NTNU – Trondheim
Norwegian University of
Science and Technology

Segmentation of Medical Image Data using Level Set Methods

Andreas Helmich Hunderi
Neshahavan Karunakaran

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Frank Lindseth, IDI
Co-supervisor: Erik Smistad, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The field of medical image analysis is becoming an increasingly important part of the medical profession. Advancements in the field of medical imaging techniques results in images and volumes with an increasing level of detail. Effective methods are needed to extract information from this ever increasing ammount of data, making the field of image analysis more important than ever.

Segmentation is an important part of the medical image analysis process. It is used to extract visualize and process relevant anatomical structures within the body. In this project we explore a spesific segmentation approach known as the level set method to extract medical data. We wanted to explore its ability to extract data from volumes of different modailites, such as CT and MRI.

The level set method was implemented using the sparse field approach which is a version optimized for serial execution on the CPU. In addition we explored the possibility of parallelizing it using CUDA on the GPU.

The results shows that the implemented sparse field method produces good results and is exellent at preventing leakage where other similar methods would struggle. However, level set methods have some problems segmenting images with low variance, causing leakages, which is also present in the implemented sparse field algorithm.

The program was parallelized in the GPU using the CUDA technology. The sparse field method is however optimized for serial implementation, which resulted in little performance increase.

Acknowledgements

The authors would want to thank our advisor Frank Lindseth and co-advisor Erik Smistad for the help and guidance given throughout the project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Data associated with a patient is ever increasing, and with this increase comes a need for effective ways of interpreting it. Medical image processing such as segmentation and registration is becomming part of daily operations in the medical proffession because of the possibilities they provide. This project is aimed at exploring ways of efficiently interpreting this data using level set methods.

1.2 Project Goals

The purpose of this project is to explore the Level Set method for medical image segmentation with the purpose of extracting anatomical structures from medical volume data. The following tasks will be attempted:

- Implement a version of the level set method
- Investigate possibilities of fast image segmentation using the implemented method
- Investigate the strengths and weaknesses of the level set method and try to achieve good segmentation results

1.3 Outline

Chapter 2 - Background Theory

In this chapter relevant background theory is presented. This includes how medical volumes are created, an introduction to general segmentation and parallel computing in the GPU.

Chapter 3 - Level Set Method

The Level Set method is described in detail. Two approaches, Narrow Band and Sparse Field, is introduced.

Chapter 4 - Sparse Field Implementation

A walkthrough of the implemented code of the sparse field method will be given.

Chapter 5 - Results

Chapter 5 presents the segmentation results and performance results.

Chapter 6 - Discussion

Results and the outcome of the implemented algorithm is discussed in this chapter.

Chapter 7 - Conclusion and Future Work

The last chapter contains conclusion and possibilities for future work.

Chapter 2

Background Theory

2.1 Medical Background

2.1.1 Brain

The brains position inside the cranium makes it difficult to access for surgery and diagnosis. Non-invasive approaches to diagnosis has therefore become the norm and is today widespread in its usage. The brain can be subdivided into several regions, but the segmentation performed in this project will only distinguish between general features, so the level of detail needed to describe the result is therefore limited.

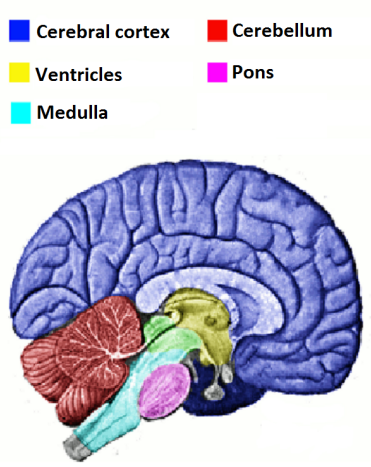


Figure 2.1: Overview of the brain.

Figure 2.1 is a basic overwiew of the brains structure and in figure 2.2 a

cross section of the cerebral cortex can be seen. Like the rest of the brain, the cerebral cortex can be further divided into greymatter shown in figure 2.2 as the band folding in on itself along the surface, and white matter as the lighter area surrounded by the greymatter. The brain also contains a set of structures containing cerebrospinal fluid known as the ventricular system, shown in the mddle of figure 2.2.

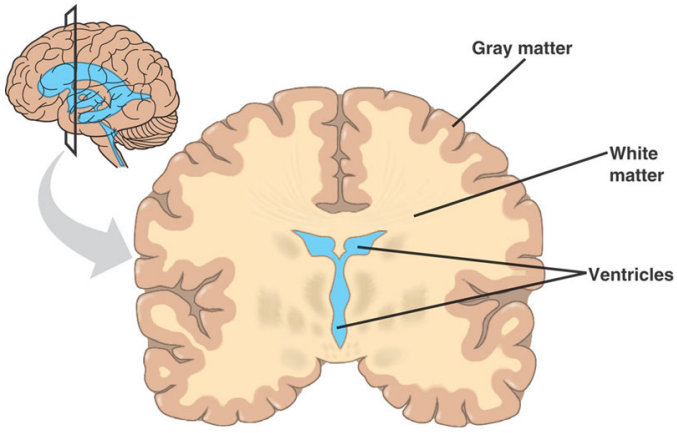


Figure 2.2: Ventricles are shown in blue.

2.1.2 Blood vessels

The blood vessels transport blood throughout the body and is part of the circulatory system. Vessels carrying blood from the heart to the tissues are called arteries and vessels returning the blood back to the heart are called veins. Damage to arteries are usually more dangerous than damage to veins because arteries deliver vital oxygen to the tissues. Reduced bloodflow to the tissues can either be caused by a ruptured blood vessel or a blood vessel occluded by a blood clot. Insufficient blood flow can in turn lead to necrosis (tissue death). Anurisms are bulges on the wall of the blood vessel. If ruptured they cause hemorrhage that can be life-threatening when they occur in vital organs, especially the brain. Medical imaging can detect aneurism that need to be treated before they rupture or detect bleeding that already has occured and need to be treated in its own way.

2.1.3 Liver

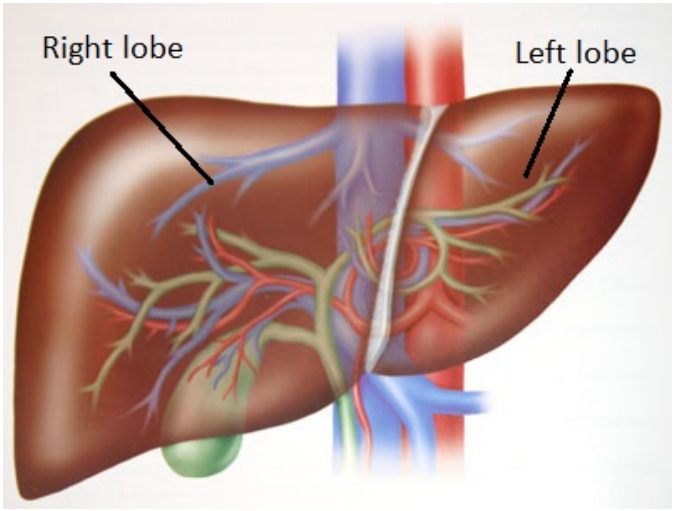


Figure 2.3: Illustration of the liver.

The functions of the liver ranges from detoxification and red blood cell decomposition to storage of glycogen. It is considered a vital organ, and diseases concerning the liver can be life-threatening. Extracting the liver volume can be difficult because of its composition and shape. It normally has a wide range of pixel intensities when captured using modalities like CT and MRI, which makes it difficult for segmentation techniques to identify it.

2.2 Medical Imaging Techniques

2.2.1 X-Ray imaging

X-ray imaging is a medical imaging method that uses X-ray radiation to generate images. X-ray photons are generated using a X-ray tube which consist of an anode and a cathode on opposite sides, with vacuum in the space between them. Electrons are liberated when the cathode is heated up, and accelerate at a high speed toward the anode. When the electrons hit the anode (which usually consist of one of the metals tungsten, copper or molybdenum) about 1% of the energy is converted into X-ray photons while the rest dissipates as heat. The X-ray photons are directed towards the patient, which is located between the X-ray tube and a detector (digital

film). The X-ray travels through the body, some of it being absorbed and the rest hits the detector. The parts of the body with high density absorbs most of the X-ray photons directed towards it, while soft tissues such as muscle and fat absorbs some of it depending on the type of tissue and its density. The detector acts as a digital film which represents the final image as white where the X-ray energy was absorbed by the body, and dark in places with little absorbtion (e.g. liquid and air). The X-ray image can thus be seen as the "shadow" of the released X-ray energy.

Even if the soft tissues does not absorb as much of the X-rays as the hard tissues, they still absorb some, so if a low energy photon source were used, it would be difficult to see the difference of hard and soft tissues in the resulting X-ray image. This is why X-ray on bones and other hard tissues requires a photon source of high energy. High energy means more radiation. A side-effect of X-ray imaging is the ionizing radiation from the X-rays, but conventional X-ray imaging does not require a large amount of radiation. Another disadvantage is that conventional X-ray imaging can only be used to create 2D images, which limits the amount of information gained. Advantages with X-ray imaging is that it is very fast to use and good at bone imaging. Thus, X-ray imaging is widely used to detect bone fractures and by dentists to exam teeth.

2.2.2 Computed Tomography

A computed tomography (CT) machine consist of an X-ray source (emitter) that generates X-rays and releases them towards the patient. The detector array at the opposite side receives the X-rays not absorbed by the patient. The machine rotates around the patient while releasing X-rays to get information from all directions. The detector array (scintillator) transforms the X-rays into proportionally strong electric currents which is represented as image slices. By moving the table step by step a full 3D volume can be created by combining the 2D slices together.

The advantages by using CT over a normal X-ray scan is that CT can take images in any direction, and that the result is a volume of data. Another advantage is the high contrast of the resulting images, CT can differentiate between tissues with less than 1% density difference. But better quality comes with a cost, increasing the quality of the images requires an increase in the amount of radiation. So there is always a tradeoff between noise in the images and the dosage of radiation. As mentioned before, X-rays are ionizing, and the high amount of ionizing radiation from CT is its biggest disadvantage. MRI is sometimes preferred over CT for small children, since

the ionizing radiation effects younger people more. Unlike conventional X-

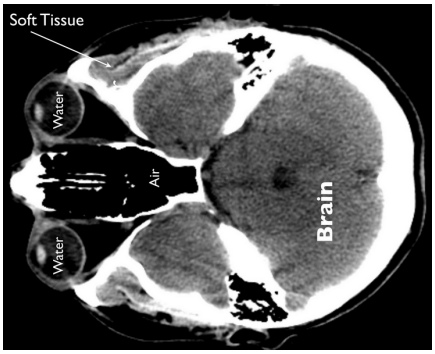


Figure 2.4: CT image of a head.

ray imaging which is mostly used to represent teeth and bone, CT is used more broadly. CT is for example used to image the heart, abdomen, acute and chronic changes in lung, detecting tumors in different parts of body, in addition to bone fractures. Figure 2.4 depicts a regular CT image of the head.

2.2.3 Magnetic Resonance Imaging

All electrons, protons and neutrons have an angular momentum around their own axis, i.e. they have a spin. All charged objects with a spin and an odd mass number creates a magnetic field around themselves. These small magnetic fields are exploited by a MRI machine to generate the MRI signal. These tiny magnetic fields are randomly aligned when no external force is acting on them.

If a small magnetic field is inside a much stronger homogenous magnetic field it will align itself according to the strong one. This is the idea behind MRI. The fact that hydrogen is the most abundant element in the body when considered as number of atoms, and its nucleus only consist of a single proton makes hydrogen the most sensitive atom to MRI machines.

The magnetic fields generated by MRI machines used clinically today vary from 0.2 to 3 tesla, and using stronger magnetic fields results in lower signal-to-noise ratio. The homogenous magnetic field, B_0, generated by the MRI machine is aligned in a certain direction referred to as the longitudinal direction. When B_0 is turned on, the hydrogen protons in the patient's body are aligned parallel with it, i.e. in the longitudinal direction. Most of the protons will align their magnetic field in the longitudinal direction, causing a net magnetisation M_z aligned parallel with B_0 ($M_z = M_0$), while

some will align in the opposite (antiparallel) direction. The favorable state of the protons is when they are parallel with B_0, which is the less energetic and more stable state. While B_0 is turned on, a small radio frequency pulse (RF-pulse) is applied through a coil perpendicular to B_0, towards the area of the body to be examined. This RF-pulse has the same frequency as the spin, or nuclear precession, of the hydrogen protons, thus affecting only the hydrogen nucleus. The hydrogen protons aligned with B_0 will absorb this RF-pulse and jump to a higher energy state, and as a result align in a direction away from B_0, causing the net magnetisation M_z to rotate away from the longitudinal direction. The amount of rotation depends on the strength and length of the RF-pulse. The RF-pulse can therefore be used to adjust the direction of net magnetisation to any angle. When the RF-pulse is turned off, the absorbed energy is released, resulting in the protons to return (relax) to being re-aligned with B_0. The RF-pulse is continously turned on and off, and the energy emitted (MR-signal) when relaxing is picked up by receiver coils, processed by a computer and stored as a 3D data volume.

In addition to the homogenous B_0 field there are additional smaller magnetic fields called gradient fields. The purpose of these non-homogenous gradient fields is to determine the exact position of where to get a 2D slice from. A gradient field changes the precession of the hydrogen protons along the axis it is applied, and by sending out RF-pulses targeting these hydrogen protons the exact position of the patients body to get a 2D slice from is determined. Moreover, the gradient fields and the RF-pulse can also be used to determine the thickness of the 2D slices.

The advantage of MRI over CT is that there is no ionizing radiation associated with it. A disadvantage is that people with metal implants can not use MRI because of the strong magnetic field. Another disadvantage is that the imaging process takes long time, which is problematic for people with claustrophobia since the patient have to be inside the machine.

T1 and T2 weighted images

The time interval between two successive RF-pulses is called the repetition time (T_R), and the time taken from the RF-pulse is applied to the peak of the signal received by the coil is the echo time (T_E). The time taken after a RF-pulse for the net magnetisation M_z to re-align with B_0 is called the longitudinal or spin-lattice relaxation time. The magnetisation in the longitudinal plane (z-axis) is given by

$$M_z = M_0(1 - e^{-t/T_1}), \tag{2.1}$$

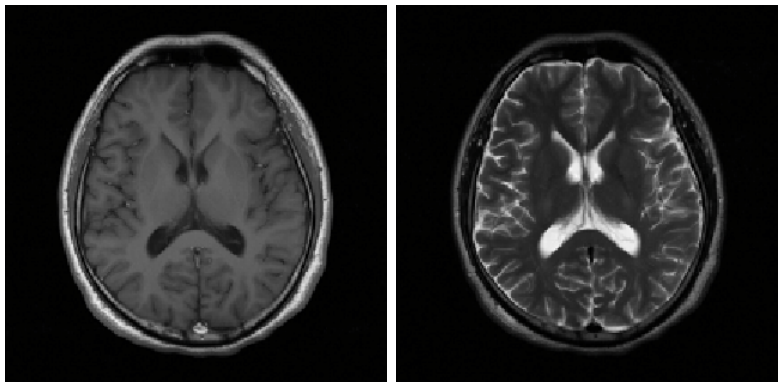
where T_1 is the time taken for M_z to recover $1-e^{-1} = 63\%$ of the equilibrium net magnetisation M_0. T_1 varies for protons of different tisse types. This is measured and used as the main source of tissue contrast information in T_1-*weighted* images. T_1-weighted images are created at the time of the greatest difference between the T_1 values of the tissues being examined, by using short T_R and T_E. Increasing the magnetic field B_0 increases T_1, hence increasing the strength of B_0 gives better contrast in T_1-weighted images.

Neighbouring molecules causes the hydrogen protons attached to different types of molecules to experience slightly different local magnetic fields. As a result, these hydrogen protons will precess at slightly different frequencies. This causes the spins to dephase and decrease the net transverse (xy-plane) magnetisation right after the RF-pulse is turned off, and is called transverse or spin-spin relaxation. This decay of magnetisation in the transverse plane is defined as

$$M_{xy} = M_0\, e^{-t/T_2}, \tag{2.2}$$

where T_2 is the time taken for transverse magnetisation to reach $e^{-1} = 37\%$ of its initial value. The contrast in T_2-weighted images are determined by differences in T_2 relaxation times of different tissue types, and is taken when the difference in the T_2 values is greatest.

The difference of T_1 and T_2-weighted images is illustrated in figure 2.5 (from [1]). Both images were taken with magnetic fields of 1.5 tesla. In



(a) (b)

Figure 2.5: (a): T1-weighted image, (b): T2-weighted image

T_1-weighted images, fluids (such as the cerebrospinal fluid in figure 2.5 a) are dark and fat-based tissues are brighter. This gives clear boundaries between different tissue types, and is thus used to represent anatomical

structures. Fluids are very bright and tissues are mid gray in T_2-weighted images. Therefore, T_2-weighted are used to demonstrate pathology.

2.3 Segmentation

Image segmentation is the process of dividing an image into meaningful non-overlapping regions or objects. The main goal is to divide an image into parts that have strong correalation with objects of the real world. Segmented regions are homogenous according to some property, such as pixel intensity or texture. Mathematically speaking, a complete segmentation of an image I is a finite set of regions $I_1, ..., I_S$ such that the condition (from [4])

$$I = \bigcup_{i=1}^{S} I_i, \quad I_i \cap I_j = \emptyset, \quad i \neq j \tag{2.3}$$

is satisfied. Image segmentation is one of the first steps leading to image analysis and interpretation. It is used in many different fields, such as machine vision, biometric measurements and medical imaging.

Automated image segmentation is a challenging problem for many different reasons. Noise, partial occluded regions, missing edges and lack of texture contrast between regions and background are some of these reasons. Noise is an artifact often found in images which makes the segmentation process harder. In the process of generating medical images noise is often introduced by the capturing devices. As a pre-processing step before segmentation, the image can be smoothed to reduce noise. In the context of medical images segmentation usually means a delineation of anatomical structures. This is important for e.g. measurements of volume or shape. Low level segmentation methods are usually not good enough to segment medical images. Thus, higher level segmentation methods that are more complex and gives better results are used. The biggest difference between low-level segmentation methods and higher level segmentation methods is the use of apriori information. Low-level methods usually have no information about the image to be segmented, while high-level segmentation methods can incorporate different types and amount of apriori information.

Traditional low-level image segmentation methods can roughly be divided according to the type of technique used:

- Global/Histogram based methods
- Region based methods
- Edge based methods

2.3.1 Histogram-based segmentation methods

Global knowledge about an image is usually represented by the histogram of the intensity values in the image. Histogram-based segmentation methods uses this information to segment simple images. These segmentation methods are usually much faster than other methods, but restricted to images with simple features.

Thresholding

The simplest segmentation approache is called thresholding. Thresholding is used to seperate objects from the background using a threshold value T. A threshold value splits the image in two groups, where all pixels with intensity value higher than T represents an object or the foreground, and the rest represents another object or the background. Choosing a good thresold value is important, as small changes in the value can significantly affects the resulting segmentation, which can be seen in figure 2.6c and 2.6d (described in more detail later). The threshold can be selected manually by either inspecting the image or the histogram of the image. But usually the threshold is selected automatically, and a variety of methods for automatically selecting T exists. When little noise is present, the mean or median intensity values can be selected as the threshold. The simplest method to select a threshold, apart from doing it manually, is iterative thresholding and is computed as follows:

1. Choose an initial threshold T_0 and segment the image.
2. The segmented image will consist of two groups, C_1 and C_2. Set the new threshold value T_i to be the sum of the mean intensity values from C_1 and C_2, divided by 2.
3. Segment the image using T_i.
4. Repeat steps 2 and 3 until $|T_i - T_{i-1}|$ is less than a predefined value.

By using multiple threshold values the image can be split up into several regions. Segmentation by thresholding is only suitable for very simple images, where the objects in the image does not overlap and their intensity values are clearly distinct from the background intensity values. If the threshold is poorly chosen, the resulting binary image would not be able to correctly distinguish the foreground from the background.

Otsu's thresholding method

Otsu's thresholding method assumes that the image contains two regions with the values in each region creating a cluster. Otsu's method tries to make each cluster (or class) as tight as possible, thus minimizing their overlap. The goal then is to select the threshold that minimizes the combined spread. The threshold that maximizes the between-class variance $\sigma_b^2(t) = \omega_1(t)\omega_2(t)\left[\mu_1(t)\mu_2(t)\right]^2$ is sought after. $\omega_1(t)$ and $\omega_2(t)$ are the weights (computed from the normalized histogram) of the two clusters, and $\mu(t)$ is the mean intensity value of the clusters. Otsu's method starts by splitting the histogram into two clusters using an initial threshold. Then $\sigma_b^2(t)$ is computed for that threshold value. The between-class variance $\sigma_b^2(t)$ is then iteratively computed for every intensity value, and the threshold that maximizes the between-class variance $\sigma_b^2(t)$ (or minimizes the within-class variance) is chosen as the final threshold value.

