding of the computation platform used, in terms of architecture and memory organization.

The challenge is to determine a suitable computation platform for an accelerated implementation. Also, the accuracy of the accelerated implementation needs to match the accuracy of the given sequential implementation.

1.3.4 Questions

The discussed problems and challenges raise the following questions;

1. Can medical experts use the Chan-Vese model for segmentation to delineate tumors visible in a volume resulting from an X-ray computed tomography, in a semi-automatic fashion, where the results of the segmentation can be influenced by a set of parameters, to guide the model towards the required solution?

2. Is it possible to extend a given sequential implementation of a numerical approximation of the Chan-Vese model for two dimensional images, to support three dimensional volumes for segmentation of surfaces?

3. Can the implementation be accelerated without loss of accuracy, using special hardware which exploits parallelization, to streamline the experience for medical experts when model parameters need to be fine tuned?

1.4 Contributions

This thesis' goal is to find solutions for, and answers to the problems and questions stated in the previous section. The following can be considered to be contributions made.

- An detailed overview and explanation of minimization of an energy functional as framework for segmentation of objects in images.
- A set of figures to visualize and help understand the role of the different parameters in the Chan-Vese model.
- A description of the link between the mathematical description of the Chan-Vese model, and an implementation in computer hardware.
- Details on how to extend a given implementation for the Chan-Vese model for two dimensional images, which uses SOR to solve the PDE, to support three dimensional volumes.
- Detailed evaluation of parallelization opportunities in the given implementation of the Chan-Vese model.
- Details on an accelerated implementation of the Chan-Vese model using the parallel computing platform CUDA for General-purpose computing on graphics processing units (GPGPU), which improves the user experience in the process of fine tuning the parameters of the model, in order to get the sought segmentation.
- Evaluation and comparison of sequential, OpenMP and CUDA implementations of the Chan-Vese model on two different hardware platforms.

2 Chan-Vese model

In this chapter the Chan-Vese model is introduced. In order to be able to understand this algorithm, first the level set method and the active contour model are discussed. Then the Chan-Vese model is discussed in detail.

2.1 Level set method

The level set method, as developed in [3], is a method where level sets are used in computations of shapes and surfaces in a discretized fixed rectangular grid. Level sets are mathematical functions of the form as given in Equation (2.1).

$$L = \{(x_1, \ldots, x_n) | f(x_1, \ldots, x_n) = c\} \tag{2.1}$$

It can be described as a set where the function f takes a given constant c.

Figure 2.1 visualizes some aspects of the level set method. Figure 2.1a shows a level set function ϕ^a. Figure 2.1b shows a contour with a nice, smooth boundary. The level set function ϕ^a can, for example, be used to define the interior of the shape as the points for which the level set function ϕ^a is positive. The boundary of the shape then is defined by the zero level set ϕ_0^a of ϕ^a i.e. all the points for which ϕ^a equals zero. By updating the level function ϕ^a to ϕ^b, and using the zero level set of ϕ^b to define the boundary of the shape, the shape can undergo topology changes easily as can be seen in Figure 2.1c and Figure 2.1d. This aspect has proven that the use of level set functions to describe the evolvement of a contour in image processing can be very effective[4][5].

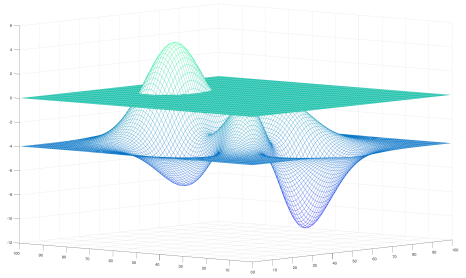
Mathematically this can be expressed as follows. In this example the level set method is used in two dimensions to represent a closed curve, Γ, by using a level set function, ϕ. The curve, Γ, is represented by the zero level set of ϕ i.e. where the constant c is zero, as can be seen for two dimensions in Equation (2.2).

$$\Gamma = \{(x, y) | \phi(x, y) = 0\} \tag{2.2}$$

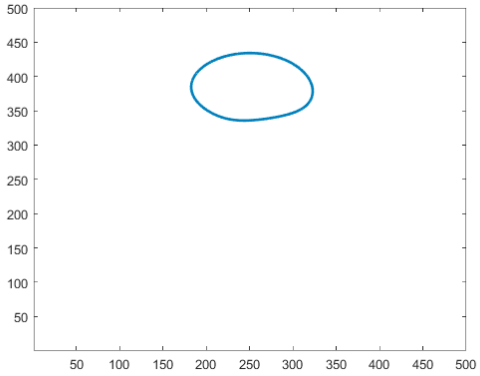
The level set method can represent the contour implicitly through the level set function. The evolution of the curve is given by the zero level curve of the function $\phi(t, x, y)$ at time t. To evolve the curve Γ in normal direction with a speed v requires to solve the differential equation as given in Equation (2.3).

$$\begin{cases} \dfrac{\delta \phi}{\delta t} = |\nabla \phi| v \\ \phi(0, x, y) = \phi_0(x, y) \end{cases} \tag{2.3}$$

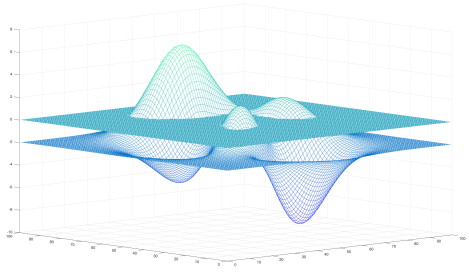
Here, $(x, y) | \phi_0(x, y) = 0$, the zero-level set, defines the initial contour.



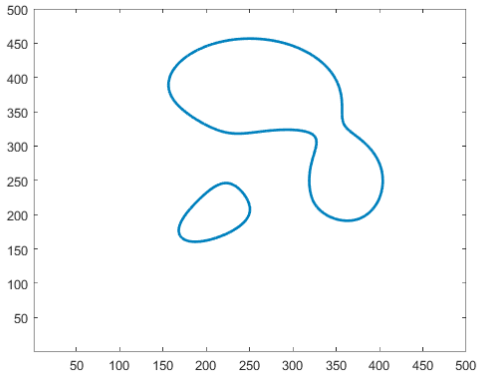
(a) Level set: ϕ^a



(b) Zero level set: ϕ_0^a i.e. $\phi^a = 0$



(c) Level set: ϕ^b



(d) Zero level set: ϕ_0^b i.e. $\phi^b = 0$

Figure 2.1: Visualization of the level set method.

2.2 Active contour model

Active contours, also referred to as snakes, are dynamic curves that evolve within an image in order to delineate objects within that image. The evolvement of the contour is based on constraints from a given image. The active contour model is widely used in all kinds of computer vision applications e.g. segmentation [2], edge detection [6], shape recognition [7], stereo matching [8] and object tracking [9].

2.2.1 Energy functional

Kass investigated the use of energy minimization as a framework to solve these delineation problems [10]. Energy minimization models allow the user to guide the model by adding energy terms to the minimization. This level of interaction makes it easy to find usable energy functions which do not have a strong dependence on the starting points and which have few local minima. It is important to note that the model is an active model that always behaves dynamically because it minimizes the energy functional.

In order to define the energy functional, let Ω be a bounded open subset of $\mathbb{R}^2$, with $\delta\Omega$ its boundary. Let u_0 be a given image. With the position of the active contour represented parametrically as $C(s) : [0,1] \to \mathbb{R}^2$, in terms of its x and y coordinates, $C(s) = (x(s), y(s))$, the energy functional E_1 of the basic active contour model, as proposed by Kass, can be written as can be seen in Equation (2.4).

$$\begin{aligned} E_1(C) &= \int_0^1 E_{\text{contour}}\left(C(s)\right)\ ds \\ &= \int_0^1 \left(E_{\text{internal}}\left(C(s)\right) + E_{\text{external}}\left(C(s)\right)\right)\ ds \\ &= \int_0^1 \left(E_{\text{internal}}\left(C(s)\right) + E_{\text{image}}\left(C(s)\right) + E_{\text{constraint}}\left(C(s)\right)\right)\ ds \end{aligned} \tag{2.4}$$

The total energy in the basic active contour model is split into 3 terms. The internal energy, E_{internal}, controls the deformation of the contour. The external energy, E_{external}, which in the original paper by Kass [10], is a combination of E_{image} and $E_{\text{constraint}}$, the image energy, E_{image}, which is a function of the features in the image which attracts the contour towards the object in the image, and the constraint energy, $E_{\text{constraint}}$, allows for some user interaction by giving users some tools to interactively guide the contour towards the object.

2.2.2 Internal energy

The internal energy term is composed of two components which control both the continuity and smoothness of the contour as can be seen in Equation (2.5).

$$\begin{aligned} E_{\text{internal}} &= E_{\text{elastic}} + E_{\text{bending}} \\ &= \alpha(s)|C(s)'|^2 + \beta(s)|C(s)''|^2 \end{aligned} \tag{2.5}$$

Here, $\alpha(s)$ and $\beta(s)$ are both user-defined positive weights which control the sensitivity of the energy function to stretch and curvature of the contour, respectively. Another way to describe it is that the first-order term makes the contour behave like a membrane, whereas the second-order term makes the contour behave like a thin plate. This means that for large values of α the changes in distances between the points in the contour are penalized whereas large values of β penalize oscillations in the contour.

$C'(s)$ denotes the first derivative of $C(s)$ with respect to s, and $C''(s)$ denotes the derivate of $C'(s)$ with respect to s i.e. the second derivative of $C(s)$.

It is important to note here that the user-defined weights are not scale-invariant.

2.2.3 External energy

The external energy term can be seen as a combination of two separate energy terms, as can be seen in Equation (2.4). The first being the image energy and the second being the constraint energy.

2.2.3.1 Image energy

The image energy is some function of the features of the image. There are many ways in which one can process or evaluate image features. As a result, there are many possible ways in which one can define the image energy term. The goal of this energy term is to generate some force or velocity which will attract or move the contour towards the boundary of the object of interest. Kass presents a combination of three energy terms for the image energy. His formulation includes lines, edges and terminations present in the image. These terms each have their own weights as can be seen in Equation (2.6).

$$E_{\text{image}} = w_{\text{line}}E_{\text{line}} + w_{\text{edge}}E_{\text{edge}} + w_{\text{termination}}E_{\text{termination}} \tag{2.6}$$

The first energy term is the line functional which is simply the intensity of the image itself. It can be represented as can be seen in Equation (2.7). The constant weight, w_{line}, determines whether the contour will be attracted by either darker or lighter lines in the image i.e. $w_{\text{line}} < 0$ penalizes high intensity, which means that the contour gets attracted by darker lines, whereas $w_{\text{line}} > 0$ penalizes low intensity and attracts the contour towards lighter lines in the image.

$$E_{\text{line}} = u_0(x, y) \tag{2.7}$$

The edge energy term is the edge functional which is based on the gradient of the image. There are several ways to approach edge detection. A possible implementation is given in Equation (2.8).

$$E_{\text{edge}} = -|\nabla u_0(x, y)|^2 \tag{2.8}$$

The absolute contribution of this edge energy term, for $w_{\text{edge}} > 0$, is the attraction of the contour towards large gradients in the image i.e. possible edges or boundaries of objects in the image. In some cases part of the contour might be attracted towards a low-energy feature of the image. When that happens, the contour will pull neighboring points on the contour towards a possible continuation of this feature. The result is a large energy wall around that specific local minimum. This can be avoided by using scale space continuation[11][12]. To avoid these local minima, the image is smoothened using a blur filter where the amount of blur is reduced as the contour evolves. The scale space continuation is applied to the Marr-Hildreth theory for edge-detection[13]. This theory defines edges to be at the zero-crossings of the convolution of the image with the Laplacian of a Gaussian i.e. the divergence of the gradient of a Gaussian. The different edge functional can be seen in Equation (2.9), where $G_\sigma(x, y)$ is a Gaussian function, $G_\sigma(x, y) = \sigma^{-1/2}e^{-|x^2+y^2|/4\sigma}$, with σ the standard deviation.

$$E_{\text{edge}}^{MH} = -|G_\sigma(x, y) \cdot \nabla^2 u_0(x, y)|^2 \tag{2.9}$$

The last energy term, $E_{\text{termination}}$, can be used to find termination of corners and line segments. This is done in a slightly blurred image using the curvature of level lines. Let $u_1(x, y)$ be u_0 filtered with a Gaussian i.e. $u_1(x, y) = G_\sigma(x, y) \cdot u_0(x, y)$. With the gradient angle defined as $\theta = arctan(u_{1_y}/u_{1_x})$, unit vectors along the gradient direction defined as $\boldsymbol{n} = (cos(\theta), sin(\theta))$, and unit vectors perpendicular to the gradient direction

defined as $\boldsymbol{n}_\perp = (-sin(\theta), cos(\theta))$, the termination energy term can be written as can be seen in Equation (2.10).

$$\begin{aligned} E_{\text{termination}} &= \frac{\delta\theta}{\delta\boldsymbol{n}_\perp} \\ &= \frac{\frac{\delta^2 u_1}{\delta^2 \boldsymbol{n}_\perp}}{\frac{\delta u_1}{\delta \boldsymbol{n}}} \\ &= \frac{u_{1_{yy}} u_{1_x}^2 - 2u_{1_{xy}} u_{1_x} u_{1_y} + u_{1_{xx}} u_{1_y}^2}{(1 + u_{1_x}^2 + u_{1_y}^2)^{3/2}} \end{aligned} \tag{2.10}$$

The absolute contribution of this energy term is the attraction of the contour towards terminations in the image.

2.2.3.2 Constraint energy

The constraint energy term allows for user interaction to put some constraints on either the initial contour or the evolvement of the contour. Again, the possible ways to define this term are endless. Kass [10] gives two possible examples for this term. The first example is the definition of a spring between two points, one point being connected to any point on the contour, and the other point either anchored at a fixed point or connected to another point on the contour. The resulting constraint energy term of the defined spring adds to the total energy function, which in the end penalizes the movement in the contour of these points away from each other. The other example given by Kass is the definition of a repulsion force. This repulsion force can be used to push the contour out of a certain local minimum into another.

2.2.4 Geometric active contours

With the constraint and termination energy terms set to zero, Equation (2.4) can be rewritten to Equation (2.11) using Equation (2.5), Equation (2.6), Equation (2.9) and Equation (2.10).

$$E_1(C) = \alpha \int_0^1 |C'(s)|^2 ds + \beta \int_0^1 |C''(s)| \; ds - \lambda \int_0^1 |\nabla u_0(C(s))|^2 \; ds \tag{2.11}$$

Active contours and energy minimization as a framework for segmentation can be successfully used to delineate objects within an image. However, there is a big limitation to the given approach. The described energy model is not capable of coping with changes in the topology of the contour when implemented directly. The topology of the contour will be the same as the initial contour. This is limiting in cases where the number of objects to be segmented is unknown. There is some work done towards special procedures, mostly heuristics, which allow the contour to split and merge [14][15][16]. These effects are hard to describe in terms of parameterization of the contour.

A new model for active contours is introduced in [17]. This model is described by a geometric flow i.e. by a Partial differential equation (PDE), and is based on mean curvature motion [18]. Instead of energy minimization as framework, it is motivated by curve evolution. When used with the level set method[3], it allows for topology changes[19] of the active contour.

Evolving the active contour in these models, where the problem is discretized on a fixed rectangular grid, requires to solve the differential equation given in Equation (2.3). In the model proposed in [17], which uses the case of motion by mean curvature, v becomes $\text{div}(\nabla\phi(x,y)/|\nabla\phi(x,y)|)$, which is the curvature of the level-curve ϕ going through (x,y). With this Equation (2.3) becomes the mean curvature equation and can be seen in Equation (2.12).

$$\begin{cases} \dfrac{\delta\phi}{\delta t} = |\nabla\phi|\text{div}\left(\dfrac{\nabla\phi}{|\nabla\phi|}\right), & t \in (0,\infty), x \in \mathbb{R}^2 \\ \phi(0,x,y) = \phi_0(x,y), & x \in \mathbb{R}^2 \end{cases} \tag{2.12}$$

The geometric active contour model [17] adds a term to the equation which acts as a constant force in the direction of the normal. The equation is also multiplied with a function $g(|\nabla u_0(x,y)|)$ which acts to stop the contour at the sought boundary. The model can be seen in Equation (2.13).

$$\begin{cases} \dfrac{\delta\phi}{\delta t} = g\left(|\nabla u_0|\right)|\nabla\phi|\left(\text{div}\left(\dfrac{\nabla\phi}{|\nabla\phi|}\right) + \nu\right), & t \in (0,\infty), x \in \mathbb{R}^2 \\ \phi(0,x,y) = \phi_0(x,y), & x \in \mathbb{R}^2 \end{cases} \tag{2.13}$$

The constant $\nu \geq 0$, can be interpreted as a force which pushes the contour towards the boundary of the object when the curvature of the contour becomes zero i.e. it ensures the term $(\text{div}(\nabla\phi/|\nabla\phi|) + \nu)$ stays positive. The function $g(x)$, which can be seen in Equation (2.14), controls the speed at which the contour moves. For the geometric active contour model $p = 2$. The movement of the contour stops at the boundary of the object where $g(x)$ goes to zero i.e. when $|\nabla G_\sigma(x,y) * u_0(x,y)|$, the gradient, is large.

$$g(|\nabla u_0(x,y)|) = \frac{1}{1 + |\nabla G_\sigma(x,y) * u_0(x,y)|^p}, \; p \geq 1 \tag{2.14}$$

2.2.5 Geodesic active contours

Another model for active contours, introduced in [19], is the geodesic model. A geodesic curve is a minimum distance path between given points. The energy functional, E_2, of the model, which is to be minimized, can be seen in Equation (2.15).

$$E_2(C) = 2\int_0^1 |C'(s)| \cdot g(|\nabla u_0(C(s))|)\, ds \tag{2.15}$$

This model also has a level set formulation given in Equation (2.16).

$$\begin{cases} \dfrac{\delta\phi}{\delta t} = |\nabla\phi| \left(\operatorname{div}\left(g(|\nabla u_0|) \dfrac{\nabla\phi}{|\nabla\phi|} \right) + \nu g(|\nabla u_0|) \right), & t \in (0,\infty), x \in \mathbb{R}^2 \\ \phi(0,x,y) = \phi_0(x,y), & x \in \mathbb{R}^2 \end{cases} \tag{2.16}$$

2.3 Chan-Vese model

All the models described in the previous section rely on some edge-function to stop the evolvement of the contour. This limits the objects that can be detected by these active contour models to objects with their boundaries defined by a gradient. Also, when discretized, the gradient is bounded, and therefore the stopping function g will never be zero on the boundary. As a result, a discretized implementation of the model might drive the contour through the actual boundary.

The active contour model proposed by Chan and Vese in [2], does not rely on an edge stopping function, but instead relies on Mumford-Shah techniques for segmentation [20] to stop the evolvement of the contour on the boundary of the object. This results in a model which can detect objects both with and without the gradient defining the edges. The model also has a level set formulation which allows for topology changes and allows for the initial contour to be placed anywhere in the image.

The basic idea of the model is best explained with an example image where two regions exist with distinct approximate constant grayscale intensities, u_0^i and u_0^o. Here i and o denote inside and outside respectively. In other words, $u_0 \approx u_0^i$ inside the object and $u_0 \approx u_0^o$ outside the contour or object.

For this thesis only grayscale images, i.e. images with a single channel, are considered, however it is also possible to use the model with vector-valued images e.g. RGB images[21].

Now let C be the evolving curve in Ω, as the boundary of an open subset ω. In other words $\omega \subset \Omega$ and $C = \delta\omega$. Furthermore inside(C) is defined as the region ω and outside(C) is defined as the region outside of ω i.e. $\Omega \setminus \bar{\omega}$. The method to segment the object is segmentation with the minimization of an energy functional[2]. The fitting term is given in Equation (2.17).

$$F_1(C) + F_2(C) = \int_{\text{inside}(C)} |u_0(x,y) - c_1|^2 \, dx \, dy + \int_{\text{outside}(C)} |u_0(x,y) - c_2|^2 \, dx \, dy \tag{2.17}$$

Here c_1 and c_2 are region averages of the intensity inside the contour and outside the contour respectively. The fitting energy is minimized if the contour C is on the boundary of the object i.e. the difference between $u_0(x,y)$ inside the contour and c_1 goes to zero, and the difference between $u_0(x,y)$ outside the contour and c_2 goes to zero. This is demonstrated in Figure 2.2[22].

The model minimizes the term given in Equation (2.17) together with some other terms to regularize the evolvement of the contour. Firstly the length of the curve C can be penalized, and secondly the area of the region inside the contour can be penalized. The resulting energy functional of the active contour model is given in Equation (2.18).

$F_1(C) > 0$	$F_1(C) > 0$	$F_1(C) \approx 0$	$F_1(C) \approx 0$
$F_2(C) \approx 0$	$F_2(C) > 0$	$F_2(C) > 0$	$F_2(C) \approx 0$

Figure 2.2: Simple source image to demonstrate minimization of the fitting term. Taken from [22].

$$\begin{aligned} F(c1, c2, C) =& \mu \cdot \text{Length}(C) + \nu \cdot \text{Area}(\text{inside}(C)) \\ &+ \lambda_1 \int_{\text{inside}(C)} |u_0(x,y) - c_1|^2 \, dx \, dy \\ &+ \lambda_2 \int_{\text{outside}(C)} |u_0(x,y) - c_2|^2 \, dx \, dy \end{aligned} \tag{2.18}$$

Here, $\mu \geq 0$, $\nu \geq 0$ are constants penalizing the length of the contour and the area inside the contour respectively. $\lambda_1 > 0$ and $\lambda_2 > 0$ are fixed parameters which penalize the average terms, inside and outside the contour respectively. The evolvement of the contour is the minimization of the given energy functional.

The relation with the Mumford-Shah functional E_{MS}, which is given in Equation (2.19), is the approximation in a reduced form, where the functional is restricted to piecewise constant functions of u i.e. all different regions in the image have the same constant intensity [20].

$$E_{MS}(u, C) = \mu \cdot \text{Length}(C) + \lambda \int_{\Omega} |u_0(x,y) - u(x,y)|^2 \, dx \, dy + \int_{\Omega \setminus C} |\nabla u(x,y)|^2 \, dx \, dy \tag{2.19}$$

This reduced case is referred to as the minimal partition problem. If the active contour model as given in Equation (2.18) is taken with $\nu = 0$ and $\lambda_1 = \lambda_2 = \lambda$, u can only take two values as can be seen in Equation (2.20), where C is the active contour.

$$u = \begin{cases} \text{average}(u_0) \text{ inside } C \\ \text{average}(u_0) \text{ outside } C \end{cases} \tag{2.20}$$

2.3.1 Level set formulation

The given minimal partition problem as given in Equation (2.20) can be solved using the level set method[3]. The contour is represented by the zero level set of a level set

function ϕ. The resulting level set formulation can be seen in Equation (2.21). This is also visualized in Figure 2.3.

$$\begin{cases} C = \delta\omega = \{(x,y) \in \Omega : \phi(x,y) = 0\} \\ \text{inside}(C) = \omega = \{(x,y) \in \Omega : \phi(x,y) > 0\} \\ \text{outside}(C) = \Omega \setminus \bar{\omega} = \{(x,y) \in \Omega : \phi(x,y) < 0\} \end{cases} \tag{2.21}$$

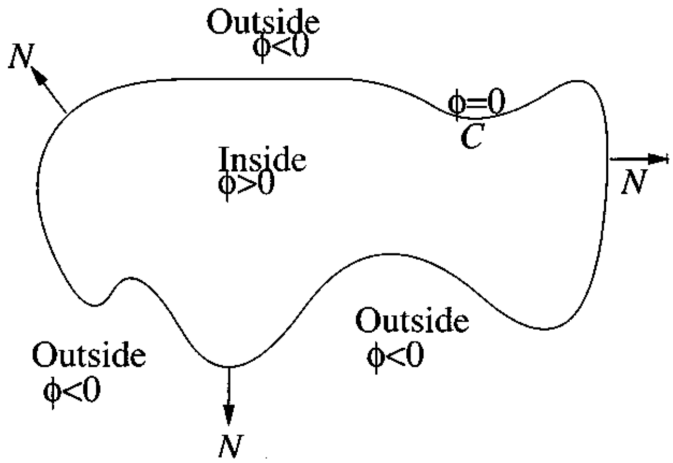


Figure 2.3: The curve C moving in normal direction according to Equation (2.21) and a level set function ϕ. Taken from [2].

