Contents

1 Introduction **2**
- 1.1 Background . . . 2
- 1.2 Project Goals . . . 2
- 1.3 Outline . . . 2

2 Background Theory **3**
- 2.1 Medical Terminology . . . 3
- 2.2 Medical Imaging Techniques . . . 3
  - 2.2.1 X-Ray imaging . . . 3
  - 2.2.2 Computed Tomography . . . 4
  - 2.2.3 Magnetic Resonance Imaging . . . 4
  - 2.2.4 T1 and T2 weighted images . . . 6
- 2.3 Segmentation . . . 7
  - 2.3.1 Histogram-based segmentation methods . . . 8
  - 2.3.2 Region based segmentation . . . 10
  - 2.3.3 Edge based segmentation . . . 12
- 2.4 Level Set Method . . . 15
  - 2.4.1 Introduction . . . 15
  - 2.4.2 Signed Distance Transform . . . 19
  - 2.4.3 Discretization by upwinding and difference of normals 20
  - 2.4.4 Chan-Vese engergy function . . . 24
- 2.5 Narrow Band - for lite? . . . 24
  - 2.5.1 Introduction . . . 24
  - 2.5.2 Overview of the Narrow Band method . . . 25
- 2.6 Sparse Field . . . 26
  - 2.6.1 Introduction . . . 26
  - 2.6.2 Overwiew of the Sparse Field method . . . 28
- 2.7 Parallel computing in GPU . . . 29
  - 2.7.1 Data and task parallelism . . . 29
  - 2.7.2 Central processing unit . . . 30

2.7.3 Flynn's taxonomy of computer architectures 30
2.7.4 Graphics processing unit 32
2.7.5 OpenMP 33
2.7.6 General Purpose GPU 33
2.7.7 CUDA 33

3 Sparse Field - Implemented code 39
3.1 Introduction 39
3.2 The layers and their representation 39
3.3 Datastructures and types used 41
3.4 Levelset evolution process 43
3.4.1 Code structure - FIKS DETTE 49
3.4.2 Input and initialization 49
3.4.3 Speed function explained 49
3.5 Problems met 51
3.6 CUDA Implementation 52
3.7 Performance 55
3.8 Third party libraries for I/O 57

4 Results 58
4.1 2D 58
4.2 3D 64
4.3 Performance 69

5 Discussion and future work 71
5.1 Modification of the speed function 71

List of Figures

2.1 CT image of a head. 5
2.2 (a): T1-weighted image, (b): T2-weighted image 7
2.3 (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 $T = 0.7686$. 11
2.4 (a): Image to be segmented, (b): LoG, (c): Sobel - highlighting horizontal edges, (d):Sobel - highlighting vertical edges . . 14
2.5 Interface of a moving surface. 16
2.6 Interface evolution difficult to represent parametrically. 16
2.7 (a): Circle with arrows pointing in direction of movement, (b): Corresponding level set function 17
2.8 The data function, from [8]. 19
2.9 (a): Binary image, (b): SDT based on city-block distance, (c): SDT based on euclidean distance 20
2.10 The narrow band extending out with a width of k from the level set. 26
2.11 Single Instruction Single Data. 31
2.12 Single Instruction Multiple Data. 32
2.13 Multiple Instruction Single Data. 32
2.14 Multiple Instruction Multiple Data. 33
2.15 CUDA programming model. 35
2.16 CUDA memory model. 36

3.1 Label image: image showing the different layers under segmentation. 41
3.2 A label image with artifacts due to code errors when handling the layers. 42
3.3 How the label image should have been. 43
3.4 Zero level set corresponding to the label image in figure 3.2. . 51
3.5 Layer image with Lz two pixels wide. 53

4.1 (a): Original image with seed point. Zero level set after: (b): 600, (c): 1200, (d): 1600 and (e): 2200 iterations (e): with $\epsilon = 0.05$. 60
4.2 (a): Seed point partly outside the object, superimposed on the input image. Interface after (b): 300, (c): 800 and (d): 1300 iterations. 62
4.3 (a): Input image with seed point superimposed. Segmentation result with (b): $\alpha = 0.80$, (c): $\alpha = 0.90$. 63
4.4 Interface smoothness highly valued. (a): Input image, (b): segmentation result. 63
4.5 Maximum intensity projection of the volume to be segmented 64
4.6 Aneurism segmentation after 500 iterations. 65
4.7 Aneurism segmentation after 500 (red), 1500 (blue) and 3000 (gray) iterations. 66
4.8 Aneurism: original volume in gray and segmentation result after 3000 iterations in blue. 67
4.9 Original volume of head and segmented brain volume in red. 68
4.10 Slices along the z-axis of the brain segmented volume, superimposed on the original volume. 68

Chapter 1

Introduction

1.1 Background

1.2 Project Goals

1.3 Outline

Chapter 2

Background Theory

2.1 Medical Terminology

Terms used in the medical practice can often be confusing and

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 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 absorbation (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-ray 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, but 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 imaginng. 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 a X-ray source (emitter) that generates X-ray and releases it towards the patient. The detector array at the opposite side receives the X-ray not absorbed by the patient. The machine rotates around the patient while releasing X-ray photons to get information from all directions. The detector array (scintillator) transforms the X-rays into proportionally strong electric current 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-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 2.1 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

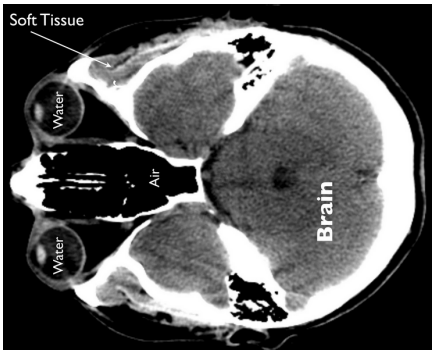


Figure 2.1: CT image of a head.

an odd number 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.

2.2.4 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.

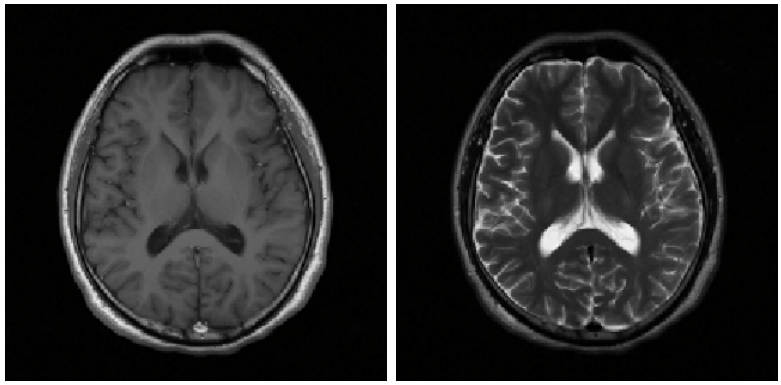
Neighbouring molecules causes the hydrogen protons attached to different types of molecules to experience slightly different local magnetic fields.

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 2.2 (from [**?**]). Both images were taken with magnetic fields of 1.5 tesla. In



(a) (b)

Figure 2.2: (a): T1-weighted image, (b): T2-weighted image

T_1-weighted images, fluids (such as the cerebrospinal fluid in figure 2.2 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 [3])

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

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, lack of texture contrast between regions and background are some of the 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 2.3c and 2.3d (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 2.3 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 2.3a, and its histogram in figure 2.3b. As can be seen from the histogram, it is not possible to select a near perfect threshold by just looking at it. Figure 2.3c illustrates the segmentation result from the iterative global thresholding method and figure 2.3d 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: $H(I_i \cup I_j) = false.$

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.

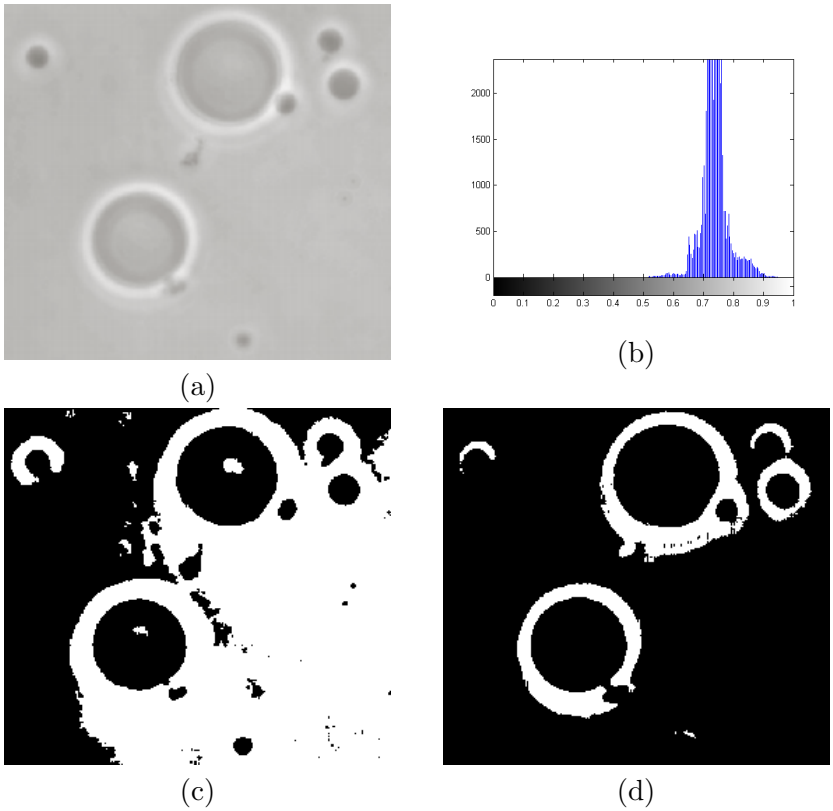


Figure 2.3: (a): Image to be segmented, (b): Histogram of image, (c): Segmented using iterative global thresholding, with $T = 0.7332$, (d): Segmented using Otus's method with $T = 0.7686$.