Figure 2.6 illustartes a gray-scale image and the segmentation results using both iterative global thresholding and Otsu's method. The threshold found using the iterative threshold method is 0.7332 where the range is from 0 (black) to 1 (white). The threshold found using Otsu's method is 0.7686. The image to be segmented is shown in figure 2.6a, and its histogram in figure 2.6b. As can be seen from the histogram, it is not possible to select a near perfect threshold by just looking at it. Figure 2.6c illustrates the segmentation result from the iterative global thresholding method and figure 2.6d is the segmentation result using Otsu's method. Even though the difference of the two threshold values is small, the segmentation results have a considerable difference, where Otsu's method gives a better result.

2.3.2 Region based segmentation

Region based segmentation methods tries to find homogenous regions based on gray-scale, color, texture or any other pixel based measure in an image. Pixels with similar properties are grouped together in regions I_i. The choice of homogenity criteria is an important factor that affects the end segmentation result. In addition to the condition in equation 2.3, images segmented by region based segmentation also satisfies the two following conditions:

- All regions I_i should be homogenous according to some specified criteria: $H(I_i) = true,\ i = 1, 2, ..., S.$
- The region that results from merging two adjacent regions R_i and R_j is not homogenous: $H(I_i \cup I_j) = false.$

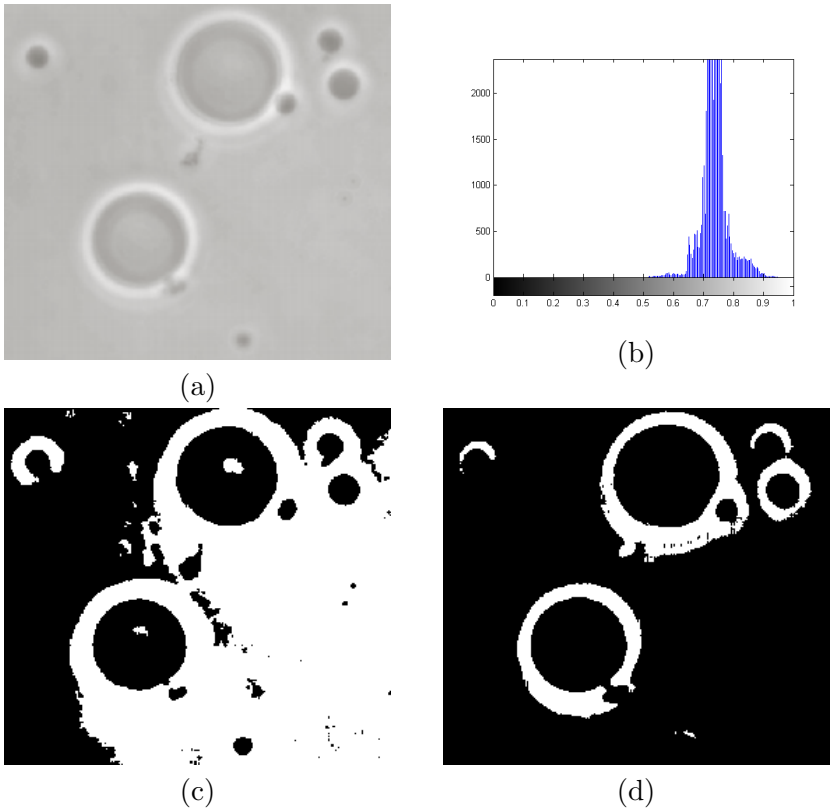


Figure 2.6: (a): Image to be segmented, (b): Histogram of image, (c): Segmented using iterative global thresholding, with $T = 0.7332$, (d): Segmented using Otus's method with $T = 0.7686$.

An example of a homogenity criteria for a region could be all adjacent pixels with intensity value within a range $\{x, y | x \pm y\}$. That is, if two adjacent pixels have intensity values in the range $x \pm y$ they are in the same region. Region based segmentation methods are usually better than edge based segmentation methods in noisy images where the borders are difficult to detect.

Region growing

An example where thresholding is insuficient is when parts of the foreground have the same pixels values as part of the background. In this case, region growing can be used. Region growing starts at a point (seed point) defined to be inside the forground and grows to include neighbouring foreground

pixels. This seed point is manually set at the beginning and consists of one or more pixels. A small region of 4x4 or 8x8 can for example be chosen as a seed region. The regions described by the seed points grows by merging with their neighbouring points (or regions) if the homogenity criteria is met. This merging is continued until merging any more would violate the homogenity criteria. When a region cannot be merged with any of its neighbours it is marked as final, and when all regions are marked as final the segmentation is completed. The result of region growing can depend on the order in which the regions are merged. Thus, the segmentation result may differ if the segmentation begins, for example, in the top right corner or the lower left corner. This is because the order of the merging can cause two similar adjacent regions R_1 and R_2 not to be merged if an earlier merge of R_1 and R_3 changed the characteristics of R_1 such that it no longer is similar (enough) to R_2.

Region splitting

Region splitting is the opposite of region growing, and starts with a single region covering the whole image. This region is iteratively split into smaller regions until all regions are homogenous according to a homogenity criteria. One disadvantage of both region growing and region splitting is that they are sensitive to noise, resulting in regions that should be merged remaining unmerged, or merging regions that should not be merged.

2.3.3 Edge based segmentation

Edge based segmentation methods are used to find edges in the image by detecting intensity changes. The edge magnitude at a certain point is the same as the gradient magnitude, and the edge direction is perpendicular to the gradient. Thus, change in intensity at a point can be detected by using first and second order derivatives. There are various edge detection operators, and they all approximate a scalar edge value for each pixel in an image based on a collection of weights applied to the pixel and its neighbours. These operators are usually represented as rectangular masks or filters consisting of a set of weight values. These masks are applied to the image to be segmented using discrete convolution.

First and second order operators

A simple second order edge detection operator is the Laplacian operator, based on the Laplacian equation:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.4}$$

This equation measures edge magnitude in all directions and is invariant to rotation of the image. Second order derivatives are commonly discretized by approximating it as $\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y)$ which is also how the Laplacian is discretized:

$$\nabla^2 f(x,y) = f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)-4f(x,y) \tag{2.5}$$

This Laplacian equation is represented by the mask in equation 2.6, and a variant of the equation that also takes into account diagonal elements is shown in 2.7.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{2.6}$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{2.7}$$

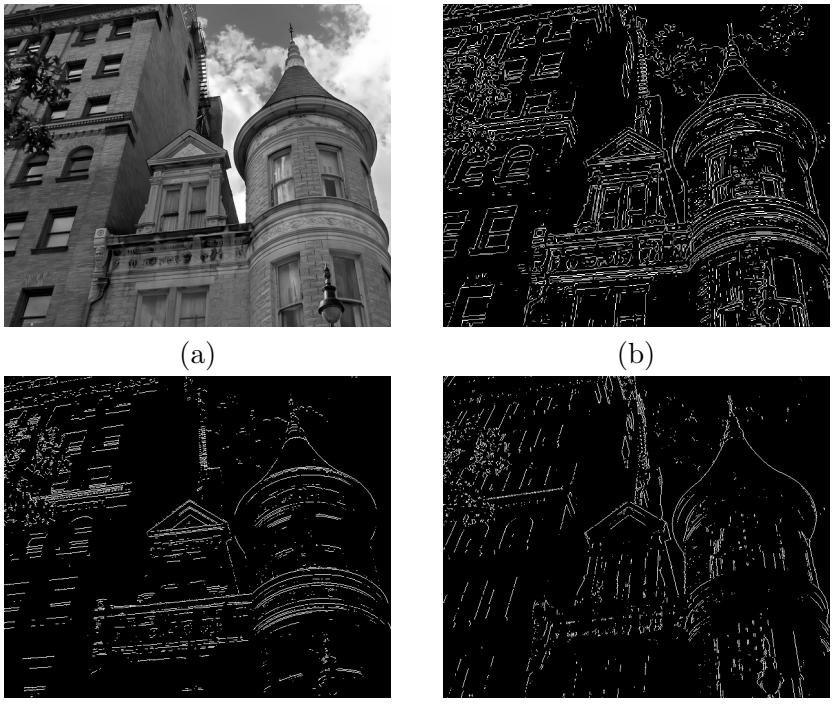
Since the Laplacian mask is based on second order derivatives it is very sensitive to noise. Moreover, it produces double edges and is not able to detect the edge direction. The center of the actual edge can be found by finding the zero-crossing between the double edges. Hence, the Laplacian is usually better then first order derivatives to find the center-line in thick edges. To overcome the sensitivity to noise problem, the image can be smoothed beforehand. This is the idea behind the Laplacian of Gaussian (LoG) operator. The LoG mask is a combination of a Gaussian operator (which is a smoothing mask) and a Laplacian mask. By convolving an image with a LoG mask it is smoothed at the same time as edges are detected. The smoothness is determined by the standard deviation of the Gaussian, which also determines the size of the LoG mask.

There are various masks based on first order derivatives, and two of them are the Prewitt and Sobel masks, represented in equation 2.8 and 2.9 respectively. These two are not rotation invariant, but the masks can be rotated to emphasize edges of different directions. The masks as they are represented in equation 2.8 and 2.9 highlights horizontal edges.

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \quad (2.8) \qquad \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad (2.9)$$

As can be seen from the above masks, the only difference between the Sobel and Prewitt is that the middle column (or row in a rotated version) in the Sobel mask is weighted by 2 and -2. This results in smoothing since the middle pixel is given more importance, hence, the Sobel is less sensitive to noise than Prewitt.

Figure 2.7a illustrates a gray-scale image and 2.7b is the edge segmented image based on LoG. 2.7c is the edge image resulted from the Sobel mask in equation 2.9b and 2.7d is the result from segmentation after rotating the mask 90°. The segmentations resulted by using the Prewitt operator to segment the image in figure 2.7a had no significant difference from the Sobel segmented images.



(a) (b)

(c) (d)

Figure 2.7: (a): Image to be segmented, (b): LoG, (c): Sobel - highlighting horizontal edges, (d):Sobel - highlighting vertical edges

Canny edge detector

A more powerful edge detection method is the Canny edge detector. This method consist of four steps. The first step is to smooth the image based on a Gaussian filter with a given standard deviation σ. In the next step the derivatives in both directions are computed using any first order operator, and using these the gradient magnitude image and its direction are computed. The gradient magnitude image typically contains wide ridges around local maxima of the gradient. In order to get a single response to an edge, only local maxima should be marked as edges. This process is called non-maxima suppression. A simple way for non-maxima suppression is to first quantize the edge directions according to 8-connectivity (or 4 connectivity). Then consider each pixel with magnitude > 0 as candidate edge pixels. For every candidate edge pixel look at the two neighboring pixels in edge-direction and the opposite direction. If the magnitude of the candidate edge pixel is not larger than the magnitude of these neighboring pixels, mark the pixel for deletion. When all candidate edge pixels are inspected, remove all the candidates that are marked for deletion. Now all the edges will contain a single response, but there still are lines/pixels that are not part of any continues edge. To remove these, hysteresis thresholding is used. Hysteresis thresholding consist of segmenting the image with two threshold values. First, the non-maxima supressed images is thresolded with a high thresold value T_h that determines which of the remaining candidate edge pixels are immediately considered as edge pixels (strong edges). The high threshold value leads to an image with broken edge contours. Therefore a low thresold value T_l is used to threshold the non-maxima supressed image again. The pixels in this segmented image that are connected to a strong edge are added to the final edge image.

The Canny edge detector gives different results based on the values of σ, T_h and T_l, but the derivative operator used to find the magnitude and how the non-maxima suppression was implemented also affects the final edge segmented image.

2.4 Parallel computing in GPU

A huge disadvantage with the level set method for segmentation is that it is very slow when working with big data volumes in 3D space. Implementations of level set algorithms for 3D in the graphical processing unit (GPU) parallelizes the level set method and makes it much faster. One of the first GPU based 3D implementations of the level set method was by Lefohn et

al. in [9] in 2003. In this paper a modified sparse field level set method was implemented for the GPU using graphic APIs such as OpenGL and DirectX. In the past few years general purpose GPUs have made implementing level set methods and other non-graphical tasks in GPUs much easier. In [10] some simple medical segmentation algorithms was implemented using NVIDIAs CUDA technology, and in [11] CUDA was used to implement the level set method.

2.4.1 Data and task parallelism

Data and task parallelism are the two main categories of computer parallelism. Data parallelism is achieved by having different units execute the same task on different data in parallel. This type of parallelism is used in image processing where for example all pixels are increased by the same value. When using task parallelism the tasks are seperated to different executional units (usually cores) and executed on different data. Task parallelism is seperated into two parts based on the type of communication used between the executional units. These two methods are the shared memory method, and the message passing method used in distributed memory. When using shared memory, the executional units have a shared space in the memory that all executional units can read from and write to. To control that no conflicts arises when multiple units accesses the shared memory, locks have to be used. By using locks the part in memory that a unit is writing to cannot be accessed by any other unit, and only when a unit is finished writing is the lock released to provide other units access to the memory data or the lock. Synchronization to prevent race conditions (occurs when operations depending on each other is executed in the wrong order) so that a unit does not change the value of a memory location before other units have used it is also an important factor when using shared memory. Pthreads is an API that supports shared memory multiprocessing, and another which will be introduced later in this chapter is OpenMP. The other method for communication between the units is message passing which is used in distributed memory systems such as supercomputers. The communication is handled by sending and receiving messages between the units. Messages sent can be one of several different types, such as synchronous or asynchronous, one-to-one or one-to-many. Several message passing systems exists, some of them being the Java Remote Method Invocation, Simple Object Access Protocol (SOAP) and the popular Message Passing Interface (MPI).

2.4.2 Central processing unit

The central processing unit (CPU) is responsible for carrying out instructions of computer programs and managing hardware. It runs most processes on the computer and has therefore been optimized to handling many kinds of tasks. Before multiple cores was intruduced to the CPU it was common to run multiple processes using time multiplexing to give the appearence of parallelism. Today most CPU's have multiple cores and can run many processes in parallel by assigning them to different cores. If more than one process runs on any one core, time multiplexing is often used.

2.4.3 Flynn's taxonomy of computer architectures

Michael J. Flynn proposed in 1996 a taxonomy of classification of computer architectures. Taxonomy is the study of the general principles of scientific classification. Flynn described four different types based on the use use of one or multiple numbers of data and instructions.

Single Instruction Single Data - SISD

The SISD architecture uses no parallelism in either the data stream or the instruction stream. SISD is used in uniprocessors and exectues a single instruction on a single data. Figure 2.8 illustrates how SISD works. In the figure processing unit is abbreviated as PU.

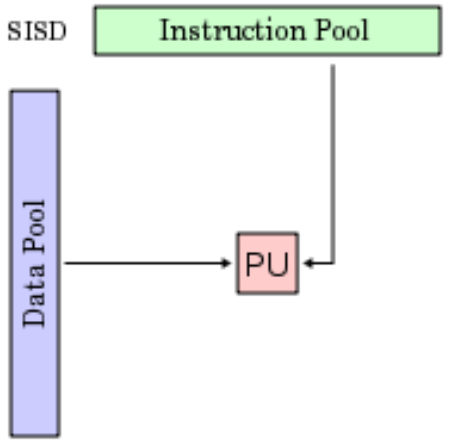


Figure 2.8: Single Instruction Single Data.

Single Instruction Multiple Data - SIMD

Architectures based on SIMD uses multiple processing units to execute a single instruction on multiple data. Thus SIMD uses data level parallelism as previously discussed. Modern CPUs are all able to perform SIMD instructions and they are able to load n numbers (n may vary depending on design) of data to memory at once and and execute the single instruction on the data. An example where SIMD instrctions can be used is in image preocessing where several pixels are to be added or subtracted the by samme value. How SIMD instructions works is shown in figure 2.9

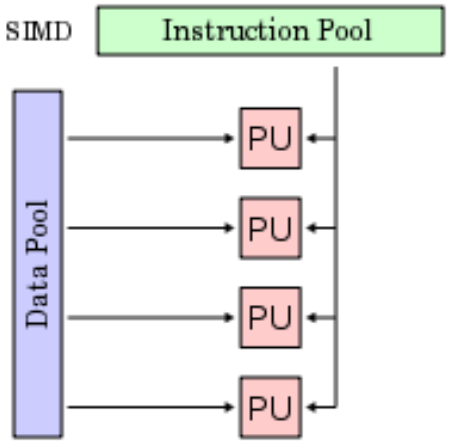


Figure 2.9: Single Instruction Multiple Data.

Multiple Instruction Single Data - MISD

MISD is the least used archetecture type of the four in Flynn's taxonomy. This is because doing multiple instructions on a single data is much less scalable and it does not utilize computational resources as good as the rest. MISD is illustrated in figure 2.10.

Multiple Instruction Multiple Data - MIMD

Being able to do multiple instructions on multiple data is possible by having different processors execute instructions on multiple data. Modern CPUs consisting of several cores are all based on MIMD for parallelism. MIMD is illustrated in figure 2.11.

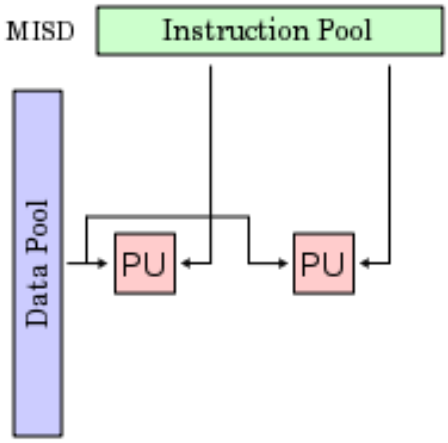


Figure 2.10: Multiple Instruction Single Data.

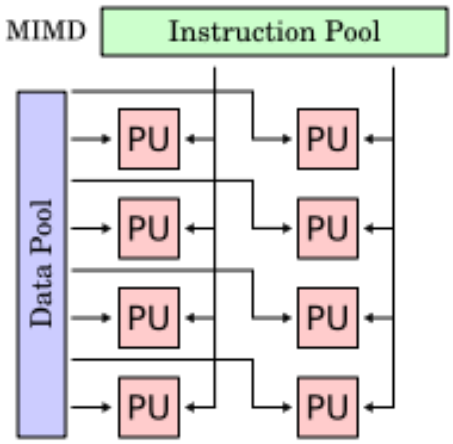


Figure 2.11: Multiple Instruction Multiple Data.

2.4.4 Graphics processing unit

A graphics processing unit (GPU) is a specialized chip that initially was designed to offload the CPU and accelerate processes associated with computer graphics. The process of computing the color of each pixel on screen is memory intensive but independent of each other and thus highly parallelizable. The architecture commonly associated with GPU's is SIMD.

2.4.5 General Purpose GPU

The GPU was developed to be able to perform graphics operations on large data structures in parallel. Earlier, if we wanted to perform something other than graphics operations we had to translate the code to something the GPU was able to understand. A GPGPU is simply a GPU that is able to perform computations normally handled by the CPU. GPGPU's were developed because of the potential speedup associated with parallelizing a program. OpenCL and CUDA are the two dominant open GPGPU computing languages. Unfortunately, GPGPU's come with certain restictions in operations and programming and is therefore not always a good alternative to CPU execution.

2.4.6 OpenMP

OpenMP is an API that supports shared memory parallel programming in C, C++, and Fortran for multiple processor architecture types and operative systems. OpenMP was designed to allow programmers to incrementally parallelize existing serial programs, which is difficult with MPI and Pthreads [14]. OpemMP makes it simple to code parallel behaviour by allowing the compiler and run-time system to determine some of the thread behaviours details. OpenMP is a directive based API, which means that a a serial code can be paralellized with little effort and a carefully written OpenMP program can be compiled and run as a serial program if the compiler does not support OpenMP.

2.4.7 CUDA

CUDA (Compute Unied Device Architecture) is a program development environment introduced by NVIDIA in 2006 for their GPUs in C/C++ and Fortran[**?**]. CUDA and OpenCL (which unlike CUDA supports all kinds of GPUs) have made GPU programming much more user-friendly than before, when the tasks had to be transformed into rendering problems. CUDA programs are initialized on the CPU (called the host) and then the data needed for the computation in the GPU (called the device) is initialized in the CPU and copied over the PCI bus to the GPU. When the computation is finished, the results are copied back to the CPU. In CUDA, a kernel is a program (function) that is executed in the device. The kernel code is run in parallel on a number of threads. Threads are grouped into blocks whose size (number of threads in a block) and dimension (up to 3D) can be decided by the programmer, within the maximum limit of 1024 threads per

block (for compute capability of 2.0 or higher). But 32 threads will always execute the same code (even if less than 32 executions is needed), and such a group of 32 threads is called a warp. Thus, the number of blocks used should be a multiplum of the warpsize to achieve maximum performance. It must be noted that warps are not a part of the CUDA model but device dependent, and even though the warpsize usually is 32, it does vary from device to device. Figure 2.12 illustrates the programming model of CUDA, which also shows that a set of blocks is called a grid.