Considering the level set formulation, and with H the Heaviside function and δ_0 the one-dimensional Dirac measure, which can be seen in Equation (2.22) and Equation (2.23), the different terms in Equation (2.18) can be rewritten as follows in Equation (2.24), Equation (2.25), Equation (2.26) and Equation (2.27).

$$H(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases} \tag{2.22}$$

$$\delta_0 = \frac{d}{dz} H(z) \tag{2.23}$$

$$\begin{aligned} \text{Length}(\phi = 0) &= \int_\Omega |\nabla H(\phi(x,y))| \; dx \; dy \\ &= \int_\Omega \delta_0(\phi(x,y)) |\nabla \phi(x,y)| \; dx \; dy \end{aligned} \tag{2.24}$$

$$\text{Area}(\phi \geq 0) = \int_\Omega H(\phi(x,y)) \; dx \; dy \tag{2.25}$$

$$\int_{\phi>0} |u_0(x,y) - c_1|^2 \; dx \; dy = \int_\Omega |u_0(x,y) - c_1|^2 H(\phi(x,y)) \; dx \; dy \tag{2.26}$$

$$\int_{\phi<0} |u_0(x,y) - c_2|^2 \; dx \; dy = \int_{\Omega} |u_0(x,y) - c_2|^2 (1 - H(\phi(x,y))) \; dx \; dy \tag{2.27}$$

This allows the energy functional as given in Equation (2.18) to be written as can be seen in Equation (2.28).

$$\begin{aligned} F(c_1, c_2, \phi) = \mu &\int_{\Omega} \delta(\phi(x,y)) |\nabla \phi(x,y)| \; dx \; dy \\ &+ \nu \int_{\Omega} H(\phi(x,y))) \; dx \; dy \\ &+ \lambda_1 \int_{\Omega} |u_0(x,y) - c_1|^2 H(\phi(x,y)) \; dx \; dy \\ &+ \lambda_2 \int_{\Omega} |u_0(x,y) - c_2|^2 (1 - H(\phi(x,y))) \; dx \; dy \end{aligned} \tag{2.28}$$

Again, considering the level set formulation, the minimal partition problem as given in Equation (2.20) can be rewritten to Equation (2.29).

$$u(x,y) = c_1 H(\phi(x,y)) + c_2 (1 - H(\phi(x,y))), \;\; (x,y) \in \bar{\Omega} \tag{2.29}$$

This then allows to write the constants c_1 and c_2 as functions of ϕ, as can be seen in Equation (2.30).

$$\begin{aligned} c_1(\phi) &= \frac{\int_{\Omega} u_0(x,y) H(\phi(x,y)) \; dx \; dy}{\int_{\Omega} H(\phi(x,y)) \; dx \; dy} \\ c_2(\phi) &= \frac{\int_{\Omega} u_0(x,y) (1 - H(\phi(x,y))) \; dx \; dy}{\int_{\Omega} (1 - H(\phi(x,y))) \; dx \; dy} \end{aligned} \tag{2.30}$$

The result is that c_1 and c_2 are now the region averages inside and outside the contour respectively, as can be seen in Equation (2.31).

$$\begin{cases} c_1(\phi) = \text{average}(u_0) \text{ in } \phi \geq 0 \\ c_2(\phi) = \text{average}(u_0) \text{ in } \phi < 0 \end{cases} \tag{2.31}$$

With H_ϵ and δ_ϵ as regularized versions of the Heaviside function H and the one dimensional Dirac measure δ_0, as defined in Equation (2.22) and Equation (2.23), the energy functional, as defined in Equation (2.28), can be rewritten to Equation (2.32).

$$\begin{aligned} F_\epsilon(c_1, c_2, \phi) = \mu &\int_{\Omega} \delta_\epsilon(\phi(x,y)) |\nabla \phi(x,y)| \; dx \; dy \\ &+ \nu \int_{\Omega} H_\epsilon(\phi(x,y))) \; dx \; dy \\ &+ \lambda_1 \int_{\Omega} |u_0(x,y) - c_1|^2 H_\epsilon(\phi(x,y)) \; dx \; dy \\ &+ \lambda_2 \int_{\Omega} |u_0(x,y) - c_2|^2 (1 - H_\epsilon(\phi(x,y))) \; dx \; dy \end{aligned} \tag{2.32}$$

The minimization can be solved by alternating between updating ϕ and the region averages, c_1 and c_2. By keeping ϕ fixed, c_1 and c_2 can be computed using Equation (2.30) with regularized versions for the Heaviside function H and the one dimensional Dirac measure δ_0. To update ϕ, c_1 and c_2 are kept fixed, and F_ϵ is minimized with respect to ϕ. This allows to deduct the associated Euler-Lagrange equation for $\phi(t, x, y)$, with $t \geq 0$ an artificial time and $\phi(0, x, y) = \phi_0(x, y)$ the initial contour, as can be seen in Equation (2.33).

$$\begin{aligned} \frac{\delta\phi}{\delta t} = \delta_\epsilon \left[\mu \text{ div} \left(\frac{\nabla\phi}{|\nabla\phi|} \right) - \nu - \lambda_1(u_0 - c_1)^2 + \lambda_2(u_0 - c_2)^2 \right] = 0 \text{ in } \Omega, \\ \phi(0, x, y) = \phi_0(x, y) \text{ in } \Omega, \\ \frac{\delta_\epsilon(\phi)}{|\nabla\phi|} \frac{\delta\phi}{\delta\vec{n}} = 0 \text{ on } \delta\Omega \end{aligned} \tag{2.33}$$

Here $\vec{n}$ denotes the exterior normal to the boundary $\delta\Omega$. $\delta\phi/\delta\vec{n}$ denotes the normal derivative of ϕ at the boundary.

2.3.2 Guiding the model

In order to be able to guide the Chan-Vese model to segment the object of interest, there are a few parameters to guide the contour. It is important to have a good understanding of the effect that varying each parameters has on the contour, in order to quickly converge to a suitable set of a parameters for a particular segmentation problem.

The parameters for the Chan-Vese model are listed below in Table 2.1.

Parameter	Penalizes
μ	Length of the contour
ν	Area inside the contour
λ_1	Average term inside the contour
λ_2	Average term outside the contour

Table 2.1: Chan-Vese model parameters

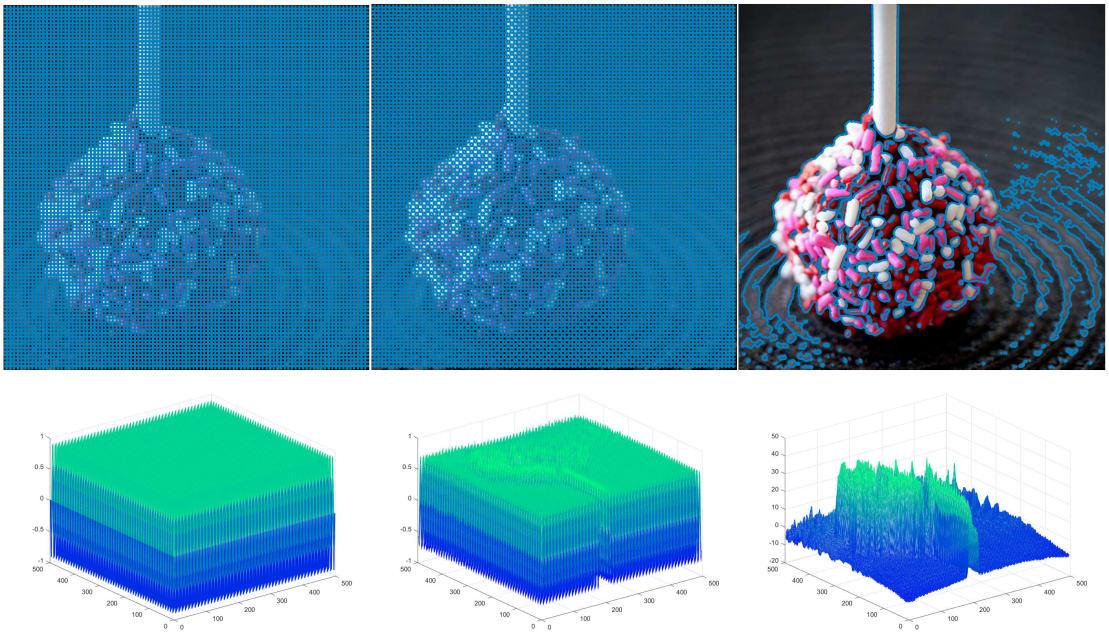
The figures in this subsection, which are used to demonstrate the different aspects which are controlled by these parameters, were generated by the implementation which is discussed in detail in Chapter 3. In all these figures, the top row shows the source image together with the contour, and the bottom row is a visualization of the level set description ϕ.

2.3.2.1 Initial contour

Another aspect which can be varied is the placement of the initial contour. A good initial contour results in quick convergence of the contour towards the solution. There are two [2][23] common ways to generate the initial contour. The first is a simple circle around the object to be segmented, and the second is a checkerboard pattern of small circles over the circle, visualized in Figure 2.4 and Figure 2.5. For some segmentation

problems the former might be more suitable than the latter, but this might be the other way around for other segmentation problems. In the case of segmentation of tumors, it is recommended to place a circle around the tumor in the image. This should simplify the search for the right set of parameters for the Chan-Vese model compared to the checkerboard initialization.

Figure 2.4 and Figure 2.5 shows the difference in evolvement of ϕ for an example image with the checkerboard and circle initialization respectively.

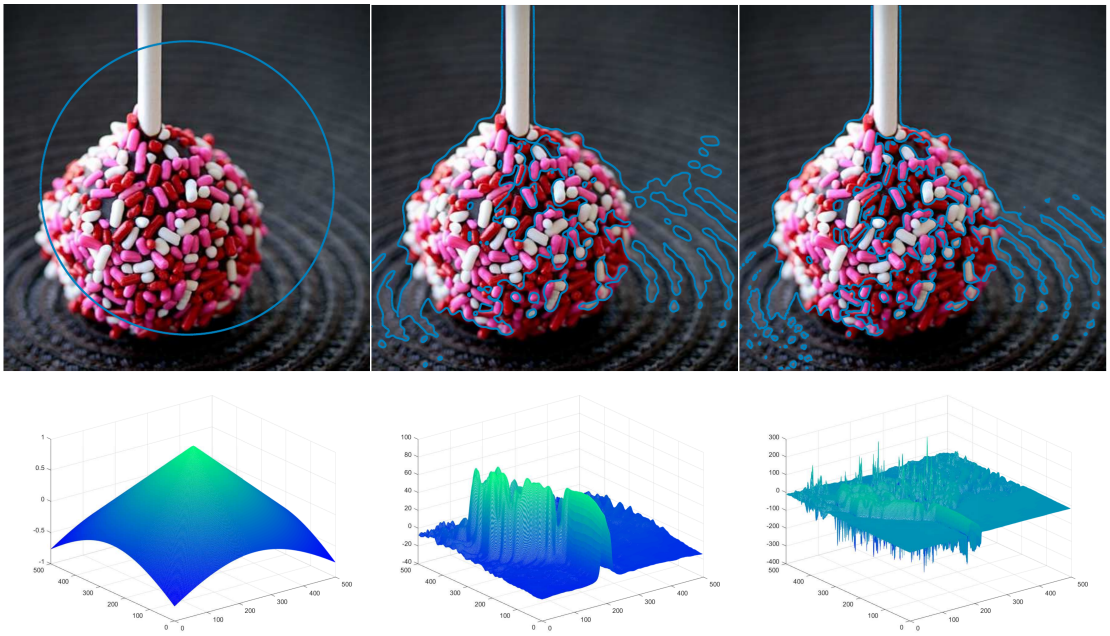


(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.4: Evolvement of ϕ for the checkerboard initialization.

2.3.2.2 Length of the contour

μ penalizes the length of the contour. Larger values for μ cause the contour to have a smoother boundary, whereas small values for μ allow for a more precise delineation. To demonstrate the effect of varying μ, the segmentation results per iteration step of the Chan-Vese model are visualized for two different types of images, for different μ. The first example demonstrates the ability to delineate a group of elements as a single object, for a large μ, or to segment the elements within the group as separate objects, with a small μ. The second example demonstrates the effect μ has on an image from a CT-scan of a brain with a tumor. Both examples show the segmentation or zero-level set, ϕ_0, projected onto the source image in the upper row, and the level set ϕ in the bottom row, at different iterations. Iteration 0 shows the initial contour. The initial contour is a circle with $r = 200$ for the image with a group of elements, whereas the initial contour for the image from a CT-scan of a brain with a tumor is a circle with $r = 500$. In these examples $\nu = 0$, $\lambda_1 = \lambda_2 = 1$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

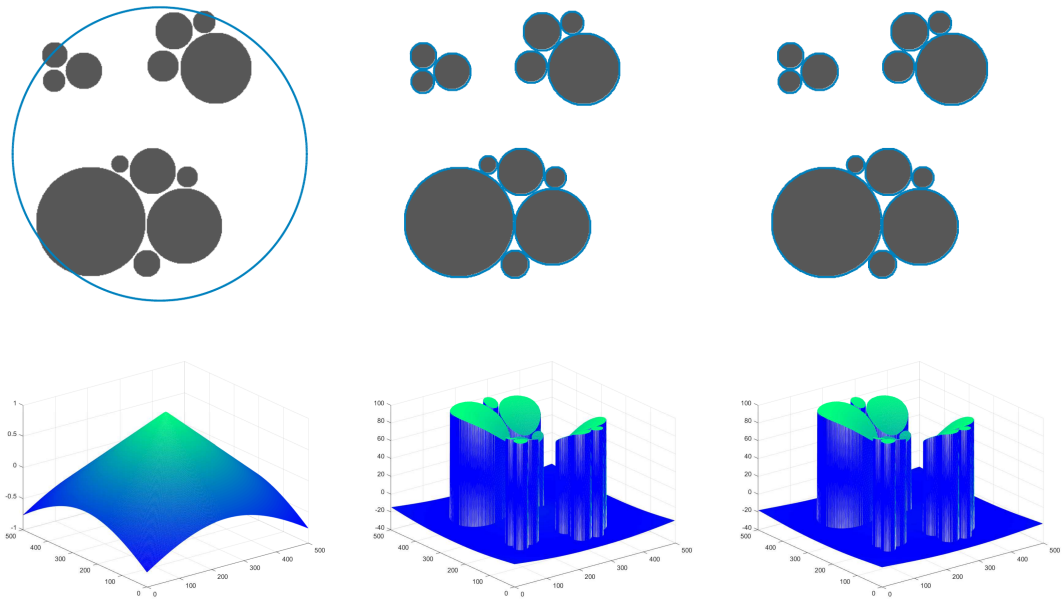
Figure 2.5: Evolvement of ϕ for the circle initialization.

Figure 2.6, Figure 2.7 and Figure 2.8 visualize the effect for the first example with $\mu = 0$, $\mu = 0.01 \cdot 255^2$ and $\mu = 1 \cdot 255^2$ respectively. Figure 2.9, Figure 2.10 and Figure 2.11 visualize the effect for the second example with $\mu = 0$, $\mu = 0.01 \cdot 255^2$ and $\mu = 1 \cdot 255^2$ respectively.

It can clearly be seen from these examples that μ controls the smoothness of the contour, by penalizing the length of the contour. Larger values for μ tend to slow the evolvement of the contour down. The level set description ϕ looks much smoother for larger values of μ, since quick evolvement is penalized. Smaller values, or even $\mu = 0$, tend to result in very quick evolvement and precise delineation of the objects. Figure 2.6 shows that $\mu = 0$ results in segmentation of the individual elements in the groups, whereas Figure 2.7 and Figure 2.8 show that $\mu = 0.01 \cdot 255^2$ and $\mu = 1 \cdot 255^1$ result in segmentation of the separate groups for the former, and segmentation of all elements as one group for the latter.

2.3.2.3 Area inside the contour

ν penalizes the area inside the contour. By setting this value to something other than $\nu = 0$, the area inside the contour can be forced to either shrink for $\nu > 0$, or the area inside the contour can be kept large by setting $\nu < 0$. By choosing too large values for ν the contour tends to shrink until it completely vanishes, whereas too negative values tend to grow the contour so that it sits at the borders. The example below demonstrates the effect of varying ν in a noisy image with a simple object. In this example $\mu = 0.001 \cdot 255^2$, $\lambda_1 = \lambda_2 = 1$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.6: Varying the penalty for the length of the contour, for an image with grouped elements. $\mu = 0$.

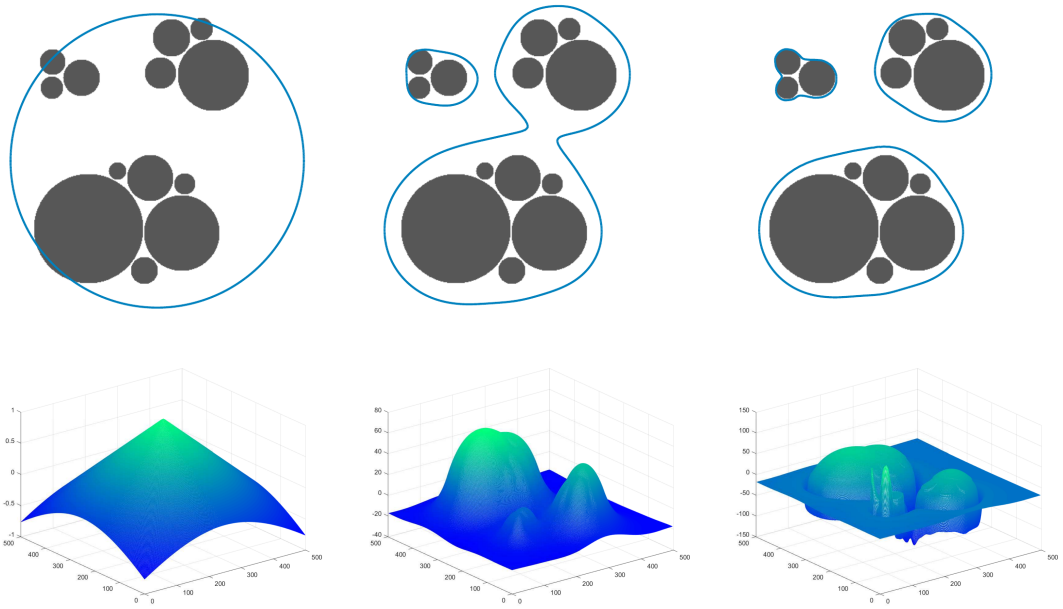
Figure 2.12, Figure 2.13 and Figure 2.14 visualize the effect of varying ν for $\nu = 0$, $\nu = 0.07 \cdot 255^2$ and $\nu = -0.015 \cdot 255^2$ respectively.

It can clearly be seen in Figure 2.13, that large values of ν tend to shrink the contour. Figure 2.14 shows that negative values for ν will grow the contour.

2.3.2.4 Average terms inside and outside the contour

λ_1 and λ_2 can be used to guide the contour towards either more or less uniform regions in terms of pixel intensities. Most of the times it makes sense to set $\lambda_1 = \lambda_2 = 1$. This will force the model towards the most uniform regions inside and outside the contour, in terms of pixel intensities. This is best understand for the image where only two intensities exists. The sums of the energy terms minimize when the difference between the actual pixel intensity and the region average is zero inside and outside of the contour i.e. in the case where the contour perfectly separates the two intensity levels, as can be seen in Figure 2.2.

By either setting λ_1 or λ_2 to a higher value, the uniformity of the region inside or outside the contour respectively, is weighted more heavily. Setting $\lambda_1 > \lambda_2$ weighs uniformity of pixel intensities inside the contour more heavily than uniformity of pixel intensities outside the contour i.e. this allows more variation in pixel intensities in the background and limits the variation in pixel intensities inside the contour. Setting $\lambda_1 < \lambda_2$ can be useful if the pixel intensities in the background are more uniform than the pixel intensities in the object to be segmented e.g. an object with non-uniform pixel



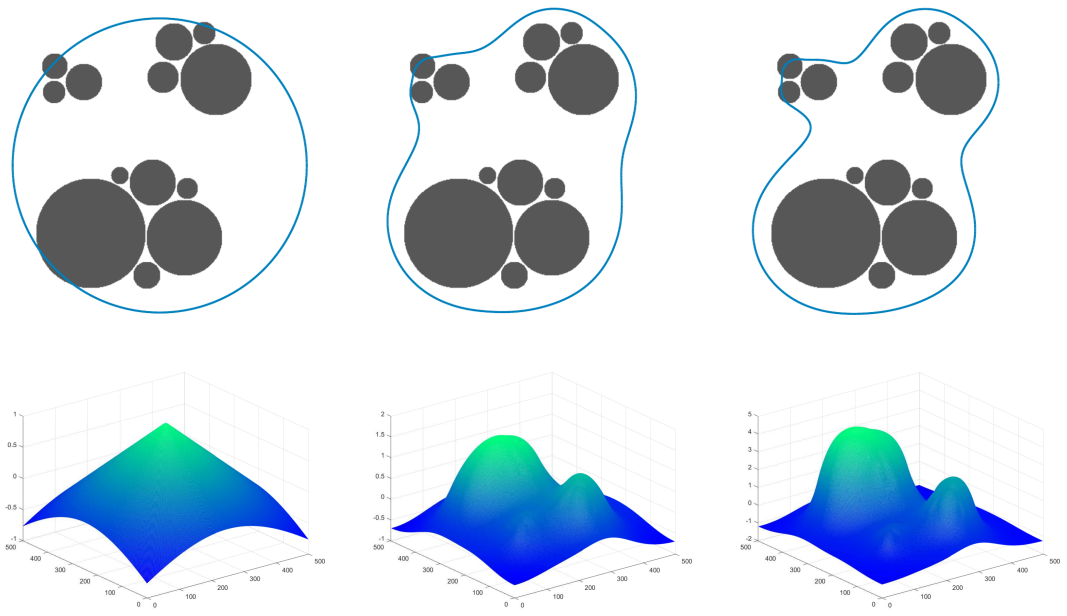
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.7: Varying the penalty for the length of the contour, for an image with grouped elements. $\mu = 0.01 \cdot 255^2$.

intensities placed on a completely black background.