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 consiste 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:

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

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.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad (2.6) \qquad \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad (2.7)$$

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.

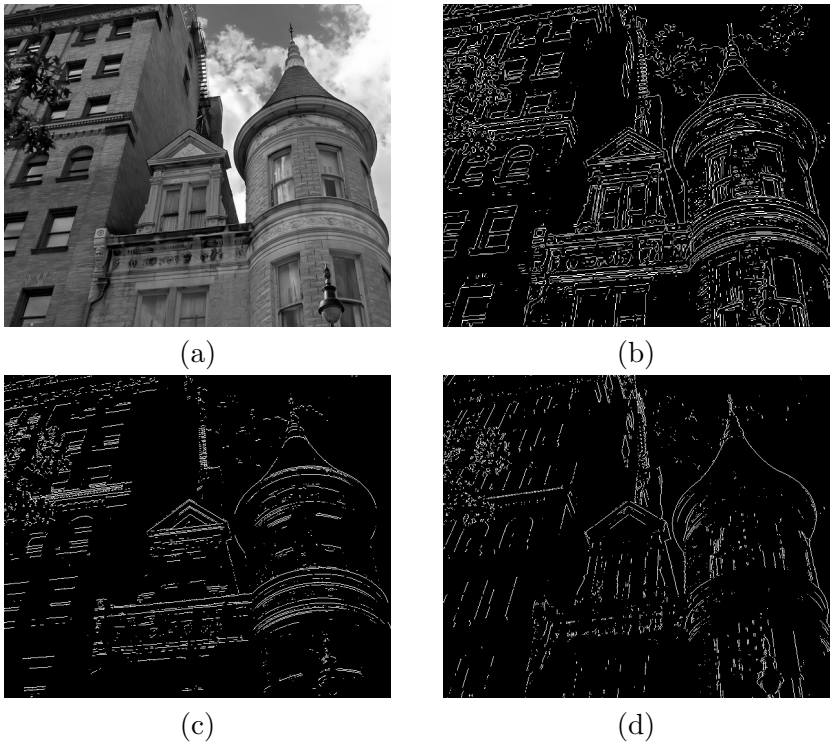
There are various masks based on first order derivatives, and two of them are the Prewitt and Sobel masks, represented in equation 2.8 and 2.9 respectively. These two are not rotation invariant, but the masks can be rotated to emphasize edges of different directions. The masks as they are represented 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 2.4a illustrates a gray-scale image and 2.4b is the edge segmented image based on LoG. 2.4c is the edge image resulted from the Sobel mask in equation 2.9b and 2.4d is the result from segmentation after rotating

the mask 90°. The segmentations resulted by using the Prewitt operator to segment the image in figure 2.4a had no significant difference from the Sobel segmented images.



(a) (b)

(c) (d)

Figure 2.4: (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, and 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 con-

nectivity). 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 Level Set Method

2.4.1 Introduction

Surfaces that evolves over time can be difficult to represent. Taking the surface in figure 2.5 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 notion. A better alternative is to use a parametric equation. The problem 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

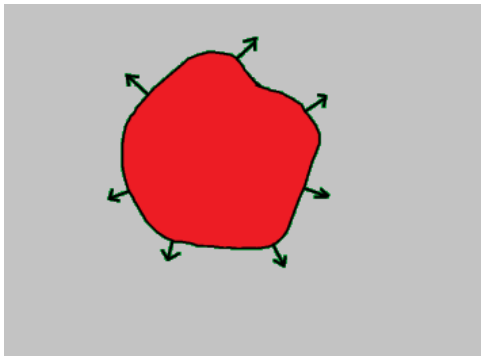


Figure 2.5: Interface of a moving surface.

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 2.6, 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.

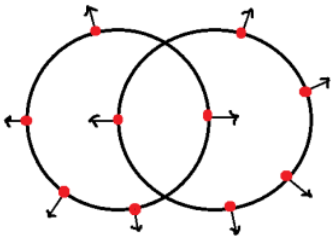


Figure 2.6: Interface evolution difficult to represent parametrically.

As a solution to all the problems mentioned above Osher and Sethian introduced the level set method in 1988 in [1]. 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) is represented by

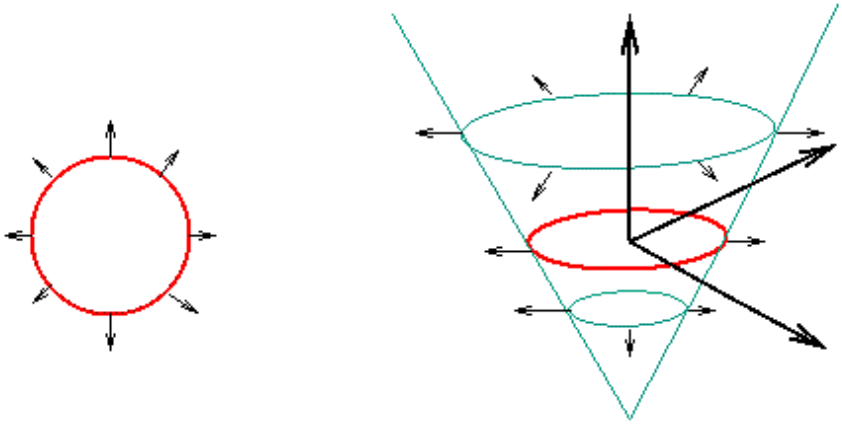
the 3D level set function

$$\phi(x, y, t) \tag{2.10}$$

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:

$$\Gamma(x, y, t) = \{\phi(x, y, t) = 0\}. \tag{2.11}$$

The initial curve is at the xy-plane, that is, at $\phi(x, y, 0)$. As an example, figure 2.7a depicts a circle with arrows pointing in the direction it is evolving, and figure 2.7b is the cone that represents the corresponding level set function with the start-position in red.



(a) (b)

Figure 2.7: (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

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| F \tag{2.12}$$

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.

An often used speed function for image segmentation that combines a data term and the mean curvature of the surface is [8, 7]

$$F = \alpha D(I) + (1 - \alpha)\nabla \frac{\nabla \phi}{|\nabla \phi|} \tag{2.13}$$

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[7]. 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[8, 7] is:

$$D(I) = \epsilon - |I - T| \tag{2.14}$$

Here T is the central intensity value of the region to be segmented, and ϵ is the deviation around T that should 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 2.8, 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 [7]. 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 will be initialized to.

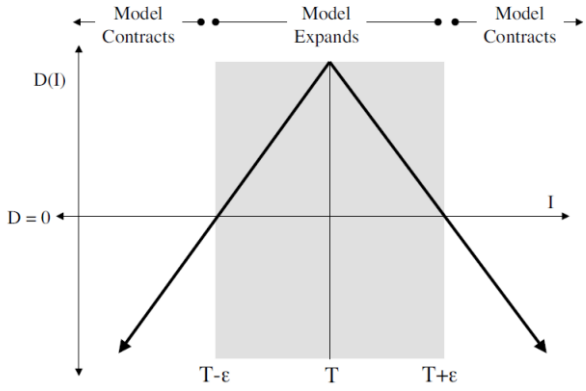


Figure 2.8: The data function, from [8].

2.4.2 Signed Distance Transform

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

$$D(r, S) = min(r - S) \quad for\ all\ r \in \mathbb{R}^3 \tag{2.15}$$

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 [1], where it was initialized as $\phi = 1 \pm D^2$. But in [5] 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 2.9a (the numbers in this image represent intensity values). Figure 2.9b is the signed distance transform of 2.9a where city-block (manhattan) distance have been used, and figure 2.9c 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