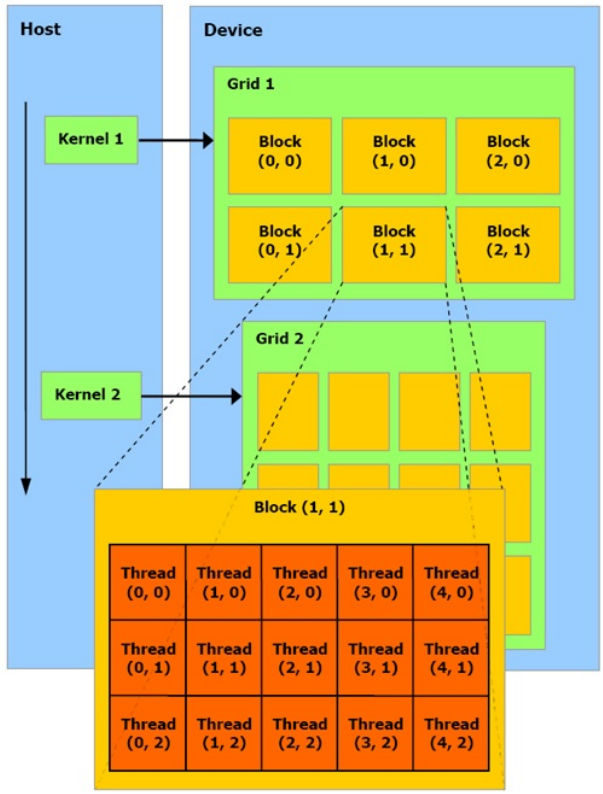


Figure 2.12: CUDA programming model.

Each thread in CUDA have its own registers and local memory, and all the threads in a block have a shared memory, all of which can be written to and read from the device. In addition all threads in a grid share global, constant and texture memory which can be read and written by the host and the device (constant and texture memory is read-only for the device). How these are connected together is indicated in figure 2.13. Registers are the smallest, but also the fastest, and the per-thread register limit for

compute capability (version) 3.0 is 63 registers per thread. If a thread needs more than 63 registers the shared memory is used (L1 cache) which is much slower. And if even more is needed the even slower global is also used.

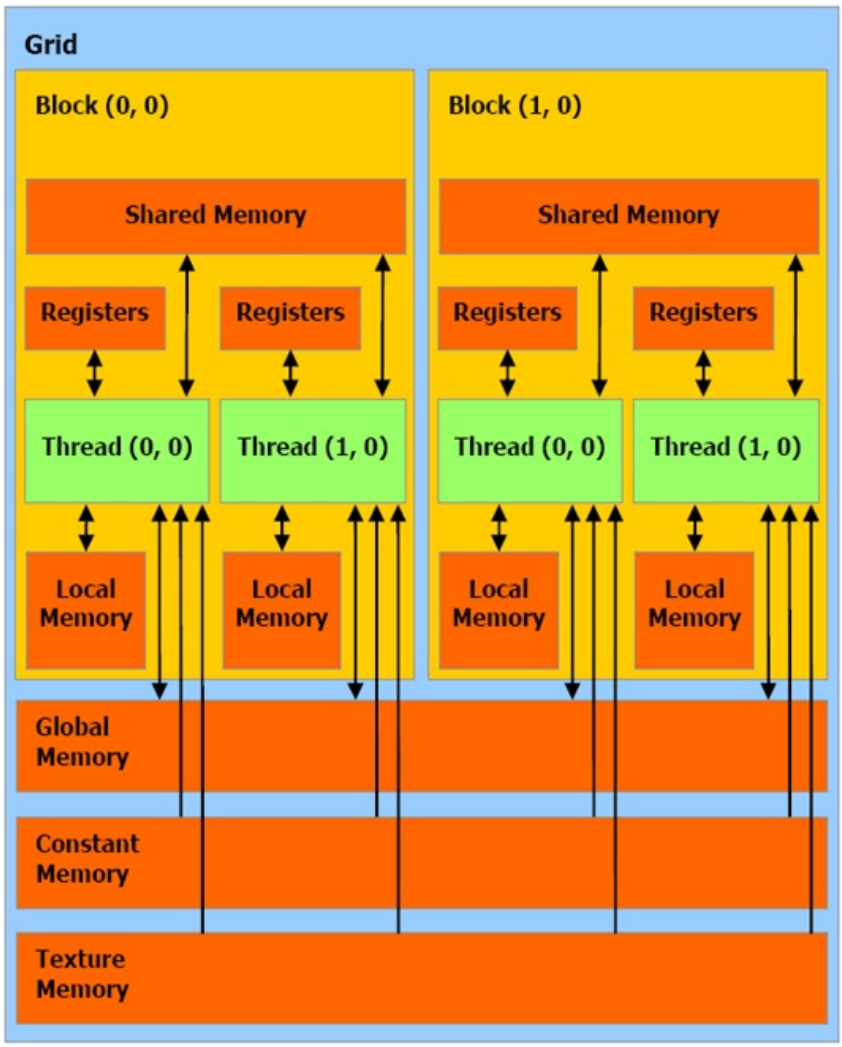


Figure 2.13: CUDA memory model.

Functions used in CUDA code can be of different types. A global function (with the identifier __global__ in front) is a function that runs on the device, but only callable from the host. The second type is device (__device__), and this type of functions are only callable by functions running on the device, i.e. device and global functions. The last type, host, is code that only

runs on the host. Host functions can be identified by the syntax __host__ in front of the return type, but this is not required. NVIDIAs own compiler, nvcc, splits up the code into host and device components, compiles the global and device functions itself, and lets the standard host compiler compile the host code. If the same functions is needed in both host and device it can be compiled as by both nvcc and the host compiler if it is identified by both __host__ and __device__. The syntax for calling a kernel from the device is kernelName<<<numBlocks, numThreadsPerBlock>>>(arg1, arg2, ..., argN). The triple angle brackets indicate that it is a kernel launch. The first number within these brackets is dimension of the grid, measured in the number of blocks in that grid. The second is the block dimension, i.e. the numbers of threads in a block. Calling a kernel with X number of blocks and Y number of threads per block results in $X * Y$ parallel executions of that kernel.

Algorithm 2.1 (from [15]) is a simple CUDA program in C++ that shows the basic CUDA syntax. In line 4 three arrays are created, and in line 5 copies of these to be used in the device is created. These have to be pointers (even if they are not arrays as in this case) because they are to be used on the device and must point to device memory. In line 9-11 space is allocated for the arrays using cudaMalloc in the device just like calling malloc would allocate space on the host. After two of the arrays have been filled with random values they are copied over to the device using the cudaMemcpy function in line 22-23. The cudaMemcpy function takes in four inputs. The first input is the destination address, which in this case was allocated in line 9-10, the second input is the source to copy, the third input is the size in bytes and the last input is the type of transfer. Type of transfer is either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost indicating transfer from host to device and device to host, respectively. On line 26 the kernel is launched. The code within the kernel (line 38-43) is written as serial code, and to differentiate between the different threads all threads can be assigned an unique thread identification. This thread-id is calculated as the id of the thread within the block (threadIdx), plus the id of the block (blockIdx) times the number of blocks (blockDim). The if-sentence in line 40 avoid errors in case the size of the array is less than the number of threads started (not necessary in this case since the number of threads is equal to the array size). Inside the if-sentence each thread does one addition with its thread-id as the position in the arrays. At line 29 the result array is copied back to the CPU, and lastly the allocated space in the GPU is freed in line 32-34.

```
1 #define N (2048*2048) //number of threads
```

```
2 #define M 512 //number of threads per block
3 int main( void ) {
4     int *a, *b, *c; // host copies of a, b, c
5     int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
6     int arraySize = N * sizeof( int ); // need space for N
          integers
7
8     // allocate device copies of a, b, c
9     cudaMalloc( (void**)&dev_a, arraySize );
10    cudaMalloc( (void**)&dev_b, arraySize );
11    cudaMalloc( (void**)&dev_c, arraySize );
12
13    a = (int*)malloc( arraySize );
14    b = (int*)malloc( arraySize );
15    c = (int*)malloc( arraySize );
16
17  //fill the a and b arrays with randon integers
18    random_ints( a, N );
19    random_ints( b, N );
20
21    // copy inputs to device
22    cudaMemcpy( dev_a, a, arraySize, cudaMemcpyHostToDevice );
23    cudaMemcpy( dev_b, b, arraySize, cudaMemcpyHostToDevice );
24
25    // launch add() kernel with blocks and threads
26    add<<< N/M, M >>>( dev_a, dev_b, dev_c );
27
28    // copy device result back to host copy of c
29    cudaMemcpy( c, dev_c, arraySize, cudaMemcpyDeviceToHost );
30
31    free( a ); free( b ); free( c );
32    cudaFree( dev_a );
33    cudaFree( dev_b );
34    cudaFree( dev_c );
35    return 0;
36 }
37
38 __global__ void add( int *a, int *b, int *c ) {
39    int threadId = threadIdx.x + blockDim.x*blockIdx.x;
40  if(threadId < arraySize){
41      c[threadId] = a[threadId] + b[threadId];
42  }
43 }
```

Listing 2.1: Simple CUDA program

Chapter 3

Level Set Method

3.1 Introduction

Surfaces that evolves over time can be difficult to represent. Taking the surface in figure 3.1 as an example, assume that the red surface is heat and the arrows on the interface as the direction of its movement, which is normal to the interface itself. One way to represent the propagation of this interface is by the function $y = f(x, t)$, where t represents time and x, y are coordinates. The problem with this representation is that it cannot represent every concievable shape of the interface. If for instance the shape of the interface has more y coordinates for a particular x coordinate (which is true for all closed interfaces), the interface cannot be correctly represented using this notation. A better alternative is to use a parametric equation. The prob-

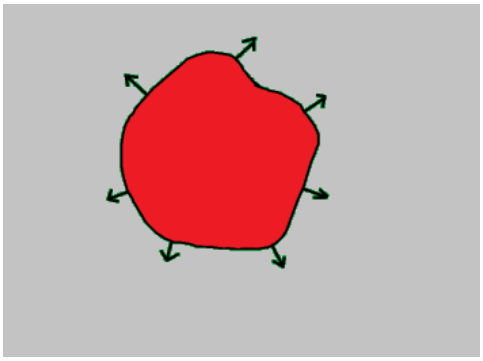


Figure 3.1: Interface of a moving surface.

lem mentioned above would then be solved because the interface would only depend on the time variable t. But parametric representation of evolving

interfaces have its own difficulties. When a surface evolves, the model have to be reparemeterized, which due to the computional overhead (especially in 3D) add limitations to what kind of shapes a parameterical model can represent effectively. Topological changes, such as splitting or merging parts during the propagation is difficult to represent using parametric models. Sharp corners, distant edges blending together and the complexity of representing boundaries in higher dimensions are some other reasons why an evolving surface is difficult to represent parametrically. A simple example is shown in figure 3.2. The two interfaces have to be represented as a single parametric function when merging and as two seperate againg when they split, and some sort of collison detection must be used to discover when the interfaces merge/split.

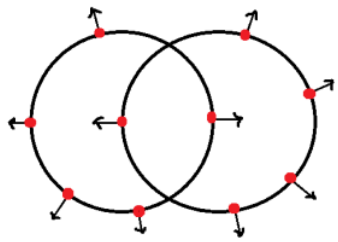


Figure 3.2: Interface evolution difficult to represent parametrically.

As a solution to all the problems mentioned above, Osher and Sethian introduced the level set method in 1988 in [2]. The main idea behind the level set method is to represent the interface of a surface implicitly by using a higher dimensional function. Adding an extra dimension simplifies the problems mentioned above significantly. This higher dimension function is called the level set function, and a 2D interface (a curve) would be represented by the 3D level set function

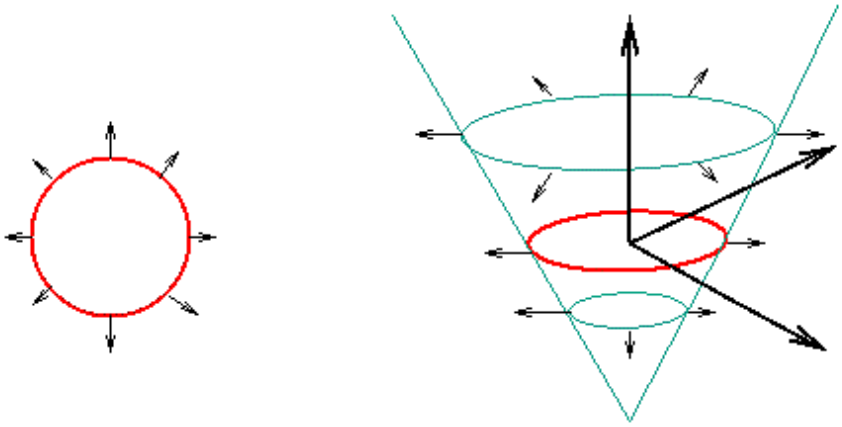
$$\phi(x, y, t) \tag{3.1}$$

where the additional dimension t represents time. Similarly any 3D or higher level function can be represented by a level set function by adding one dimension. At a given time step, the evolving surface/model can be represented as a closed curve by the boundary of the level set at that time step. This representation of the model is called the zero level set and is

defined as the set of points where the level set is zero:

$$\Gamma(x, y, t) = \{\phi(x, y, t) = 0\}. \tag{3.2}$$

The initial curve is at the xy-plane, that is, at $\phi(x, y, 0)$. As an example, figure 3.3a depicts a circle with arrows pointing in the direction it is evolving, and figure 3.3b is the cone that represents the corresponding level set function with the start-position in red.



(a) (b)

Figure 3.3: (a): Circle with arrows pointing in direction of movement, (b): Corresponding level set function

Assuming that the zero level set moves in a direction normal to the speed F, then ϕ satisfies the level set equation

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| F \tag{3.3}$$

which is used to update the level set at each time step (iteration). Here $|\nabla \phi|$ represents the gradient of ϕ, and the speed function F describes how each point in the boundary of the surface evolves. The level set method is applied in many different contexts, such as image processing, fluid dynamics and other simulations, and the speed function F depends on the type of problem being considered.

3.2 Speed function for image segmentation

An often used speed function for image segmentation that combines a data term and the mean curvature of the surface is [9, 8]

$$F = \alpha D(I) + (1 - \alpha)\nabla\frac{\nabla\phi}{|\nabla\phi|} \tag{3.4}$$

where $\nabla\cdot(\nabla\phi/|\nabla\phi|)$ is the normal vector that represents the mean curvature term which keeps the level set function smooth. $D(I)$ is the data function that forces the model towards desirable features in the input data. The free weighting parameter $\alpha \in [0, 1]$ controls the level of smoothness, and I is the input data (the image to be segmented). The smoothing term α restricts how much the curve can bend and thus alleviates the effect of noise in the data, preventing the model from leaking into unwanted areas[8]. This is one of the big advantages the level set method has over classical flood fill, region grow and similar algorithms, which does not have a constraint on the smoothness of the curve.

A simple data function for any point (pixel, voxel) based solely on the input intensity I at that point[9, 8] is:

$$D(I) = \epsilon - |I - T| \tag{3.5}$$

Here T is the central intensity value of the region to be segmented, and ϵ is the deviation around T that is also considered to be inside the region. This makes the model expand if the intensity of the points are within the region $T \pm \epsilon$, and contract otherwise. The data function, plotted in 3.4, is gradual, thus the effects of $D(I)$ diminish as the model approaches the boundaries of regions with gray-scale levels within the $T \pm \epsilon$ range [8]. This results in the model expanding faster with higher values of ϵ and slower with lower values.

The level set algorithm is initialized by placing a set of seed points that represents a part inside the region to be segmented. These seed points are represented by a binary mask of the same size as the image to be segmented. This mask is used to compute the signed distance function which ϕ is initialized to be the signed distance transfrom.

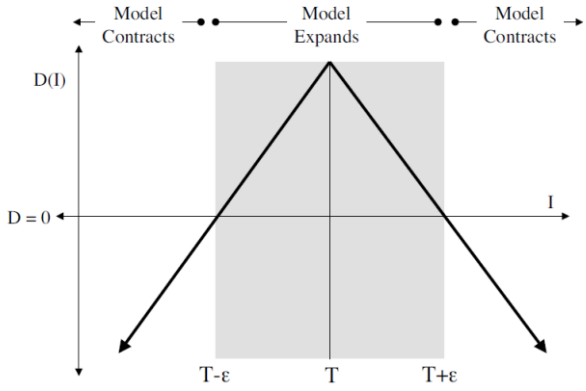


Figure 3.4: The data function, from [9].

3.3 Signed Distance Transform

A distance function $D : \mathbb{R}^3 \rightarrow \mathbb{R}$ for a set S is defined as

$$D(r, S) = min(r - S) \quad for\ all\ r \in \mathbb{R}^3 \tag{3.6}$$

If a binary image have one or more objects, a distance function can be used to assign a value for every pixel (or voxel in 3D) that represents the minimum distance from that pixel to the closest pixel in the boundary of the object(s). That is, the pixels in the boundary of an object are zero valued, and all other pixels represent the distance to the boundary as a value. Using a distance transform was the idea of how to initialize ϕ in [2], where it was initialized as $\phi = 1 \pm D^2$. But in [6] it was showed that initializing ϕ to a signed distance function gives more accurate results. Signed distance transforms (SDT) assign for each pixel a value with a positive or negative sign that depend on whether the pixel is inside or outside the object. The values are usually set to be negative for pixels that are inside an object, and positive for those outside. The pixels of the model, which represents the boundary (the zero level set), have values 0. A binary image containing an object is shown in figure 3.5a (the numbers in this image represent intensity values). Figure 3.5b is the signed distance transform of 3.5a where city-block (manhattan) distance have been used, and figure 3.5c is the signed Euclidean distance transform (SEDT).

As can be seen from the figures above, using different kind of functions for the SDT can result in different distances. These differences effects the accuracy of the level set function, which may leads to different end-results

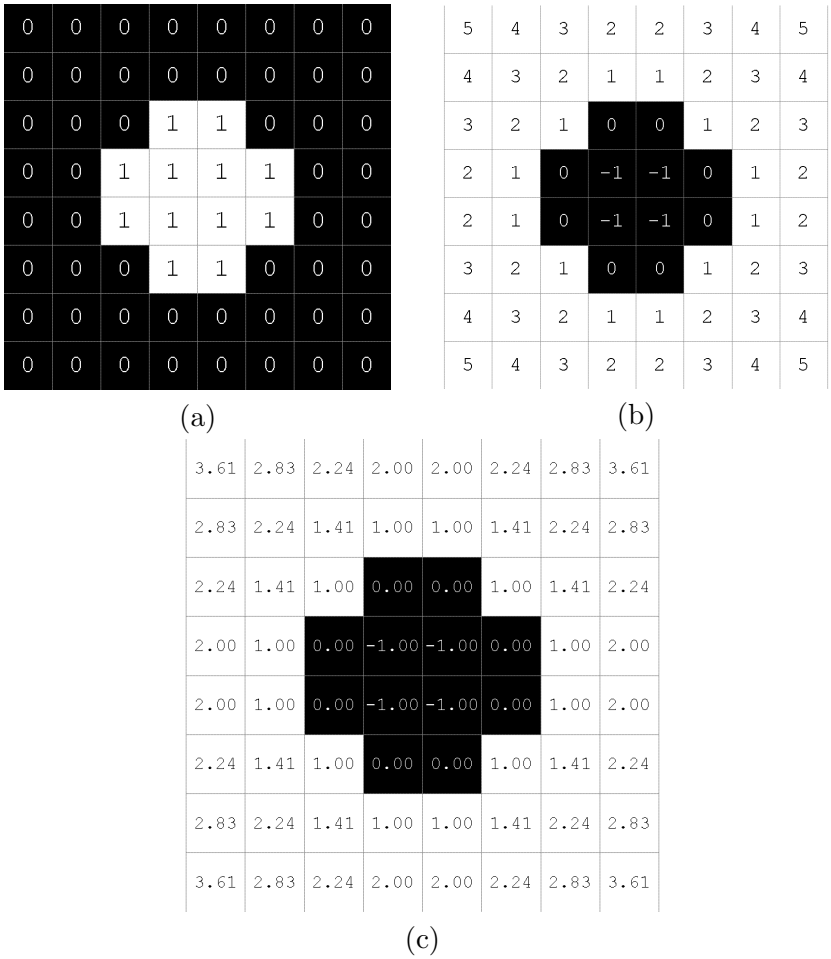


Figure 3.5: (a): Binary image, (b): SDT based on city-block distance, (c): SDT based on euclidean distance

of the segmentation, hence, the function used to represent the distance have to be carfully chosen. However, sometimes a less accurate SDT have to be used as a tradeoff for faster computation time.