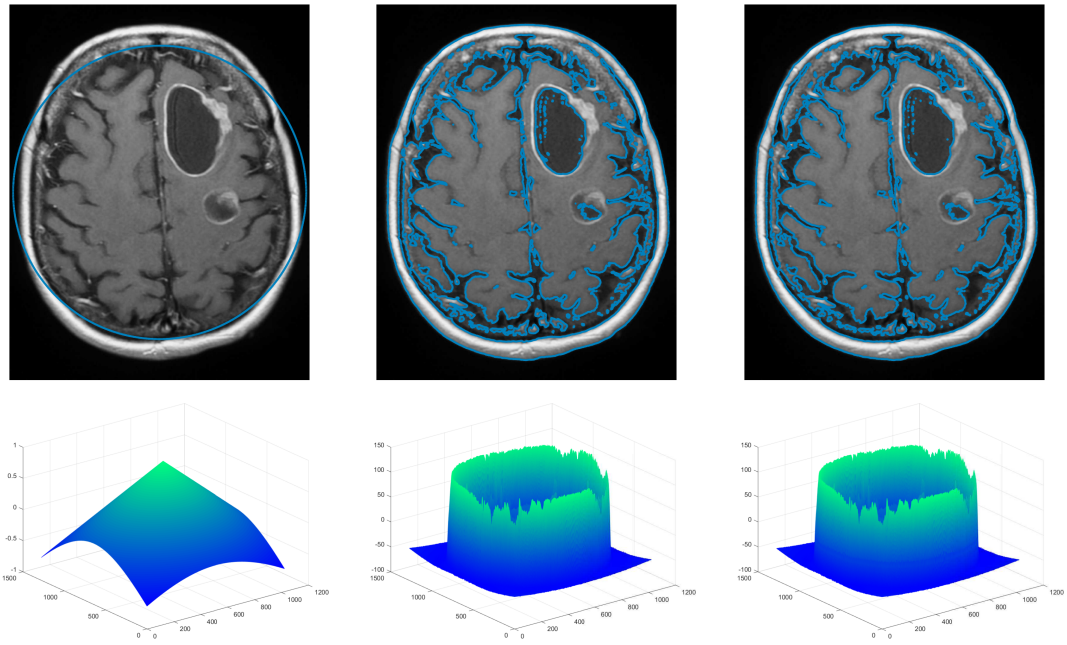
In the next example, the result of varying λ_1 and λ_2 are demonstrated for an image with some circles with different intensities. In this example $\mu = 0.0001 \cdot 255^2$ and $\nu = 0$. The effect of $\lambda_1 = \lambda_2$, $\lambda_1 < \lambda_2$ and $\lambda_1 > \lambda_2$ are visualized in Figure 2.15, Figure 2.16 and Figure 2.17 respectively.

Figure 2.15 clearly shows that $\lambda_1 = \lambda_2 = 1$ results in the outer two darker circles to be segmented. Since the inner circle intensity is relatively close to the intensity of the white background, it is excluded from the segmentation. The level set description ϕ in iteration 1 nicely shows the gap this generates in the middle. Figure 2.16 demonstrates that $\lambda_1 < \lambda_2$ forces more uniformity outside the contour. In this case this results in inclusion of the inner circle to the segmentation even though the intensity of the inner circle is relatively close to the intensity of the background. Figure 2.17 demonstrates the effect of setting $\lambda_1 > \lambda_2$. In this case uniformity inside the contour weighs heavier, thus resulting in a single circle to be segmented. This inner circle has a single intensity, minimizing the difference in the energy term. The result is less uniformity in the pixel intensities outside the contour.



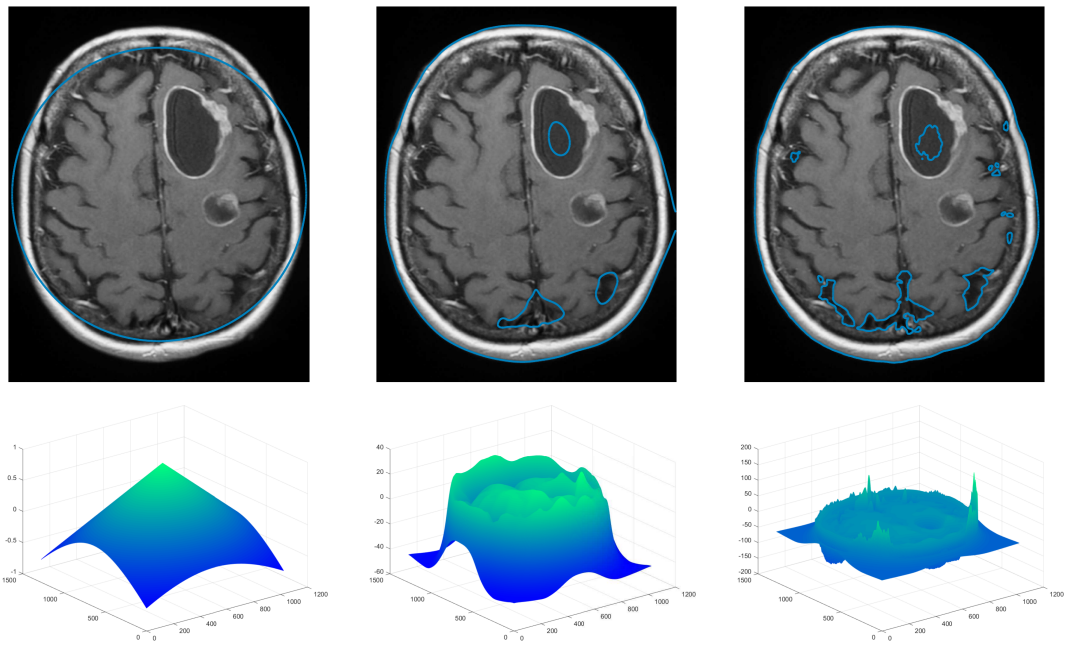
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.8: Varying the penalty for the length of the contour, for an image with grouped elements. $\mu = 1 \cdot 255^2$.



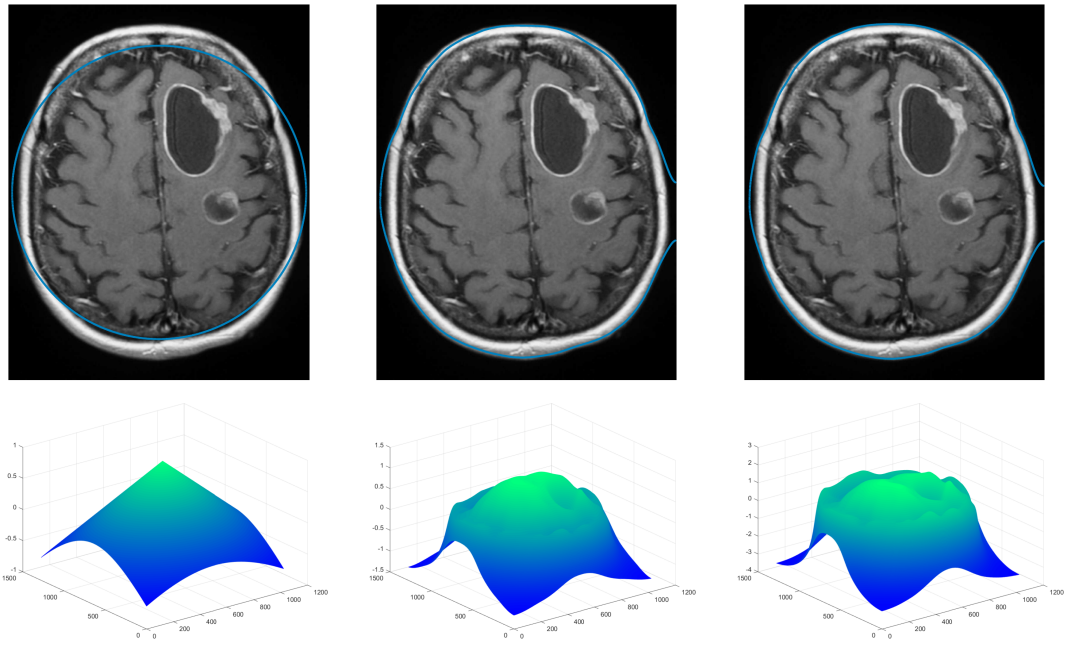
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.9: Varying the penalty for the length of the contour, for an image from a CT-scan of a brain with a tumor. $\mu = 0$.



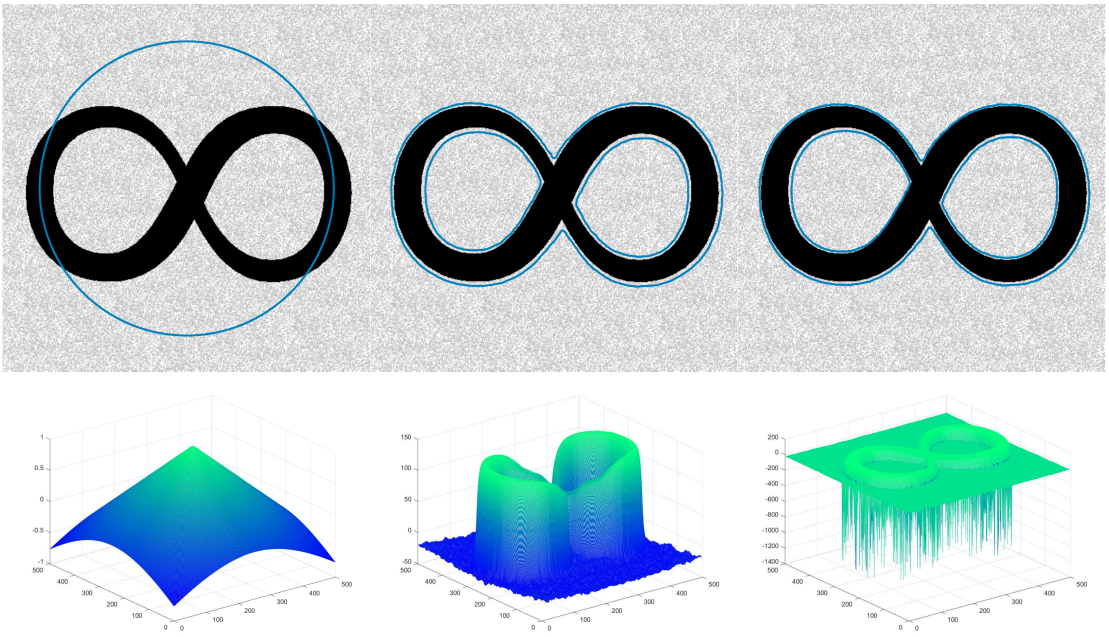
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.10: Varying the penalty for the length of the contour, for an image from a CT-scan of a brain with a tumor. $\mu = 0.01 \cdot 255^2$.



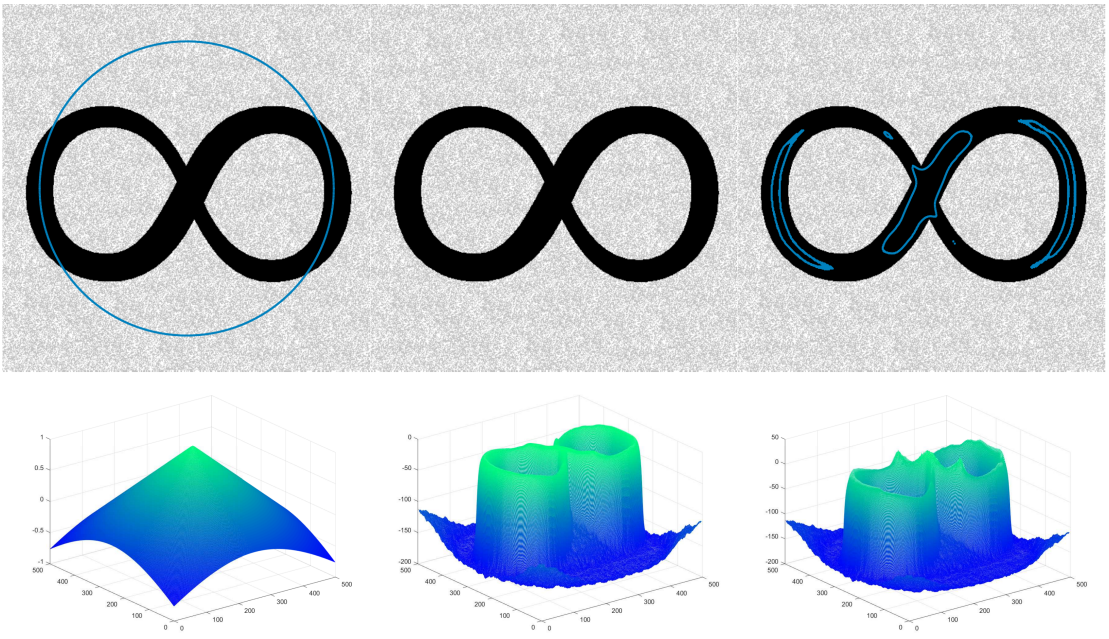
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.11: Varying the penalty for the length of the contour, for an image from a CT-scan of a brain with a tumor. $\mu = 1 \cdot 255^2$.



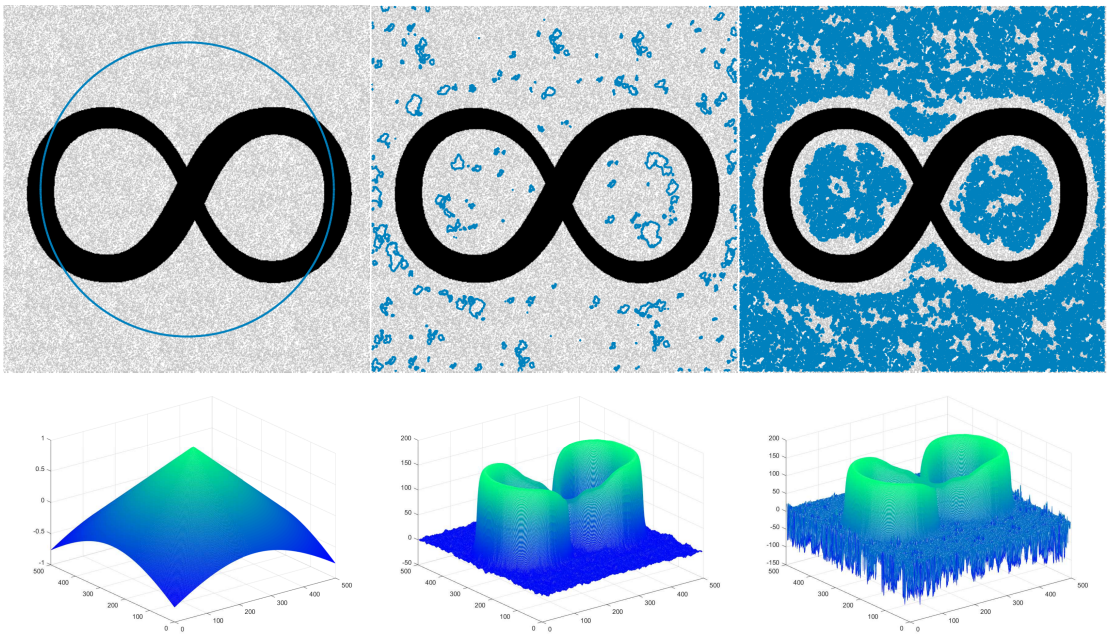
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.12: Varying the penalty for the area inside the contour, for a noisy image with a simple object. $\nu = 0$.



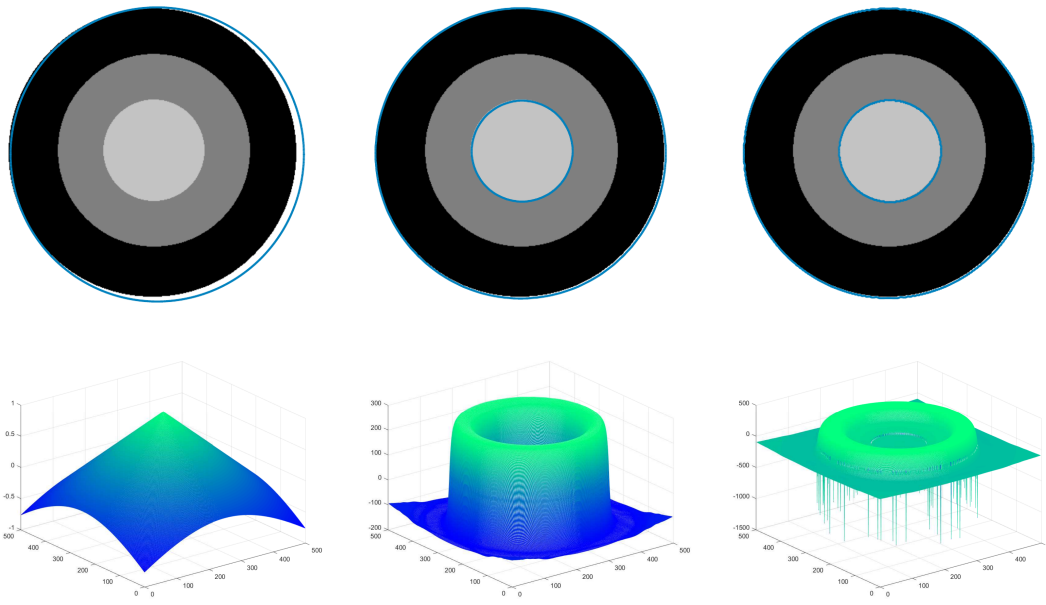
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.13: Varying the penalty for the area inside the contour, for a noisy image with a simple object. $\nu = 0.07 \cdot 255^2$.



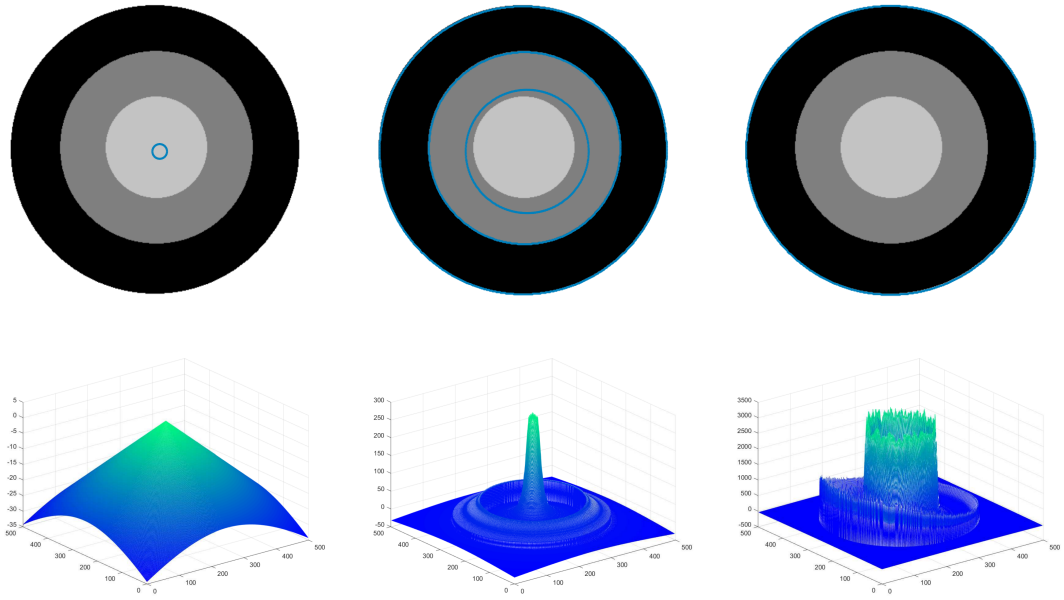
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.14: Varying the penalty for the area inside the contour, for a noisy image with a simple object. $\nu = -0.015 \cdot 255^2$.



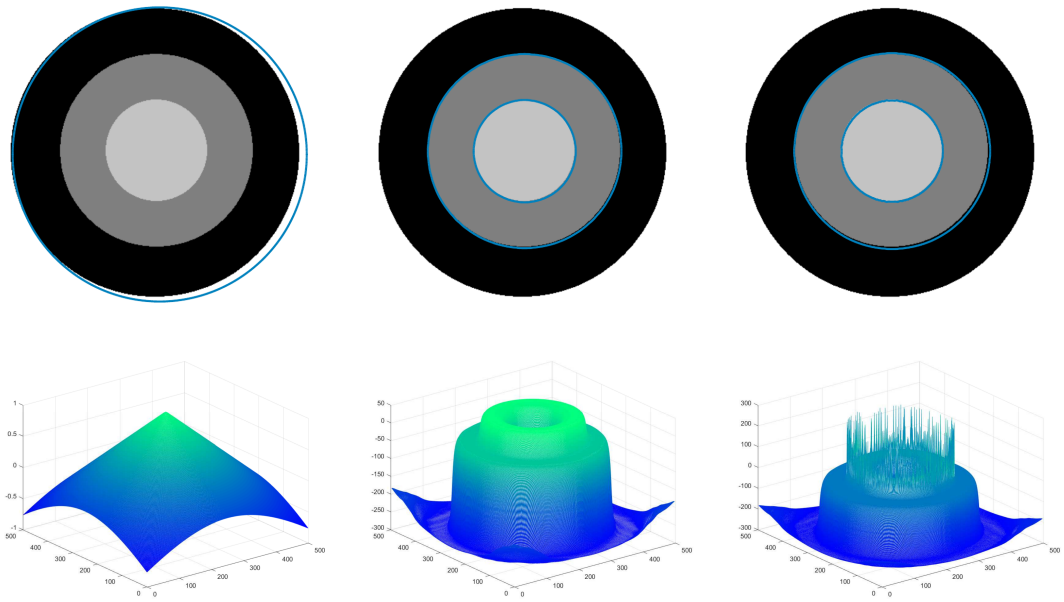
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.15: Varying the penalty for the average terms inside and outside the contour, for an image with some circles with different intensities. $\lambda_1 = \lambda_2 = 1$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.16: Varying the penalty for the average terms inside and outside the contour, for an image with some circles with different intensities. $\lambda_1 = 0.3$ and $\lambda_2 = 10$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

Figure 2.17: Varying the penalty for the average terms inside and outside the contour, for an image with some circles with different intensities. $\lambda_1 = 1.4$ and $\lambda_2 = 1$.

3 Implementation

This chapter discusses all the details regarding the implementation of an accelerator for the Chan-Vese model. Section 3.1 introduces a sequential implementation of a numerical approximation for the Chan-Vese model in two dimensions. Section 3.2 discusses the required steps to extend the given sequential implementation for contours in two dimensional images, to surfaces in three dimensional volumes. Section 3.3 analyses the opportunities for parallelization of both implementations, after which Section 3.4 reveals all the details of an accelerated implementation.

3.1 Sequential implementation of a numerical approximation

This section will lay out a possible sequential implementation of the Chan-Vese model for segmentation. The implementation uses the Successive over-relaxation (SOR) iterative method to solve the Partial differential equation (PDE). First SOR is explained and the discretization from [2] is presented, before the different steps of the implementation are given and linked to the model description as given in Chapter 2.

3.1.1 SOR

SOR is a variant of the Gauss-Seidel method, which is an iterative method used to solve a system of linear equations. SOR aims to speed up the convergence of the Gauss-Seidel method by setting the relaxation factor ω.

Let $A\boldsymbol{x} = \boldsymbol{b}$ be a square system of n linear equations with $\boldsymbol{x}$ the unknown. Here A is a $n \times n$ matrix and both $\boldsymbol{x}$ and $\boldsymbol{b}$ are column vectors of size n.

With this iterative technique, the left hand side of the equation can be solved for x using values from the previous iteration on the right hand side i.e. with k the iteration count and x_i^k the i-th element of $\boldsymbol{x}$ at iteration k, $x_i^{(k+1)}$ can be written in terms of A, $\boldsymbol{b}$ and $x_i^{(k)}$. This can be achieved by decomposing A into a diagonal component and strictly lower and upper triangular components, D, L and U respectively i.e. $A = D + L + U$. This allows the system of equations to be written as $(D+\omega L)\boldsymbol{x} = \omega\boldsymbol{b} - [\omega U + (\omega - 1)D]\boldsymbol{x}$ where $\omega > 1$ is the constant relaxation factor.

SOR then solves the equation for $\boldsymbol{x}$, using an iterative method, which means that the equation can also be written as can be seen in Equation (3.1).

$$\boldsymbol{x}^{(k+1)} = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)\boldsymbol{x}^{(k)} + \omega(D - \omega L)^{-1}\boldsymbol{b} \tag{3.1}$$

Writing this in terms of the matrix and vector elements yields Equation (3.2).

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\Big(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\Big),\ i = 1, 2, \ldots, n \tag{3.2}$$

Here a_{ij} denotes the element of matrix A from the i-th row and the j-th column, and b_i denotes the i-th element of $\boldsymbol{b}$.

3.1.2 Discretization

This section presents the discretization of the iterative algorithm introduced in [2].

A finite differences implicit scheme is used to discretize the equation in ϕ. Finite differences methods are numerical methods in which difference equations are used to solve differential equations i.e. the derivatives are approximated by finite differences.