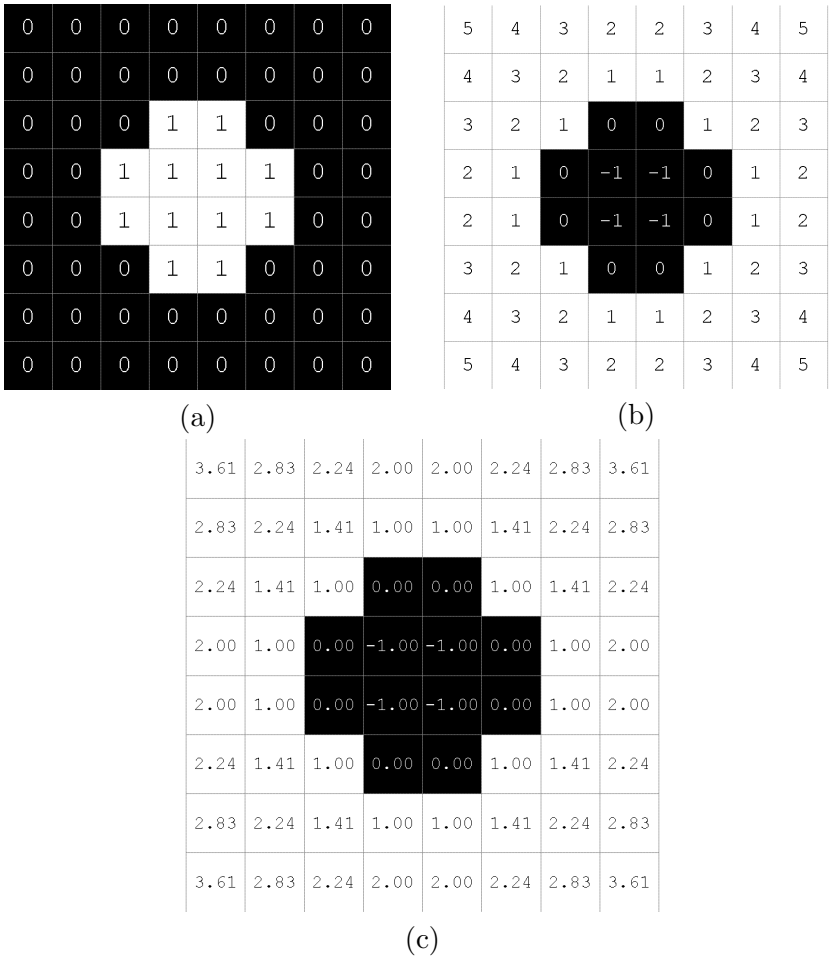


Figure 2.9: (a): Binary image, (b): SDT based on city-block distance, (c): SDT based on euclidean distance

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.

2.4.3 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 [1]. 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 2.12, given by (from [4])

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + F^n \cdot \nabla \phi^n = 0 \tag{2.16}$$

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):

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u^n \phi_x^n + v^n \phi_y^n + w^n \phi_z^n = 0, \tag{2.17}$$

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 [4]. 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 2.17 can be written as

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + u_i^n (\phi_x)_i^n = 0, \tag{2.18}$$

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

$$\begin{gathered}
D_x = \frac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{2} \quad D_y = \frac{\phi_{i,j+1,k} - \phi_{i,j-1,k}}{2} \quad D_z = \frac{\phi_{i,j,k+1} - \phi_{i,j,k-1}}{2} \\
D_x^+ = \phi_{i+1,j,k} - \phi_{i,j,k} \quad D_y^+ = \phi_{i,j+1,k} - \phi_{i,j,k} \quad D_z^+ = \phi_{i,j,k+1} - \phi_{i,j,k} \\
D_x^- = \phi_{i,j,k} - \phi_{i-1,j,k} \quad D_y^+ = \phi_{i,j,k} - \phi_{i,j-1,k} \quad D_z^+ = \phi_{i,j,k} - \phi_{i,j,k-1}
\end{gathered} \tag{2.19}$$

which is taken from the appendix of [7]. This is a *consistent* finite difference approximation to the level set equation in 2.12, because the approximation error converges to zero as $\Delta t \to 0$ and $\Delta x \to 0$ [4]. 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 $|u|$,

$$\Delta t = \frac{\Delta x}{max\{|u|\}}, \tag{2.20}$$

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,

$$\nabla\phi = \begin{cases} ||\nabla\phi_{max}||_2 & F_{i,j,k} > 0 \\ ||\nabla\phi_{min}||_2 & F_{i,j,k} < 0 \end{cases} \tag{2.21}$$

where $\nabla\phi_{max}$ and $\nabla\phi_{min}$ is given by (from [7])

$$\nabla\phi_{max} = \begin{bmatrix} \sqrt{max(D_x^+, 0)^2 + max(-D_x^-, 0)^2} \\ \sqrt{max(D_y^+, 0)^2 + max(-D_y^-, 0)^2} \\ \sqrt{max(D_z^+, 0)^2 + max(-D_z^-, 0)^2} \end{bmatrix} \tag{2.22}$$

$$\nabla\phi_{min} = \begin{bmatrix} \sqrt{min(D_x^+, 0)^2 + min(-D_x^-, 0)^2} \\ \sqrt{min(D_y^+, 0)^2 + min(-D_y^-, 0)^2} \\ \sqrt{min(D_z^+, 0)^2 + min(-D_z^-, 0)^2} \end{bmatrix} \tag{2.23}$$

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:

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

$$
\begin{aligned}
D_x^{+z} &= (\phi_{i+1,j,k+1} - \phi_{i-1,j,k+1})/2 \quad D_x^{-z} = (\phi_{i+1,j,k-1} - \phi_{i-1,j,k-1})/2 \\
D_y^{+x} &= (\phi_{i+1,j+1,k} - \phi_{i+1,j-1,k})/2 \quad D_y^{-x} = (\phi_{i-1,j+1,k} - \phi_{i-1,j-1,k})/2 \\
D_y^{+z} &= (\phi_{i,j+1,k+1} - \phi_{i,j-1,k+1})/2 \quad D_y^{-z} = (\phi_{i,j+1,k-1} - \phi_{i,j-1,k-1})/2 \\
D_z^{+x} &= (\phi_{i+1,j,k+1} - \phi_{i+1,j,k-1})/2 \quad D_z^{-x} = (\phi_{i-1,j,k+1} - \phi_{i-1,j,k-1})/2 \\
D_z^{+y} &= (\phi_{i,j+1,k+1} - \phi_{i,j+1,k-1})/2 \quad D_z^{-y} = (\phi_{i,j-1,k+1} - \phi_{i,j-1,k-1})/2
\end{aligned}
\tag{2.24}
$$

Then these derivatives are used to compute the normals n^+ and n^- in equation 2.25, which is used to compute the mean curvature H in equation 2.26 taken from [7].

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

$$
n^- = \begin{bmatrix}
\frac{D_x^-}{\sqrt{(D_x^-)^2 + (\frac{D_y^{-x} + D_y}{2})^2 + (\frac{D_z^{-x} + D_z}{2})^2}} \\
\frac{D_y^-}{\sqrt{(D_y^-)^2 + (\frac{D_x^{-y} + D_x}{2})^2 + (\frac{D_z^{-y} + D_z}{2})^2}} \\
\frac{D_z^-}{\sqrt{(D_z^-)^2 + (\frac{D_x^{-z} + D_x}{2})^2 + (\frac{D_y^{-z} + D_y}{2})^2}}
\end{bmatrix}
\tag{2.25}
$$

$$
H = \frac{1}{2} \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} = \frac{1}{2} [(n_x^+ - n_x^-) + (n_y^+ - n_y^-) + (n_z^+ - n_z^-)] \tag{2.26}
$$

Finally, the level set equation is updated as

$$
\phi(t + \Delta t) = \phi(t) + \Delta t F |\nabla \phi|. \tag{2.27}
$$

2.4.4 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:

$$E^{CV}(c_1, c_2, C) = \int_{inside(C)} (\mu(x,y) - c_1)^2 \, dx \, dy + \int_{outside(C)} (\mu(x,y) - c_2)^2 \, dx \, dy \, [12] \tag{2.28}$$

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

$$E^{CV}(x,y) = (\mu(x,y) - c_1)^2 - (\mu(x,y) - c_2)^2. \tag{2.29}$$

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.

2.5 Narrow Band - for lite?

2.5.1 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[2]. 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.

2.5.2 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[6]. 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 2.10), 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 $O(nk^d)$ [2] 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

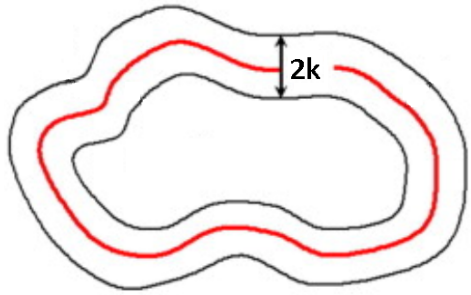


Figure 2.10: The narrow band extending out with a width of k from the level set.

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 [4].

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 [2]) because of two competing computional costs[6]. 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.

2.6 Sparse Field

2.6.1 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 [6] 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. The band is kept track of by defining the points nearest the level set as active points. Combining them gives us the active set. (dette m muligens forandres igjen. stusser veldig p hvordan dette skal skrives)

Using only the active points to compute the derivatives would not give sufficient accuracy. Because of this, 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 [6]. 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.

A disadvantage of the narrow band method is that the stability at the boundaries of the band have to be maintained (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 done 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.

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 many other level set algorithms.

The narrow band method (og ogs vanlig Level Set Method) uses the same SDT for multiple iterations inside the band because reclaculating the SDT at every iteration would make the method very time-consuming. This

is a tradeoff between speed and accuracy, as the accuracy of the SDT decreases with every iteration. Sparse field aproximates the SDT to be the city block distance from the active set, and recalculates this for the points in the layers at every iteration. So both methods uses tradeoffs between speed and accuracy, but the aproximations of the sparse field method has been shown to not be worse than other approaches to the level set method. (insert link)(knotete skrevet dette her).