3.4 Discretization by upwinding and difference of normals

To use the level set method in image processing it have to be discretized, but simple forward finite difference schemes cannot be used because such

schemes tends to overshoot and are unstable. To overcome this problem the up-winding scheme was proposed in [2]. To avoid the overshooting problems associated with forward finite differences the up-winding scheme uses one-sided derivatives that looks in the up-wind direction of the moving interface. Let ϕ^n and F^n represent the values of ϕ and F at some point in time t^n. The updating process consist of finding new values for ϕ at each point after a time interval Δt. The forward Euler method is used to get a first-order accurate method for the time discretization of equation 3.3, given by (from [5])

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + F^n \cdot \nabla \phi^n = 0 \tag{3.7}$$

where ϕ^{n+1} is ϕ at time $t^{n+1} = t^n + \Delta t$, and $\nabla \phi^n$ is the gradient at time t^n. This equation is expanded as follows (for three dimensions):

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u^n \phi_x^n + v^n \phi_y^n + w^n \phi_z^n = 0, \tag{3.8}$$

where the techniques used to approximate the $u^n \phi_x^n$, $v^n \phi_y^n$ and $w^n \phi_z^n$ terms can be applied independently in a dimension-by-dimension manner [5]. When looking at only one dimension (for simplicity), the sign of u^n would indicate whether the values of ϕ are moving to the right or to the left. The value u^n can be spatially varying, hence by looking at only one point x_i in addition to only look at one dimension, equation 3.8 can be written as

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + u_i^n (\phi_x)_i^n = 0, \tag{3.9}$$

where $(\phi_x)_i^n$ denotes the spatial derivative of ϕ at point x_i at time t^n. The values of ϕ are moving from left to right if $u_i > 0$, thus the points to the left for x_i are used to determine the value of ϕ at point x_i for the the next time step. Similarly, if $u_i < 0$ the movement is from right to left, and the points to the right of x_i are used. As a result, ϕ_x is approximated by the derivative function D_x^+ when $u_i < 0$ and D_x^- when $u_i > 0$. When $u_i = 0$ the term $u_i(\phi_x)_i$ equals zero, and approximation is not needed. Extending this to three dimensions, the derivatives used to update the level set equation are

$$D_x = \frac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{2} \quad D_y = \frac{\phi_{i,j+1,k} - \phi_{i,j-1,k}}{2} \quad D_z = \frac{\phi_{i,j,k+1} - \phi_{i,j,k-1}}{2}$$

$$D_x^+ = \phi_{i+1,j,k} - \phi_{i,j,k} \quad D_y^+ = \phi_{i,j+1,k} - \phi_{i,j,k} \quad D_z^+ = \phi_{i,j,k+1} - \phi_{i,j,k}$$

$$D_x^- = \phi_{i,j,k} - \phi_{i-1,j,k} \quad D_y^+ = \phi_{i,j,k} - \phi_{i,j-1,k} \quad D_z^+ = \phi_{i,j,k} - \phi_{i,j,k-1}$$

(3.10)

which is taken from the appendix of [8]. This is a *consistent* finite difference approximation to the level set equation in 3.3, because the approximation error converges to zero as $\Delta t \to 0$ and $\Delta x \to 0$ [5]. In addition to being consistent, it also have to be *stable* in order to get the correct solution. Stability guarantees that small errors in the approximations are not amplified over time. The stability can be enforced using the Courant-Friedreichs-Lewy (CLF) condition which says that the numerical wave speed $\frac{\Delta x}{\Delta t}$ must be greater than the physical wave speed $|u|$,

$$\Delta t = \frac{\Delta x}{max\{|u|\}}, \tag{3.11}$$

where $max\{|u|\}$ is the largest value of $|u|$ on the model.

The gradient $\nabla\phi$ is approximated to either $\nabla\phi_{max}$ or $\nabla\phi_{min}$ depending on whether the speed function for a given point $F_{i,j,k}$ is positive or negative,

$$\nabla\phi = \begin{cases} ||\nabla\phi_{max}||_2 & F_{i,j,k} > 0 \\ ||\nabla\phi_{min}||_2 & F_{i,j,k} < 0 \end{cases} \tag{3.12}$$

where $\nabla\phi_{max}$ and $\nabla\phi_{min}$ is given by (from [8])

$$\nabla\phi_{max} = \begin{bmatrix} \sqrt{max(D_x^+, 0)^2 + max(-D_x^-, 0)^2} \\ \sqrt{max(D_y^+, 0)^2 + max(-D_y^-, 0)^2} \\ \sqrt{max(D_z^+, 0)^2 + max(-D_z^-, 0)^2} \end{bmatrix} \tag{3.13}$$

$$\nabla\phi_{min} = \begin{bmatrix} \sqrt{min(D_x^+, 0)^2 + min(-D_x^-, 0)^2} \\ \sqrt{min(D_y^+, 0)^2 + min(-D_y^-, 0)^2} \\ \sqrt{min(D_z^+, 0)^2 + min(-D_z^-, 0)^2} \end{bmatrix} \tag{3.14}$$

The curvature term $\nabla \cdot (\nabla\phi/|\nabla\phi|)$ of the speed function F is discretized using the difference of normals method. The second order derivatives are computed first:

$$D_x^{+y} = (\phi_{i+1,j+1,k} - \phi_{i-1,j+1,k})/2 \quad D_x^{-y} = (\phi_{i+1,j-1,k} - \phi_{i-1,j-1,k})/2$$

$$
\begin{aligned}
D_x^{+z} &= (\phi_{i+1,j,k+1} - \phi_{i-1,j,k+1})/2 \quad D_x^{-z} = (\phi_{i+1,j,k-1} - \phi_{i-1,j,k-1})/2 \\
D_y^{+x} &= (\phi_{i+1,j+1,k} - \phi_{i+1,j-1,k})/2 \quad D_y^{-x} = (\phi_{i-1,j+1,k} - \phi_{i-1,j-1,k})/2 \\
D_y^{+z} &= (\phi_{i,j+1,k+1} - \phi_{i,j-1,k+1})/2 \quad D_y^{-z} = (\phi_{i,j+1,k-1} - \phi_{i,j-1,k-1})/2 \\
D_z^{+x} &= (\phi_{i+1,j,k+1} - \phi_{i+1,j,k-1})/2 \quad D_z^{-x} = (\phi_{i-1,j,k+1} - \phi_{i-1,j,k-1})/2 \\
D_z^{+y} &= (\phi_{i,j+1,k+1} - \phi_{i,j+1,k-1})/2 \quad D_z^{-y} = (\phi_{i,j-1,k+1} - \phi_{i,j-1,k-1})/2
\end{aligned}
\tag{3.15}
$$

Then these derivatives are used to compute the normals n^+ and n^- in equation 3.16, which is used to compute the mean curvature H in equation 3.17 taken from [8].

$$
n^+ = \begin{bmatrix}
\frac{D_x^+}{\sqrt{(D_x^+)^2 + (\frac{D_y^{+x} + D_y}{2})^2 + (\frac{D_z^{+x} + D_z}{2})^2}} \\
\frac{D_y^+}{\sqrt{(D_y^+)^2 + (\frac{D_x^{+y} + D_x}{2})^2 + (\frac{D_z^{+y} + D_z}{2})^2}} \\
\frac{D_z^+}{\sqrt{(D_z^+)^2 + (\frac{D_x^{+z} + D_x}{2})^2 + (\frac{D_y^{+z} + D_y}{2})^2}}
\end{bmatrix}
$$

$$
n^- = \begin{bmatrix}
\frac{D_x^-}{\sqrt{(D_x^-)^2 + (\frac{D_y^{-x} + D_y}{2})^2 + (\frac{D_z^{-x} + D_z}{2})^2}} \\
\frac{D_y^-}{\sqrt{(D_y^-)^2 + (\frac{D_x^{-y} + D_x}{2})^2 + (\frac{D_z^{-y} + D_z}{2})^2}} \\
\frac{D_z^-}{\sqrt{(D_z^-)^2 + (\frac{D_x^{-z} + D_x}{2})^2 + (\frac{D_y^{-z} + D_y}{2})^2}}
\end{bmatrix}
\tag{3.16}
$$

$$
H = \frac{1}{2} \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} = \frac{1}{2} [(n_x^+ - n_x^-) + (n_y^+ - n_y^-) + (n_z^+ - n_z^-)] \tag{3.17}
$$

Finally, the level set equation is updated as

$$
\phi(t + \Delta t) = \phi(t) + \Delta t F |\nabla \phi|. \tag{3.18}
$$

3.5 Chan-Vese engergy function

The speed function of the the level set discussed so far is one of the most popular speed functions used, along with many modifications and improvements of it based on the problems at hand. But this far from the only speed function out there. The Chan-Vese energy funtion is another function that is used as speed function to evolve the level set. The Chan-Vese model is a powerful and flexible method which is able to segment many types of images, and is used widely in the medical imaging field, especially for the segmentation of the brain, heart and trachea.

In this project a simplified version of the Chan-Vese function is used. This simplified version of the Chan-Vese energy function (E^{CV}) is defined as:

$$E^{CV}(c_1, c_2, C) = \int_{inside(C)} (\mu(x,y) - c_1)^2 \, dx \, dy + \int_{outside(C)} (\mu(x,y) - c_2)^2 \, dx \, dy \text{ [13]} \tag{3.19}$$

where μ is the image, and C is a closed segmentation curve. In the context of the level set function C is the curve defined by the zero level set. The constants c_1 and c_2 are the average greyscale intensity values inside and outside of C, respectively. Discretizing this energy function and writing it as a pixelwise function gives

$$E^{CV}(x,y) = (\mu(x,y) - c_1)^2 - (\mu(x,y) - c_2)^2. \tag{3.20}$$

The average values c_1 and c_2 can chosen by sampling intensity values both outside and inside of the object to be segmented and averaging them. When the algorithm is calculating the speed of a point in the interface (determining whether its going to expand or retract) it looks at its pixel value and compares it to the measured mean values of the foreground and background. The speed, either positive, negative, or zero, depends on which of the two mean values it resembles the most. This simplified version of the Chan-Vese function acts much like a simple region grow method without any consideration for curvature and smoothness of the zero level set.

3.6 Narrow Band

3.6.1 Introduction

When working with the level set of a single interface a huge drawback with the originally proposed level set method is the computional inefficiency due to computing over the whole domain of ϕ. As a solution to this problem

Adalstein and Sethian proposed the narrow band method in 1994[3]. The narrow band looks at the interface of a single level set instead of the whole domain, and thereby decreases the computational labor of the standard level set method for propagating interfaces considerably. Another reason the narrow band was proposed are problems where the velocity field is only given on the interface. In such cases the construction of an appropriate speed funcion for the entire domain made use of the classical level set method a significant modeling problem.

3.6.2 Overview of the Narrow Band method

Unlike the original level set method, which describe the evolution of an embedded family of contours, the narrow band works with only a single surface model[7]. That is, instead if calculating ϕ over the whole domain it focuses only on a small part surrounding the surface. There are many cases in which the description of the evolution of only one surface in the domain is needed, and in such cases the narrow band method operates much faster while delivering the same results. The method ignores points that are far away from the zero level set at each iteration and only looks at the points within a narrow band. This is possible because points far away from the zero level set do not have any influence on the result. That is, only the area of ϕ where $\phi \approx 0$ is important for accurate representation of the level set. The narrow band method restrict the computation to a thin band of points by extending out approximately k points from the zero level set (shown in figure 3.6), and an embedding of the evolving interface is constructed via a signed distance transform. All points outside the band is set to constant values to indicate that they are not within the band and thus should not be used in the computation. This reduces the number of operations at each iteration from $O(n^{d+1})$ to $O(nk^d)$ [3] where d is the number of dimensions and n is the (average) number of points in one dimension. The points within the band is used to calculate the distance function and then to initialize ϕ to the signed distance. As the zero level set evolves, ϕ will get further and further away from its initialized value as signed distance. As this happens ϕ must be ensured to stay within the band. One way to do this would be to make a new band for each iteration. But determining which points are to be inside the band, and deciding how to take the differentials at the edge points makes the reconstruction process of the band time consuming. Thus a given band is used for several iterarions with the same initialization of ϕ. When the interface gets close to the band it has to be reset from the current position of the zero level set and ϕ must be reinitialized. Reinitializing ϕ

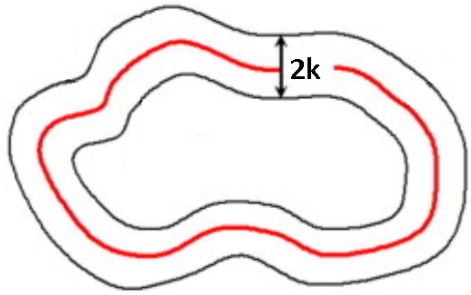


Figure 3.6: The narrow band extending out with a width of k from the level set.

at every iteration takes too much time and the alternative task of finding out if any of the pixels in the zero level set are getting close to the edge of the band (for every iteration) also takes time. Hence, ϕ is usually just reinitialized after a fixed number of iterations, which keeps ϕ approximately equal to the SDT.

As metioned in the section about signed distance transforms, different SDTs can lead to slightly different end-results and must be carefully chosen. If the technique used to approximate ϕ to a signed distance function is too sensitive, ϕ needs to be reinitialized accurately and often. If it is less sensitive, it does not have to be initialized so often and a less accurate method can be used, but this may lead to noisy features [5].

The narrow band, despite its improvements over the original level set method, is not optimal. The band used being too wide is the main reason. Even if k=2 is enough to compute the necessary derivatives, the band have to be of a certain width (k=12 was used in the test of topological changes in [3]) because of two competing computional costs[7]. The first is the cost of computing the position of the curve and the SDT, and reset the band. The second is the cost of computing the evolution process over the entire band.

3.7 Sparse Field

3.7.1 Introduction

The narrow band method assumes that the computation of the SDT is so slow that it cannot be computed for every iteration. The sparse field method introduced in [7] uses a fast approximation of the distance transform that makes it feasible to compute the neighborhood of the level set model for

each iteration. In the sparse field method the idea of using a thin band is taken to the extreme by working on a band that is only one point wide. To keep track of the band a set of points, active points, are defined. The set of these active points, the active set, is equal to the zero level set. Using only the active set to compute the derivatives would not give sufficient accuracy, hence the method extends out from the active points in layers one pixel wide to create a neighborhood that is precisely the width needed to calculate the derivatives for each time step.

Several advantages to this approach are mentioned in [7]. Like stated above, no more than the precise number of calculations to find the next position of the zero level set surface is used. This also results in that only those points whose values control the position of the zero level set surface are visited at each iteration, which minimizes the calculations necessary. The number of points being computed is so small that a linked-list can be used to keep track of them.

The sparse field algorithm is based on an important approximation. It assumes that points adjacent to the active points undergo the same change in value as their nearby active set neighbours. But despite this, the errors introduced by the sparse field algorithm are no worse than the error in the original level set algorithm.

As a result of only visiting the grid points whose values are changing (the active points and their neighbors) at each time step, the computation time is $\mathcal{O}(d^{n-1})$, where d is the number of pixels along one dimension of the problem domain [7]. This is the same as for parameterized models where the computation times increase with the resolution of the domain, rather than the range. Comparing this with the fact that the originally proposed level set method has $\mathcal{O}(d^n)$ computation time, the sparse field should perform considerable faster. This was confirmed in [7], where these two methods were compared. Their sparse field algorithm achieved a speed increase of 34.8 times in a $300x300$ image containing a circle of radius $N/3$ moving inwards with constant speed.

3.7.2 Overwiew of the Sparse Field method

A disadvantage of the narrow band method is that the stability at the boundaries of the band have to be maintained (e.g. by smoothing) since some points are undergoing the evolution while other neighbouring points remain fixed. The sparse field method avoid this by not letting any point entering or leaving the active set affect its value. A point enters the active set if it is adjacent to the model. As the model evolves, points that are

no longer adjacent to the model are removed from the active set. This is performed by defining the neighborhoods of the active set in layers and keeping the values of points entering or leaving the active set unchanged. A layer is a set of pixels represented as L_i where i is the city-block (manhattan) distance from the active set. The layer L_0 represents the active set, and $L_{\pm 1}$ reprsents pixels adjacent to the active set on both sides. Using linked lists to represents the layers and arrays (matrices) to represent distance values makes the algorithm very efficient.

As described in section 3.4, the Up-Winding scheme is used to calculate the curvature in an area a point in a grid. This scheme uses both first and second order derivatives, and to calculte them it needs a 3x3x3 grid (in 3D) of points surrounding the active point whose speed is being calculated. This creates a lower limit for the number of layers needed surrounding the active set. In addition to the active set which is stored in $L0$, four additional lists are needed, L_1 L_2 L_{-1} L_{-2}. These lists keeps track of where the points of computational significance are located at any time during execution. They are defined by their closeness to, and on which side of $L0$ they are located. L_{-1} and L_1 are defined as the layers of neighbouring points to L_0 on the inside and outside, respectively, of the object under segmentaion. Likewise, L_{-2} and L_2 are defined as the neighbours to L_{-1} and L_1. Like the other aproaches to the level set method, the datastructure that tracks the evolution of the interface is an array (ϕ) with the same dimensions as the problem domain. It is important to note that the lists are used to keep track of which points are in the active set and their neighbours, and changes in ϕ must be reflected in these lists.

The initialization process of the interface is fairly straight forward and starts by defining a seed point. This is usually a binary mask, of equal size as the problem domain, consisting of points defined as either inside (seed) or outside the mask. The points in the border of the seed (points inside the object with neighbours defined as outside points) are defined as the active set (i.e. zero level set). This is reflected by assigning the corresponding points in ϕ to 0, and by adding them to the $L0$ list. The other lists are then filled with points according to their definitions, and ϕ is updated accordingly (more about this later).

Each iteration consist of three main steps. First, points in ϕ who are members of the active set are updated by the speed function. Secondly these changes are reflected in the neighbouring layers, and finally, the lists are updated accordingly. How these steps are executed will be described in chapter 4.

Chapter 4

Sparse Field - Implemented code

4.1 Introduction

The sparse field level set method was implemented in C++ for this project, and the implemented code is mainly based on the pseudocode in [12], which again is based on Whitaker's introduction to the sparse field method in [7].

The sparse field was first implemented in 2D and after bugfixing and some test-runs it was extended to 3D, which executes and runs in the exact same way as the 2D version. Later the code was parallelized by implementing it in CUDA. The implemented code of the all versions can be found in the Appendix.

This chapter will give a detailed explanation of the implemented code and how it works. Henceforth, when the word pixel is meantioned it can have slightly different meanings. Elements in different arrays will be referred to as pixels (even if they actually are integer of floating point values), as will the elements in all the lists.

4.2 The layers and their representation

As previously mentioned, the sparse field method can be implemented using linked lists to hold the pixels being used in the calculations. These pixels are seperated into five layers, each represented by a linked list. These layers (described in 3.7) are visually illustrated in figure 4.1. In this figure the lists L_{-2}, L_{-1}, L_0, L_1 and L_2 are represented as Ln2, Ln1, Lz, Lp1 and Lp2 respectively, and will henceforth be referred as such.

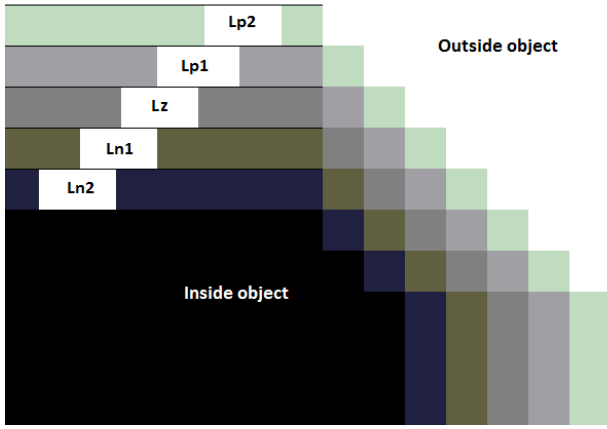


Figure 4.1: Visual representation of the layers.

The elements in these lists are C/C++ structures (struct) called *Pixel* which contains integer values x and y (and z in 3D) representing their coordinates in ϕ. Hence, when an element in ϕ is to be added to any of the layers, the coordinates in ϕ of that element is used to create a new *Pixel* which is added to the list corresponding to the layer in question.

The range of values a point in ϕ must have to be in any of the lists, as defined in [7], is shown in table 4.1. By looking at table 4.1 it can be seen that Lz has a slightly wider range than the other lists. This range of exactly 1 does in some cases cause problems that lead to disortions and artifacts in the segmentation. What these problems are will be discussed in 4.5. To overcome these problems the ranges of the lists were slightly changed to make all the lists equal in range. The range-corrected lists used in the implementation are shown in table 4.2, and even though the change seems small and insignificant it improves the result significantly (as will be discussed in 4.5).