The problem is mapped onto a grid with h the space step. For an input image with N and M the width and height respectively, grid points are defined according to $(x_i, y_i) = (ih, jh)$, where $1 \leq i \leq N$ and $1 \leq j \leq M$. With Δt the time step and $n \geq 0$ the iteration step, $\phi_{i,j}^n = \phi(n\Delta t, x_i, y_j)$ approximates $\phi(t, x, y)$, and $\phi^0 = \phi_0$.

3.1.2.1 Finite differences

This allows to define the finite differences as given in Equation (3.3).

$$\begin{aligned}
\Delta_-^x \phi_{i,j} &= \phi_{i,j} - \phi_{i-1,j} \\
\Delta_-^y \phi_{i,j} &= \phi_{i,j} - \phi_{i,j-1} \\
\Delta_+^x \phi_{i,j} &= \phi_{i+1,j} - \phi_{i,j} \\
\Delta_+^y \phi_{i,j} &= \phi_{i,j+1} - \phi_{i,j}
\end{aligned} \tag{3.3}$$

3.1.2.2 Heaviside function and one-dimensional Dirac measure

A smooth approximation, $H_\epsilon(z)$, for the Heaviside function $H(z)$ as defined in Equation (2.22), and a smooth approximation $\delta_\epsilon(z)$, for the one-dimensional Dirac measure $\delta_0(z)$ as defined in Equation (2.23), are presented in Equation (3.4) and Equation (3.5). For $\epsilon \to 0$ these approximations converge to $H(z)$ and $\delta_0(z)$.

$$H_\epsilon(z) = \frac{1}{2}\Big(1 + \frac{2}{\pi}\arctan(\frac{z}{\epsilon})\Big) \tag{3.4}$$

$$\delta_\epsilon(z) = \frac{d}{dz}H_\epsilon(z) = \frac{\epsilon}{\pi(\epsilon^2 + z^2)} \tag{3.5}$$

3.1.2.3 Region averages

For the numerical approximation of the divergence operator in Equation (2.33), the discretization of [24] is used. For the iterative algorithm the method of [25] is used. As stated before in Chapter 2, the steps are to alternatingly update ϕ and the region

averages c_1 and c_2. Given a ϕ^n, c_1^n and c_2^n can be computed using the equations as can be seen in Equation (3.6).

$$\begin{aligned} c_1(\phi^n) &= \frac{\int_\Omega u_0(x,y) H_\epsilon(\phi^n(x,y))\ dx\ dy}{\int_\Omega H_\epsilon(\phi^n(x,y))\ dx\ dy} \\ c_2(\phi^n) &= \frac{\int_\Omega u_0(x,y)(1 - H_\epsilon(\phi^n(x,y)))\ dx\ dy}{\int_\Omega (1 - H_\epsilon(\phi^n(x,y)))\ dx\ dy} \end{aligned} \tag{3.6}$$

3.1.2.4 Discretization of ϕ in space

The evolvement of ϕ, as described in Equation (2.33), is discretized in space according to [2] and [23], who adopted their methods from [25] and [24], as can be seen in Equation (3.7).

$$\begin{aligned} \frac{\delta\phi_{i,j}}{\delta t} = \delta_h(\phi_{i,j}) \Bigg[& \frac{\mu}{h^2}\Delta_-^x \cdot \left(\frac{\Delta_+^x \phi_{i,j}}{\sqrt{(\Delta_+^x \phi_{i,j})^2/(h^2) + (\phi_{i,j+1} - \phi_{i,j-1})^2/(2h)^2}} \right) \\ & + \frac{\mu}{h^2}\Delta_-^y \cdot \left(\frac{\Delta_+^y \phi_{i,j}}{\sqrt{(\phi_{i+1,j} - \phi_{i-1,j})^2/(2h)^2 + (\Delta_+^y \phi_{i,j})^2/(h^2)}} \right) \\ & - \nu - \lambda_1 (u_{0,i,j} - c_1(\phi))^2 + \lambda_2 (u_{0,i,j} - c_2(\phi))^2 \Bigg] \end{aligned} \tag{3.7}$$

Here δ_h is defined by Equation (3.5) with $\epsilon = h$. With the space step $h = 1$, the addition of a small constant η to prevent division by zero, and substitution of the finite differences as defined in Equation (3.3), this simplifies to Equation (3.8).

$$\begin{aligned} \frac{\delta\phi_{i,j}}{\delta t} = \delta_h(\phi_{i,j}) \Bigg[& \mu \left(\Delta_-^x \cdot \left(\frac{\Delta_+^x \phi_{i,j}}{\sqrt{\eta^2 + (\Delta_+^x \phi_{i,j})^2 + (\Delta_+^y \phi_{i,j} + \Delta_-^y \phi_{i,j})^2/4}} \right) + \right. \\ & \left. \Delta_-^y \cdot \left(\frac{\Delta_+^y \phi_{i,j}}{\sqrt{\eta^2 + (\Delta_+^y \phi_{i,j})^2 + (\Delta_+^x \phi_{i,j} + \Delta_-^x \phi_{i,j})^2/4}} \right) \right) \\ & - \nu - \lambda_1 (u_{0,i,j} - c_1(\phi))^2 + \lambda_2 (u_{0,i,j} - c_2(\phi))^2 \Bigg] \end{aligned} \tag{3.8}$$

In order to simplify Equation (3.8) further, the weights as suggested in [23] and given in Equation (3.9) are used to rewrite the discretization to Equation (3.10).

$$\begin{aligned}\rho^x_{i,j} &= \frac{\mu}{\sqrt{\eta^2 + (\Delta^+_x \phi_{i,j})^2 + (\Delta^y_+ \phi_{i,j} + \Delta^y_- \phi_{i,j})^2/4}} \\ \rho^y_{i,j} &= \frac{\mu}{\sqrt{\eta^2 + (\Delta^+_y \phi_{i,j})^2 + (\Delta^x_+ \phi_{i,j} + \Delta^x_- \phi_{i,j})^2/4}}\end{aligned} \tag{3.9}$$

$$\begin{aligned}\frac{\delta \phi_{i,j}}{\delta t} = \delta_h(\phi_{i,j}) \Big[&(\rho^x_{i,j} \Delta^x_+ \phi_{i,j} - \rho^x_{i-1,j} \Delta^x_- \phi_{i,j}) + (\rho^y_{i,j} \Delta^y_+ \phi_{i,j} - \rho^y_{i,j-1} \Delta^y_- \phi_{i,j}) \\ &-\nu - \lambda_1 (u_{0,i,j} - c_1(\phi))^2 + \lambda_2 (u_{0,i,j} - c_2(\phi))^2 \Big]\end{aligned} \tag{3.10}$$

3.1.2.5 Discretization of ϕ in time

Equation (3.10) is discretized in time according to [23] and [25]. The idea is to compute all the elements of ϕ^{n+1} in a row-by-row sweep from the top to the bottom and from left to right. A result of this method is that, in order to compute the ϕ^{n+1}, the elements above and to the left are evaluated at time step $n+1$, since they are computed earlier in the sweep, and all other values are evaluated at time step n i.e. computation of $\phi^{n+1}_{i,j}$ depends $\phi^{n+1}_{i-1,j}$, $\phi^{n+1}_{i,j-1}$, $\phi^n_{i+1,j}$ and $\phi^n_{i,j+1}$. This is also visualized in Figure 3.1. This allows Equation (3.10) to be rewritten to Equation (3.11).

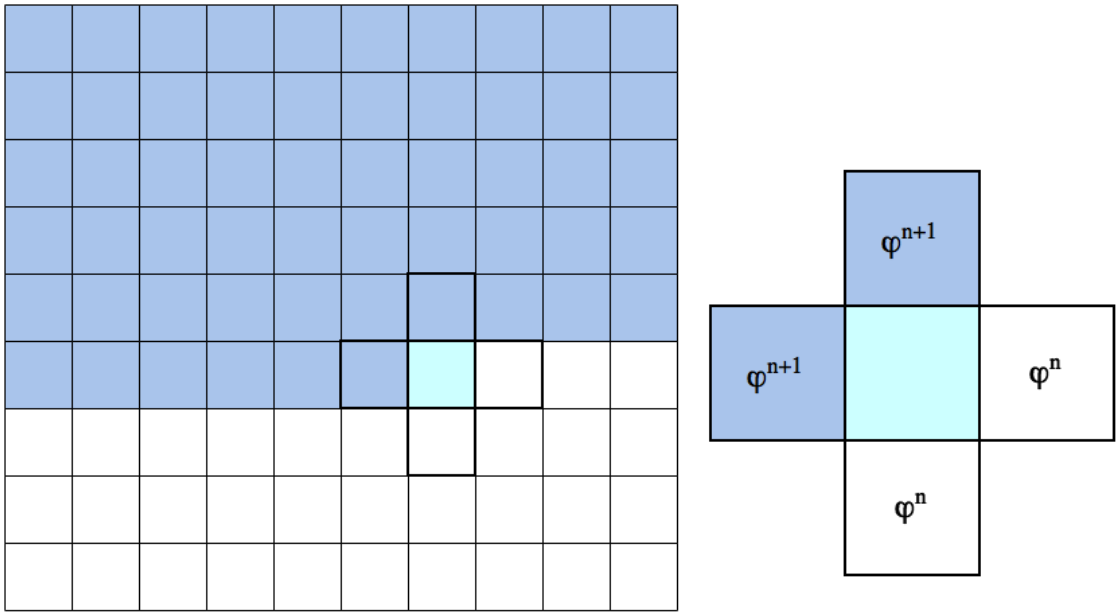


Figure 3.1: Sequential SOR sweep dependencies.

$$\begin{aligned}\frac{\phi_{i,j}^{n+1}-\phi_{i,j}^{n}}{\Delta t} = \delta_h(\phi_{i,j}^n)\Big[&\rho_{i,j}^x\phi_{i+1,j}^n + \rho_{i-1,j}^x\phi_{i-1,j}^{n+1} + \rho_{i,j}^y\phi_{i,j+1}^n + \rho_{i,j-1}^y\phi_{i,j-1}^{n+1} \\ &-(\rho_{i,j}^x + \rho_{i-1,j}^x + \rho_{i,j}^y + \rho_{i,j-1}^y)\phi_{i,j}^{n+1} \\ &-\nu - \lambda_1(u_{0,i,j} - c_1(\phi^n))^2 + \lambda_2(u_{0,i,j} - c_2(\phi^n))^2\Big]\end{aligned} \tag{3.11}$$

Moving $\phi_{i,j}^{n+1}$ in Equation (3.11) to the right hand side of the equation and applying SOR for the iteration, results in the actual computation to be performed for all elements in the grid except for the borders, as shown in Equation (3.12).

$$\begin{aligned}\phi_{i,j}^{n+1} =&(1-\omega)\phi_{i,j}^n + \\ &\omega\cdot\Big[\phi_{i,j}^n + \Delta t\delta_h(\phi_{i,j}^n)\Big(\rho_{i,j}^x\phi_{i+1,j}^n + \rho_{i-1,j}^x\phi_{i-1,j}^{n+1} + \rho_{i,j}^y\phi_{i,j+1}^n + \rho_{i,j-1}^y\phi_{i,j-1}^{n+1} \\ &-\nu - \lambda_1(u_{0,i,j} - c_1(\phi^n))^2 + \lambda_2(u_{0,i,j} - c_2(\phi^n))^2\Big)\Big] \\ &/\Big[1 + \Delta t\delta_h(\phi_{i,j}^n)\Big(\rho_{i,j}^x + \rho_{i-1,j}^x + \rho_{i,j}^y + \rho_{i,j-1}^y\Big)\Big]\end{aligned} \tag{3.12}$$

The value of ω determines the convergence rate of the SOR iteration. For SOR, $1 < \omega < 2$ normally gives good convergence[26].

3.1.3 Algorithm

An overview of the different steps in the implementation of the algorithm is given below.

1. Initialization
   - Initialize ϕ^0 by ϕ_0 and set $n = 0$.
2. Main loop
   (a) Copy
   - Copy ϕ^{n+1} to ϕ^n

   (b) Boundary conditions
   - Enforce Neumann boundary conditions

   (c) Region averages
   - Compute region averages for ϕ^n i.e. $c_1(\phi^n)$ and $c_2(\phi^n)$

   (d) Finite differences
   - Compute finite differences for ϕ^n

   (e) Weights
   - Compute weights for ϕ^n

   (f) Solve the PDE
   - Solve the PDE using SOR to obtain ϕ^{n+1} using ϕ^n, $c_1(\phi^n)$ and $c_2(\phi^n)$
3. Generate binary output image of segmentation.

3.1.4 Setup

This subsection describes the details regarding the setup in terms of input and output parameters, required data structures and used libraries.

Table 3.1 lists all input and output arguments of the sequential implementation. The input arguments are parsed using the `tclap` library.

Name	Type	Default	Description
input	`String`		Path to input image
output	`String`		Path to output image
timestep	`float`	10	Timestep (Δt) size for the PDE
steps	`int`	10	Main loop iterations
iterations	`int`	100	Fixed number of SOR iterations per step
lambda1	`float`	1	Penalty for average term inside the contour (λ_1)
lambda2	`float`	1	Penalty for average term outside the contour (λ_2)
mu	`float`	50	Penalty for the length of the contour (μ)
nu	`float`	0	Penalty for the size of area inside the contour (ν)
omega	`float`	1.97	Relaxation factor for SOR (ω)
checkerboard	`int`	0	Use checkerboard initialization
circle_x	`int`	x_c	x-coordinate of circle origin
circle_y	`int`	y_c	y-coordinate of circle origin
circle_r	`int`	$\min(x_c, y_c)/2$	Circle radius

Table 3.1: Input and output arguments

In this implementation all grids are defined with a space step of $h = 1$. Let u_0 define a grayscale input image, with n_x the width and n_y the height. Empty borders are added to u_0. The question how to handle the boundaries often arises in image processing. Here the borders, which are added to the input image, make sure that evaluation of neighboring pixels at the edges is defined and segmentation faults are prevented. With b_x and b_y the widths of the borders on the sides, and top and bottom respectively, n_{xx} is defined as the width of the image including the borders i.e. $n_{xx} = n_x + 2 \cdot b_x$, and n_{yy} is the height of the image including the borders i.e. $n_{yy} = n_y + 2 \cdot b_y$. x_c and y_c denote the center coordinates of the image i.e. $x_c = n_{xx}/2$ and $y_c = n_{yy}/2$. This can also be seen in Figure 3.2. The complete grid is referred to as Γ_0 whereas the grid without the borders is referred to as γ_0.

The first data object described in Table 3.2 holds the input image grayscale data where each pixel value is stored in a byte. The empty borders in the input image, in this case, do not have to be added for possible problems around the edges, but they are added to keep pointer arithmetic simple and similar to other $n_{xx} \times n_{yy}$ objects, where the borders are required due to the reason given before.

The input image is loaded into memory from the disk using OpenCV[27], an open source computer vision library, and is then stored in a `Mat` container.

In this numerical approximation a two-dimensional five-point stencil is used to solve the PDE using SOR. The five-point stencil for a point (x, y) in a grid with space step h can be seen in Equation (3.13) and Figure 3.3. With the grid space step set to $h = 1$, the

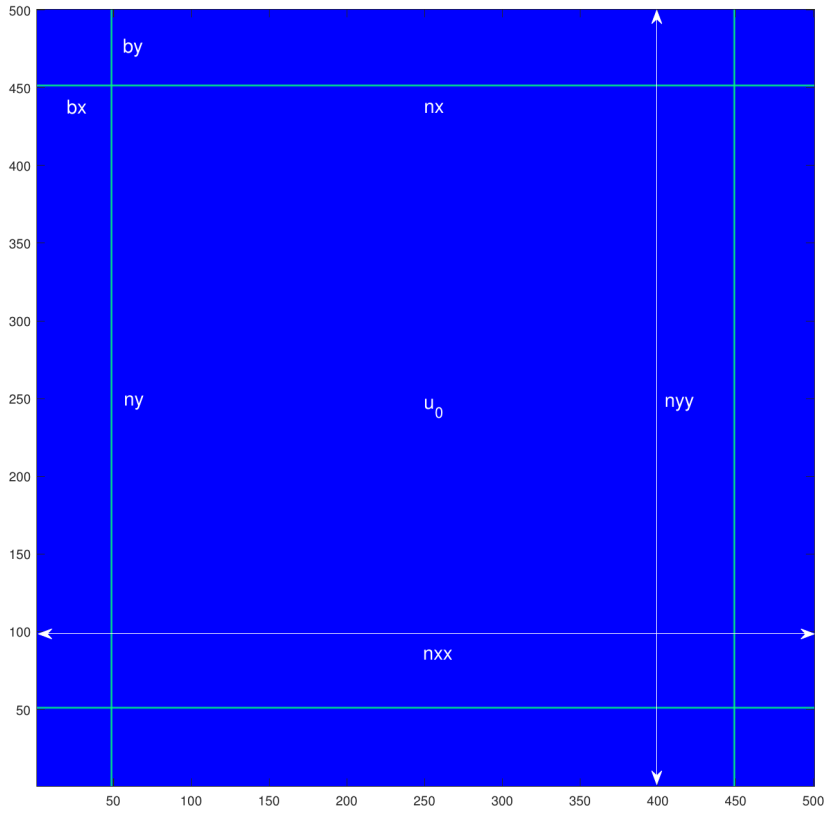


Figure 3.2: Overview of dimension symbols. Here $n_x = n_y = 400$, $b_x = b_y = 50$ and $n_{xx} = n_{yy} = 500$. This gives $x_c = y_c = 250$.

Object	Type	Dimensions
u_0	`uchar`	$n_{xx} \times n_{yy}$
ϕ	`float`	$n_{xx} \times n_{yy}$
ϕ_{old}	`float`	$n_{xx} \times n_{yy}$
ψ	`float`	$n_{xx} \times n_{yy} \times 2$
ρ	`float`	$n_{xx} \times n_{yy} \times 5$

Table 3.2: Objects in memory used in the implementation, with their types and sizes

five-point stencil determines the requirement for the border size to be $b_x = b_y = h = 1$.

$$\{(x, y), (x - h, y), (x + h, y), (x, y - h), (x, y + h)\} \tag{3.13}$$

The two other objects listed in Table 3.2 are used to store intermediate computation

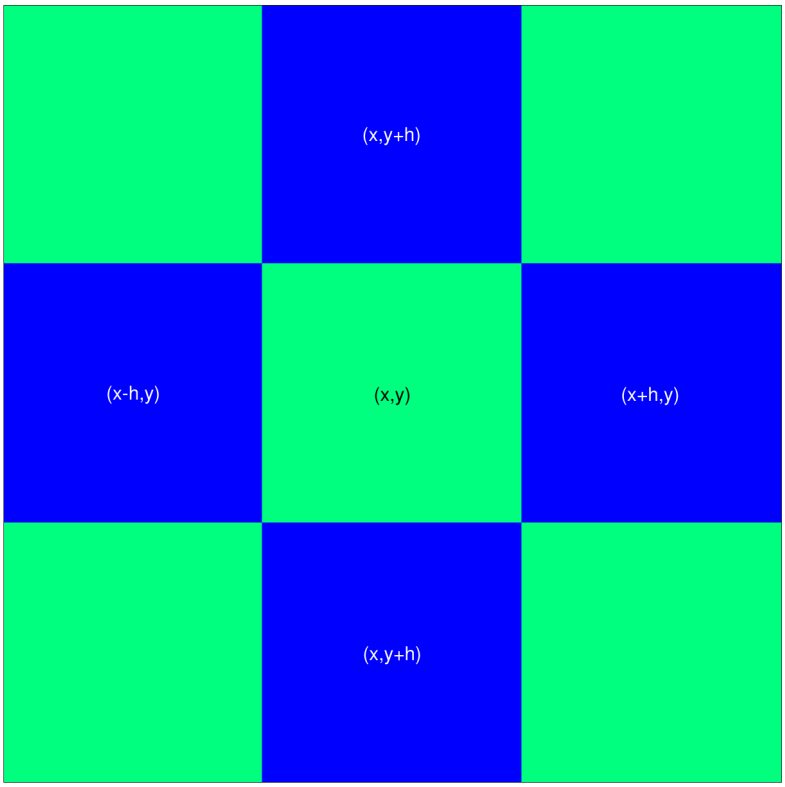


Figure 3.3: The five-point stencil for a point (x, y) in a grid with space step h.

results i.e. the finite differences in ψ, and some weights, used in the SOR iteration, in ρ.

Both tables list single precision floating-point representation for most data objects. Normally when using floating-point representation one should use double precision where possible. In this implementation single precision representation is favored over double precision, not to reduce memory usage, but due to the extra performance for single precision in the special hardware targeted in the accelerated implementation. The texture memory, for example, can not be leveraged with double precision values.

3.1.5 Initialization

This subsection discusses the initialization step of the algorithm.

The active contour is represented by the zero level set of a level set function, which is stored in ϕ. Since different initial contours may result in quicker convergence, as discussed in Chapter 2, different initial contours can be selected in the implementation.

The two initial contours as seen in Figure 2.4 and Figure 2.5 are implemented according to Equation (3.15) and Equation (3.14) respectively. The former results in a zero level set that looks like a checkerboard pattern, whereas for the latter the resulting zero level set, or initial contour, is a circle with radius r and (x_c, y_c) as center coordinates. The radius and origin coordinates can be set using an input argument.

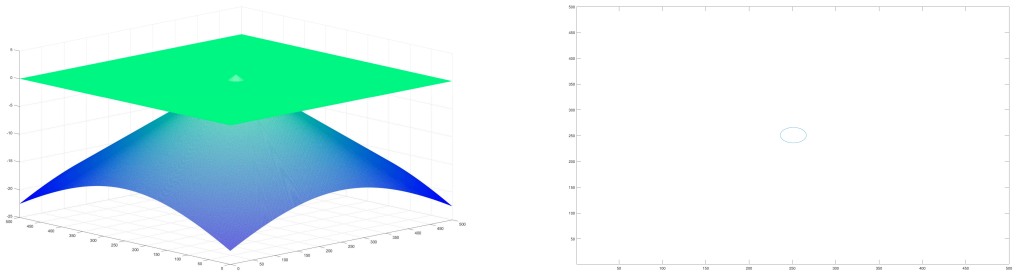
Values for ϕ are stored per pixel in a `Mat` container, as single precision floating points as can be seen in Table 3.2. After convergence of the model, the final contour can be collected by evaluating the zero level set of ϕ i.e. points for which $\phi = 0$. The inside of the contour can be collected by evaluating the points for which $\phi > 0$. The outside of the contour can be collected by evaluating the points for which $\phi < 0$. This can also be seen from Equation (2.21) and Figure 2.3.

The ϕ_{old} object is allocated to hold a copy of ϕ between iterations of the main loop, which is discussed in the next subsection.

$$\phi_{i,j} = -\left(\frac{\sqrt{2^{i-b_x-x_c} + 2^{j-b_y-y_c}}}{r} - 1\right) \tag{3.14}$$

$$\phi_{i,j} = sin\left(\frac{i\pi}{5}\right) sin\left(\frac{j\pi}{5}\right) \tag{3.15}$$

The result of the initialization step for ϕ, with Equation (3.14) and $r = 15$, visualized for a 500×500 image can be seen in Figure 3.4.



(a) Level set: ϕ

(b) Zero level set: ϕ_0 i.e. $\phi = 0$

Figure 3.4: Initialization with circle of radius $r = 15$ in a 500×500 image, using Equation (3.14)

3.1.6 Main loop

The main loop, or outer loop, implements the evolvement of the contour by updating the level set description, ϕ, of the contour in each iteration. Iterations of the outer loop are referred to as steps. The outer loop can be set by an input argument to iterate for a fixed number of iterations.

3.1.6.1 Copy

At the beginning of each step, a copy is made of the level set description, ϕ, to ϕ_{old}. In the main loop, ϕ is updated to the evolved level set description of the active contour whereas ϕ_{old} holds the level set description of the active contour of the previous step i.e. ϕ is the equivalent of ϕ^{n+1} and ϕ_{old} is the equivalent of ϕ^n.