Since only the grid points whose values are changing (the active points and their neighbors) are visited at each time step the growth computation time is d^{n-1}, where d is the number of pixels along one dimension of the image (er dette rett?). This is the same as for parameterized models where the computation times increase with the resolution of the domain, rather than the range.

Since we only do calculations on pixels in the active set and the neighbouring layers, the computation time increases with the size of the interface rather than the range of the domain. With comparable aproximation errors and good speed, the sparse field method is a viable approach to active shape segmentation.

2.6.2 Overwiew of the Sparse Field method

Like described in section 2.4.3 (link to up-winding), the Up-Winding scheme gives the curvature in an area surrounding a point in the active set. This scheme uses both first and second order derivatives, and to calculte them it needs a 3x3x3(3D) grid of points surrounding the active point whose speed is being calculated. This creates a lower limit for the number of layers surrounding the active set. In addition to the active set which is stored in $L0$ we need four lists,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. Like the other aproaches to the level set method, the datastructure that tracks the evolution of the interface is an array with the same dimensions as the problem domain. (kan vi skrive dette litt mer profft kanskje? for eksempel med en formel som viser at array dimension med stor R equals Image dimensions)

Its important to note that the lists are used to keep track of which points are in the active set and their neighbours, and are a redundant datastructure, separate from phi. Thus tracking the layers has no effect on the accuracy of the end result.(skriver dette fordi narrow band har et problem i interfacen, stemmer det?)

The initialization process of the interface is fairly straight forward. Like most ASM's the method 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 or outside the mask. The values on the border of the mask is defined as the zero level set so the corresponding points in phi is set to 0. This set of points is the initial active set. The neighbouring layers around it is set by defining one layer at the time as the points immediately adjacent to its inner layer. Every point in each layer has its level set value (phi) set to the value of the layer it's in. Initialization is then complete.

Each iteration consists of four steps. First the speed of each point in the actve set is calculated and the level set is updated with the new level value of the point. Second all the layers around the active set are updated with their new position according to the change in value of its inner neighbour. So if an active point is determined to move out of the range of the acive set, the phi value is updated, and then its neiighbouring points are all updated to be either -1 (inside) or +1 (outside) the value of the previously active point. This will make one of the points fall within the range of the active set and will update its layer to reflect this.

2.7 Parallel computing in GPU

TODO: fiks p dette 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 [8] 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 [9] some simple medical segmentation algorithms was implemented using NVIDIAs CUDA technology, and in [10] CUDA was used to implement the level set method.

2.7.1 Data and task parallelism

Data and task parallelism are the two main categories of computer parallelism. Data parallelism is achieved by having differnt units execute the same task at different data in parallel. This type of parallelism is used in im-

age 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).

2.7.2 Central processing unit

The central processing unit (CPU) .....

2.7.3 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 2.11 illustrates how SISD works. In the figure processing unit is abbreviated as PU.

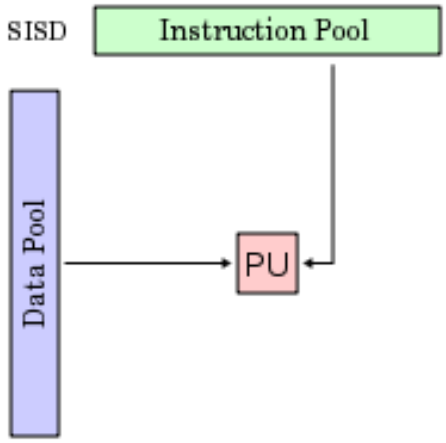


Figure 2.11: Single Instruction Single Data.

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 2.12

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 2.13.

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

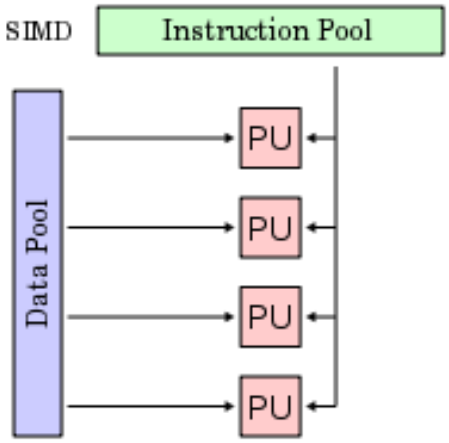


Figure 2.12: Single Instruction Multiple Data.

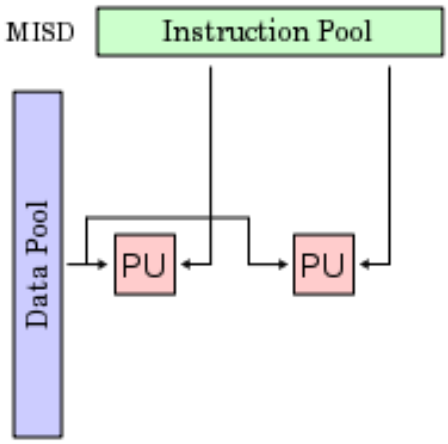


Figure 2.13: Multiple Instruction Single Data.

consisting of several cores are all based on MIMD for parallelism. MIMD is illustrated in figure 2.14.

2.7.4 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

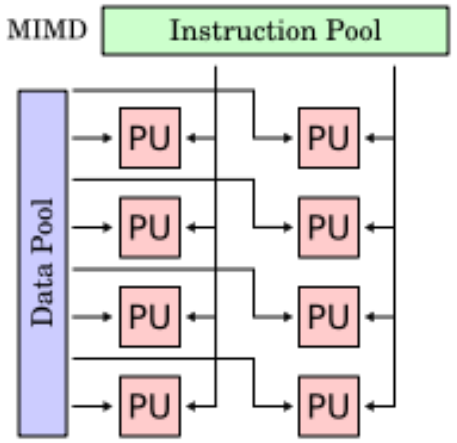


Figure 2.14: Multiple Instruction Multiple Data.

is memory intensive but independent of each other and thus highly parallelizable.

2.7.5 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 [13]. 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.

2.7.6 General Purpose GPU

TODO

2.7.7 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 2.15 illustrates the programming model of CUDA, which also shows that a set of blocks is called a grid.

Each thread in CUDA have its own registers and local memory, and all the threads in a block have a shared memory, all 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 2.16. 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.

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 de-

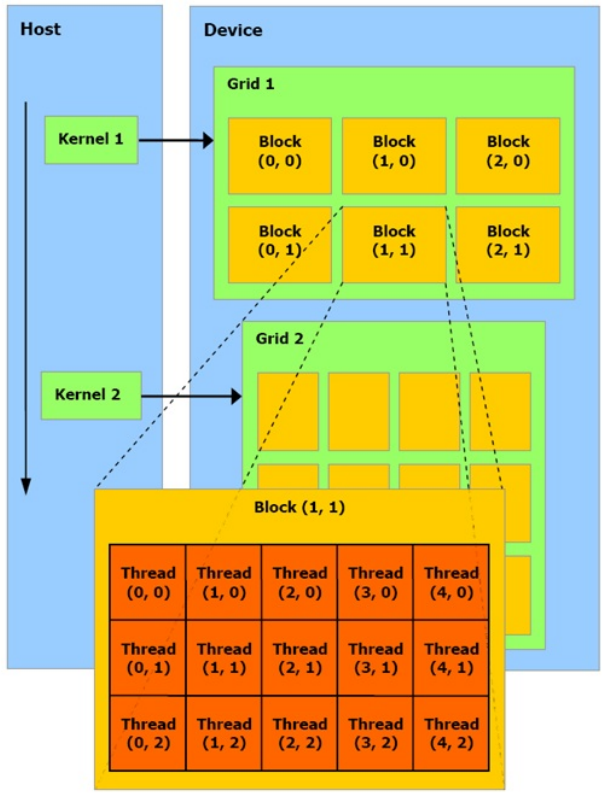


Figure 2.15: CUDA programming model.

vice 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 [14]) 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