List Name	Range
Lz	$[-0.5, 0.5]$
Ln1	$[-1.5, -0.5\}$
Lp1	$[0.5, 1.5]$
Ln2	$[-2.5, -1.5\}$
Lp2	$\{1.5, 2.5]$

Table 4.1: Range of lists used in [7] and [12]

List name	Range
Lz	[-0.5, 0.5}
Ln1	[-1.5, -0.5}
Lp1	[0.5, 1.5}
Ln2	[-2.5, -1.5}
Lp2	[1.5, 2.5}

Table 4.2: Range of lists used in the implementation

4.3 Datastructures and types used

In addition to the five lists representing the five layers, two arrays of equal size and dimension as the problem domain are used. One of them is shown in figure 4.2. This image represents the 5 different layers with different

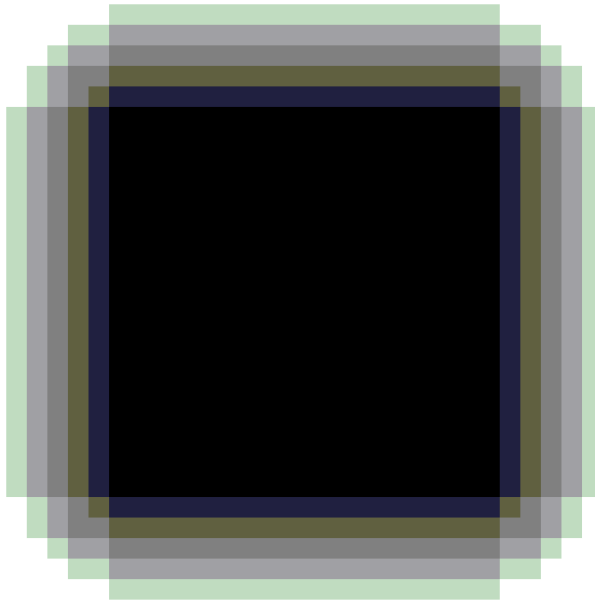


Figure 4.2: Label image: image showing the different layers under segmentation.

colors, with the same color codes as in figure 4.1. At a given iteration, Ln1, Ln2 and the dark parts are defined to be inside the object being segmented. Similarly, Lp2, Lp1 and the white parts are defined to be outside the object. This type of image will be referred to as the *label* image/array because it shows the label assigned to each pixel of the image being segmented. This

array is used to track where the pixels containing the different layers are on the domain. Given a point, to find out which layer (if any) it is a member of, a simple lookup to the *label* array is enough. An excellent feature of the *label* array is that it can be used to visually verify if all the layers are correctly aligned and if there are any pixels of any layer that are poorly placed. The *label* image can thus be used to find artifacts that might have resulted from code errors by an user visually looking at it, which proved to be of excellent help when debugging. An example of a *label* image which clearly states that there is something wrong with how the layers are handled in the code is shown in figure 4.3 (zoomed in for clarity). How that *label* image actually should have been is illustrated in figure 4.4.

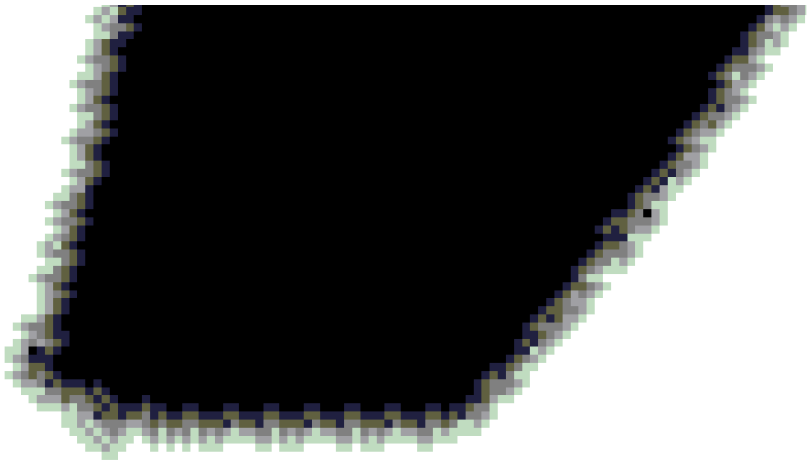


Figure 4.3: A label image with artifacts due to code errors when handling the layers.

The other array used is the ϕ - array, which contains the actual ϕ values of each pixel in the domain. The range of the values is exactly the same as in the *label* image, but while the *label* image only contains integer values describing which layer a pixel is part of, the ϕ image contains the actual values (floating point numbers) of the level set. The images represented by the *label* and ϕ arrays would thus be very similar (though small differences may be seen) when looking at, but they do have different tasks. The *label* is as mentioned used as a lookup table, while the ϕ array represents the actual level set, and is used to determines which layer a pixel belongs to after its value have been updated with the speed function.

To correctly move pixels between the layers some temporary lists have to be used, one for each layer. By using these temporary lists, called Sn2,

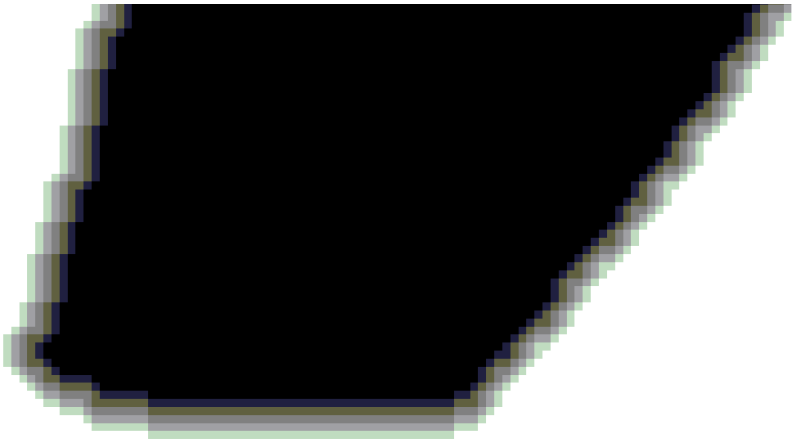


Figure 4.4: How the label image should have been.

Sn1, Sz, Sp2 and Sp2, elements in the corresponding Ln2, Ln1, Lz, Lp1 and Lp2 lists are prevented from being moved more than once in a single iteration.

4.3.1 Code structure

The code (of every version) is seperated into three C++ files, main.cpp, update.cpp, IO.cpp, and corresponding header files. The update.cpp file consist of everything that happens at each iteration, this includes calculating the speed function, updating the ϕ array with the speed, updating the *label* array and update the lists. The IO.cpp file handles the input parameters to the program, as well as reading the input file and writing result to file. The main.cpp file consist of actions executed before and after the actual segmentation, such as initializing everything, and the running main loop by calling functions in update.cpp.

4.3.2 Input, initialization and output

The program takes four inputs, the total number of iterations, threshold, epsilon and alpha. More inputs can be defined as input, e.g. seed points and the location of the input/output files, but these are currently set in the code because too many inputs is unnecessary when running the problem with a few different data sets. Handling of the input code is however set up in a way that makes adding more inputs a simple task.

An array of same size as the problem domain is used to initialize *label*,

ϕ and Lz. This array, called *init*, is initialized to zero valued elements at start, and then filled with 1's given the x,y (and z in 3D) coordinates of the seed point(s). The seed point creates a circle (or sphere if 3D) of 1's in the *init* array that represents the starting position. Then, based on the values in *init* the two arrays *label* and ϕ are initialized. All pixels in *label* and ϕ corresponding with those in *init* with value 1 are set to -3 to indicate that they are inside the segmentation object. All other pixels in *label* and ϕ are set to 3 indicating that they are outside the object. Then the corresponding pixels to all values in *init* that are 1 but have 0 valued neighbours are set to 0, indicating that they are part of the zero level set. Then these pixels are added to Lz as initial zero level set values. Then $Ln1$, $Lp1$, $Ln2$ and $Lp2$ are filled according to their definitions, and the *label* and ϕ arrays are updated to refelct these changes. After these initializing actions are finished the segmentation process can start.

The output of the program is an image/volume of same size as the input. The 2D version outputs two files, the *label* image was as mentioned an important part of the debuging process and is therfore made as an output. The second output is an image containing the zero level set, which defines the border of the segmented object. The output from the 3D version is a volume containing the zero level set. The output is only the zero level set itself, and the object defined by the zero level set is not filled. In the 2D version the *label* image does fill the whole object, and in the 3D version there are more advantages by not filling. It is much easier to compare the segmentation result with the original when looking at 2D slices by superimposing the result on the original. Moreover, it would not make a difference when rendering the result volume from the outside, and when rendering inside the volume it is easier to get a clear overview.

4.4 Levelset evolution process

First all pixels in Lz are updated by the speed function. Some (or all) of these pixels may at that point in time have values outside of Lz's range. The pixels in Lz with values smaller than -0.5 will then be removed from Lz and added to Sn1, while those with values greater than or equal to 0.5 will be removed from Lz and added to Sp1. This process is shown in the pseudocode in algorithm 1. This process of updating the pixels with new values, and moving them to neighbouring lists if they are not within the range of the list, is executed for all the other lists as well. The difference with Lz and the rest is that while the pixels in Lz are directly updated by the speed function the pixels in the other lists are not. By definition, pixels in Ln1

Algorithm 1 Update elements in Lz with the speed function and transfer to Sp1 or Sn1.

```
1: for all p ∈ Lz do                                          ▷ p = Pixel
2:     φ(p.x, p.y) = φ(p.x, p.y) + speedFucntion(p.x, p.y)
3:     if φ(p.x, p.y) ≥ 0.5 then
4:         sp1.add(p)
5:         Lz.remove(p)
6:     else if phi(p.x, p.y) < −0.5 then
7:         Sn1.add(p)
8:         Lz.remove(p)
9:     end if
10: end for
```

and Lp1 are neighbours of pixels in Lz, which makes a recomputation of the speedfunction for these pixels unnecessary. When Lz moves in one direction, Ln1 and Lp1 must move in the same direction, resulting in Ln2 and Lp2 doing the same. This process of following Lz can be accomplished in code by several means, but doing it as in the pseudocode in algorithm 2 proved to give good results. For pixels in Ln2 and Ln1 the greatest value from a pixel's four neighbours (over, under, left, right, and pixels in front and back for 3D) in *label* is found, and then that pixel is assigned the found value minus one. Similarly for Lp2 and Lp1 the smallest value of the neighbours is found, and the pixel is assigned that value plus one. In algorithm 3 Ln1 and Lp1 are updated similarly to Lz in algorithm 1, using algorithm 2 to update ϕ. Algorithm 4 shows the same for Ln2 and Lp2. Notice that the *label* and ϕ arrays of the pixels moving out of Ln2 and Lp2 are updated in algorithm 4. This is not the case in algorithms 1 and 3 because the pixels of Ln1 and Lp1 are depended on the values of the ϕ and *label* arrays of the pizel who are in Lz. Likewise, Lp2 and Ln2 are dependent on the values of Lp1 and Ln1, but no lists are dependent of Lp2 and Ln2, hence they can be updated in 4.

When all pixels moving from one layer to another layer have been adressed, pixels moving into Lp2 and Ln2 from the outside have to be added to Sp2 and Sn2. This is accomplished by simply adding all neighbours of Lp1 and Ln1 who are not part of any layer, to Sp2 and Sn2 respectively, and update their value by incrementing or decrement by one to reflect the range of the lists they are moved to. How this actually is implemented can be seen in algorithm 5, which includes the updating of the lists by their corresponding temporary list. Alogrithms 1, 3, 4 and 5 (in that order) is

Algorithm 2 How Ln2, Ln1, Lp1, Lp2 follows after Lz.

```
1:  procedure FOLLOW(p, greaterOrLess, checkAgainst)
                ▷ p = Pixel, greaterOrLess and checkAgainst are integers
2:      result = checkAgainst
3:      if greaterOrLess = 1 then          ▷ true for: Ln1 and Ln2 pixles
4:          for all n ∈ N(p) do
                        ▷ N(p) = neighbouring pixels: over, under, left, right
5:              if label(n.x, n.y) > result then
6:                  result = φ(n.x, n.y)
7:              end if
8:          end for
9:      end if
10:     if greaterOrLess = −1 then         ▷ true for: Lp1 and Lp2 pixles
11:         for all n ∈ N(p) do
12:             if label(n.x, n.y) < result then
13:                 result = φ(n.x, n.y)
14:             end if
15:         end for
16:     end ifreturn result
17: end procedure
```

the complete set of actions executed in each iteration of the segmentation process.

Algorithm 3 Update elements in Ln1 and Lp1

```
1: for all p ∈ Ln1) do
2:     if p has no neighbour that is part of Lz then
3:         Sn2.add(p)
4:         Ln1.remove(p)
5:     else
6:         M = follow(p, 1, 0)
7:         phi(p.x, p.y) = M − 1
8:         if phi(p.x, p.y) ≥ −0.5 then
9:             Sz.add(p)
10:            Ln1.remove(p)
11:        else if phi(p.x, p.y) < −1.5 then
12:            Sn2.add(p)
13:            Ln1.remove(p)
14:        end if
15:    end if
16: end for
17: for all p ∈ Lp1) do
18:    if p has no neighbour that is part of Lz then
19:        Sp2.add(p)
20:        Lp1.remove(p)
21:    else
22:        M = follow(p, −1, 0)
23:        phi(p.x, p.y) = M + 1
24:        if phi(p.x, p.y) < 0.5 then
25:            Sz.add(p)
26:            Lp1.remove(p)
27:        else if phi(p.x, p.y) ≥ 1.5 then
28:            Sp2.add(p)
29:            Lp1.remove(p)
30:        end if
31:    end if
32: end for
```

Algorithm 4 Update elements in Ln2 and Lp2.

1: **for all** $p \in Ln2$) **do**
2: **if** p has no neighbour that is part of Ln1 **then**
3: $label(p.x, p.y) = -3$
4: $phi(p.x, p.y) = -3$
5: Ln2.remove(p)
6: **else**
7: $M = follow(p, 1, -1)$
8: $phi(p.x, p.y) = M - 1$
9: **if** $phi(p.x, p.y) \geq -1.5$ **then**
10: Sn1.add(p)
11: Ln2.remove(p)
12: **else if** $phi(p.x, p.y) < -2.5$ **then**
13: $label(p.x, p.y) = -3$
14: $phi(p.x, p.y) = -3$
15: Ln2.remove(p)
16: **end if**
17: **end if**
18: **end for**
19: **for all** $p \in Lp2$) **do**
20: **if** p has no neighbour that is part of Lp1 **then**
21: $label(p.x, p.y) = 3$
22: $phi(p.x, p.y) = 3$
23: Lp2.remove(p)
24: **else**
25: $M = follow(p, -1, 1)$
26: $phi(p.x, p.y) = M + 1$
27: **if** $phi(p.x, p.y) < 1.5$ **then**
28: Sp1.add(p)
29: Lp2.remove(p)
30: **else if** $phi(p.x, p.y) \geq 2.5$ **then**
31: $label(p.x, p.y) = 3$
32: $phi(p.x, p.y) = 3$
33: Lp2.remove(p)
34: **end if**
35: **end if**
36: **end for**

Algorithm 5 Updating by using the temporary lists.

```
1: for all p ∈ Sz) do
2:     label(p.x, p.y) = 0
3:     Lz.add(p)
4: end for
5: Reset Sz
6: for all p ∈ Sn1) do
7:     label(p.x, p.y) = −1
8:     Ln1.add(p)
9:     for all n ∈ N(p) do
10:        if phi(n.x, n.y) = −3 then
11:            Sn2.add(n)
12:        end if
13:    end for
14: end for
15: Reset Sn1
16: for all e ∈ Sp1) do
17:    label(p.x, p.y) = 1
18:    Lp1.add(p)
19:    for all n ∈ N(e) do
20:        if phi(n.x, n.y) = 3 then
21:            Sp2.add(n)
22:        end if
23:    end for
24: end for
25: Reset Sp1
26: for all p ∈ Sn2) do
27:    label(p.x, p.y) = −2
28:    Ln2.add(p)
29: end for
30: Reset Sn2
31: for all p ∈ Sp2) do
32:    label(p.x, p.y) = 2
33:    Lp2.add(p)
34: end for
35: Reset Sp2
```

4.4.1 Speed function explained

Two different speed functions are implemented. These are seperatly implemented in their own methods, and a speed function is only referenced in one place in the code. This makes it easy to implement new speed functions, and to change between which of them to use when running the program. The speed function methods takes in as parameters the coordinates of the pixel to calculate the speed chnage on, and returns a value which then is added to the speed from the last iteration. The two speed functions implemented are the ones explained in chapter 3. The simplified Chan-Vese speed function was first implemented, and only used to test whether the rest of the implemented sparse field code worked as expected. Because this simple function behaves much like a region grow function it will not be a part of the discussion in the result chapter.

The other speed function was implemented as explained in chapter 3, except for some parts that were dropped because it was not needed in this verion of the sparse field. A short description of how the speed function is calculated is shown in algorithm 6

Algorithm 6 Speed function calculation.

```
1: procedure SPEEDFUNCTION(p)
2:     Calculate the data term
3:     Calculate first order derivatives
4:     Calculate second order derivatives
5:     Calculate normals
6:     Calculate the curvature
7:     speed = -α * dataTerm + (1-α) * curvature);
8: end procedure
```

Notice that the α * $dataTerm$ is multiplied with -1. Assume that a point in the zero level set have its value in ϕ increased by a value that would make it be transfered over to the Lp1 layer. This means that a point that was in Lz is now part of Lp1, and its neighbour that was Ln1 is now Lz, i.e. the zero level set have contracted. This is the opposite of the wanted behaviour, and thus the α * $dataTerm$ term is set to -α * $dataTerm$. This is normal in an implementation of this speed function, and not something used only in this project.

To improve the speed of the evalution process of the zero level set the calculation of the data function (defined in 3.5) was modified. The modified version of the data function divides the result of the previously defined data term by ϵ. This makes the data term which before had an range of $\{-1, \epsilon\}$

to get a new range of $\{-1, 1\}$ (after clamping the minimum range to -1), which greatly improves the speed. This modification makes the segmentation process go much faster (less iterations needed for a full segmentation) and the only difference for the user is that the α values used in the speed function have a different scaling. This modification will be discussed in more detail in the discussion (chapter 6).

New speed functions can be implemented and easily merged with the rest of the code, but an importatnt factor that must be remembered is that the value returned from the speed function must be in the range $\{-1, 1\}$ to reflect therange of the layers. Another important factor is that the lists representing the layers must support equal sized range-width (< 1) because the speed function is calculated the exact same way for all elements regardless of which layer it is a member of.

The calculations needed for the computations of first and second order derivatives and the normals for the speed function are in a header file. By keeping these outside the speed function, the calculations can be reused in any other speed funtion that may be implemented in the future.

4.5 Problems met

As previously mentioned, when looking at figure 4.3 it can be clearly seen that something is wrong with how the lists (the layers) are arranged. This becomes even more clear when looking at figure 4.5 which shows only the zero level set. The zero level set in figure 4.5 is the segmentation result

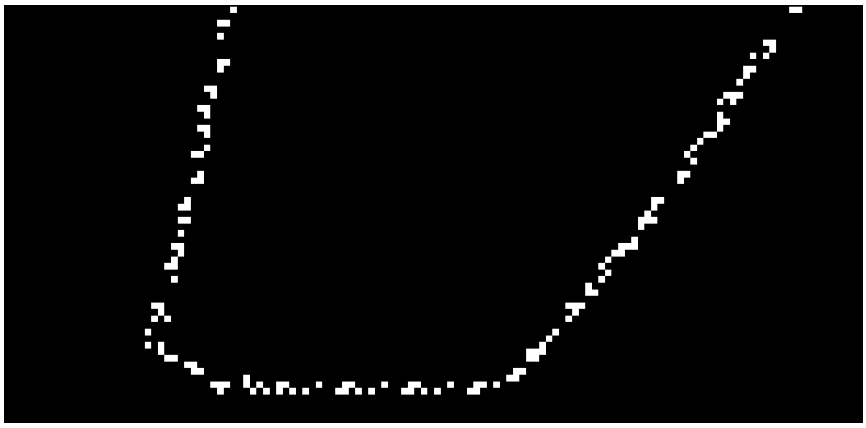


Figure 4.5: Zero level set corresponding to the label image in figure 4.3.

(zoomed in) that corresponds to the label image in figure 4.3. The zero

level set is supposed to be a one pixel wide continous line, which is not true in this case. Several problems and bugs in the code combined were reasons were for this result. Much time and effort was used to debug this and to fix these problems. In addition to creating artifacts in the results, the bugs also made the program run slower, which made the debugging process even more time consuming. The main problem was that the layers were not of equal range-width and that the speed function was not normalized to be within the range $\{-1, 1\}$, which will now be explained in more detail.

When an element in any of the five layers is updated by the speed function the new value may not reflect the range at which is allowed for the layer it is part of. In that case it have to be moved to another layer or in case the value is not in the allowed range of any of the layers removed from its current layer and not added to any other. The problem caused by the value returned by the speed function not being normalized was that elements in any layer was able to be transferred from its previous layer to a layer that is not a neihbouring layer. For example, transferring a pixel from Ln1 is restricted to its neighbouring layers of Ln2 and Lz. But if a pixel A in Ln1 with value -0.65 had its value increased by 1.2, its new value of 0.55 would indicate that it should be moved to Lp1, jumping over Lz.