3.1.6.2 Boundary conditions

After making a copy of the level set description of the active contour, ϕ, the Neumann boundary condition is enforced on the borders of ϕ[28][29]. This assumes that the data in the borders, which is outside the input image overlap, is assumed to be a reflection of the data within the borders i.e. where the input image overlaps with ϕ i.e. $b_x \leq x \leq n_x + b_x$ and $n_y \leq y \leq n_y + b_y$. Since the border size is set to $b_x = b_y = 1$, the implementation becomes relatively simple and applies the reflections as can be seen in Equation (3.16) and Equation (3.17). Equation (3.16) is applied for $0 \leq i \leq n_{xx}$ and reflects the copy of values in the top and bottom row, whereas Equation (3.17) is applied for $0 \leq j \leq n_{yy}$ and reflects the copy of values in the first and last column.

$$\begin{aligned} \phi_{i,0} &= \phi_{i,1} \\ \phi_{i,n_{yy}} &= \phi_{i,n_{yy}-1} \end{aligned} \tag{3.16}$$

$$\begin{aligned} \phi_{0,j} &= \phi_{1,j} \\ \phi_{n_{xx},j} &= \phi_{n_{xx}-1,j} \end{aligned} \tag{3.17}$$

The order in which these boundary conditions are applied determines the values of the corner pixels. However, this race condition can be considered non-critical since these values are never accessed in any of the other steps.

3.1.6.3 Region averages

The region averages, as described in Section 2.3, are computed by evaluating the regularized Heaviside step function, as proposed by [2] and given in Equation (3.4) with $\epsilon = 1$, for all the values in γ_0 i.e. ϕ^n for which $b_x \leq i \leq n_x + b_x$ and $b_y \leq j \leq ny + b_y$. These results are multiplied with pixel intensities and accumulated according to Equation (3.6) to compute c_1 and c_2, as can be seen in Equation (3.18).

$$
\begin{aligned}
c_1 &= \frac{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} H_\epsilon(\phi_{i,j}) \cdot u_{0_{i,j}}}{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} H_\epsilon(\phi_{i,j})} \\
c_2 &= \frac{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} (1 - H_\epsilon(\phi_{i,j})) \cdot u_{0_{i,j}}}{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} (1 - H_\epsilon(\phi_{i,j}))}
\end{aligned}
\tag{3.18}
$$

3.1.6.4 Finite differences

In this step, the required finite differences, as given in Equation (3.3), are computed for all values in γ_0, and stored in ψ.

For example, let $i = 4$ and $j = 8$. This defines $\Delta_+^x \phi_{4,8} = \phi_{5,8} - \phi_{4,8}$. Now, if the cell to the right is considered i.e. $i = 5$ and $j = 8$, $\Delta_-^x \phi_{5,8} = \phi_{5,8} - \phi_{4,8}$. It can clearly be seen that $\Delta_+^x \phi_{4,8} = \Delta_-^x \phi_{5,8}$. The same hold true for the Δ_-^y and Δ_+^y operators. In general this can be formulated as follows in Equation (3.19).

$$
\begin{aligned}
\Delta_+^x \phi_{i,j} &= \Delta_-^x \phi_{i+1,j} \\
\Delta_-^x \phi_{i,j} &= \Delta_+^x \phi_{i-1,j} \\
\Delta_+^y \phi_{i,j} &= \Delta_-^y \phi_{i,j+1} \\
\Delta_-^y \phi_{i,j} &= \Delta_+^y \phi_{i,j-1}
\end{aligned}
\tag{3.19}
$$

As a results, only two finite differences per cell need to be computed and stored i.e. Δ_+^x and Δ_+^y. Then in order to access $\Delta_-^x \phi_{i,j}$ one can simply access $\Delta_+^x \phi_{i-1,j}$.

3.1.6.5 Weights

In this step, the weights ρ^x and ρ^y, as given in Equation (3.9), and as used in Equation (3.12), are computed for all values in γ_0, for all directions i.e. $\rho_{i,j}^x$, $\rho_{i-1,j}^x$, $\rho_{i,j}^y$ and $\rho_{i,j-1}^y$. The center weight, $(\rho_{i,j}^x + \rho_{i-1,j}^x + \rho_{i,j}^y + \rho_{i,j-1}^y)$, as used in the denominator of Equation (3.12) is also computed and stored.

3.1.7 Iterative solver loop

This implementation uses the SOR iterative solver loop to solve the PDE, as described in Subsection 3.1.1.

In this implementation the number of iteration is fixed to the value provided as input argument. However, instead one could also add a check to investigate if the solution has converged i.e. if the smallest maximum difference of values in the solution between iterations is below a certain threshold.

Equation (3.12) is evaluated in each iteration for all cells in γ_0, and ϕ is updated to converge to the solution. ω can be set using an input argument or it defaults to $\omega = 1.97$.

In each iteration, ϕ gets updated, and in the next iteration ϕ is used again as ϕ^{n+1} where needed.

3.1.8 Segmentation

The final segmentation can be obtained by evaluation of ϕ in γ_0, according to Equation (2.21). The output is a binary image with pixel values set to 0 for all $\phi < 0$ and pixels values to set to 1 for all $\phi \geq 0$.

3.2 Extending to 3D

This section extends the given sequential implementation of the Chan-Vese model with SOR as method to solve the PDE, as described for 2D in the previous section, to an implementation for segmentation of surfaces in 3D.

There are two ways, in general, to extend the 2D implementation to a 3D implementation. The first, most trivial way to solve 3D segmentation problems, is by using a 2D segmentation model for each slice. However, there are many downsides to this approach e.g. different slices require different input parameters to find the sought segmentation. It is better to redefine the model. For the Chan-Vese model the energy functional[2], as defined in Equation (2.32), becomes Equation (3.20), now with Ω a bounded open subset of $\mathbb{R}^3$, and v_0 the input volume.

$$\begin{aligned} F_\epsilon^{3D}(c_1, c_2, \phi) = \mu &\int_\Omega \delta_\epsilon(\phi(x,y,z))|\nabla\phi(x,y,z)| \; dx \; dy \; dz \\ &+ \nu \int_\Omega H_\epsilon(\phi(x,y,z))) \; dx \; dy \; dz \\ &+ \lambda_1 \int_\Omega |v_0(x,y,z) - c_1|^2 H_\epsilon(\phi(x,y,z)) \; dx \; dy \; dz \\ &+ \lambda_2 \int_\Omega |v_0(x,y,z) - c_2|^2 (1 - H_\epsilon(\phi(x,y,z))) \; dx \; dy \; dz \end{aligned} \tag{3.20}$$

The steps in the algorithm are the same as described for the 2D implementation in Subsection 3.1.3.

3.2.1 Discretization

This subsections lists differences in the discretization, compared to the one described for 2D in Subsection 3.1.2, which are used in the implementation for 3D.

3.2.1.1 Finite differences

For the finite differences, the operators as introduced in Equation (3.3) are updated to reflect the addition of an extra dimension, and two new operators, which reflect the finite differences in the z direction, are introduced as can be seen in Equation (3.21).

$$
\begin{aligned}
\Delta_-^x \phi_{i,j,k} &= \phi_{i,j,k} - \phi_{i-1,j,k} \\
\Delta_-^y \phi_{i,j,k} &= \phi_{i,j,k} - \phi_{i,j-1,k} \\
\Delta_-^z \phi_{i,j,k} &= \phi_{i,j,k} - \phi_{i,j,k-1} \\
\Delta_+^x \phi_{i,j,k} &= \phi_{i+1,j,k} - \phi_{i,j,k} \\
\Delta_+^y \phi_{i,j,k} &= \phi_{i,j+1,k} - \phi_{i,j,k} \\
\Delta_+^z \phi_{i,j,k} &= \phi_{i,j,k+1} - \phi_{i,j,k}
\end{aligned}
\tag{3.21}
$$

3.2.1.2 Volume averages

Region averages c_1 and c_2, as introduced in Equation (3.6), are updated to integrate over the extra dimension as can be seen in Equation (3.22), and are now referred to as volume averages.

$$
\begin{aligned}
c_1(\phi^n) &= \frac{\int_\Omega v_0(x,y,z) H_\epsilon(\phi^n(x,y,z))\ dx\ dy\ dz}{\int_\Omega H_\epsilon(\phi^n(x,y,z))\ dx\ dy\ dz} \\
c_2(\phi^n) &= \frac{\int_\Omega v_0(x,y,z)(1 - H_\epsilon(\phi^n(x,y,z)))\ dx\ dy\ dz}{\int_\Omega (1 - H_\epsilon(\phi^n(x,y,z)))\ dx\ dy\ dz}
\end{aligned}
\tag{3.22}
$$

3.2.1.3 Discretization of ϕ in space

The discretization of ϕ in space as described in Subsection 3.1.2.4, is updated to reflect the addition of the extra dimension.

Equation (3.8) becomes Equation (3.23).

$$
\begin{aligned}
\frac{\delta \phi_{i,j,k}}{\delta t} = \delta_h(\phi_{i,j,k}) \cdot \\
\Bigg[\mu \Bigg(\Delta_-^x \cdot \left(\frac{\Delta_+^x \phi_{i,j,k}}{\sqrt{\eta^2 + (\Delta_+^x \phi_{i,j,k})^2 + (\Delta_+^y \phi_{i,j,k} + \Delta_-^y \phi_{i,j,k})^2/4 + (\Delta_+^z \phi_{i,j,k} + \Delta_-^z \phi_{i,j,k})^2/4}} \right) + \\
\Delta_-^y \cdot \left(\frac{\Delta_+^y \phi_{i,j,k}}{\sqrt{\eta^2 + (\Delta_+^y \phi_{i,j,k})^2 + (\Delta_+^x \phi_{i,j,k} + \Delta_-^x \phi_{i,j,k})^2/4 + (\Delta_+^z \phi_{i,j,k} + \Delta_-^z \phi_{i,j,k})^2/4}} \right) + \\
\Delta_-^z \cdot \left(\frac{\Delta_+^z \phi_{i,j,k}}{\sqrt{\eta^2 + (\Delta_+^z \phi_{i,j,k})^2 + (\Delta_+^x \phi_{i,j,k} + \Delta_-^x \phi_{i,j,k})^2/4 + (\Delta_+^y \phi_{i,j,k} + \Delta_-^y \phi_{i,j,k})^2/4}} \right) \Bigg) \\
-\nu - \lambda_1 (v_{0,i,j,k} - c_1(\phi))^2 + \lambda_2 (v_{0,i,j,k} - c_2(\phi))^2 \Bigg]
\end{aligned}
\tag{3.23}
$$

With the weights ρ, as given in Equation (3.9), updated as given in Equation (3.24), Equation (3.10) can be rewritten to Equation (3.25).

$$\begin{aligned}
\rho^x_{i,j,k} &= \frac{\mu}{\sqrt{\eta^2 + (\Delta^+_x \phi_{i,j,k})^2 + (\Delta^y_+ \phi_{i,j,k} + \Delta^y_- \phi_{i,j,k})^2/4 + (\Delta^z_+ \phi_{i,j,k} + \Delta^z_- \phi_{i,j,k})^2/4}} \\
\rho^y_{i,j,k} &= \frac{\mu}{\sqrt{\eta^2 + (\Delta^+_y \phi_{i,j,k})^2 + (\Delta^x_+ \phi_{i,j,k} + \Delta^x_- \phi_{i,j,k})^2/4 + (\Delta^z_+ \phi_{i,j,k} + \Delta^z_- \phi_{i,j,k})^2/4}} \\
\rho^z_{i,j,k} &= \frac{\mu}{\sqrt{\eta^2 + (\Delta^+_z \phi_{i,j,k})^2 + (\Delta^x_+ \phi_{i,j,k} + \Delta^x_- \phi_{i,j,k})^2/4 + (\Delta^y_+ \phi_{i,j,k} + \Delta^y_- \phi_{i,j,k})^2/4}}
\end{aligned} \tag{3.24}$$

$$\begin{aligned}
\frac{\delta \phi_{i,j,k}}{\delta t} = \delta_h(\phi_{i,j,k}) \big[&(\rho^x_{i,j,k} \Delta^x_+ \phi_{i,j,k} - \rho^x_{i-1,j,k} \Delta^x_- \phi_{i,j,k}) + \\
&(\rho^y_{i,j,k} \Delta^y_+ \phi_{i,j,k} - \rho^y_{i,j-1,k} \Delta^y_- \phi_{i,j,k}) + \\
&(\rho^z_{i,j,k} \Delta^z_+ \phi_{i,j,k} - \rho^z_{i,j,k-1} \Delta^z_- \phi_{i,j,k}) - \\
&\nu - \lambda_1 (v_{0,i,j,k} - c_1(\phi))^2 + \lambda_2 (v_{0,i,j,k} - c_2(\phi))^2 \big]
\end{aligned} \tag{3.25}$$

3.2.1.4 Discretization of ϕ in time

The discretization of ϕ in time uses the same method as described in Subsection 3.1.2.5. ϕ^{n+1} is computed in a sweep from z-top to z-bottom, from left to right and from top to bottom. Computation of $\phi^{n+1}_{i,j,k}$ depends on $\phi^{n+1}_{i-1,j,k}$, $\phi^{n+1}_{i,j-1,k}$, $\phi^{n+1}_{i,j,k-1}$, $\phi^n_{i+1,j,k}$, $\phi^n_{i,j+1,k}$ and $\phi^n_{i,j,k+1}$. This allows Equation (3.11) to be rewritten to Equation (3.26).

$$\begin{aligned}
\frac{\phi^{n+1}_{i,j,k} - \phi^n_{i,j,k}}{\Delta t} = &\delta_h(\phi^n_{i,j,k}) \cdot \\
&\Big[\rho^x_{i,j,k} \phi^n_{i+1,j,k} + \rho^x_{i-1,j,k} \phi^{n+1}_{i-1,j,k} + \\
&\rho^y_{i,j,k} \phi^n_{i,j+1,k} + \rho^y_{i,j-1,k} \phi^{n+1}_{i,j-1,k} + \\
&\rho^z_{i,j,k} \phi^n_{i,j,k+1} + \rho^z_{i,j,k-1} \phi^{n+1}_{i,j,k-1} - \\
&(\rho^x_{i,j,k} + \rho^x_{i-1,j,k} + \rho^y_{i,j,k} + \rho^y_{i,j-1,k} + \rho^z_{i,j,k} + \rho^z_{i,j,k-1}) \phi^{n+1}_{i,j,k} \\
&-\nu - \lambda_1 (v_{0,i,j,k} - c_1(\phi^n))^2 + \lambda_2 (v_{0,i,j,k} - c_2(\phi^n))^2 \Big]
\end{aligned} \tag{3.26}$$

Applying the same step from Subsection 3.1.2.5 allows Equation (3.12) to be rewritten to Equation (3.27).

$$
\begin{aligned}
\phi_{i,j,k}^{n+1} =& (1-\omega)\phi_{i,j,k}^{n} + \\
& \omega \cdot \Big[\phi_{i,j,k}^{n} + \Delta t \delta_h(\phi_{i,j,k}^{n}) \cdot \\
& \Big(\rho_{i,j,k}^{x}\phi_{i+1,j,k}^{n} + \rho_{i-1,j,k}^{x}\phi_{i-1,j,k}^{n+1} + \\
& \rho_{i,j,k}^{y}\phi_{i,j+1,k}^{n} + \rho_{i,j-1,k}^{y}\phi_{i,j-1,k}^{n+1} + \\
& \rho_{i,j,k}^{z}\phi_{i,j,k+1}^{n} + \rho_{i,j,k-1}^{z}\phi_{i,j,k-1}^{n+1} + \\
& -\nu - \lambda_1(v_{0,i,j,k} - c_1(\phi^n))^2 + \lambda_2(v_{0,i,j,k} - c_2(\phi^n))^2\Big)\Big] \\
& / \Big[1 + \Delta t \delta_h(\phi_{i,j,k}^{n})\Big(\rho_{i,j,k}^{x} + \rho_{i-1,j,k}^{x} + \rho_{i,j,k}^{y} + \rho_{i,j-1,k}^{y} + \rho_{i,j,k}^{z} + \rho_{i,j,k-1}^{z}\Big)\Big]
\end{aligned}
\tag{3.27}
$$

3.2.2 Implementation

This section discusses the differences between the sequential implementation of the algorithm for 2D and the implementation for 3D.

3.2.2.1 Setup

The setup is slightly different compared to the setup as described for 2D in Subsection 3.1.4. Input and output arguments no longer point to images, but instead they point to volumes. Also, the penalties for the length of the contour and the size of the area inside the contour, become penalties for the surface area of the surface and the volume inside the surface.

Let v_0 define a volume consisting of n_z grayscale images, with n_x and n_y the width and height respectively. n_z could also be referred to as the depth. Empty borders, $b_x = b_y = b_z = 1$ are added in all dimensions to prevent out of bound accesses. n_{xx}, n_{yy} and n_{zz} are defined as the dimensions of the volume including the borders. The five-point stencil becomes a seven-point stencil, as can be seen in Equation (3.28), for a grid point (x, y, z) with space step $h = 1$. The volumetric grid with the borders is referred to as Γ_1 whereas the grid without borders is referred to as γ_1.

$$
\{(x,y,z),(x-h,y,z),(x+h,y,z),(x,y-h,z),(x,y+h,z),(x,y,z-h),(x,y,z+h)\} \tag{3.28}
$$

The data objects, as described in Table 3.2, increase in size as can be seen in Table 3.3.

v_0, ϕ and ϕ_{old} need to store the extra dimension. The finite differences, stored in ψ, requires an extra dimension for the volume, but also another dimension for the extra finite difference in the z direction. ρ requires two additional dimensions for a seven-point stencil, one to compensate for extra dimension of the volume, and the other to compensate for the added weights in the z direction.

3.2.2.2 Initialization

The initialization of ϕ with a circle with radius r, as described for 2D in Equation (3.14), is updated to initialize ϕ according to Equation (3.29). Here $z_c = n_{zz}/2$.

Object	Type	Dimensions
v_0	`uchar`	$n_{xx} \times n_{yy} \times n_{zz}$
ϕ	`float`	$n_{xx} \times n_{yy} \times n_{zz}$
ϕ_{old}	`float`	$n_{xx} \times n_{yy} \times n_{zz}$
ψ	`float`	$n_{xx} \times n_{yy} \times n_{zz} \times 3$
ρ	`float`	$n_{xx} \times n_{yy} \times n_{zz} \times 7$

Table 3.3: Objects in memory used in the implementation for 3D, with their types and sizes

$$\phi_{i,j,k} = -\left(\frac{\sqrt{2^{i-b_x-x_c} + 2^{j-b_y-y_c} + 2^{k-b_z-z_c}}}{r} - 1\right) \tag{3.29}$$

Again, the radius and origin coordinates can be changed by input arguments.

The checkerboard initialization as given in Equation (3.15) is updated as can be seen in Equation (3.30).

$$\phi_{i,j,k} = sin\left(\frac{i\pi}{5}\right) sin\left(\frac{j\pi}{5}\right) sin\left(\frac{k\pi}{5}\right) \tag{3.30}$$

3.2.2.3 Main loop

In the main loop, the computations are updated to reflect the changes in the model. Equation (3.31) shows the Neumann boundary conditions which are applied.

$$\begin{aligned} \phi_{0,j,k} &= \phi_{1,j,k} \\ \phi_{n_{xx},j,k} &= \phi_{n_{xx}-1,j,k} \\ \phi_{i,0,k} &= \phi_{i,1,k} \\ \phi_{i,n_{yy},k} &= \phi_{i,n_{yy}-1,k} \\ \phi_{i,j,0} &= \phi_{i,j,1} \\ \phi_{i,j,n_{zz}} &= \phi_{i,j,n_{zz}-1} \end{aligned} \tag{3.31}$$

Here the subscripts indicate iterations over all elements in γ_1 in the specific dimension.

The volume averages are computed according to Equation (3.32).

$$
\begin{aligned}
c_1 &= \frac{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} \sum_{k=b_z}^{n_z+b_z} H_\epsilon(\phi_{i,j,k}) \cdot v_{0_{i,j,k}}}{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} \sum_{k=b_z}^{n_z+b_z} H_\epsilon(\phi_{i,j,k})} \\
c_2 &= \frac{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} \sum_{k=b_z}^{n_z+b_z} (1 - H_\epsilon(\phi_{i,j,k})) \cdot v_{0_{i,j,k}}}{\sum_{i=b_x}^{n_x+b_x} \sum_{j=b_y}^{n_y+b_y} \sum_{k=b_z}^{n_z+b_z} (1 - H_\epsilon(\phi_{i,j,k}))}
\end{aligned}
\tag{3.32}
$$

For the finite differences, the same method as described in Section 3.1.6.4 is applied to save memory by only storing the forward finite differences. The formulation is given in Equation (3.33).

$$
\begin{aligned}
\Delta_+^x \phi_{i,j,k} &= \Delta_-^x \phi_{i+1,j,k} \\
\Delta_-^x \phi_{i,j,k} &= \Delta_+^x \phi_{i-1,j,k} \\
\Delta_+^y \phi_{i,j,k} &= \Delta_-^y \phi_{i,j+1,k} \\
\Delta_-^y \phi_{i,j,k} &= \Delta_+^y \phi_{i,j-1,k} \\
\Delta_+^z \phi_{i,j,k} &= \Delta_-^z \phi_{i,j,k+1} \\
\Delta_-^z \phi_{i,j,k} &= \Delta_+^z \phi_{i,j,k-1}
\end{aligned}
\tag{3.33}
$$

The updated equation for the weights, as given in Equation (3.24), is used to compute the weights for the 3D implementation. Again, SOR is used to solve the PDE. An extra loop is added to loop over the third dimension from the top to the bottom. The segmentation is obtained by evaluation of ϕ in γ_1, again setting pixel values to 0 for all $\phi < 0$ and pixels values to 1 for all $\phi \geq 0$.

3.3 Parallelization opportunities

This section investigates parallelization opportunities for the given sequential implementation for the 2D segmentation using the Chan-Vese model. The 2D model is considered here since updating it to support 3D is trivial at this point. This section is organized as follows. First some profiling results are investigated to spot parallelization opportunities. Next, different acceleration platforms are discussed and considered, after which an accelerated implementation is discussed in detail.

Computation steps from the algorithm are now referred to as kernels. The kernels as listed in Table 3.4 are distinguished throughout this analysis.

3.3.1 Profiling

Profiling results of the implementation as described for 2D, and executed for 4 different setups, are shown in Figure 3.5. The 4 different setups are as follows.

Kernel	Reference
Initialization	`init`
Copy	`copy`
Boundary conditions	`boundaries`
Region averages	`averages`
Finite differences	`differences`
Weights	`weights`
Solve the PDE	`sor`
Generate output	`output`

Table 3.4: The different kernels and their reference considered in the analysis for parallelization.

1. Input image size: 1000×1000. 100 SOR iterations.
2. Input image size: 1000×1000. 10 SOR iterations.
3. Input image size: 500×500. 100 SOR iterations.
4. Input image size: 500×500. 10 SOR iterations.

The initialization and output kernels are missing from the profiling results, since they are outside the main loop.