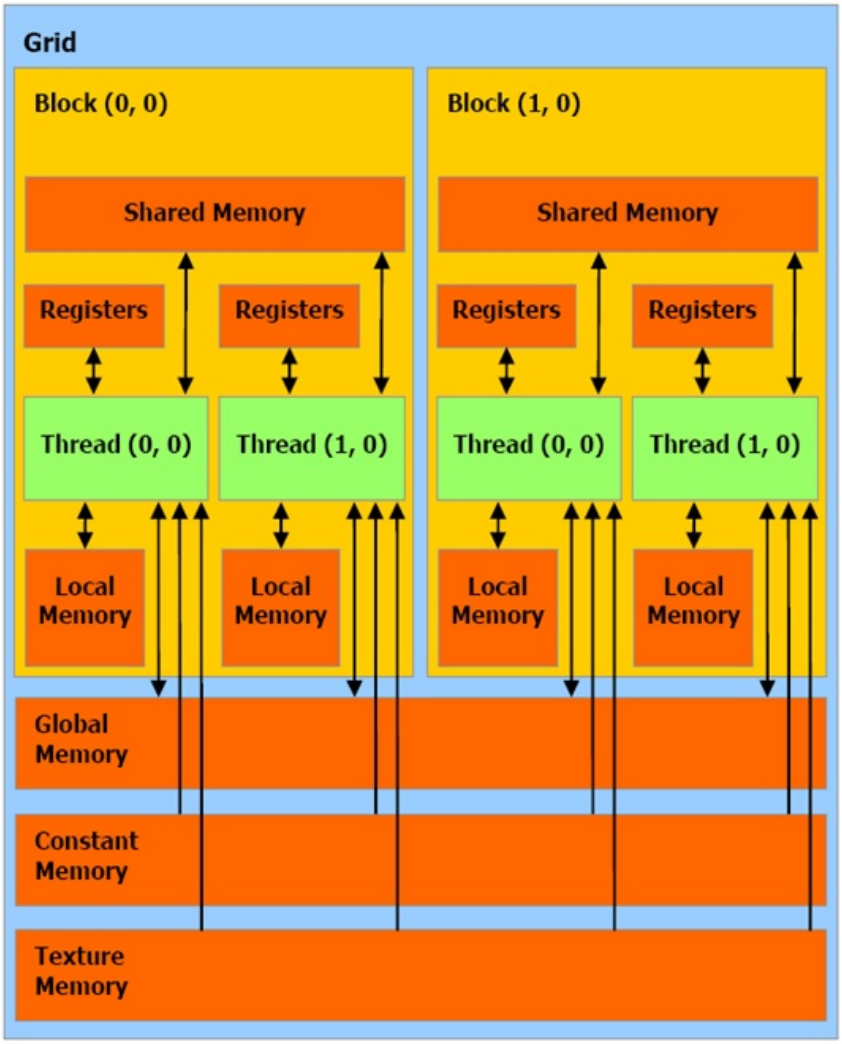


Figure 2.16: CUDA memory model.

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

Sparse Field - Implemented code

3.1 Introduction

The sparse field level set method was implemented in C++ for the project, and the implemented code is mainly based on the pseudocode in [11], which again is based on Whitaker's introduction to the sparse field method in [6]. 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. The implemented code of the 2D and 3D versions the implemented code can be found in appendix A and B respectively. A parallelized verson of the 2D version was also implemented in CUDA, and the source code is found in Appendix C. 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.

3.2 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. One of the lists holds the active points, i.e the zero level set, and is referred to as the Lz list. The rest of the needed pixels are seperated according to their closeness to the pixels in Lz and on which side of the Lz pixels they are

located. The Ln1 list contains the pixels that are adjacent to Lz pixels on the inside of the object being segmented. Similarly Lp1 contains pixels that are adjacent, but on the outside. All pixels that are adjacent to those in Ln1 except for those in Lz are elements in the Ln2 list, and similarly the ones adjacent to Lp1 on the opposite side of Lz are part of Lp2. This becomes more clear when looking at table 3.1 and figure 3.1. The elements in these lists are C/C++ structures (struct) called *Pixel* which contains two (three in 3D) integer values x and y representing its coordinates in ϕ. So when an element in ϕ is to be added to a 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.

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 3.1: Range of lists used in [11]

Table 3.1 shows the ranges used by Whitaker in [6] when describing the layers used in the sparse field method. Figure 3.1 represents the 5 different layers with different colors. The pixels in the zero level set are represented as dark-purple colored, Lp1 is represented by light purple and the outer-most (light-blue) pixels are members of Lp2. Ln2 is dark-blue and Ln1 is brown, and these two layers plus the dark part are defined to be inside the object being segmented at a given iteration. The two other layers and the white part is 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.

By looking closer at table 3.1 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 3.5. 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 3.2, and even though the change seems small and insignificant it improves the result significantly (as will be discussed in 3.5).

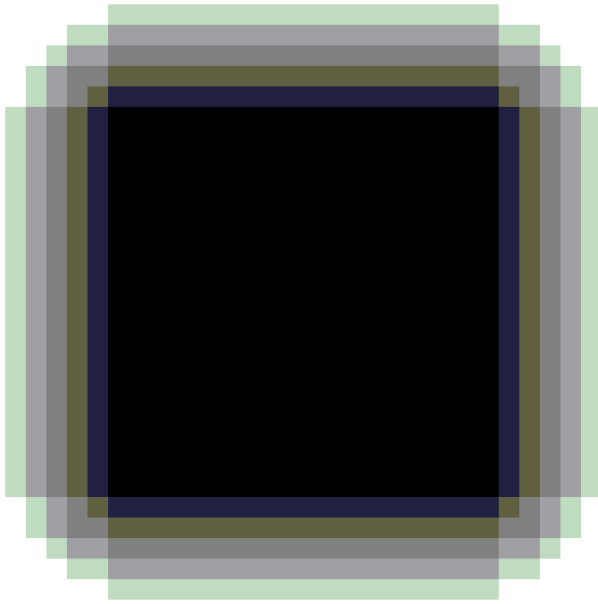


Figure 3.1: Label image: image showing the different layers under segmentation.

3.3 Datastructures and types used

In addition to the five lists representing the five layers, two arrays of equal size and dimension as the image to be segmented are used. One of them is the *label* image described above, which is used to track where the pixels containing the different layers are on the domain. Given a pixel, to find out which layer (if any) it is a member of, a simple lookup to the *label* array is enough. Another excellent feature of the *label* array is that it can be used to visually verify if all the layers are correclty 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 3.2 (zoomed in for clarity). How that *label* image actually should have been is illustrated in figure 3.3.

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

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 3.2: Range of lists used in the implementation

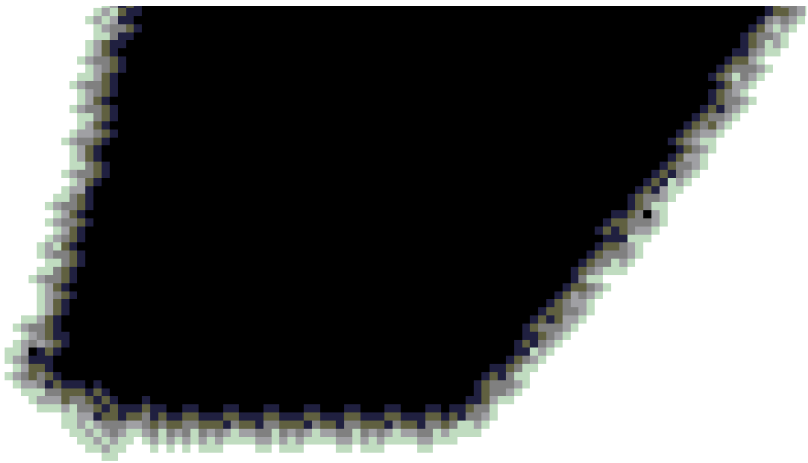


Figure 3.2: A label image with artifacts due to code errors when handling the layers.

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 determines which layer a pixel belongs to after its pixels 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, 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.

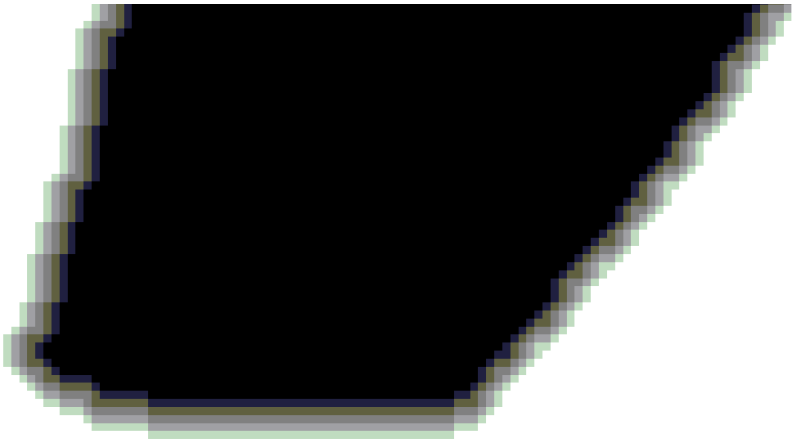


Figure 3.3: How the label image should have been.

3.4 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 show in the pseudocode in algorithm 1. This process of updating the pixels with new values and

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(e)
5:         Lz.remove(e)
6:     else if phi(e.x, e.y) < −0.5 then
7:         Sn1.add(e)
8:         Lz.remove(e)
9:     end if
10: end for
```

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 while the pixels in the other lists are not. The definition of pixels in Ln1 and Lp1 say that they 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 the algorithm in 2 to update ϕ. Algorithm 4 shows the same for Ln2 and Lp2. Notice that the *label* arraysof the pixels moving out of Ln2 and Lp2 is updated in 4. This is not the case in algorithm 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. Also notice that pixels moving out of Ln2 (not those moving to Ln1, but the other direction) are having their corresponding values in the ϕ and *label* arrays set to -3, and those moving out of Lp2 to 3.

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

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

```
1:  procedure FOLLOW(p, greaterOrLess, checkAgainst)
                          ▷ p = Pixel, greaterOrLess, checkAgainst = integer
2:      result = checkAgainst
3:      if greaterOrLess = 1 then                  ▷ true for: Ln1 or 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 or 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
```

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
```

3.4.1 Code structure - FIKS DETTE

The code is seperated into two C++ files, main.cpp and update.cpp, and two corresponding header files. The update.cpp file consist of everything that happens at each iteration, this icnludes calculating the speed function, updating the ϕ - array with the speed, updating the *label* array and update the lists. The main.cpp file consist of actions that are executed before and after the actual segmentation, such as initializing everything, handle input and reading/writing to/from the input image and segmentation result.