As can be seen in table 4.2 all the lists have the exact same range-width of < 1, which is not the case in table 4.1. If the ranges in table 4.1 is used it will disort the segmentation process. This happens for example when an element in Lz have the value -0.5 and is increased by 1 by the speed function. A result from the speed function with value 1 (or -1) indicates fast movement and that element should be moved to Lp1 (or Ln1). But according to table 4.1 that will not happen in Lz when the value is -0.5 (or 0.5), even if a change in 1 of an element in any other layer would definetly move it out of that layer. But even if an element that should be removed is not removed, an element from either Ln1 or Lp1 is moved into Lz (which is correct behaviour), hence the Lz becomes two pixels wide. An example layer image of this is illustrated in figure 4.6.

4.6 CUDA Implementation

The sparse field level set method was parallelized by implementing it in CUDA. This process proved to be somewhat more complicated than creating a serial sparse field program. The sparse field method is as mentioned an optimized version of the narrow band method that focuses on using dynamic arrays (linked lists) to hold the elements needed for computation. This is a very serial way of thinking, and parallelization was not a factor when sparse

Figure 4.6: Layer image with Lz two pixels wide.

field method was created.

The most important change in the CUDA implementaion was the addition of another array in the same size as the image to be segmented. This array, called *layer*, is used as an replacement for the five lists and their corresponding temporary lists. This change was made because of two reasons. The first reason is that the only dynamic array structure supported by CUDA (as of May 2013), called thrust and is a C++ template library for CUDA[19], does not support resizing within the device. This is in itself is a reason to not use arrays (dynamic or not) to represent the different layers when the code is structured as it is in the serial versions. The other reason is that the use of an array of same size as the image to be segmented, with each coordinate being handled by a single thread in the GPU utilizes the power of paralleilsm that the GPU provides much better.

The most used method to utilize the parallel capabilities of GPUs when working with multidimensional arrays is to split the array into small tiles, each manageable by a CUDA block, and do the processing using the shared memory. This avoids too much use of the slow global memory whic can take hundreds of clock cycles to load data. Comparing this to the 1-2 cycles needed to access data in the local memory a huge difference in speed can be achieved by using the local and shared memory. But doing it this way does

however require an almost complete re-implementaion of the code, which due to the limited time was not achievable in this project. Instead an array was used as a whole without splitting it up.

This is where the *layer* array mentioned above comes in. This array contains integer values corresponding to each value in the *label* and ϕ arrays. Each element in *layer* who are member of any of the five layers is represented by a two digit number, and the rest is zero. The first of these two digits is is either 1 or 2. The second digit represents the layer in which that element is a part of, if the digit is 3, 4, 5, 6, or 7 it means that the element is part of Ln2, Ln1, Lz, Lp1 or Lp2 respectively. If the first digit is 2, it corresponds to the element being part of a temporary list. For example, if $label(x, y) = 25$ then the element with coordinate (x, y) in the ϕ array is part of what in the serial version was called Sz. The decision to make a somewhat unusual array like this was taken to avoid a set of conditional checks in the code, in addition to the resulting consumption of less memory. Using an array like this instead of linked lists makes this implementation close to an extreme narrow band implementation, though the pixel are processed in the way of a sparse field implementation. The only other change from the data-structures used in the serial implementation is the removal of the C++ struct called *Pixel*. This struct only contained the coordinates of elements in any of the five layers, but this is not necassary when not using lists. Apart from these changes the CUDA code is very similar to the serial version, hence there is no difference beteween results of full segmentations from the serial and parallelized version.

4.7 Performance

Both C++ and Matlab were candidates languuages to implement the level set function in. The adavatage of using Matlab is the simple syntax used for mathematical operations and the ease of loading/writing and displaying images in both 2D and 3D. But ultimately C++ was chosen because of its advantages in speed and the possibility of parallelization. The performance improvements to be discussed in this section was all performed before the implementation of CUDA code. This section will only discuss changes made to improve performance in the serial versions of the code. A comparision of the performance of the serial and CUDA versions of the program will be discussed in chapter LINK TIL ENTEN RESULT ELR DISCUSSION HER.

Several improvements to increase the performance were made after a working 3D version was complete, some which gave insignificant or small

performance increases, and a few which greatly improved runtime. One of the changes made to achieve significantly improved runtime was as simple as changing all structures defined as *double* to $float$. In many cases this change may seem insignificant, but in this case with data structures of sizes as big as 512^3 and several linked-lists with hundreds of pixels being pushed and popped each itereation, the change reduced the runtime significantly. By changing from using double values which take up 8 byte each, to using float which uses 4 bytes, the memory usage of the array and list structures was reduced by nearly 50%. A chance that improved the runtime even more significanly was the replacement of the C++ datastructure $STLvector$ with the datastructure $STLlist$. When the implementation process started $STLvector$ was chosen as the container for the elements in the different layers, without considering any other candidates. The runtime in 2D using *vwctor* was not considered slow, hence vector was also used for 3D. But due to the slow speed of the 3D version (when using *vector*) changes were needed. One improvement was the above-mentioned double to float change, and even if this improved performance greatly, more changes that could improve performance were sought after. This resulted in the replacement of the *list* container with the *vector* container for the pixels in the different layers. Some of the advantages and disadvantages of using the *list* and *vector* data structures are summarized in table 4.3 and 4.4.

Vector	
Advantages	Disadvantages
Insertion/erasure from the end uses constant time. Efficient accessing of its elements.	Insertion/erasure from other than end is costly (O(n)).

Table 4.3: Advantages and disadvantages of C++ *std* :: *vector*

List	
Advantages	Disadvantages
Fast insertion, extraction and moving of elements in any position.	Consume some extra memory to keep the linking information associated to each element. Cannot access elements by their position.

Table 4.4: Advantages and disadvantages of C++ *std* :: *list*

The reason for the drastical inmprovement in performance when changing from *vector* to *list* is the removal of the overhead associated with inserteion and erasure of elements not at the end when using *vector*. After the first few iterations, these two actions happens hundreds of times per iteration, and by changing to *list* this overhead along with the smaller $log(n)$ overhead when increasing the size of the vector is eliminated. The speedup gained by changing the element types from using double to float and the speedup aquired when replacing *vector* with *list* is shown in table 4.5. Note that even though the *double* to $float$ change was made before the *vector* to *list* change, the numbers in the *double* $\rightarrow$ $float$ column of the table were all aquired after the *list* structure was implemented.

	double $\rightarrow$ $float$	*vector* $\rightarrow$ *list*
2D (512x512)	1.52 $\rightarrow$ 1.31 sec	2.08 $\rightarrow$ 1.31 sec
3D (256x256x256)	3,44 $\rightarrow$ 2.12 min	109.29 $\rightarrow$ 2.12 min

Table 4.5: Runtime improvements in 2D and 3D.

Another change that was considered but later dropped, was to replace the use of *list* with $std :: forward_list$. This structure was considered due to its slightly less overhead when inserting and removing elements which makes it more efficient than *list*. But this improvement in insertion and deletion time over *list* comes as a consequence of the fact that $forward_list$ is a single linked list, and is thus not able to point to the previous element in the list. The sparse field level set methd can be implemented using single-linked lists instead of double-linked lists, but the implementation in this project depends on the lists being double-linked.

4.8 Third party libraries and programs used

Third party libraries for I/O

In both 2D and 3D version third party libraries were used to read and write input and output data. In the 2D version a simple open source (under the revised BSD license) C++ library called EasyBMP ([17]) was used for easily reading and writing Windows bitmap (BMP) image files. In the 3D version the Simple Image Processing Library (SIPL, [18] created by the co-supervisor for this project, Erik Smistad, is used. SIPL is a C++ library that among other features allows simple load and store of volumes of different types. In addition to volume (and image) processing it supports visualization of the data. In this project SIPL is used for reading and storing

medical volume data by reading directly from raw data or by using an mhd metafile. It is also used for visualizing 2D slices of the input volume and segmentation result for comparision.

Program for visualizing volumes

In addition to SIPL to visualize the volume data, a Matlab plugin named Viewer3D [20] was used. Viewer3D enable the segmentation result to be superimposed on the the original volume, either as 3D volume rendering or 2D slicing. In the final stage of the project an open source program named Amide [21] was used, which in addition to the features previously mentioned also supports customizable coloring of objects.

Chapter 5

Results

In this chapter the results of multiple runs of the program will be discussed, both in 2D and 3D. First some runs in 2D will be discussed along with how the variables in the speed function affects the segmentation. Then some results from 3D runs will be illustrated, before the performance of different runs are compared. The Chan-Vese speed function implemented will not be discussed in any detail because the very simplified version implemented behaves much like a simple region grow function.

A note about the values used to get the segmentation results in this chapter, the values used for the speed function were found to give a good result when manually comparing to the input images/volumes, and may or may not be the optimal values for speed and accuracy. Also note that the colors used in the volumes are only to clearly illustrate the differences between input and result, or between number of iterations.

5.1 2D

First a simple binary image of size 512x512 of a circle shown in figure 5.1a was segmented. The red dot in the middle represents the seed point chosen, and is not part of the image (superimposed). The values used for the speed function were: threshold $(T) = 0.99$, $\epsilon = 0.15$ and $\alpha = 0.80$. Since this image is binary the $T \pm \epsilon$ would not affect the end result of a full segmentation, as long as $T - \epsilon < 1 < T + \epsilon$. This also assumes that $T - \epsilon$ is not too close to 0, which would (also depending on α) either stop the surface evolution midways or collaps it. The advantage of using higher values of ϵ within these limits is that higher values of ϵ makes the segmentation process faster by needing less iterations to achieve full segmentation. The reason is that the data term $D(I)$ (see 3.5) in the speed function is gradual, as mentioned

when describing the speed function in chapter 3.

Figures 5.1 b, c and d represents the zero level set after 700, 1200 and 1600 iterations respectively. Figure 5.1e illustrates the full segmentation result which required 2250 iterations.

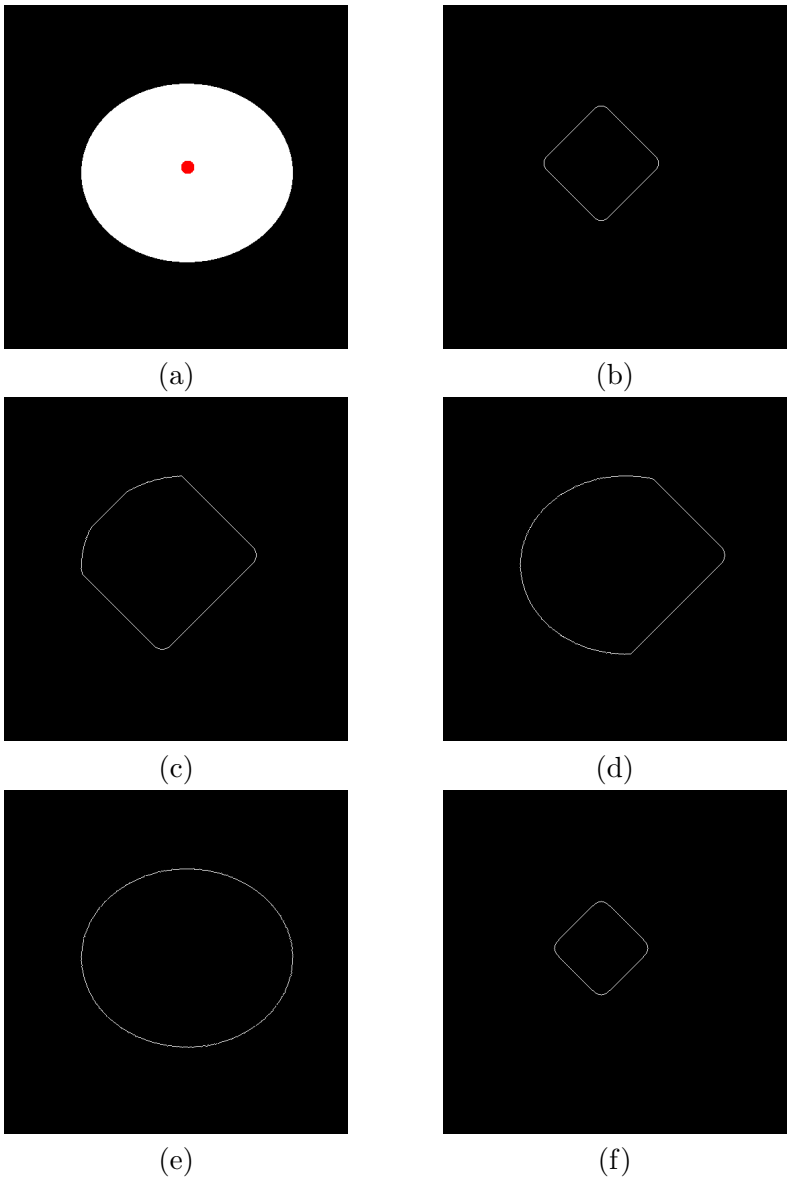


Figure 5.1: (a): Original image with seed point. Zero level set after: (b): 600, (c): 1200, (d): 1600 and (e): 2200 iterations (e): with $\epsilon = 0.05$.

By comparing the input image in figure 5.1a with the segmentation result in figure 5.1e it can be seen that the program succesfully segmented the image. To measure the difference in iterations needed to get a full segmentation with a different value of ϵ an additional segmentation with all variables equal to the previous segmentation except for ϵ was performed. By assigning ϵ a value 0.05 the result was the exact same after full segmentation, but the numbers of iterations for full segmentation increased by nearly four times, from 2200 to about 8600. By omparing figure 5.1e with figure 5.1f which is the result after 2200 iterations with $\epsilon = 0.05$, it can be seen how much slower the interface evolves. And as expected, increasing ϵ (to 0.30) decreased the numbers of iterations needed (to 1100).

To test if the program works as it should when parts of the seed point is outside the object to be segmented, the seed point was set as shown in red in figure 5.2a. How the interface looked like after 300, 800 and 1300 iterations is depicted in figure 5.2b, c and d respectively. The final segmentation result was as expected a correct segmentation of the object as in figure 5.1e.

To further test the robustness of the program and to illustrate the effect α has on the smoothness of the interface the 512x512 binary image in figure 5.3a, with the seed point superimposed in red, was segmented. Notice the one-pixel wide "cut" at the top that seperates the main object in the image from the smaller one. Also notice the one-pixel wide line that holds together the main object with the rectangle at the button. Two full segmentations were run, both with $T = 0.99$ and $\epsilon = 0.15$, figure 5.3b is the result with $\alpha = 0.65$ and 5.3c with $\alpha = 0.90$. As explained in chapter 3, α restricts how much the interface can bend and prevents the model from leaking into unwanted areas, which can be seen by the fact that 5.3c with a higer value of α have been able to include the rectangle at the button by evolving through the thin line, while 5.3b did not manage it. (Note that the modified speed function was used in this example, this will be discussed in the next chapter).

Higher values gives more importance to $D(I)$ and lower values makes $\nabla \frac{\nabla\phi}{|\nabla\phi|}$ affect the level set more. The more importance $\nabla \frac{\nabla\phi}{|\nabla\phi|}$ gets, the less likely is the model to leak into unwanted areas. But giving $\nabla \frac{\nabla\phi}{|\nabla\phi|}$ too much importance makes the model so smooth that it does not reach all areas of the object being segmented. This is illustrated in figure 5.4b, which is the segmentation result of the image in figure 5.4a using $\alpha = 0.4$. These two figures are of small size (100x100) to clearly illustrate the effects of α on the smoothness of the segmentation result.

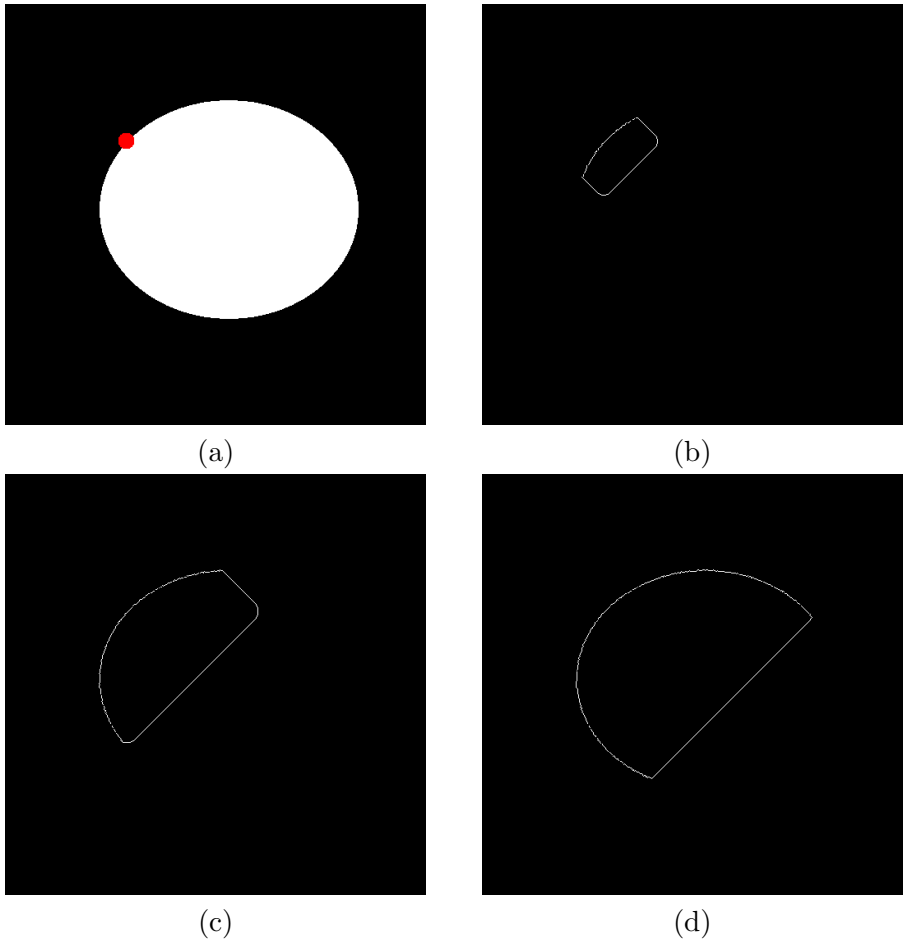
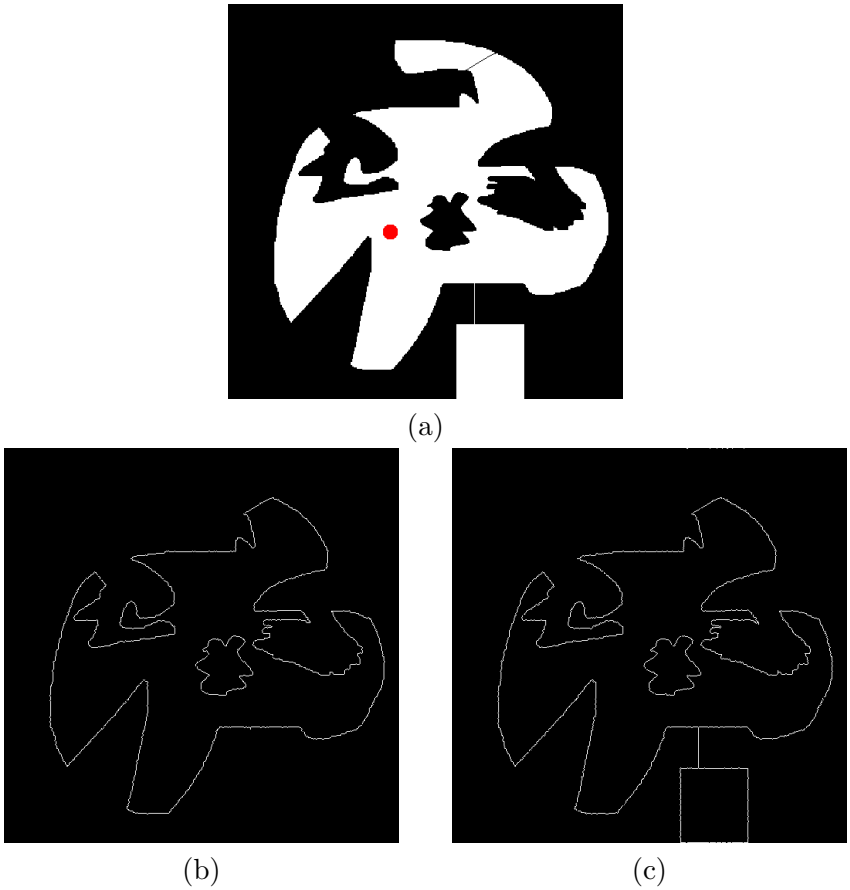


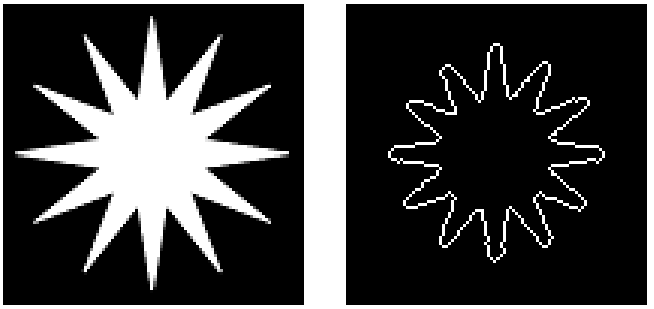
Figure 5.2: (a): Seed point partly outside the object, superimposed on the input image. Interface after (b): 300, (c): 800 and (d): 1300 iterations.