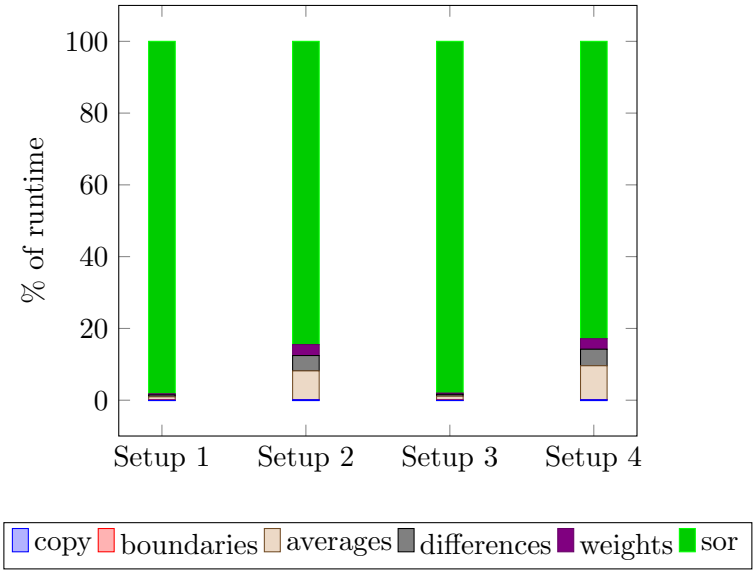


Figure 3.5: Profiling results for different input image sizes and different number of SOR iterations.

It can clearly be seen that the most compute intensive kernel is the one which solves the PDE i.e. `sor`. For 100 SOR iterations the contribution of the `sor` kernel is 98% of the total runtime for both the 500×500 and 1000×1000 images. Even if the number of

iterations is reduced by an order of magnitude, the `sor` kernel still contributes to over 80% of the runtime. For the reduced number of SOR iterations, it becomes visible that the `averages` (10%) and the `weights` (5%) are computationally more intensive than for example the `copy` (0.1%) and the `boundaries` (0.001%) kernels.

3.3.2 Kernels

This subsection discusses parallelization opportunities by investigation of data dependencies on the kernel level. However, first parallelization opportunities at the algorithm level are investigated.

The complexity of the kernels is analyzed using the Parallel random-access machine (PRAM) model and the Work-Time (WT) paradigm. For simplicity $N = n_{xx} = n_{yy}$ and $M = n_x = n_y$.

3.3.2.1 Algorithm step concurrency

Before investigating the individual kernels, the different steps of the algorithm are investigated for parallelization opportunities.

It is evident that the initialization kernel, `init`, can not be overlapped with the execution of another step. All computation in the main loop depend on the values of ϕ set by this step.

The copy kernel, `copy`, prepares a copy of ϕ for evaluation in the `sor` kernel. However, since the boundary condition, applied in the `boundaries` kernel, modify the borders of this copy for evaluation in the finite differences kernel, `differences`, the kernel can not be executed in parallel with these kernels.

It can be concluded that the boundary conditions kernel, `boundaries`, requires the results of the copy kernel, so these two kernels can not run in parallel.

The next kernel is region averages kernel, `averages`. This kernel does not depend on either the copy or boundary condition kernel. Therefor, it can be run concurrent with these two kernels.

As stated before, the finite differences kernel, depends on the copy and boundary condition.

The weights kernel, `weights`, requires the results of the finite differences kernel, so it can not run in parallel with any of the other kernels.

The SOR kernel, `sor`, requires the results of the weights kernel and the region averages kernel, therefor it can not run in parallel with any of the other kernels.

The kernel which generates the binary output image, `output`, requires the final result of ϕ, which is computed in the main loop. Therefor it can not be run concurrently with any other kernels.

The above is summarized in Figure 3.6, where it can be seen that the execution of the region average kernel can be overlapped with the execution of the copy, boundary conditions, finite differences and weights kernel. Barriers indicate a synchronization point and arrows indicate a kernel launch.

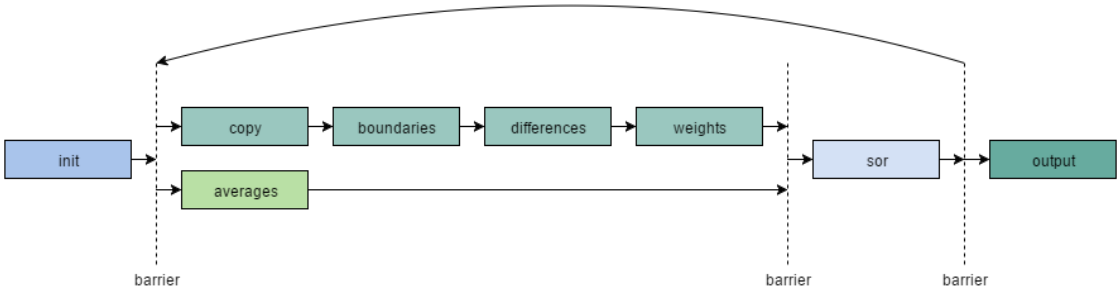


Figure 3.6: Algorithm step concurrency.

3.3.2.2 Initialization

The initialization kernel computes a single value for each element in ϕ, excluding the borders, according to Equation (3.14). This requires $O(M^2)$ work. All these computation are independent which means that in theory all values can be computed in parallel. This results in a time complexity of $O(1)$ for the parallel kernel.

3.3.2.3 Copy

In this kernel all elements of ϕ are copied to ϕ_{old}. This requires $O(N^2)$ work. All these operations are independent which means that they could in theory all be executed in parallel. This results in a time complexity of $O(1)$ for the parallel copy. In theory, only the values of ϕ excluding the border have to be copied, since the border values are set by the boundary condition later on. This reduces the work to $O(M^2)$.

3.3.2.4 Boundary conditions

This kernel copies values at the boundary of the ϕ. This requires $W = 2M = O(M)$ work. All these operations are independent, which allows parallel computation of all operations, resulting in a time complexity of $O(1)$ for the parallel kernel.

3.3.2.5 Region averages

Computation of c_1 and c_2 are independent. However, this kernel is a bit more complicated in terms of parallelization opportunities because the computation requires a reduction sum. The total number of operations is $W = 2M^2 = O(M^2)$. Summation of multiple elements in parallel can be solved with a reduction tree with a time complexity of $O(log(M))$.

3.3.2.6 Finite differences

The finite differences kernel computes two values per grid point resulting in $W = 2M^2 = O(M^2)$. Again there are no dependencies between the computations per grid point resulting in a time complexity for a parallel kernel of $O(1)$.

3.3.2.7 Weights

The weights kernel computes 5 values for each grid point resulting in $W = 5M^2 = O(M^2)$. The computations are independent within the grid, resulting in a time complexity of $O(1)$ for the parallel kernel.

3.3.2.8 Solve the PDE

Parallelization of the SOR kernel is essential in terms of acceleration opportunities due to the high contribution of this kernel to the runtime, however it is also the most complex. With K the total number of SOR iterations steps, the amount of work is $W = KM^2 = O(M^2)$.

Discretization in time for ϕ in the sequential implementation is based on the sweep from top to bottom and left to right. This determined where to use new values of ϕ i.e. ϕ^{n+1}, and where to evaluate the values of ϕ from the previous iteration i.e. ϕ^n. In the sequential implementation, grid values above and to the left are evaluated at ϕ^{n+1} and all others at ϕ^n. If the values of ϕ were to be computed in parallel there is no control over which time step is evaluated when ϕ is accessed i.e. either n or $n+1$. This however is essential for the convergence to the solution [25]. This is a common problem when solving PDE numerically. In general SOR is known to be bad for parallelization[30] when compared to the Jacobi method for example[31]. However, there are ways to exploit parallelism in this iterative method [32][33][34][35][36][37][38][39][40].

By using a Red-Black ordering the kernel can be parallelized. This results in $K = O(1)$ as the time complexity of the parallel kernel. The implementation is discussed later in this chapter.

3.3.2.9 Generate output

This kernel evaluates ϕ at each pixel, and set the value of the output image to either 0 for $\phi < 0$, or to 1 for $\phi > 0$. There are no dependencies which allows for parallel computation of all elements in the grid. This results in $W = M^2$ and a time complexity of $O(1)$.

3.3.3 Overview

This subsection lists a summary of the discussed opportunities for parallelization. The two different types of parallelization investigated are parallelization by launching kernels in parallel, and parallelization within the kernels.

Figure 3.6 shows that the region averages kernel can be launched in parallel, within the main loop, with the following kernels: `copy`, `boundaries`, `differences` and `weights`. However, it is not expected to have a big impact on the speedup, since the profiling results, as given in Figure 3.5, show little contribution to the total runtime by this kernel.

Most kernels have great potential for parallelization, since in those kernels the computations are independent. The results in terms of the WT paradigm for the PRAM model are shown in Table 3.5.

Kernel	Work	Time
`init`	$O(M^2)$	$O(1)$
`copy`	$O(N^2)$	$O(1)$
`boundaries`	$O(M)$	$O(1)$
`averages`	$O(M^2)$	$O(log(M))$
`differences`	$O(M^2)$	$O(1)$
`weights`	$O(M^2)$	$O(1)$
`sor`	$O(M^2)$	$O(1)$
`output`	$O(M^2)$	$O(1)$

Table 3.5: Parallelization opportunities per kernel in terms of the WT paradigm for the PRAM model.

3.4 Accelerated implementation

This section discusses all details regarding acceleration of the discussed implementation for 2D on an hardware acceleration platform. An accelerated implementation which runs on the CPU and uses OpenMP has also been implemented for both 2D and 3D. The performance of the different implementations for both 2D and 3D, hardware acceleration and CPU, are compared in Chapter 4.

3.4.1 Evaluation of acceleration platforms

This subsection compares and evaluates different acceleration platforms to be used to accelerate the discussed implementation, by combining parallelization of the kernels and the algorithm steps as described in the previous section. Two different acceleration platforms are considered. The first is an Field-programmable gate array (FPGA), and the second is a Graphics processing unit (GPU) for General-purpose computing on graphics processing units (GPGPU).

3.4.1.1 FPGAs

FPGAs are integrated circuits which can be configured to implement any arbitrary function. FPGAs are often used as accelerators to exploit parallelism in a certain kernel of an application. In most cases, the FPGA is configured to be filled with a large array of compute or process elements. Because of the flexibility of this platform, a lot of kernels can be implemented very efficiently in terms of energy consumption and utilization of available resources.

FPGAs have been successfully leveraged as accelerators for kernels of image analysis algorithms[41][42]. Also, with the introduction of Coherent Accelerator Processor Interface (CAPI)[43], FPGAs can now be connected with a cache coherent interface to the host processor running the host application. This makes movement and modification of data relatively simple and convenient, without great loss of performance.

From the analysis of the given implementation it can be learned that most of the computations in the kernels are dealing with floating-point numbers. FPGAs nowadays have some support for floating-point arithmetic, however the number of floating-point

units is still limited and performance gains compared to normal microprocessors come from massive parallelization rather than from the floating-point operations themselves, for which the microprocessor outperforms the FPGA[44]. It might be possible to convert some of the floating-point operations to fixed-point arithmetic to overcome this limitation. However, this is a very complex task, and well beyond the scope of this thesis.

3.4.1.2 GPUs

GPUs are integrated circuits designed to do computations and memory alterations related to the creation and output of images and video data. Their highly parallel structure makes them very effective at image processing tasks.

With the introduction of both Compute Unified Device Architecture (CUDA) by NVidia[45], and the open source OpenCL framework[46], GPUs became more accessible as generalized computation devices, allowing them to act as accelerators for all kinds of different computationally intensive tasks. This is also referred to as GPGPU.

CUDA, as mentioned before, is the parallel computing platform and application programming interface from NVidia. Software engineers can write device kernels for the GPU and invoke them with a certain configuration from their host application.

CUDA has been leveraged successfully to accelerate kernels for image analysis[47][48].

3.4.1.3 Overview

FPGAs and GPUs have both been leveraged to accelerate kernels in image analysis algorithm. Both acceleration platforms have their advantages and disadvantages when used in image processing[49]. There are also cases where both these platforms are combined for a image processing task[50][51].

Available host to device and on-device memory bandwidth is another important aspect to consider. Applications are either limited by the available bandwidth or by the available computation power. If the ratio of the number of floating point operations to the number of bytes required in the algorithm is bigger than the available bandwidth, the application is considered bandwidth limited, whereas if this ratio is smaller than the available bandwidth, the application is limited by the number of floating point operations.

Since the implementation used in this thesis for the Chan-Vese model requires a great number of floating-point operations, and can fully be mapped on the acceleration platform, the GPU is selected as the acceleration platform for the accelerated implementation described in this section.

From now on, device refers to the GPU.

3.4.2 Data movement and kernel invocation

Two types of parallelism were investigated in the previous section. Parallel invocation of kernels and parallelization of computations within kernels. Since the expected benefit from parallel kernel invocation is low, this is not implemented. In order to implement this, one could create two different CUDA streams, and launch the copy and region average kernel in parallel.

Data movement and allocation is explicit in CUDA. Movement of data between the host and GPU always introduce some overhead. Profiling results, given in Figure 3.5, show that the kernel which solves the PDE using SOR, is the most compute intensive. Initially one could choose to accelerate this kernel solely on the GPU, however, this is far from ideal, since the invocation of this kernel originates from within the main loop, which runs for a number of iterations. In this case the `sor` kernel computes a new ϕ, which then would have to be moved back to the host after the SOR iteration, and then from the host back to the device, after applying the Neumann boundary conditions. Another way to approach acceleration of the problem is to move all the kernels to the GPU for computation. The data movement would then be limited to the absolute minimum i.e. a copy of the source image and model parameters from host to device during initialization, and a copy of the resulting segmentation image after completion of the main loop from device to host. In this case all other data objects reside is the device's global memory, except for the source image. Since the source image is never altered it can be mapped to the GPU texture memory, which is a cached read only memory optimized for 2D spatial locality. Other data structures which are read only for other kernels are also bound to texture memory.

Considering all kernels have some parallelization opportunities, and ignoring overhead from data movement, there is a small theoretical benefit from accelerating these kernels on the GPU, even though their contribution to the compute time is small. Considering the data movement overhead, there is a large benefit from mapping all kernels on the GPU and therefor minimizing the data transfers between host and device.

Because the required number of iterations in the main loop might be large for certain segmentation problems, for this implementation all kernels are mapped to the GPU, including the initialization and output kernels.

The next design choice is related to the main loop and kernel invocation. Invoking GPU kernels can be done from either the host or from the device. CUDA allows definition of so called global and device functions. Global functions define kernels which can be launched from both the host and device. Global kernels are always launched with a certain configuration. CUDA kernels are launched in a grid, which can be 1-,2- or 3-dimensional[52]. The grid consists of blocks, which can also be 1-,2- or 3-dimensional. Each block has a number of threads. The configuration determines the number of blocks in the grid and the number of threads in a block. The syntax for a kernels launch is `kernel<<<dim3 blocks, dim3 threads>>>(function parameters)`. When a kernel is launched, blocks are assigned to multiprocessors by a scheduler. The blocks are then split into warps which consist of a number of threads that can run on the GPU, and handle part of a computation based on their coordinate within the launched grid. This scheduling and mapping introduces some overhead. Global kernels can also be launched from the device. This is referred to as dynamic parallelism[53]. CUDA device functions are kernels, which are the equivalent of normal functions on the host, and which can only be called from the device. There is little to no overhead from calling these device functions. This brings up the design choice and consideration for kernel invocations. However, first synchronization needs to be considered. CUDA has the ability to synchronize all threads within a block. However, there is no barrier for synchronization between blocks other than kernel invocations. One might use atom operations, however, this has turned out

to be cumbersome and not very efficient.

Considering kernel invocation overhead and synchronization limitations, two possible kernel invocation schemes were considered. The first invocation scheme uses dynamic parallelism, and launches a single kernel, with a single thread acting as the control thread, from the host. This kernel then launches all other kernels from the device using dynamic parallelism. In this case the iteration counter for the main loop resides on the device. There is no interaction with the host, other than the initial kernel launch and data movements for initialization and output collection. With the second invocation scheme, the iteration counter for the main loop resides on the host. Each iteration, all the different kernels are launched by the host. Two main benefits of this scheme are the implicit synchronization between kernel launches, and the possibility to change configurations per kernel launch. In this scheme the initialization and output kernels are also launched separately from the host, before and after the main loop kernel invocation respectively. Unfortunately, the first scheme does not benefit from the possible advantages of dynamic parallelism, since there is no change in granularity, nested parallelism or dynamic work generation. An other scheme in which the main loop resides on the device is not considered because there is no good option for synchronization across blocks.

This results in the second invocation scheme to be used in this implementation.

More details regarding data movement and kernel invocation configurations are discussed in the next subsection, which discusses details per CUDA kernel.

3.4.3 CUDA kernels

This subsection discusses details regarding the implementation of the CUDA kernels. Each kernel is discussed separately.

3.4.3.1 Setup

In order to be able to use the GPU for acceleration of the given implementation, some memory needs to be allocated on the device for storage of data objects used in the implementation, and some data needs to be moved from the host to the device. First memory for the data objects, as listed in Table 3.2, is allocated on the device. Then the source image is copied to the device and bound to device texture memory. Other data objects are also bound to device texture memory.

Function parameters can be passed with each kernel launch, but they can also be stored in the constant memory of the device. This sounds like a good idea, however it turns out to decrease the performance[54]. Therefor, this implementation passes the required function parameters with each kernel launch.

During the setup, a launch configuration variable is set based on the image size. In this implementation most kernels are launched with a thread per pixel. This does not always ensures the best performance, however it makes building the kernels extremely easy. In this implementation the blocks are launched in a 2D grid. The number of required blocks in the grid depends on the number of threads per block. The number of threads per block is limited by the GPU architecture. In this case 1024 is the maximum number of threads per block. Launching this number of threads in a 2D configuration results in a 32×32 grid of threads per block, because $\sqrt{1024} = 32$. The dimensions of the

grid of blocks are then determined according to Equation (3.34). One could also launch the kernel with a different number of threads per block, however in order to maximize the benefit from the L2 cache for reads from global memory, this method results in the smallest number of halo cell reads. Comparison with other block dimensions showed worse performance for configurations other than the 32×32 configuration.

$$\begin{aligned} \text{threadsPerBlock}_x &= 32 \\ \text{threadsPerBlock}_y &= 32 \\ \text{numBlocks}_x &= ceil(\frac{n_{xx}}{\text{threadsPerBlock}_x}) \\ \text{numBlocks}_y &= ceil(\frac{n_{yy}}{\text{threadsPerBlock}_y}) \end{aligned} \tag{3.34}$$

This configuration allows to determine the index and coordinates of a thread according to Listing 1.

```
const int x = blockIdx.x * blockDim.x + threadIdx.x;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
const int i = y * nxx + x;
```

Listing 1: Determine coordinates and index within a CUDA thread.

For some kernels, threads might be launched for grid coordinates which require no computation. For example if there are no computations for the threads mapped on the borders of the image or outside of the image due to the *ceil* function, the thread can be terminated according to Listing 2 or Listing 3, in order to check if the thread is mapped on the border or outside the image respectively.

```
const int x = blockIdx.x * blockDim.x + threadIdx.x;
if (x < bx || x >= nx + bx) return;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
if (y < by || y >= ny + by) return;
const int i = y * nxx + x;
```

Listing 2: Thread termination for threads mapped on the border.

```
const int x = blockIdx.x * blockDim.x + threadIdx.x;
if (x >= nxx) return;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
if (y >= nyy) return;
const int i = y * nxx + x;
```

Listing 3: Thread termination for threads mapped outside the image.

3.4.3.2 Initialization

The initialization kernel sets all values of ϕ according to Equation (3.14) or Equation (3.15). Based on the input arguments the correct kernel is selected to initialize ϕ.

The implementation is very straightforward for both possibilities. Since border pixels are not set, threads outside the grid or on the border are terminated.

The function parameters are a pointer to the level set description ϕ and information about the dimensions of the source image. For the circle initialization method, the radius and origin coordinate are also passed as arguments.

Each thread writes a single value to the global memory of the device.

3.4.3.3 Copy

The copy kernel is not implemented as a global CUDA function. Instead, the `cudaMemcpy` function is used with `cudaMemcpyDeviceToDevice` as the type of transfer.

This step reads and writes N^2 values from and to global device memory.

3.4.3.4 Enforce Neumann boundary condition

The `boundaries` kernel is launched with a different configuration. Since it only reads and writes $2M$ values from global memory it is a bad idea to launch N^2 or more threads. Instead the kernel is splitted in to a kernel which copies the rows and a kernel which copies the columns. For the rows, the number of threads in a block in the y-direction is set to 2, one for the top row and one for the bottom row. For the column copy kernel the launch configuration it is the other way around.

Equation (3.16) is applied in the column copy kernel and Equation (3.17) is applied in the row copy kernel.

Each thread read and writes a single value to the global device memory.

3.4.3.5 Region averages

The region averages kernel is a bit more interesting in terms of parallelization. The reduction from [55] is used to reduce the values within blocks. The kernel makes use of shared memory on the device. The shared memory on the device is shared between different treads within a block. Each block uses $4 \cdot 1024 \cdot 4 = 16384$ bytes of shared memory. 4 is the amount of values that are reduced i.e. the numerator and denominator for both c_1 and c_2 according to Equation (3.18). 1024 is the number of threads per block. And the other 4 is the number of bytes per value. Since single precision-floating-point numbers are used, this is 32 bits per value or 4 bytes. Each thread computes part of the sum of Equation (3.18). Then threads within the block are synchronized using `__syncthreads()`, and the results are reduced and stored in global device memory. Then a second kernel is launched to reduce the summation results from the different blocks in the grid. A second kernel launch is needed since there is no good way to synchronize across blocks as mentioned before. This kernel is launched for a single thread which collects, adds and divides the results from all the different blocks to compute c_1 and c_2, which are then stored in the device's global memory.

Each thread in the average kernel fetches both the grayscale value of the image and the value of ϕ from texture memory.

The reduce kernel writes the averages c_1 and c_2 to the device's global memory.

3.4.3.6 Finite differences

Implementation of the CUDA kernel for the finite differences kernel is straightforward. Each thread computes the values for the forward finite differences from Equation (3.19) and stores them in global device memory.

Each threads fetches the required values of ϕ from the texture memory and stores the finite differences in the global device memory.

3.4.3.7 Weights

The CUDA kernel which computes the weights is reads finites differences from global device memory and writes the weights to the device memory. Each thread computes the different weights from Equation (3.9).

Each threads reads 5 elements from global memory and writes 5 values to global memory.

3.4.3.8 Solve the PDE

The most interesting kernel for acceleration using CUDA is the `sor` kernel. Since the problems mentioned before, a straightforward implementation would not work in this case. Instead the red-black ordering is used to update ϕ in the iterations[56]. In the red-black ordering the grid is split in red and black cells. The color of a cell is determined by $(x + y)\ mod\ 2$. If $x + y$ is even cells are colored red, whereas if $x + y$ is odd the cells are colored black. This results in a checkerboard of red and black cells as shown in Figure 3.7.

Since a five-point stencil is used, red cells solely depend on values of black cells, and black cells solely depend on values of red cells. This allows for concurrency when updating cells of the same color. In the implementation the SOR iteration counter resides on the host. Each iteration two CUDA kernels are launched after each other. One kernel which updates all the red cells, and one which updates all the black cells, both according to Equation (3.12), with the difference that all the accessed cells of ϕ can be considered ϕ^{n+1}, instead of using ϕ^{n+1} for the upper cell and the cell to the left in the sequential sweep. Because the kernels are launched from the host, there is an implicit barrier between the two invocations.

The kernel launches half the number of blocks in the x-direction since only half the cells in a row are updated in a single kernel launch. The indexing is determined by setting an offset to either 0 or 1 and add this to the x-coordinate of the thread. Then for the index the result of $y\ mod\ 2$ is added to generate the checkerboard read pattern for the rows. This can be seen in Listing 4.

The kernel fetches grayscale source image values and values of ϕ and the weights from texture memory. Each thread writes a single value of ϕ^{n+1} to device global memory.