3.4.2 Input and initialization

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 file, 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 *label* and the ϕ arrays is used to initialize *label*, the 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. 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.

3.4.3 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 are implemented are the ones explained in chapter 2.4. The simplified Chan-Vese speed function was first implemented, and only used to test wether 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 2.4, 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 Speef 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 set as negative value. 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 neighbor 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 a 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 2.14) 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 input values to the speed function are must be a little different. This modification will be discussed in more detail in the discussion (chapter 5).

New speed functions can be implemented and easily mergeed 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 $\{-1, 1\}$, because of the range 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.

3.5 Problems met

As previously mentioned, when looking at figure 3.2 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 3.4 which shows only the zero level set. The zero level set in figure 3.4 is the segmentation result

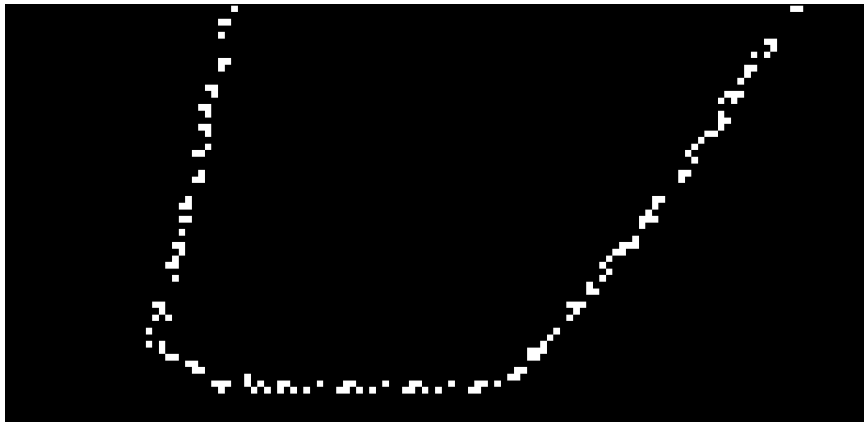


Figure 3.4: Zero level set corresponding to the label image in figure 3.2.

(zoomed in) that corresponds to the label image in figure 3.2. The zero level set is supposed to be a one pixel wide continous line, but in this case that is not true. 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 much 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 the neighbouring layers of Ln1, namely 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 3.2 all the lists have the exact same range-width of < 1, which is not the case in table 3.1. If the ranges in table 3.1 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 functin with value 1 (or -1) indicates fast movement and that element should be moved to Lp1 (or Ln1). But according to table 3.1 that will not happen in Lz when the value is -o.5, even if a change in 1 (or -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 3.5. To clearly illustrate how the double Lz looks like, this figure was sampled after the normalization of the speed function results was implemented.

Another thing that caused problems was a bug in the code that in some cases moved a pixel from $Lp1$ to $Ln2$ when it was supposed to move to $Lp2$. This bug occured only in the 3D implementation and only under certain circumstances, which made the debugging process more complicated and cumbersome. MER??

3.6 CUDA Implementation

The sparse field level set method was also 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 before an optimized version of the narrow band level set method that focuses on using dynamic arrays (linked lists) to hold the elements needed for

Figure 3.5: Layer image with Lz two pixels wide.

computation. This is a very serial way of thinking, and parallelization was not a factor when sparse 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[17], 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 2D and 3D arrays is to split the array in 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 reimplementaion 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. In the serial version, the calculations of normals and first and second order derivatives were put in a header file, but in the CUDA version these computations were put inside the speed function for performance. The speed function is defined as a __device__ function, which makes it inline with the function calling it. 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.

As described before, the updating process of the layers are dependent on each other. Ln2 and Lp2 are respectively dependent on Ln1 and Lp1, while Ln1 and Lp1 depends on Lz. So even if all the calculations in each element part 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 [18]. But these

methods are only applicable when the number of blocks and threads is smaller 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. This does affect the performance, but because the global memory is persistent between kernel launches, only slightly. Because only a few neighbouring pixels are elements of the same layers, warp divergence will be affecting the performance more. With the lack of local and shared memory usage, and the bottleneck when accessing the slow global memory results is the implemented CUDA program to use slightly more time to run than the serial version. More abouut the performance will be discussed in chapter REFERANSE TIL ENTEN RESULTS ELLER DISCUSSION HER.

3.7 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 *std* :: *vector* with the datastructure *std* :: *list*. When the implementation process started *std* :: *vector* 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 *std* :: *list* and *std* :: *vector* are summarized in table 3.3 and 3.4.

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 3.3: 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 3.4: 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 3.5.

	double → float		vector → list	
	100 iteration	full segmentation	100 iteration	full segmentation
2D	TODO	X2	X3	X4
3D	X1	X2	X3	X4

Table 3.5: 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.

3.8 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 ([15]) was used for easily reading and writing Windows bitmap (BMP) image files. In the 3D version the Simple Image Processing Library (SIPL, [16] 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 (can read directly from raw data or using a mhd metafile), and for visualizing the the input volume and the segmentation result for comparision.

Chapter 4

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 illustrate the difference in number of iterations, and the colors for the same number of iterations vary from figure to figure.

4.1 2D

First a simple binary image of size 512x512 of a circle shown in figure 4.1a 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 funcition 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 **??**) in the speed function is gradual, as mentioned when describing the speed function in chapter 2.4.

Figures 4.1 b, c and d represents the zero level set after 700, 1200 and 1600 iterations respectively. Figure 4.1d illustrates the full segmentation result which required 2200 iterations.

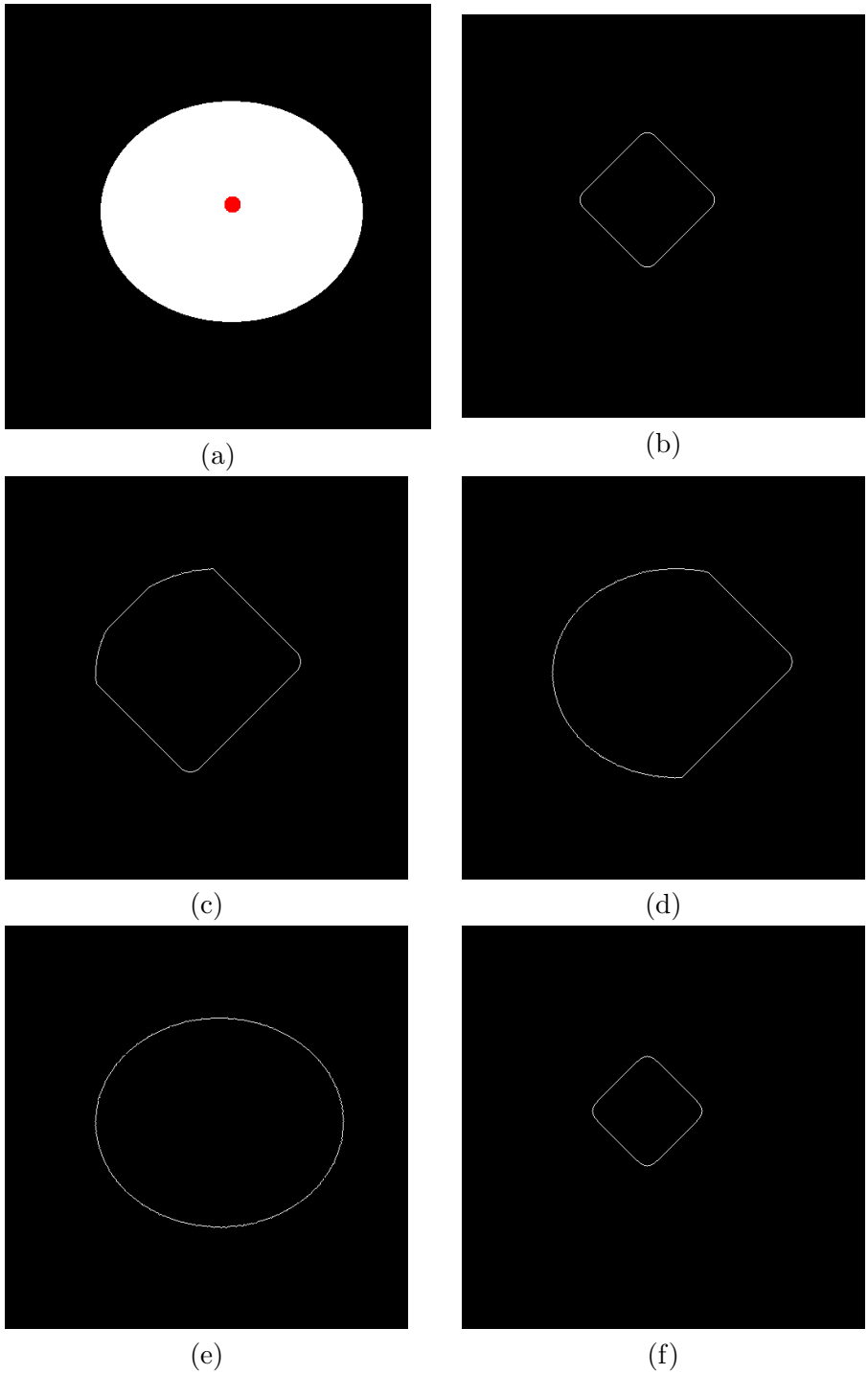


Figure 4.1: (a): Original image with seed point. Zero level set after: (b): 600, (c): 1200, (d): 1600 and (e): 2200 iterations (e): with $\epsilon = 0.05$.