(a)

(b) (c)

Figure 5.3: (a): Input image with seed point superimposed. Segmentation result with (b): $\alpha = 0.65$, (c): $\alpha = 0.90$.



(a) (b)

Figure 5.4: Interface smoothness highly valued. (a): Input image, (b): segmentation result.

5.2 3D

First volume to be segmented is a volume of an aneurism in a "semi segmented brain volume". The goal is to segment the aneurism itself as well as

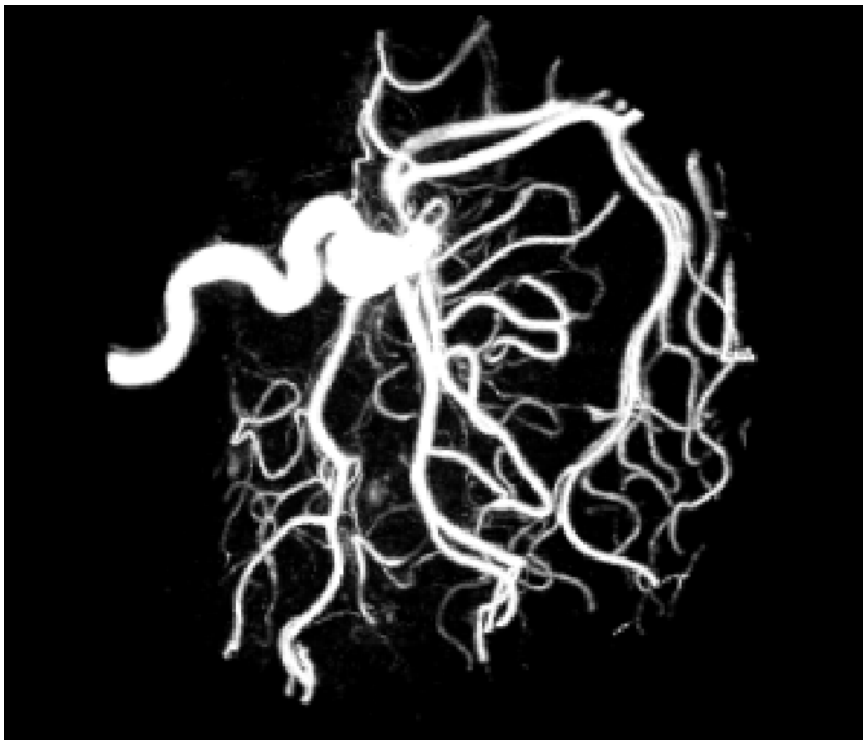


Figure 5.5: Maximum intensity projection of the volume to be segmented

adjacent connected arteries without expanding into insignificant parts of the volume. In figure 5.5 the maximum intensity projection of the unsegmented volume is depicted.

Figure 5.6 shows how the segmented image looks like after 500 iterations. The values used are $T = 1.0$, $\epsilon = 0.3$ and $\alpha = 0.75$. The dimension of the aneurism volume is 256x256x256 and the maximal expanding speed the interface can have using the implemented speed funciton is 1 pixel per iteration. In theory this would mean that only about half the number of iterations of the greatest image dimension is needed (assuming the seed point is located near the center of the volume) in order to achieve convergence. But this assumes an average speed of 1 pixel per iteration, which in practice does not happen. The speed is reduced by both the curcature of the object being segmented and the intensity values of the neighbouring pixels. And

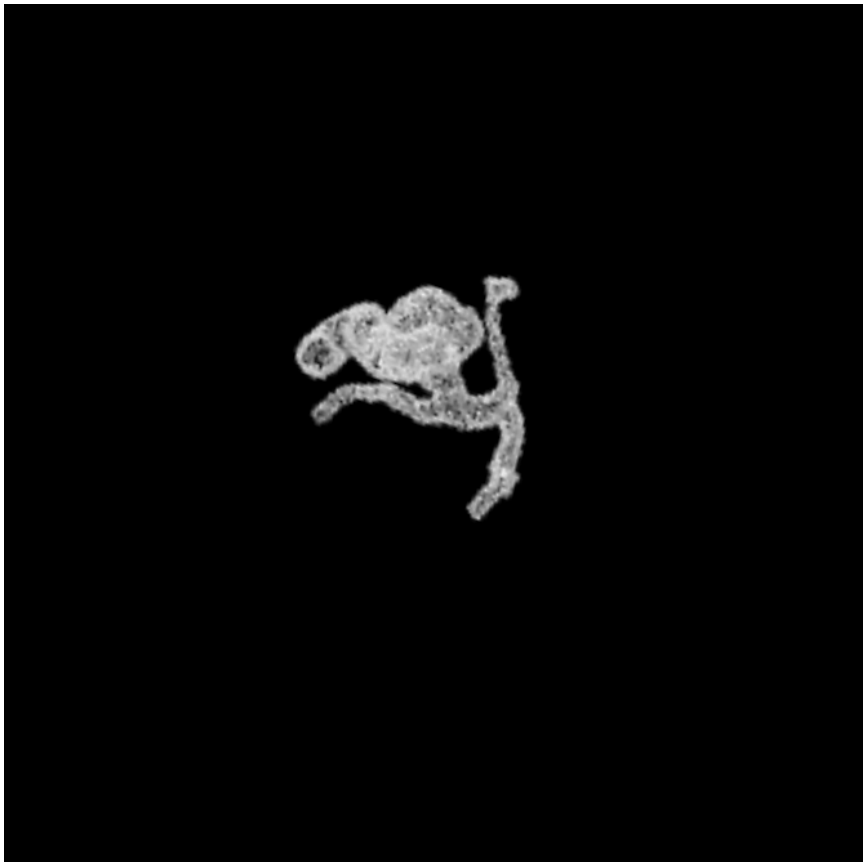


Figure 5.6: Aneurism segmentation after 500 iterations.

in this case, whith an aneurism volume with narrow paths, the curvature greatly reduces the speed of the evalution process. It turns out that to achieve a satisfying result of the aunerism volume with the given the input parameters stated above, about 3000 iterations is needed.

Figure 5.7 shows the segmentated volume after 500 ((as in figure 5.6)), 1500 and 3000 iterations. The figure shows how stable the algorithm is throughout the run. The area along the walls of the arteries covered after 500 iterations is not retracting at a later point, nor is it expanding further. This is shown by observing how well the 500 iterations volume and the 1500 iterations volume overlap.

Next, the original volume (not MIP as in figure 5.5) in gray is compared to the result after 3000 iterations (in blue). The completely gray arteries are not connected to the aneurism where the seed point was set and are thus not

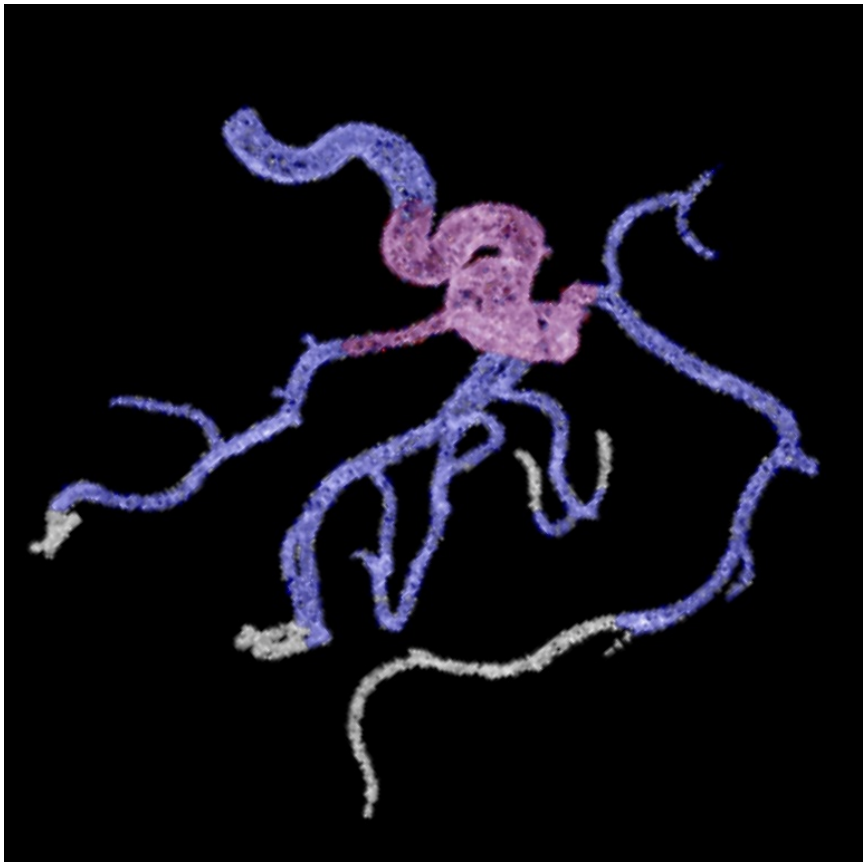


Figure 5.7: Aneurism segmentation after 500 (red), 1500 (blue) and 3000 (gray) iterations.

reachable unless additional seed points are set at their locations. But apart from those arteries it can be seen that the segmentation result have been able to access the majority of the arteries, except for a few locations (e.g. at the top right corner) which was too narrow for the level set to access given the current values used in the speed funcion. Some segmentations given different values were executed to access these areas, but that resulted in the level set leaking into other areas not connected to the seed location.

Two more volumes were segmented using the implemented sparse field method, but these two were segmented using the modified speed function, thus the number of iterations and α values used for these two cannot be directly compared to the segmentations mentioned above.

The first among these two is a T1-weighted MRI volume of a head,

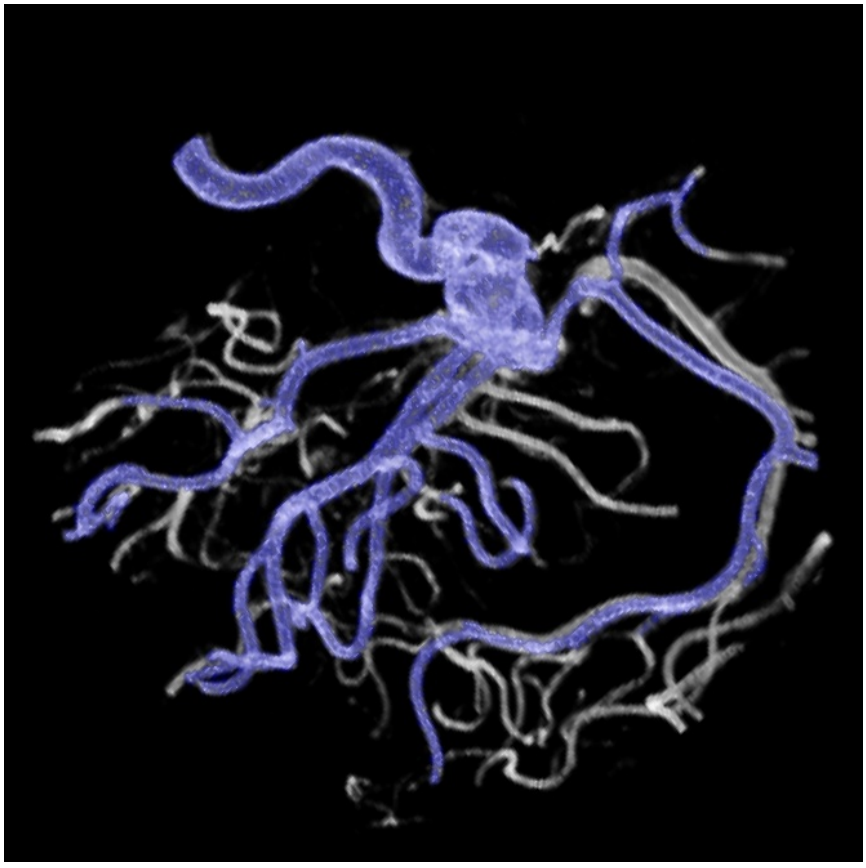


Figure 5.8: Aneurism: original volume in gray and segmentation result after 3000 iterations in blue.

which was segmented to extract out the brain. The head volume along with the segmentation result of the brain is illustrated in figure 5.9. The number of iterations needed to achieve this result was 700. Figures 5.10a and b illustrates the segmentation result as a superimposition on the original volume along different coordinates of the z-axis. Figure 5.11 is the slice of a similar segmentation result with a slightly higher α resulting in leakage at the bottom left.

The last volume tested is a CT volume (with dimensions 320x220x72) of an abdomen where the goal is to extract out the volume of the liver. This process is somewhat tricky because the liver has greyscale values very similar to the organs surrounding it, making it difficult to prevent the interface from leaking into the surroundings. Although the curvature term will prevent

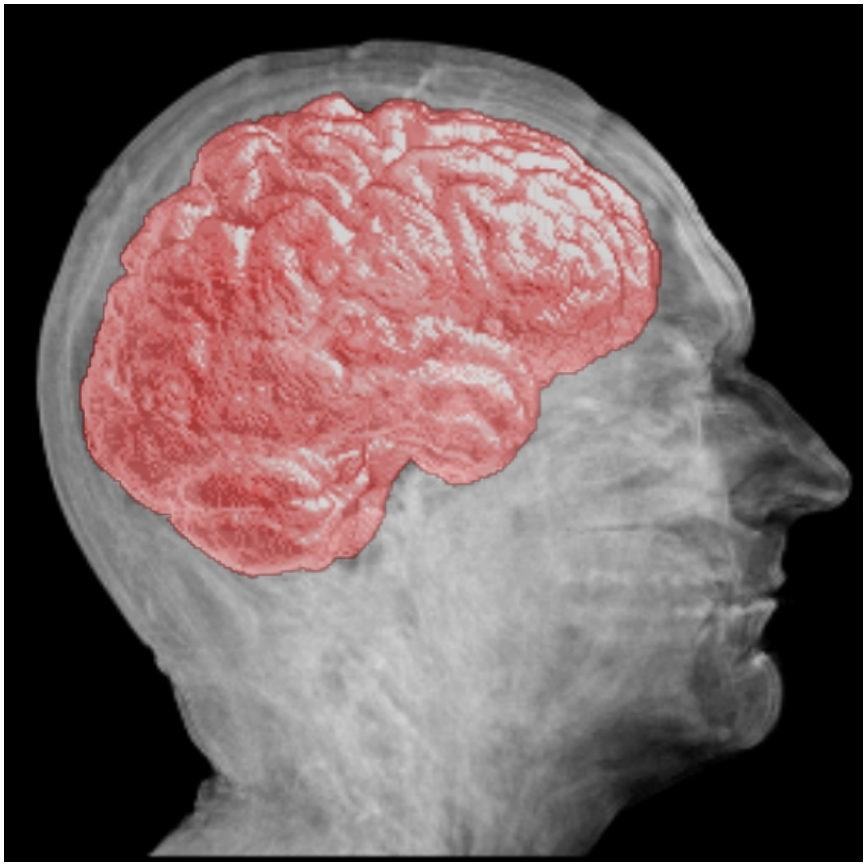
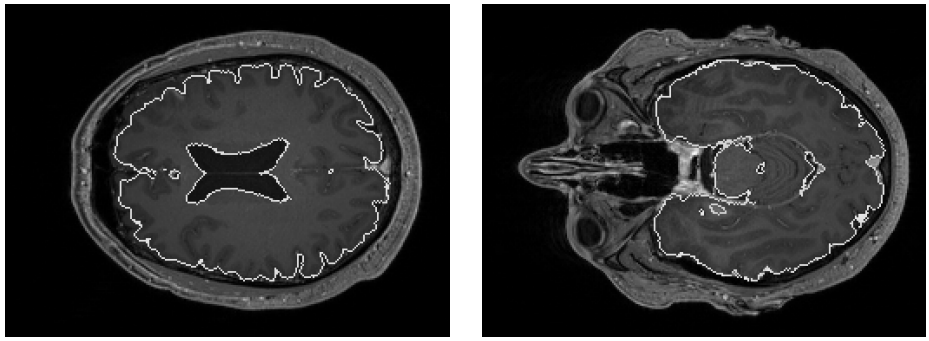


Figure 5.9: Original volume of head and segmented brain volume in red.

pixel sized leaks and small holes, it will not prevent a broad part of the interface to move out of the liver volume unless the curvatures is weighted heavily. In addition, the internal structures of the liver migh vary even more than the liver and its surroundings. The parameters must thus be chosen wisely to get a good segmentation. Figure 5.12 illustrates a slice in the z-axis of the volume with the zero level set superimposed. Dark parts of the figure have low intensity values, and lighter values have higher intensity values. The liver is the big semi-uniform area that stretches from the left and over the middle of the stomach. This is not a full segmentation, but it clearly displays how different the intensity values witin the liver are. It can be seen how the little dark dot in the middle at the bottom have hindered the zero level set to evolve further to the right. This could have been fixed by increasing α and make it less smooth or by increasing ϵ and include the



(a) (b)

Figure 5.10: Slices along the z-axis of the brain segmented volume, superimposed on the original volume.

Figure 5.11: Slightly higher α, resulting in leakage.

dark dot in the segmentation, but both these alternatives would result in leakages over to other neighbouring organs. An example of this is displayed in figure 5.13.

5.3 Performance

Testing environment

The tests were performed on a laptop PC with the specifications as described in table 5.1. The software related specifications are Windows 7 as operative

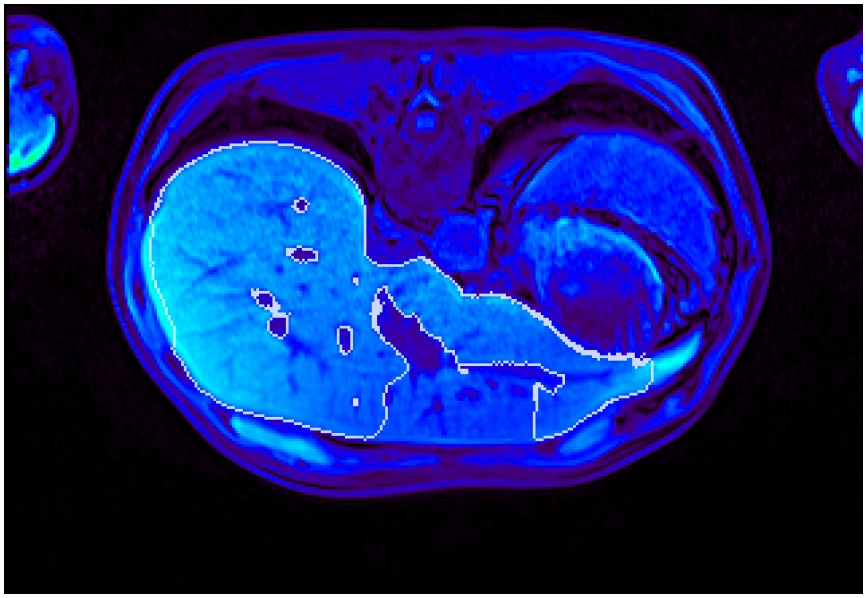


Figure 5.12: Slice of CT liver volume in the z-axis with result superimposed.

system, g++ version 4.6.2 as C++ compiler, and CUDA compute capability 5.0 with nvcc version 0.2.1221 as compiler.

PC the tests were executed on	
CPU model	Intel Core i5-320M
Cores in CPU	2
CPU frequency	2.5GHz
Memory	4GB
GPU model	NVIDIA GeForce GT 630M
Cores in GPU	96
GPU memory	1GB

Table 5.1: Specifications of the PC the tests were performed on.