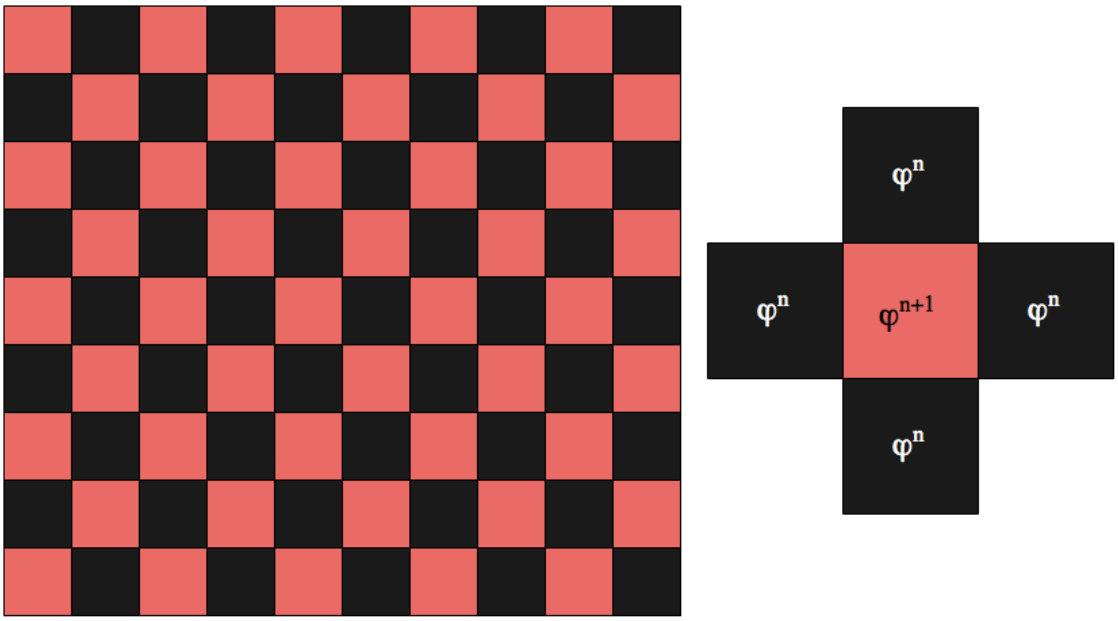


Figure 3.7: Red-black ordering.

```
const int x = ((blockIdx.x * blockDim.x + threadIdx.x) << 1) + offset;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
const int i = y * nxx + x + (y & 1);
```

Listing 4: Determine coordinates and index within the `sor` kernel.

3.4.3.9 Generate output

The CUDA kernel which implements the generation of the binary output image is straightforward. Each thread sets the value at its index to either 1 or 0 based on ϕ. After the generation of the output image, the image is copied from the device back to the host.

Finally, all allocated device memory gets freed and texture memory gets unbounded.

3.4.3.10 Acceleration of the 3D implementation

In the accelerated implementation for 3D the CUDA kernels are launched in a 3D grid of 2D blocks. The resulting index is determined as can be seen in Listing 5.

For the `sor` kernel, the coordinates and index are computed according to Listing 6.

```
const int x = blockIdx.x * blockDim.x + threadIdx.x;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
const int z = blockIdx.z;
const int i = z * nyy * nxx + y * nxx + x;
```

Listing 5: Determine coordinates and index within a CUDA thread for the 3D implementation.

```
const int x = ((blockIdx.x * blockDim.x + threadIdx.x) << 1) + offset;
const int y = blockIdx.y * blockDim.y + threadIdx.y;
const int z = blockIdx.z;
const int i = z * nxx * nyy + y * nxx + x + (y & 1) + (z & 1);
```

Listing 6: Determine coordinates and index within the SOR kernel for the 3D implementation.

4 Results

4.1 Setup

Two different systems were used to analyze the performance of the implementations for both 2D and 3D of the Chan-Vese model, as described in Chapter 3. Details on both systems are listed in Table 4.1.

System Name	IBM Power8 S824L	Similde
CPU		
Processor	IBM POWER8	Intel Xeon E5420
Number of processors	2	1
Number of cores per processor	10	8
Clock frequency	3.42 GHz	2.50 GHz
L2 cache	512 KB per core	3 MB per core
L3 cache	8 MB per core	
L4 cache	16 MB per DIMM	
Memory		
RAM	256 GB	32 GB
Memory bandwidth	230 GB/s	
GPU		
Device	K40	GTX 750 Ti
CUDA architecture	Kepler	Maxwell
Number of CUDA cores	2880	640
Base clock frequency	745 MHz	1.02 GHz
Boost clock frequency	810 MHz and 875 MHz	1.085 GHz
Memory	12 GB	2 GB
Memory bandwidth	288 GB/s	86.4 GB/s
CUDA compiler	V7.5.17	V7.0.27
OS		
Distribution	Ubuntu 15.04	Ubuntu 14.04.3 LTS
Kernel	Linux 3.19.0-28-generic ppc64le	Linux 2.6.32-24-server x86_64

Table 4.1: Comparison of the two different test systems.

The POWER8 system, which is used, is part of the FPGA Research Infrastructure Cloud (FAbRIC) project. This project, which is supported by grants from International Business Machines (IBM) and National Science Foundation (NSF), allows researchers access to powerful state of the art hardware, to explore the potential of these high performance systems and reconfigurable hardware, without having to invest in these relatively

expensive systems. Each POWER8 node in FAbRIC has an NVidia K40 Graphics processing unit (GPU) and two Coherent Accelerator Processor Interface (CAPI) accelerator cards installed. The FAbRIC is located in Texas Advanced Computing Center (TACC) at the University of Texas at Austin.

The system is configured with the Simultaneous multithreading (SMT) mode off. This technology allows multiple instruction streams, up to 8 per physical core, within the processor. Applications where the available chip's memory bandwidth is not fully utilized might see an increase in performance with SMT enabled. Investigation of the utilization of the available memory bandwidth for the OpenMP implementation running with 20 threads i.e. a single thread per physical core, shows up to 25% utilization of the available bandwidth for the weights kernel, and even less for the other kernels. This is well below the available memory bandwidth and suggests that there might be a performance increase from enabling the SMT mode on the POWER8. This however has not been tested due to the configuration on the FAbRIC nodes.

The Similde system is owned by the Computer Engineering laboratory of Delft University of Technology. The server has a consumer grade GPU installed, the NVidia GTX 750 Ti.

4.2 Segmentation of surfaces in 3D

The ability to quickly converge to a good set of parameters which allow the Chan-Vese model to segment tumors in CT-scan volumes has not been tested due to the lack of a good data set. The 3D implementation is tested with a generated data set and a small data set of slices of a healthy brain.

The generated volume contains a single solid box inside a $100 \times 100 \times 100$ volume. The evolvement of the surface can be seen in Figure 4.1.

For the slices of a healthy brain the isosurface shown in Figure 4.2a has been segmented using the implementation for 3D. The contours of different slices are plotted in Figure 4.2b. Some slices of the brain volume used in this example are also shown in Figure 4.2. The brain volume consists of 100 slices with a dimension of 256×256.

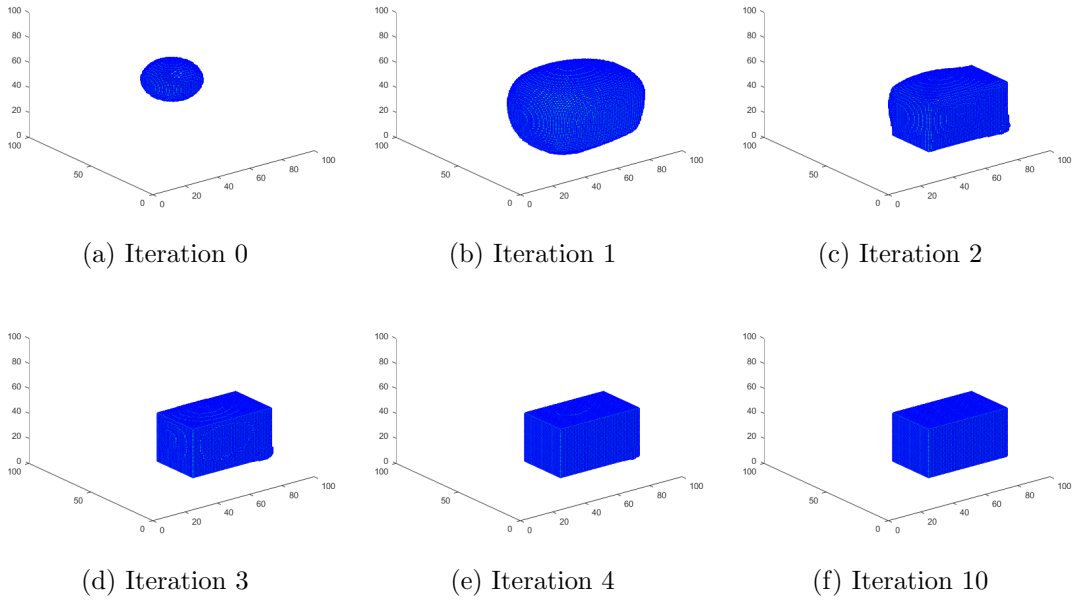
4.3 Performance

The performance of the accelerated implementations is evaluated on both implementation and kernel level. The following implementations are distinguished.

Name	Description
`seq`	Sequential implementation
`omp#`	OpenMP implementation with # the number of threads
`cuda`	Compute Unified Device Architecture (CUDA) implementation

Table 4.2: Different implementations.

In the following results, the IO is excluded from the results i.e. reading the image or volume and writing the result.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

(d) Iteration 3 (e) Iteration 4 (f) Iteration 10

Figure 4.1: Evolvement of surface segmenting a solid box in a generated data set. $\mu = 0.01 \cdot 255^2$, $\nu = 0$ and $\lambda_1 = \lambda_2 = 1$.

Speedup comparison is indicated by implementation1-implementation2 e.g. `seq-cuda` indicates the speedup between the sequential and CUDA implementation.

Figure 4.3 and Table 4.3 show the speedup of the CUDA implementation per kernel for the 2D implementation on both systems compared to sequential implementations. A 2048×2048 source image is used, and the number of steps is set to 10.

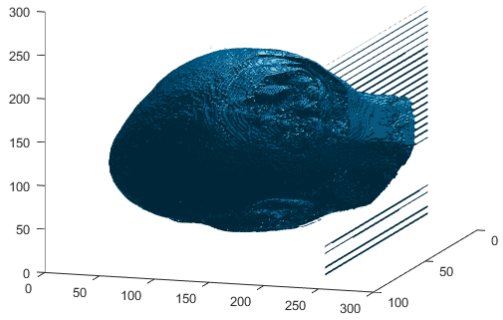
Kernel	p8-single	p8-double	similde-single	similde-double
`init`	160.7	116.5	136.0	18.5
`copy`	17.6	13.3	40.5	38.1
`boundaries`	1.1	3.4	3.3	5.4
`averages`	17.7	15.9	18.8	10.7
`differences`	165.7	123.0	21.9	22.4
`weights`	73.0	35.9	7.0	82.3
`sor`	58.8	24.7	33.6	20.1

Table 4.3: Speedup `seq-cuda` per kernel comparison for the 2D implementations.

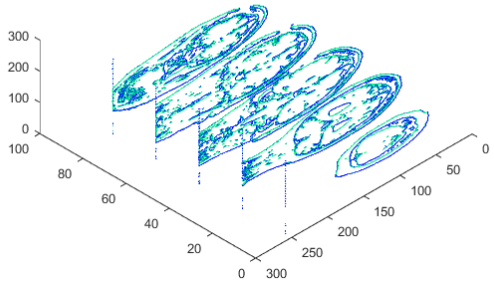
All CUDA kernels outperform their sequential CPU variants. It can be seen that the most important Successive over-relaxation (SOR) kernel has a speedup of over 56 for single precision, and over 24 for double precision.

Figure 4.4 and Table 4.4 show the execution times for the different 2D implementations. The same input image is used and the steps are set to 10.

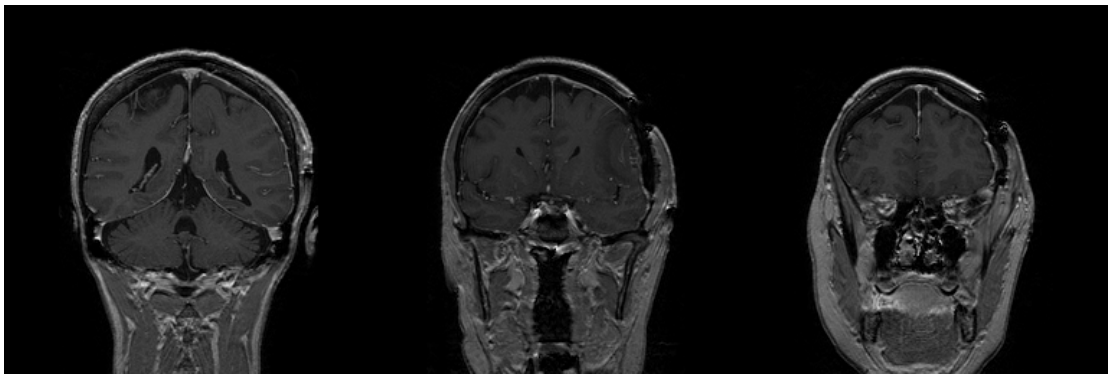
The CUDA implementation is able to iterate 10 steps of the main loop in little



(a) Isosurface of the segmentation of the healthy brain slices after 10 iterations.



(b) Plot of some contour slices from the segmented surface of the healthy brain slices.



(c) Slice 36 (d) Slice 65 (e) Slice 82

Figure 4.2: Segmentation results for 100 slices of a healthy brain. $\mu = 0.01 \cdot 255^2$, $\nu = 0$ and $\lambda_1 = \lambda_2 = 1$.

Implementation	p8-single	p8-double	similde-single	similde-double
seq	93.44s	85.35s	162.70s	174.31s
omp2	47.55s	41.35s	81.55s	98.22s
omp4	23.50s	21.05s	45.16s	79.85s
omp8	12.87s	10.98s	40.23s	78.77s
omp10	10.34s	8.84s		
omp16	6.33s	6.13s		
omp20	5.11s	5.57s		
cuda	1.64s	3.44s	4.97s	8.86s

Table 4.4: Execution times for the 2D implementations.

over 1.5 seconds for the input image with a size of 2048×2048. This is significantly more responsive than the sequential implementation with single precision taking over 90 seconds.

Figure 4.5 and Table 4.5 show the speedup of the different implementations for 2D.

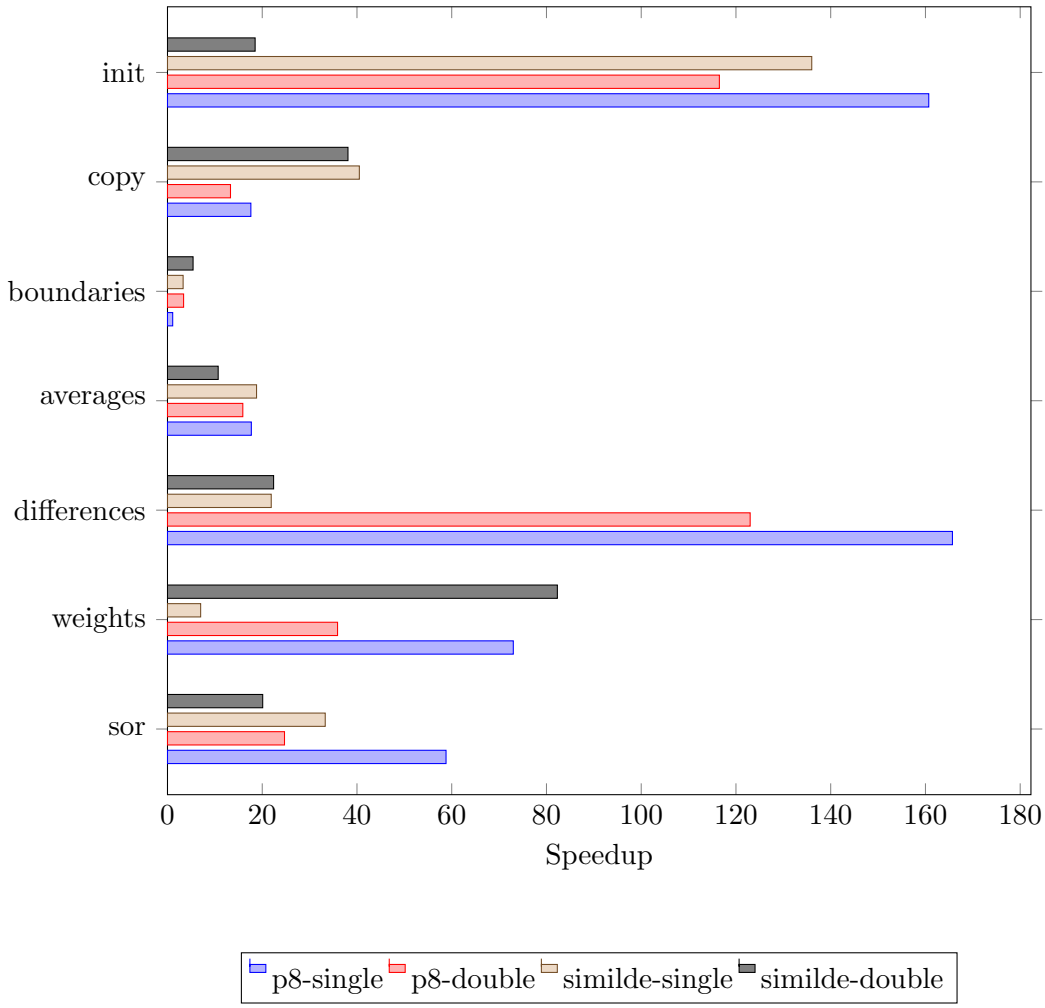


Figure 4.3: Speedup `seq-cuda` per kernel comparison for the 2D implementations.

The same 2048×2048 source image is used, and the number of steps is fixed to 10.

Implementation	p8-single	p8-double	similde-single	similde-double
seq-cuda	56.9	24.8	32.7	19.7
seq-omp2	2.0	2.1	2.0	1.8
seq-omp4	4.0	4.1	3.6	2.2
seq-omp8	7.3	7.8	4.0	2.2
seq-omp10	9.0	9.7		
seq-omp16	14.8	13.9		
seq-omp20	18.3	15.3		
omp8-cuda	7.8	3.2	8.1	8.9
omp20-cuda	3.1	1.6		

Table 4.5: Implementation speedup for the 2D implementations.

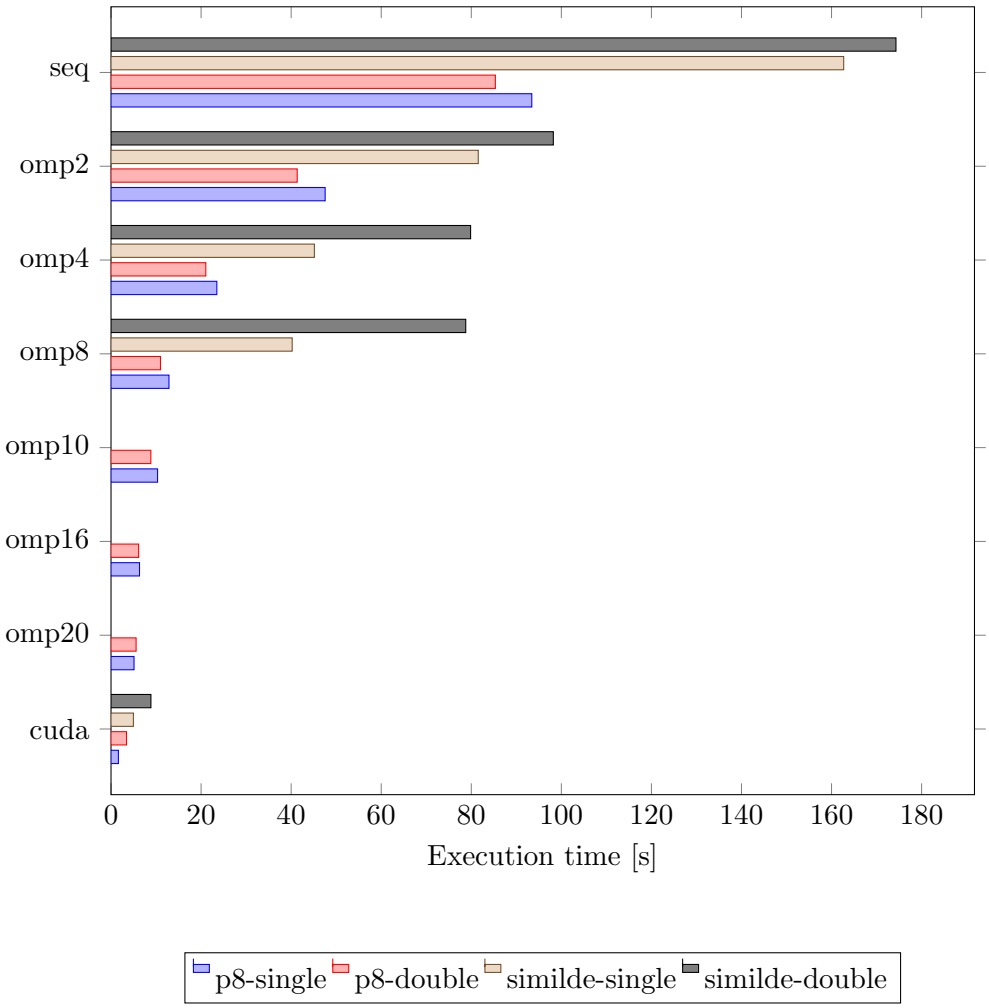


Figure 4.4: Execution times for the 2D implementations.

It can be seen that the OpenMP speedup scales almost linear on both systems. The speedup comparison between the OpenMP implementations running with 8 threads and the CUDA implementation shows a speedup of roughly 8 for single and double precision on the Similde system, whereas a speedup of roughly 8 and 4 is achieved for single and double precision on the POWER8 system. This however is compared with a POWER8 system with SMT mode off, and investigation of utilization of available memory bandwidth shows that the POWER8 should be able to reduce the advantage of the CUDA implementation by enabling SMT mode. However, there is also room for improvement in terms of coalesced memory accesses for the CUDA implementation, as can be seen from the efficient memory bandwidth utilization results below.

For 3D segmentation of large volumes it is important to note that the OpenMP implementation is the better one since the available amount of memory is limited on the K40 GPU to 12GB whereas the POWER8 system has 256GB of RAM.

Figure 4.6 and Table 4.6 show the speedup of the implementation for 3D on the

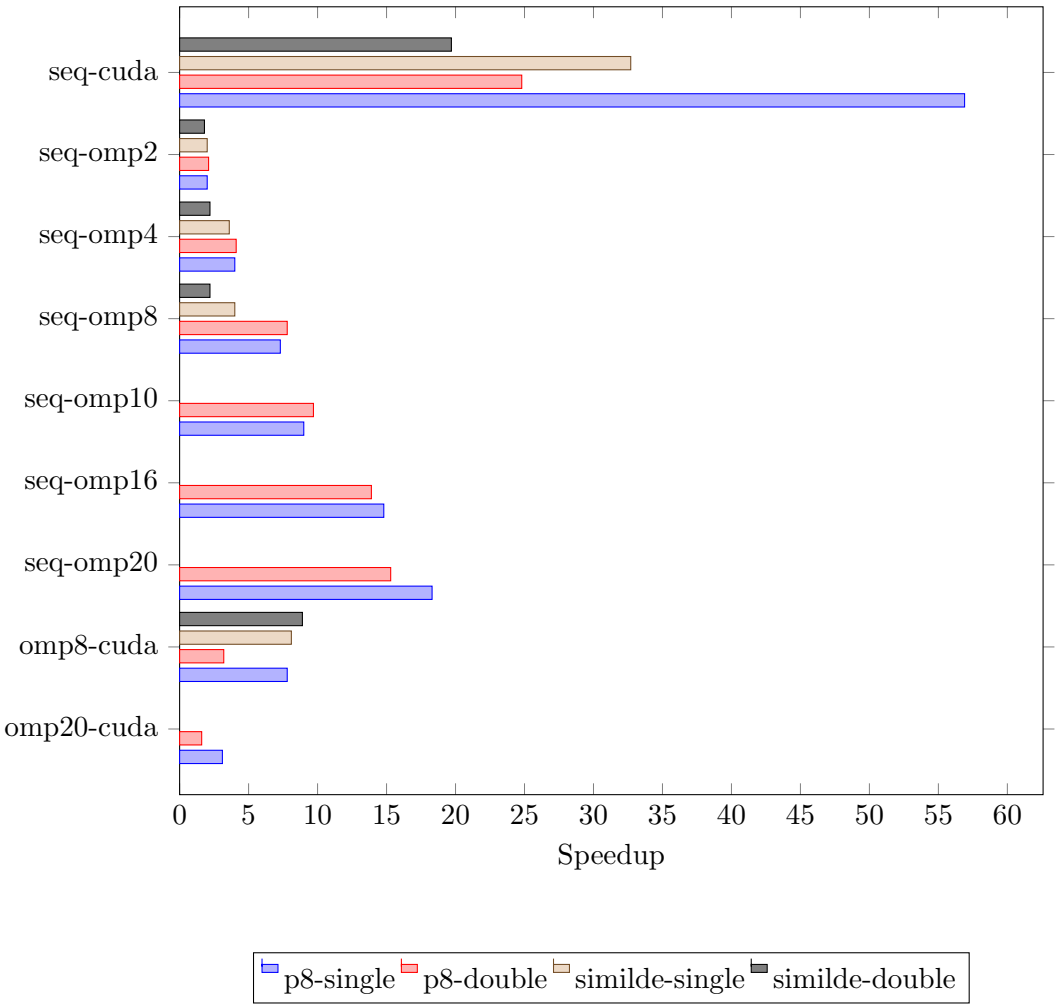


Figure 4.5: Implementation speedup for the 2D implementations.