By comparing the input image in figure 4.1a with the segmentation result in figure 4.1e 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 4.1e with figure 4.1f which is the result after 2200 iterations with $\epsilon = 0.05$ it can be seen how much slower the interface evolves.

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 4.2a. How the interface looked like after 300, 800 and 1300 iterations is depicted in figure 4.2b, c and d respectively. The final segmentation result was as expected a correct segmentation of the object as in figure 4.1e.

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 4.3a, 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 4.3b is the result with $\alpha = 0.80$ and 4.3c with $\alpha = 0.90$. As explained in chapter 2.4, 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 4.3c with a higer value of α have been able to include the rectangle at the button by evolving through the thin line, while 4.3b with only a value of 0.10 α less did not manage it. 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 4.4b, which is the segmentation result of the image in figure 4.4a using $\alpha = 0.4$. To clearly illustrate the effects of α on the smoothness of the segmentation result, figures 4.4a and b are small of size (100x100).

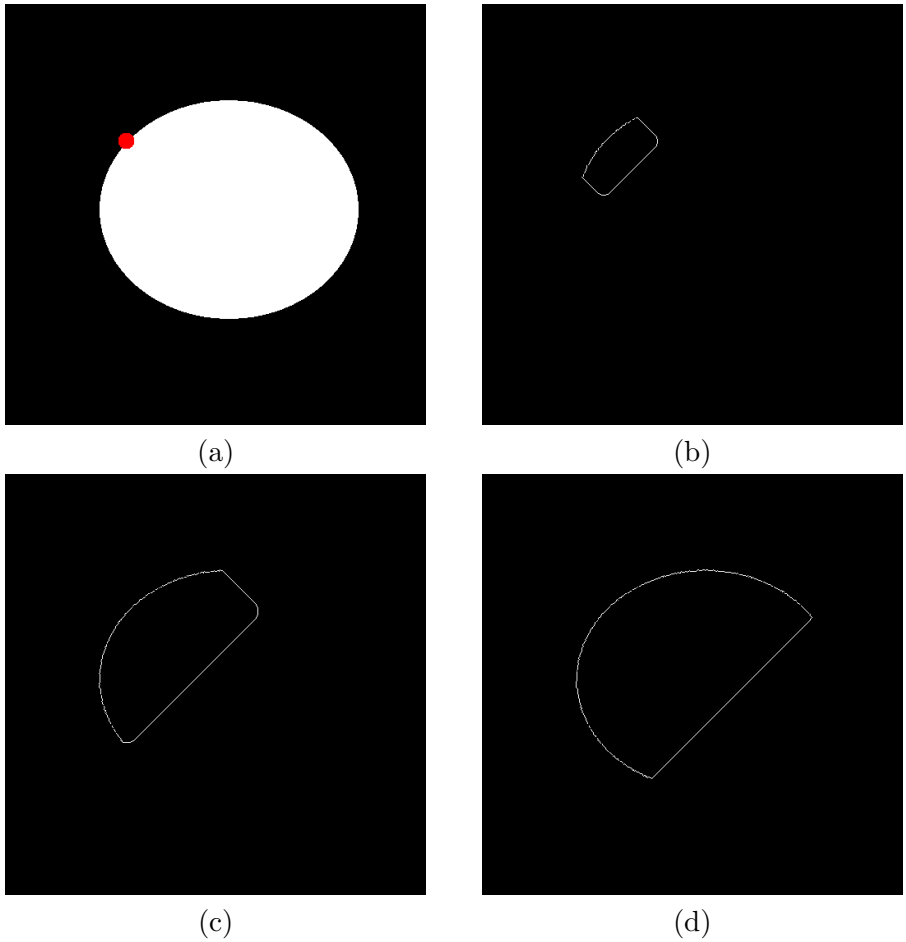
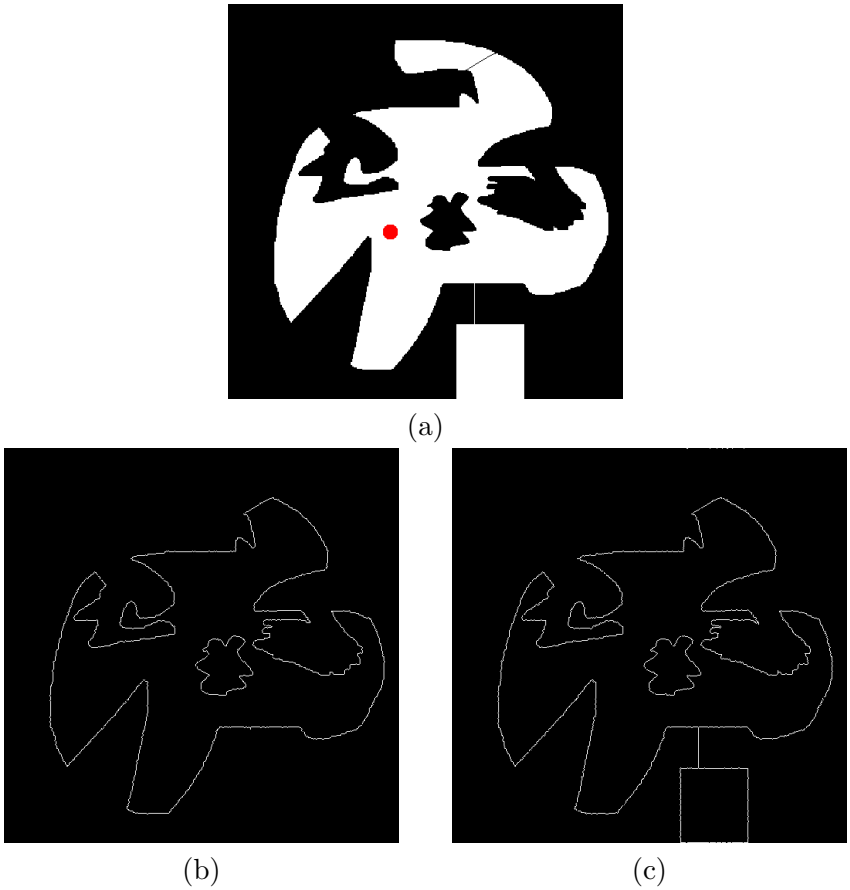


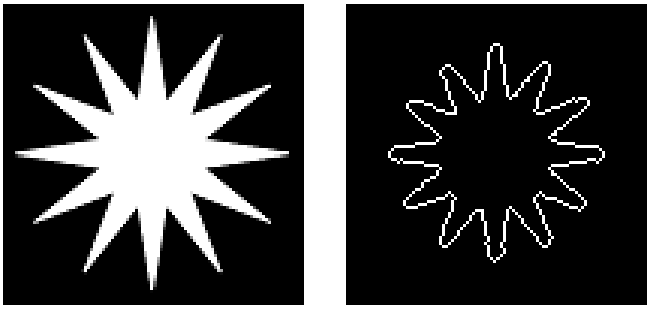
Figure 4.2: (a): Seed point partly outside the object, superimposed on the input image. Interface after (b): 300, (c): 800 and (d): 1300 iterations.



(a)

(b) (c)

Figure 4.3: (a): Input image with seed point superimposed. Segmentation result with (b): $\alpha = 0.80$, (c): $\alpha = 0.90$.



(a) (b)

Figure 4.4: Interface smoothness highly valued. (a): Input image, (b): segmentation result.

4.2 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

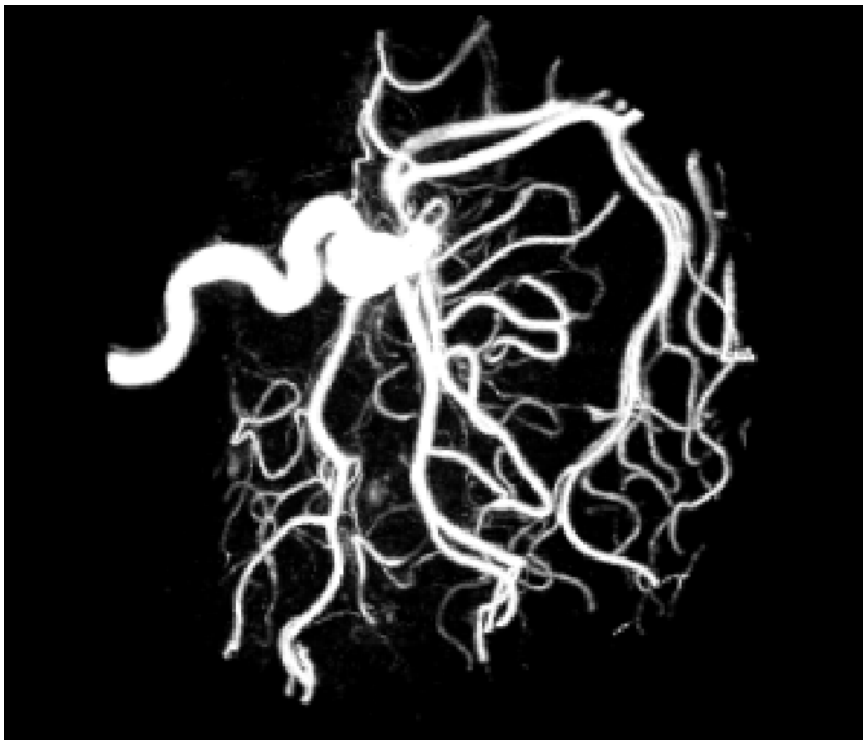


Figure 4.5: Maximum intensity projection of the volume to be segmented

adjacent connected arteries without expanding into insignificant parts of the volume. In figure 4.5 the maximum intensity projection of the unsegmented volume is depicted.