The performance results of the 2D serial and 2D CUDA versions are described in table 5.2 and 5.3, with results both before and after the modification of the speed function (equal results). The results in these tables were all aquired using a single seed point set at an optimal location close to the center of the object being segmented. The values used for the speed function were found to be the ones resulting in a correct and complete segmentation. The time was taken for only the segmentation process, thus not

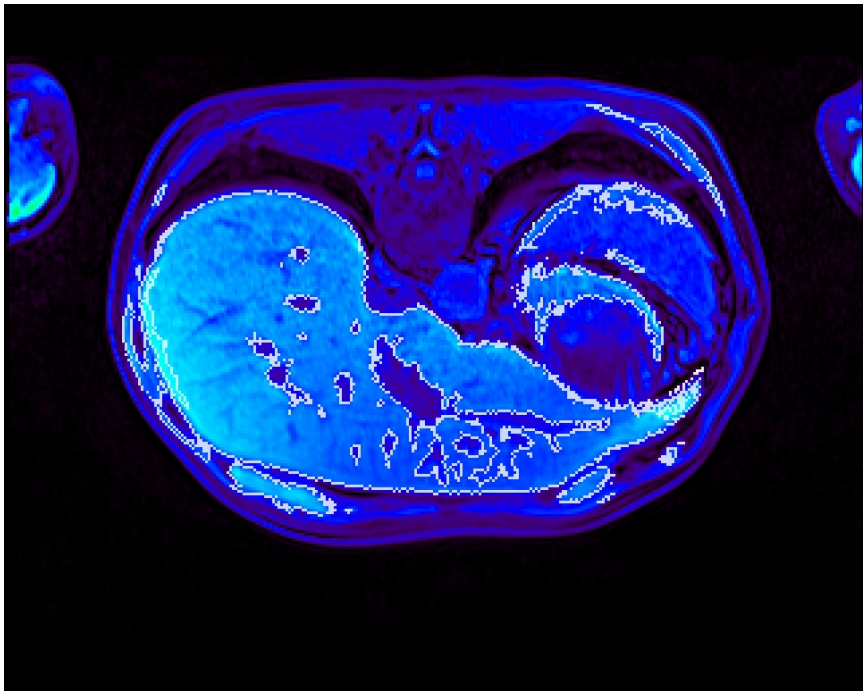


Figure 5.13: Another slice of liver volume, with leakage.

including the time used to read input, initialize and write result back to file.

Input image 5.1a	2D serial		2D CUDA	
	Unmodified	Modified	Unmodified	Modified
Iterations	2300	320	2300	320
T	0.99	0.99	0.99	0.99
ϵ	0.15	0.15	0.15	0.15
α	0.80	0.80	0.80	0.80
Time (seconds)	1.31	0.37	2.56	0.14

Table 5.2: 2D perfromance results on figure 5.1a as input.

Table 5.4 describes the results for the 3D version for the aneurism volume and the results (modified/unmodified) are equal.

Input image 5.1a	2D serial		2D CUDA	
	Unmodified	Modified	Unmodified	Modified
Iterations	6500	4000	6500	4000
T	0.99	0.99	0.99	0.99
ϵ	0.15	0.15	0.15	0.15
α	0.60	0.165	0.165	0.60
Time (seconds)	12.29	12.27	7.68	4.30

Table 5.3: 2D perfromance results on figure 5.3a as input.

Input volume 5.5	3D serial		3D CUDA	
	Unmodified	Modified	Unmodified	Modified
Iterations	3000	2100	N/A	N/A
T	1.0	1.0	N/A	N/A
ϵ	0.3	0.25	N/A	N/A
α	0.75	0.4	N/A	N/A
Time (min/sec)	2.12	1.14	N/A	N/A

Table 5.4: 3D perfromance results on figure 5.5 as input.

Chapter 6

Discussion

6.1 Modification of the speed function

Although the level set method is good at handling leakages when the curvature term is taken into account, and naturally deals with merging and splitting of the interface, it has its shortcommings. If for instance the object to be extracted has a greater internal intensity difference than the border of the object and neighbouring objects, the interface might leak out of the object in one region and at the same time refuse to grow in another. Figure 5.13 is an example of this behaviour. To reduce the effects of this weakness it is possible to combine different segmentation methods to benefit from their strengths and avoid their shortcommings. Better segmentation using the level set method alone however might be possible by modifying the behaviour of the speed function.

As mentioned when discussing the speed function in chapter 4 the data function was modified a little from the one described in chapter 3. The reason for this modification is that under certain circumstances the zero level set moved very slow using the original speed function. This is true especially when a low ϵ value is used, because the maximum speed of the data term (which was shown in figure 3.4) is ϵ.

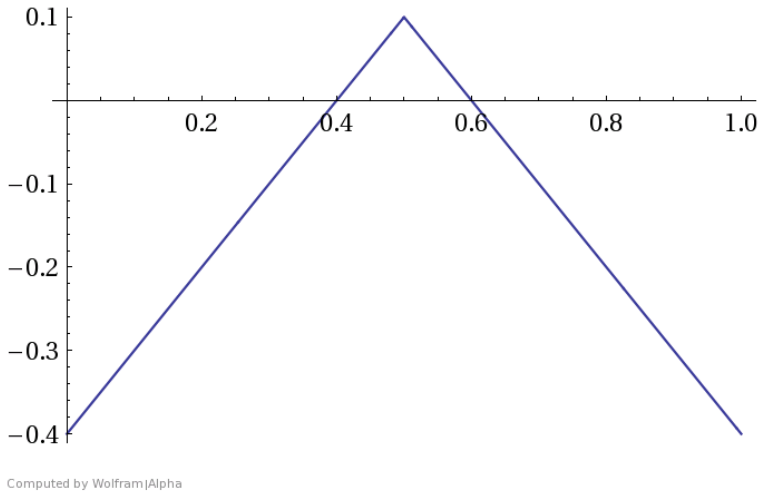


Figure 6.1: Original data term, $D(I)$, with $\epsilon = 0.1$.

Hence, with a small ϵ and the data term weighted high (high α value) the resulting speed is potentially only a fraction of its maximally possible value. As an example, assume $\epsilon = 0.1$, which makes the maximum speed resulting from $D(I)$ equal to 0.1, this is shown in figure 6.1. In areas where the curvature is low or D(I) is weighted heavily, the interface would be moving at about 10% of max speed.

By dividing the data function by ϵ its maximally positive value is increased from ϵ to 1. The new data function can be seen in equation 6.1.

$$D(I) = \frac{\epsilon - |I - T|}{\epsilon} \tag{6.1}$$

The original data function's smallest possible value is ϵ-1. Thus, in the modified data function ϵ values smaller than 0,5 will result in values less than -1. Hence, the result of the equation has to be clamped to -1 for $D(I) <$ -1.

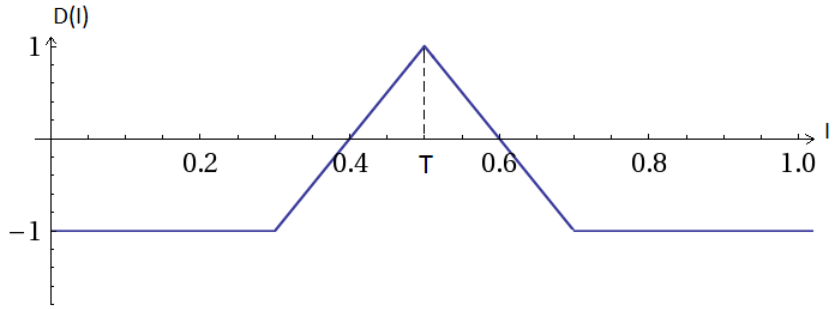


Figure 6.2: New data function

The graph in figure 6.2 illustrates the value of the new $D(I)$ as the intensity increases form 0 to 1. Threshold = 0.5 and $\epsilon = 0.1$ is chosen to clearly show how negative values are clamped.

As an alternative to equation 6.1, $D(I)$ could have two different functions depending on whether it is positive or negative. Positive values would be divided by ϵ, and negative values would be divided by 1-ϵ to give both positive and negative values a linear increase towards the maxmium value of 1.

$$D(I) = \begin{cases} \dfrac{\epsilon - |I - T|}{\epsilon} & if\ I\ is\ inside\ [T \pm \epsilon] \\ \dfrac{\epsilon - |I - T|}{1 - \epsilon} & if\ I\ is\ outside\ [T \pm \epsilon] \end{cases}$$

In practice however, this is not nescessarily as elegant as it looks. Even though the maximal theoretical value of this $D(I)$ function is ± 1 the conditions required to get maximum speed is unusual and does not often occur in practice. It requires the threshold and intensity to be on oposite sides of the intensity domain, for instance $T = 1$ and $I = 0$. Because resulting values from this $D(I)$ function close to -1 only appear in extreme cases, the interface will grow away from negative regions slower than it will grow into positive ones.

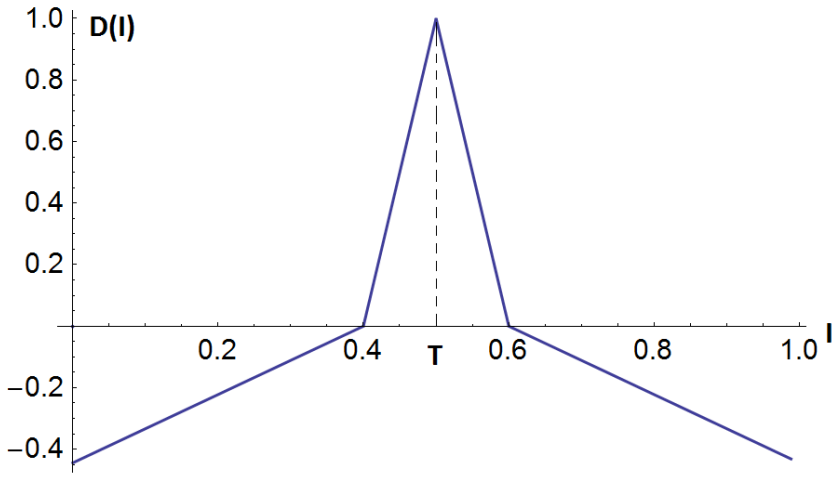


Figure 6.3: Alternative D(I) function

Figure 6.3 shows this difference in growth for negative and positive vaules. Although segmentations performed in this project uses equation 6.1, it does not mean that the alternative equation is inherently bad, it just has a different behaviour.

Equation 6.1 was used in this project becuse of its constant change in speed for both positive and negative values. This makes $D(I)$ easier to reason about in order to get good results. Because $D(I)$ now has a much higher max value it will contributes alot more to the speed and have a greater impact on the result. The modified speed function will therefore need different α values to reflect this change. To perform a segmentation with this modified $D(I)$ to get the same result as from a segmentation which used the unmodified $D(I)$, α needs to be smaller. Apart from the improved execution time, using such an α value with the modified $D(I)$, the end results will not be affected.

As with the data term, the curvature term (C) can also be altered to achieve different behaviour. The curvature term can for instance be multiplied by a factor n, before it is added to the data term in the speed function. For $n > 1$ it would then have to be clamped to keep its maximal possible value at 1.

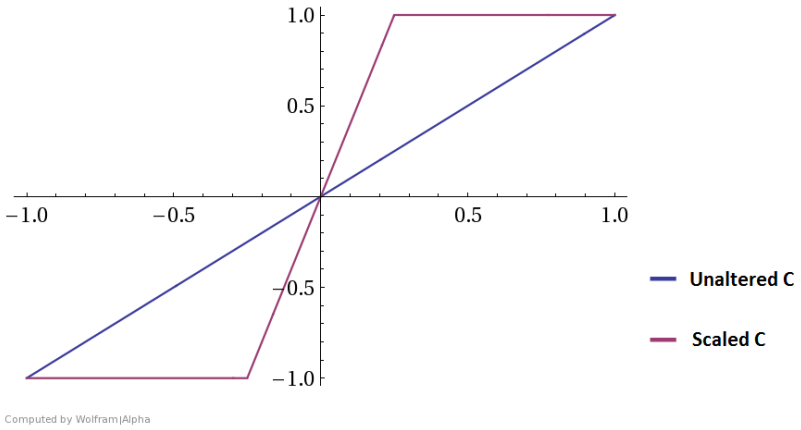


Figure 6.4: Scaled curvature function. Unaltered curvature is shown in blue and altered is shown in red.

Such a curvature function is plotted in figure 6.4, where $n = 4$ to show the increase in steepness. The old curvature is shown in blue and the scaled in red. This type of scaling would make changes in curvature have a bigger impact on the movement of the interface, so effectively it resembles the effect of decreasing the α term to weigh the curvature more.

Another and perhaps more interresting way of altering C is to approach it from the same angle as the data term, and have it grow differently depending on whether it is positive or negative. One of the problems with the level set method, as mentioned in the section above, is its poor performance when the difference in intensity within the object to be extract is greater than the difference between pixels inside and outside the object. If the curvature is scaled differently depending on whether it is positive or negative, it could potentially force the interface into regions of the object it would not normally grow into. It has the following definition:

$$D(I) = \begin{cases} C & if\ C\ is\ positive \\ C\,n & if\ C\ is\ negative \end{cases}$$

The resulting graph is shown in figure 6.5.

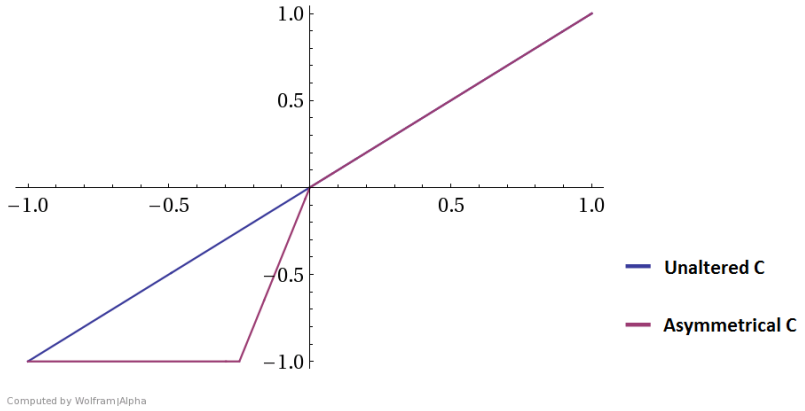


Figure 6.5: Asymmetrically scaled curvature function. Unaltered curvature is shown in blue and altered is shown in red.

This alteration would cause the interface to grow into regions where the curvature pushes it forward, and at the same time prevent the curvature from restricting the growth too much. The level of detail would decrease because holes in the segmentation would be filled more easily, but in return the segmentation might include more of the object given that the space it normally fail to grow into is at least partially surrounded by the interface.

6.2 Problems with the CUDA implementation

As described before, the updating process of the layers are dependent on each other. Ln2 and Lp2 are dependent on Ln1 and Lp1 respectively, while Ln1 and Lp1 depends on Lz. Thus even if all the calculations in each element of the zero level set can be parallelized, all operations in the other layers have to wait. One way to overcome this when in a parallel context is to use barriers to synchronize. But even if all threads within a block are synchronizable using the CUDA defined barrier $\_\_synchthreads()$, there are no native ways to synchronize blocks in CUDA. Some ways to manually synchronize CUDA block exists, for example by using atomic functions to increment a mutex and busy-waiting until the mutex reaches a predefined value or by using lock-free sychronizing as described in [16]. But these methods are only applicable when the number of blocks and threads is less than what can be run in parallel (hence no native CUDA block synchronization) which is not the case in this project, where multiple full scale arrays are

used. In this case, the only way to achieve the desired feature of ordered execution is to seperate the code into different CUDA kernels. In the serial versions the pseudocode in algorithms 1, 3 and 4 are all executed in the same function, but in the CUDA version the code is split up into several kernel functions. Having to re-create new threads for each iteration once for all the CUDA kernels affects the performance greatly. However the global memory is persistent between kernel launches, resulting in no data transfer while segmenting. Because only a few neighbouring pixels are elements of the same layer warp divergence will also be affecting the performance. Furthermore, with the lack of shared memory usage and the bottleneck when accessing the slow global memory the speedup gained is low.

Apart from the fact that shared memory is not utilized, it can be seen from the results in tables 5.2 and 5.2 that thread creation by multiple kernel launches at each iteration greatly affects the performance. From table 5.2 it can be seen that the execution time in the CUDA version nearly doubled that of the serial version. But when using the modified speed function which only needed 320 iterations for a full segmentation, the CUDA execution time was less than half of the serial execution time.

A 3D version of CUDA was implemented in addition to the 2D version. But various problems when implementing prevented a version able to segment correctly. A few hours with this code should make it able to correctly extract out volumes. And by using the modified speed function, this version of the program should be able to give better performance increases than the 2D CUDA version.

Chapter 7

Conclusion and future work

7.1 Conclusion

In this project we looked at the sparse field method and how it can be used to segment medical volume data. We chose the level set method as the focus of this project because it is widely used in the field of medical image segmentation. The sparse field method was chosen because it is one of the fastest and least computationally demanding of the approaches to the level set method. In addition we explored the possibilities of parallelizing the sparse field method to speed up computation time.

The implemented sparse field algorithm is very effective in handling large data volumes and segmenting medical data by only handling a small portion of the volume at each iteration.

We explored some modifications to a popular speed function used in the segmentation of images and volumes. We concluded that one of the modifications reults in a significant speedup of the execution time. We also looked into parallilizing the sparse field method using CUDA. However, due to the nature of the sparse field method it is not well suited to be parallelized. Dynamic lists are one of the key features in the sparse field algorithm which is not supported by current CUDA versions. To overcome this restriction, the CUDA implementaton resembles more an extreme version of a narrow band method, where the band is as small as possible without affecting the accuracy of the result.

7.2 Future work

7.2.1 Reduce leakage

Some weaknessse were mentioned in the discussion chapter. The scenario where it fails to grow in one region of the image and simultaneously leaks in another is one of the challenges with the level set method. A possible solution is to run multiple segmentations. First, extract a rough shape of the object by using a small α. The resulting surface is then used to confine the second, more sensitive segmentation. Any leakage during the second segmentation will be confined to the result of the first segmentation, and thus heavy leakage is avoided.

7.2.2 Parallelization

GPU's are great at parallelization, but are not as flexible as CPU's. GPU's are dependent on the CPU to hand it instructions, and the data transefer between them is a bottleneck. However, the continuously improving functionality of GPU's makes is feasible in the near future to implement the sparse field method on the GPU without any major modifications to the structures. The parallelized version was implemented in CUDA, which only supports NVIDIA GPU's. As an alternative to CUDA, the OpenCL platform can be used to parallelize programs not only on NVIDIA systems, but a wide range of GPU's and CPU's. Using OpenCL it would be possible to parallelize the sparse field method on the CPU which might allow an implementation using list structures. Using OpenCL will also allow for a broader range of hardware.

NVIDIA has a library called NVIDIA Performance Primitives (NPP) which is a collection of GPU-accelerated functions that can be used for segmentation in CUDA. Using this library to implement the sparse field level set method might be more compatible than plain CUDA.

Bibliography

[1] Eng L. Wong, *Linear Spectral Unmixing Approaches to Magnetic Resonance Image Classification*. ProQuest, 2008.

[2] S. Osher & James A. Sethian, *Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulation*. Journal of computational physics 79.1, 1988.

[3] David. Adalsteinsson & James A. Sethian, *A fast level set method for propagating interfaces*. Journal of Computational Physics, 1994.

[4] Dzung L. & Chenyang Xu & Jerry L. Prince, *A Survey of Current Methods in Medical Image Segmentation*. Annual review of biomedical engineering 2.1, 2000.

[5] Stanley Osher & Ronald Fedkiw, *Level set methods and dynamic implicit surfaces*. Vol. 153. Springer, 2002.

[6] W. Mulder & S. Osher & James A. Sethian, *Computing interface motion in compressible gas dynamics*. Journal of Computational Physics 100.2, 1992.

[7] Ross T. Whitaker, *A level-set approach to 3D reconstruction from range data*. International Journal of Computer Vision 29.3, 1998.

[8] Aaron E. Lefohn & Joe M. Kniss & Charles D. Hansen & Ross T. Whitaker, *A streaming narrow-band algorithm: Interactive computation and visualization of level sets*. IEEE Transactions on Visualization and Computer Graphics, 2004.

[9] Aaron E. Lefohn & Joshua Cates & Ross T. Whitaker, *Interactive, GPU-based level sets for 3D segmentation*. Medical Image Computing and Computer-Assisted Intervention, 2003.

[10] Lei Pan & Lixu Gu & Jianrong Xu, *Implementation of medical image segmentation in CUDA*. Information Technology and Applications in Biomedicine, 2008.

[11] M. Roberts & J. Packer & Mario C. Sousa & Joseph R. Mitchell, *A work-efficient GPU algorithm for level set segmentation*. Proceedings of the Conference on High Performance Graphics, pp. 123-132, Eurographics Association, 2010.

[12] Shawn Lankton, *Sparse Field Methods - Technical Report*. Georgia institute of technology, 2009.

[13] X.F Wang & D.S Huang & H Xu, *An efficient local ChanVese model for image segmentation*. Pattern Recognition 43.3 pp. 603-618, 2010.

[14] Peter S. Pacheco, *An introduction to parallel programming*. Morgan Kaufmann, 2011.

[15] Jason Sanders & Edward Kandrot, *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[16] Shucai Xiao & Wu-chun Feng, *Inter-block GPU communication via fast barrier synchronization*. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. IEEE, 2010.

[17] Paul Macklin, *EasyBMP - Cross-Platform Windows Bitmap Library*. `http://easybmp.sourceforge.net/`

[18] Erik Smistad, *The Simple Image Processing Library*. `http://www.thebigblob.com/simple-image-processing-library/`.

[19] NVIDIA Corporation, *Thrust Quick Start Guide*. `http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf`, version 5.0, October 2012.

[20] Dirk-Jan Kroon, *Viewer3D*. `http://www.mathworks.com/matlabcentral/fileexchange/21993-viewer3d`, version 11, Januar 2011.

[21] Andreas Loening, *Amide*. `http://amide.sourceforge.net/index.html`, version 1.0.4, April 2013.