POWER8 system for both single and double precision. An randomly generated $100 \times 100 \times 100$ is used with 10 iterative steps.

A speed up of 107 is achieved over the sequential implementation with single precision. Speedup for double precision is 48 when compared to the sequential implementation. Comparison with a 20 thread OpenMP implementation results in a speedup of 6 and 3 for single and double precision respectively.

Figure 4.7 and Table 4.7 show the efficient bandwidth for the 2D single precision implementation on POWER8. The efficient bandwidth is defined as the ratio of between the number of bytes read and written by a kernel over the execution time of the kernel in seconds.

By coalesced memory accesses to global device memory one can improve the efficient bandwidth utilization.

Other metrics for the performance of the kernels have also been investigated with the help of the NVidia profiler tools. These results show that there is still room for improve-

Implementation	p8-single	p8-double
seq-cuda	107.9	48.9
seq-omp2	2.0	1.9
seq-omp4	3.9	2.7
seq-omp8	7.6	6.4
seq-omp10	9.7	7.4
seq-omp16	13.2	8.7
seq-omp20	17.5	15.2
omp8-cuda	14.2	7.7
omp20-cuda	6.1	3.2

Table 4.6: Implementation speedup for the 3D implementations on the POWER8.

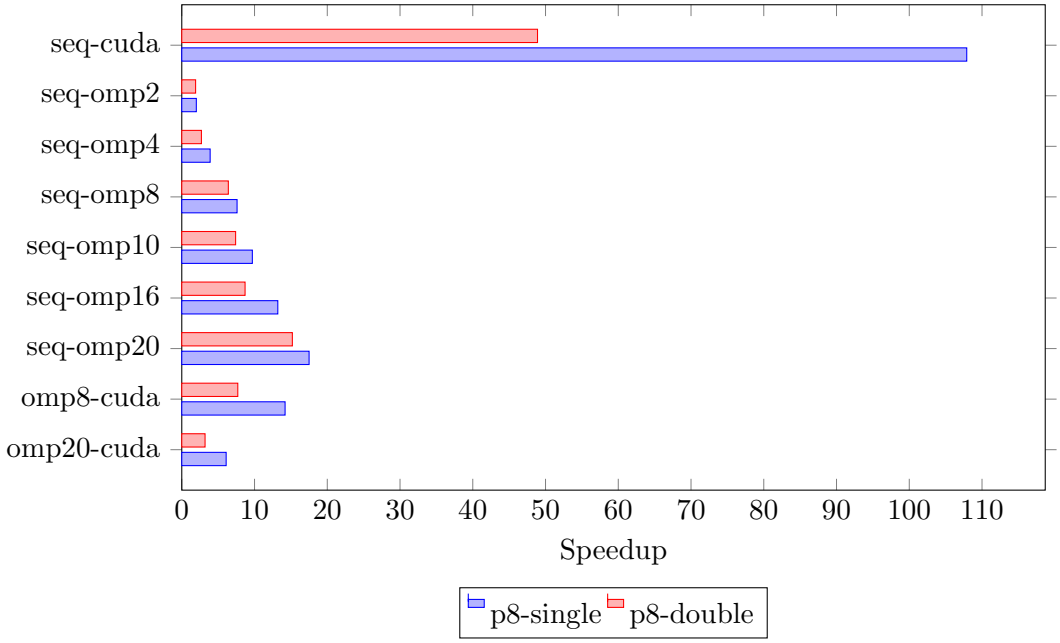


Figure 4.6: Implementation speedup for the 3D implementations on the POWER8.

Kernel	Efficient bandwidth [GB/s]
`init`	73.9
`copy`	164.4
`boundaries`	1.4
`averages`	4.0
`differences`	240.5
`weights`	236.5
`sor`	21.4

Table 4.7: Efficient bandwidth of the single precision implementation for 2D on the POWER8.

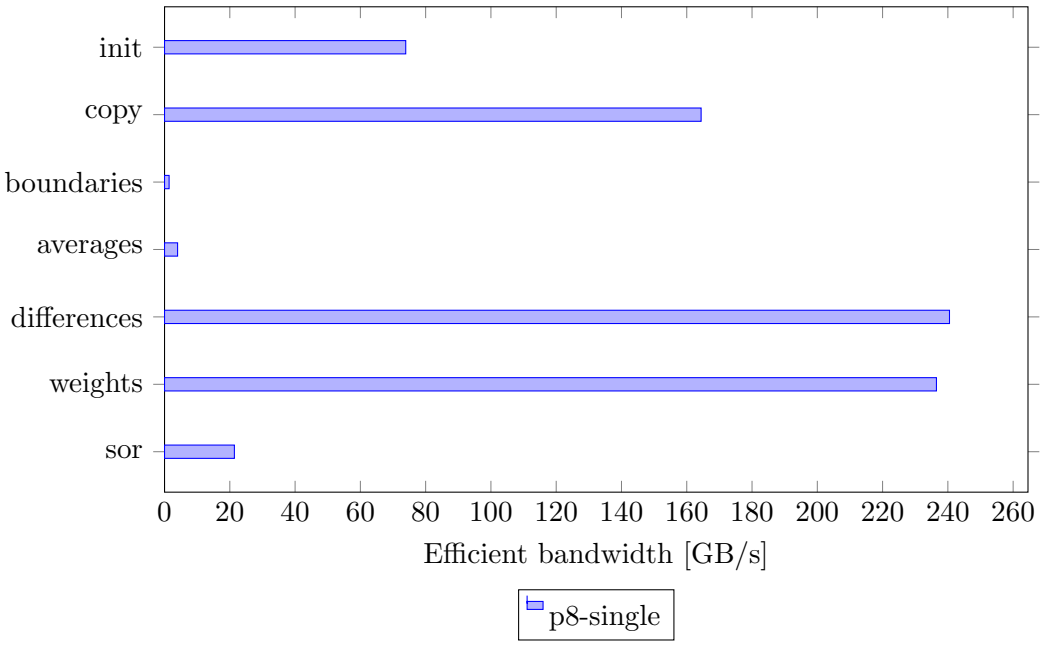


Figure 4.7: Efficient bandwidth of the single precision implementation for 2D on the POWER8.

ment in terms of efficient bandwidth utilization of the SOR kernel. Recommendations are given in the next chapter.

5 Conclusion

This chapter lists some conclusions and tries to answers the questions as stated in Chapter 1. It also lists a number of suggestions and recommendations for future work.

5.1 Conclusions

The following questions were stated in Chapter 1.

1. Can medical experts use the Chan-Vese model for segmentation to delineate tumors visible in a volume resulting from an X-ray computed tomography, in a semi-automatic fashion, where the results of the segmentation can be influenced by a set of parameters, to guide the model towards the required solution?

2. Is it possible to extend a given sequential implementation of a numerical approximation of the Chan-Vese model for two dimensional images, to support three dimensional volumes for segmentation of surfaces?

3. Can the implementation be accelerated without loss of accuracy, using special hardware which exploits parallelization, to streamline the experience for medical experts when model parameters need to be fine tuned?

This thesis provides the following answers, insights and solutions to these questions.

1. The usability of the Chan-Vese model in terms of ability to segment tumor visible in a volume has been evaluated in this thesis. From the results given in Chapter 4 it can be seen that the Chan-Vese model can successfully be used to delineate objects within a volume resulting from an X-ray computed tomography. The model is not fully automatic, but from Chapter 2 it can be learned that the available parameters of the Chan-Vese model give the user good control over the evolvement of the contour, and therefor the implementation of the model can be considered semi-automatic. The parameters allow the user to set penalties for the length of the contour, the area inside the contour and the uniformity of the pixel intensities in- and outside the contour. Besides, the shape and placement of the initial contour also allows to influence the result of the segmentation. These tools allow the user to guide the model in many different ways, allowing complex segmentation problems to be solved with the use of this model. The fact that the Chan-Vese model does not depend on a gradient to define edges makes it more versatile than other edge based segmentation methods. The ability to delineate tumors within a computed tomography scan of the brain has not been evaluated.

2. The given sequential implementation which uses Successive over-relaxation (SOR) to solve the Partial differential equation (PDE) can be extended to support three dimensional volumes for the segmentation of surfaces, as can be learned from Chapter 3. The steps to go from the mathematical description of the model for 2D to the discretization of the model for implementation with SOR to solve the PDE, are given together with the steps to take this discretization and implementation from 2D to 3D.

3. The given implementation was investigated for parallelization opportunities, and the results were shared in Chapter 3. The same chapter discusses details regarding a possible implementation on an Field-programmable gate array (FPGA) or a Graphics processing unit (GPU). GPUs were selected over FPGAs due to their natural capability to deal with image data and their superior floating-point units. The whole model was mapped onto the GPU to minimize the data transfers to an absolute minimum. Chapter 4 shows that utilization of the accelerated Compute Unified Device Architecture (CUDA) implementation on a POWER8 platform with a K40 GPU results in a speedup of 56 for 2D and 107 for 3D implementations with single precision, compared to sequential implementations, which is a significantly reduces the delay in the feedback loop, allowing medical experts to quickly converge to the right set of parameters for the given segmentation problem. The OpenMP implementation shows that the feedback loop is also reduced significantly for a multiple threaded implementation on the POWER8, which like the K40 GPU has a high memory bandwidth.

5.2 Future work

This section lists some suggestions and recommendations for future work.

- Test the implementation of the model with real medical data. Tests with real data could point to possible problems or areas of possible improvement.
- Integrate the ITK library to read medical data containers[57].
- Create a module for 3D Slicer to allow medical experts to interactively guide a contour towards a perfect segmentation of the surface of a tumor. More details regarding this suggestion can be found in Subsection 5.2.2.
- Investigate the usability and performance in terms of successful delineation of the Chan-Vese model for vector valued images.
- Experiment with different and additional terms in the energy functional of the Chan-Vese model to reflect certain aspects which are not influencing the evolvement in the original Chan-Vese model.
- Evaluate the performance on the POWER8 system with Simultaneous multithreading (SMT) enabled, and compare the limitations and difference in bottlenecks for CPU and GPU implementations.

5.2.1 Optimizations

There are several unimplemented optimizations that could be considered for the CUDA implementation to increase the performance and speedup over the sequential implementation. Below some possible optimizations are listed.

- For this implementation most kernels were launched with a single thread per pixel. This makes indexing extremely easy, however, it results in a lot of overhead due to the relatively large number of threads that need to be launched and scheduled. A possible optimization would be to change the launch configurations of the kernels in a way which would give more work to each thread and at the same time reducing the number of threads to be spawned with each kernel invocation.
- The efficiency of the `sor` kernel can be improved by storing the required data structures separate for the red and black grid[58]. This improves memory coalescing and the efficient memory bandwidth utilization.
- Leverage shared memory in the `sor` for the values of ϕ^n. Another potential optimization moves more computations to the `sor` kernel to reduce the number of global memory reads.
- Allow for problem size reduction during initialization to reduce the number of unwanted delineated objects. This would also improve the performance, and at the same time it causes for quicker parameters set convergence.

5.2.2 Integration with 3D Slicer

3D Slicer is an open source software package for medical image analysis and visualizations[59].

A module could be created for 3D Slicer which allows the user to interactively make use of the accelerated implementation of the Chan-Vese model to segment tumors in 3D CT-scans. Ideally the user would have the ability to place any initial contour and then start tuning the parameters to segment the object of interest. Because the accelerated implementation is used, the feedback loop should be relatively short and finding the right set of parameters is done in an interactive way where changes of parameters are reflected in terms of segmentation quickly. The segmentation results could be visualized nicely from within 3D Slicer.

Another important addition could be the ability to manually edit the contour to fix small issues which can not be fixed easily by changing the parameters or the initial contour.

Bibliography

[1] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986, ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851.

[2] T. Chan and L. Vese, "Active contours without edges," *Image Processing, IEEE Transactions on*, vol. 10, no. 2, pp. 266–277, Feb. 2001, ISSN: 1057-7149. DOI: 10.1109/83.902291.

[3] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations," *J. Comput. Phys.*, vol. 79, no. 1, pp. 12–49, Nov. 1988, ISSN: 0021-9991. DOI: 10.1016/0021-9991(88)90002-2. [Online]. Available: http://dx.doi.org/10.1016/0021-9991(88)90002-2.

[4] R. Tsai, S. Osher, *et al.*, "Review article: Level set methods and their applications in image science," *Communications in Mathematical Sciences*, vol. 1, no. 4, pp. 1–20, 2003.

[5] R. Malladi and J. A. Sethian, "Level set and fast marching methods in image processing and computer vision," in *Image Processing, 1996. Proceedings., International Conference on*, IEEE, vol. 1, 1996, pp. 489–492.

[6] V. Chalana, D. T. Linker, D. R. Haynor, and Y. Kim, "A multiple active contour model for cardiac boundary detection on echocardiographic sequences," *Medical Imaging, IEEE Transactions on*, vol. 15, no. 3, pp. 290–298, 1996.

[7] C. Xu and J. L. Prince, "Snakes, shapes, and gradient vector flow," *Image Processing, IEEE Transactions on*, vol. 7, no. 3, pp. 359–369, 1998.

[8] A. T. Brint and M. Brady, "Stereo matching of curves," *Image and vision computing*, vol. 8, no. 1, pp. 50–56, 1990.

[9] N. Paragios and R. Deriche, "Geodesic active contours and level sets for the detection and tracking of moving objects," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 3, pp. 266–280, 2000.

[10] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models," English, *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321–331, 1988, ISSN: 0920-5691. DOI: 10.1007/BF00133570. [Online]. Available: http://dx.doi.org/10.1007/BF00133570.

[11] A. P. Witkin, "Scale-space filtering," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'83, Karlsruhe, West Germany: Morgan Kaufmann Publishers Inc., 1983, pp. 1019–1022. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623516.1623607.

[12] A. Witkin, D. Terzopoulos, and M. Kass, "Signal matching through scale space," English, *International Journal of Computer Vision*, vol. 1, no. 2, pp. 133–144, 1987, ISSN: 0920-5691. DOI: 10.1007/BF00123162. [Online]. Available: http://dx.doi.org/10.1007/BF00123162.

[13] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London B: Biological Sciences*, vol. 207, no. 1167, pp. 187–217, 1980, ISSN: 0080-4649. DOI: 10.1098/rspb.1980.0020.

[14] F. Leitner and P. Cinquin, "Dynamic segmentation : Detecting complex topology 3d-object," in *Engineering in Medicine and Biology Society, 1991. Vol.13: 1991., Proceedings of the Annual International Conference of the IEEE*, Oct. 1991, pp. 295–296. DOI: 10.1109/IEMBS.1991.683943.

[15] R. Szeliski, D. Tonnesen, and D. Terzopoulos, "Modeling surfaces of arbitrary topology with dynamic particles," in *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR '93., 1993 IEEE Computer Society Conference on*, Jun. 1993, pp. 82–87. DOI: 10.1109/CVPR.1993.340975.

[16] T. McInerney and D. Terzopoulos, "Topologically adaptable snakes," in *Computer Vision, 1995. Proceedings., Fifth International Conference on*, Jun. 1995, pp. 840–845. DOI: 10.1109/ICCV.1995.466850.

[17] V. Caselles, F. Catté, T. Coll, and F. Dibos, "A geometric model for active contours in image processing," English, *Numerische Mathematik*, vol. 66, no. 1, pp. 1–31, 1993, ISSN: 0029-599X. DOI: 10.1007/BF01385685. [Online]. Available: http://dx.doi.org/10.1007/BF01385685.

[18] L. C. Evans, J. Spruck, *et al.*, "Motion of level sets by mean curvature i," *J. Diff. Geom*, vol. 33, no. 3, pp. 635–681, 1991.

[19] V. Caselles, R. Kimmel, and G. Sapiro, "Geodesic active contours," *International journal of computer vision*, vol. 22, no. 1, pp. 61–79, 1997.

[20] D. Mumford and J. Shah, "Optimal approximations by piecewise smooth functions and associated variational problems," *Communications on Pure and Applied Mathematics*, vol. 42, no. 5, pp. 577–685, 1989, ISSN: 1097-0312. DOI: 10.1002/cpa.3160420503. [Online]. Available: http://dx.doi.org/10.1002/cpa.3160420503.

[21] T. F. Chan, B. Y. Sandberg, and L. A. Vese, "Active contours without edges for vector-valued images," *Journal of Visual Communication and Image Representation*, vol. 11, no. 2, pp. 130–141, 2000.

[22] T. Chan and L. Vese, "An active contour model without edges," in *Scale-Space Theories in Computer Vision*, Springer, 1999, pp. 141–151.

[23] P. Getreuer, "Chan-vese segmentation," *Image Processing On Line*, vol. 2, pp. 214–224, 2012.

[24] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D: Nonlinear Phenomena*, vol. 60, no. 1, pp. 259–268, 1992.

[25] G. Aubert and L. Vese, "A variational method in image recovery," *SIAM Journal on Numerical Analysis*, vol. 34, no. 5, pp. 1948–1979, 1997.

[26] S. Yang and M. K. Gobbert, "The optimal relaxation parameter for the sor method applied to the poisson equation in any space dimensions," *Applied Mathematics Letters*, vol. 22, no. 3, pp. 325–331, 2009.

[27] L. OpenCV, "Computer vision with the opencv library," *GaryBradski & Adrian Kaebler-O'Reilly*, 2008.

[28] M. K. Ng, R. H. Chan, and W.-C. Tang, "A fast algorithm for deblurring models with neumann boundary conditions," *SIAM Journal on Scientific Computing*, vol. 21, no. 3, pp. 851–866, 1999. DOI: 10.1137/S1064827598341384. eprint: http://dx.doi.org/10.1137/S1064827598341384. [Online]. Available: http://dx.doi.org/10.1137/S1064827598341384.

[29] X.-F. Wang, D.-S. Huang, and H. Xu, "An efficient local chan–vese model for image segmentation," *Pattern Recognition*, vol. 43, no. 3, pp. 603–618, 2010.

[30] W. Niethammer, "The sor method on parallel computers," *Numerische Mathematik*, vol. 56, no. 2, pp. 247–254, ISSN: 0945-3245. DOI: 10.1007/BF01409787. [Online]. Available: http://dx.doi.org/10.1007/BF01409787.

[31] A. H. Sameh, "On jacobi and jacobi-like algorithms for a parallel computer," *Mathematics of computation*, vol. 25, no. 115, pp. 579–590, 1971.

[32] C. Zhang, H. Lan, Y. Ye, and B. D. Estrade, "Parallel sor iterative algorithms and performance evaluation on a linux cluster," DTIC Document, Tech. Rep., 2005.

[33] T. Hopkins, "Parallel overrelaxation algorithms for systems of linear equations,"

[34] L. Boyd, *Solving the poisson problem in parallel with sor*.

[35] S. Mittal, "A study of successive over-relaxation method parallelisation over modern hpc languages," *International Journal of High Performance Computing and Networking*, vol. 7, no. 4, pp. 292–298, 2014.

[36] ——, "A study of red-black sor parallelization using chapel, d and go languages," 2015.

[37] D. J. Evans, "Parallel sor iterative methods," *Parallel computing*, vol. 1, no. 1, pp. 3–18, 1984.

[38] R. Benshila *et al.*, "Optimization of the sor solver for parallel run," 2005.

[39] I. Epicoco, S. Mocavero, and G. Aloisio, "The performance model for a parallel sor algorithm using the red-black scheme," *International Journal of High Performance Systems Architecture 12*, vol. 4, no. 2, pp. 101–109, 2012.

[40] I. Epicoco and S. Mocavero, "The performance model of an enhanced parallel algorithm for the sor method," in *Computational Science and Its Applications–ICCSA 2012*, Springer, 2012, pp. 44–56.

[41] P. Dillinger, J. Vogelbruch, J. Leinen, S. Suslov, R. Patzak, H. Winkler, and K. Schwan, "Fpga-based real-time image segmentation for medical systems and data processing," *Nuclear Science, IEEE Transactions on*, vol. 53, no. 4, pp. 2097–2101, 2006.

[42] H. Y. Chang, I. H. R. Jiang, H. P. Hofstee, D. Jamsek, and G. J. Nam, "Feature detection for image analytics via fpga acceleration," *IBM Journal of Research and Development*, vol. 59, no. 2/3, 8:1–8:10, Mar. 2015, ISSN: 0018-8646. DOI: 10.1147/JRD.2015.2398631.

[43] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[44] F. De Dinechin, J. Detrey, O. Creţ, and R. Tudoran, "When fpgas are better at floating-point than microprocessors,"

[45] C. Nvidia, *Programming guide*, 2008.

[46] K. O. W. Group *et al.*, "The opencl specification," *Version*, vol. 1, no. 29, p. 8, 2008.

[47] L. Pan, L. Gu, and J. Xu, "Implementation of medical image segmentation in cuda," in *Information Technology and Applications in Biomedicine, 2008. ITAB 2008. International Conference on*, IEEE, 2008, pp. 82–85.

[48] V. Vineet and P. Narayanan, "Cuda cuts: Fast graph cuts on the gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, IEEE, 2008, pp. 1–8.

[49] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, IEEE, 2009, pp. 126–131.

[50] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, and J. Lemeire, "Comparing and combining gpu and fpga accelerators in an image processing context," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, IEEE, 2013, pp. 1–4.

[51] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung, "Gpu versus fpga for high productivity computing," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, IEEE, 2010, pp. 119–124.

[52] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[53] S. Jones, "Introduction to dynamic parallelism."

[54] W. B. Langdon, "Parallel architectures and bioinspired algorithms," in, F. Fernández de Vega, I. J. Hidalgo Pérez, and J. Lanchares, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Creating and Debugging Performance CUDA C, pp. 7–50, ISBN: 978-3-642-28789-3. DOI: 10.1007/978-3-642-28789-3_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28789-3_2.

[55] M. Harris *et al.*, "Optimizing parallel reduction in cuda,"

[56] L. Adams and J. Ortega, "A multi-color sor method for parallel computation.," in *ICPP*, Citeseer, 1982, pp. 53–56.

[57] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, "The itk software guide," 2003.

[58] E. Konstantinidis and Y. Cotronis, "Accelerating the red/black sor method using gpus with cuda," in *Parallel Processing and Applied Mathematics*, Springer, 2011, pp. 589–598.

[59] S. Pieper, M. Halle, and R. Kikinis, "3d slicer," in *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*, IEEE, 2004, pp. 632–635.