Figure 4.6 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

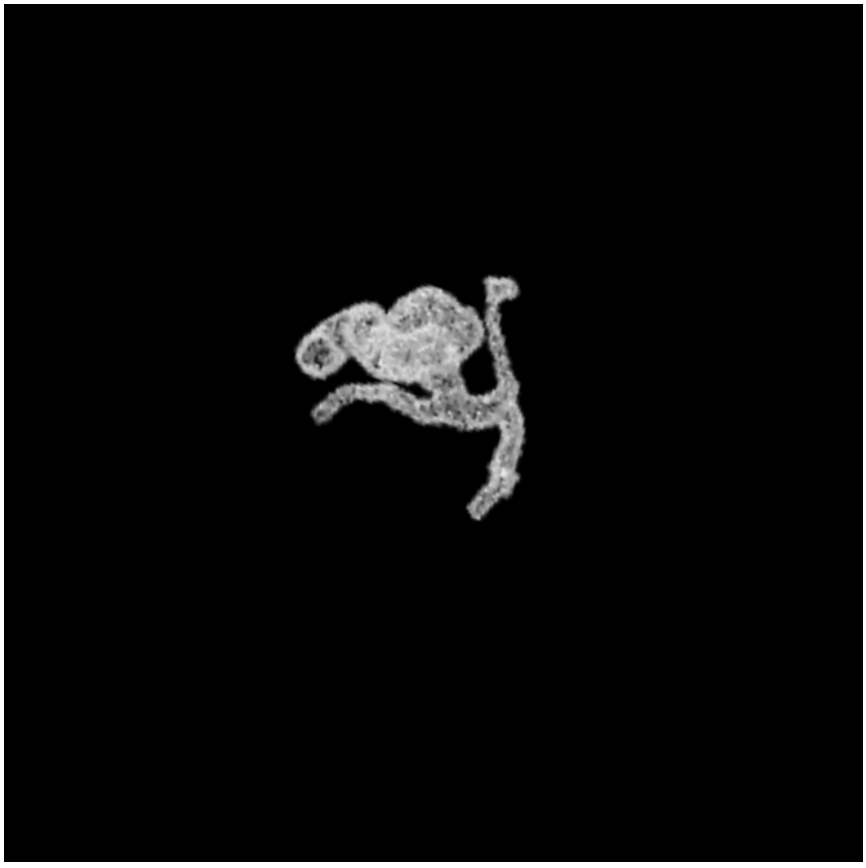


Figure 4.6: Aneurism segmentation after 500 iterations.

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 4.7 shows the segmentated volume after 500, 1500 and 3000 iterations. The red part of the structure is the result of the segmentation after 500 iterations (same as in figure 4.6). The light blue colored part shows the segmentation after 1500 iterations and the gray colored part is the result after 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.

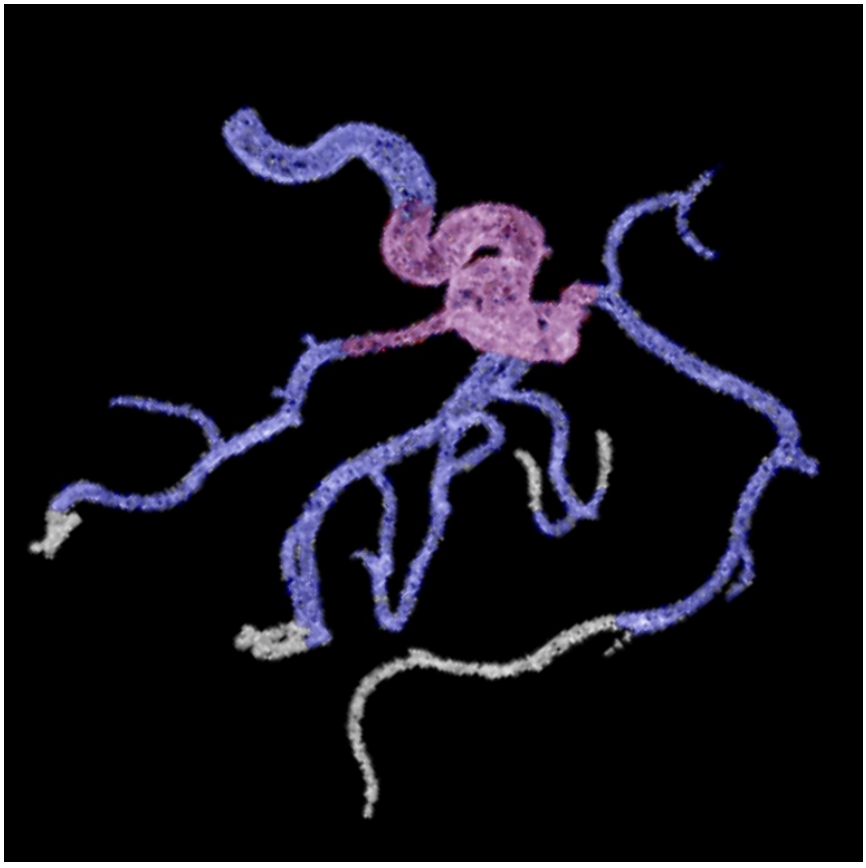


Figure 4.7: Aneurism segmentation after 500 (red), 1500 (blue) and 3000 (gray) iterations.

Next, the original volume (not MIP as in figure 4.5) 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 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.

By looking at all the aneurism segmentation results above it can be seen how good the implemented version of the sparse field level set metod is

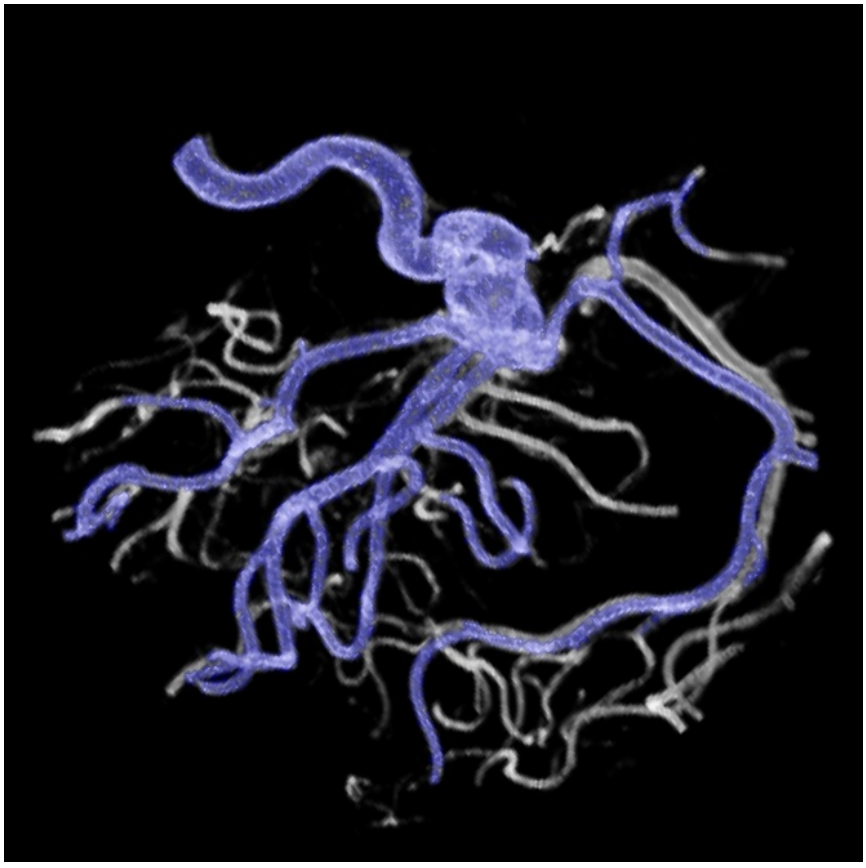


Figure 4.8: Aneurism: original volume in gray and segmentation result after 3000 iterations in blue.

to expand through narrow paths without leaking outside the segmentation object given the correct speed function values.

A T1-weighted MRI volume of a head was segmented to extract out the brain. The head volume along with the segmentation result of the brain (in red) is illustrated in figure 4.9. The number of iterations needed to achieve this result was 4000, and the values used are $T = 0.24$, $\epsilon = 0.09$ and $\alpha = 0.65$. Figure 4.10a and b illustrates the segmentation result as a superimposition on the original volume along different coordinates of the z-axis.

The program was also tested on a CT volume of an abdomen by to extracting out the volume of the liver. This process is somewhat tricky because the liver has greyscale values very similar to the organs surrounding

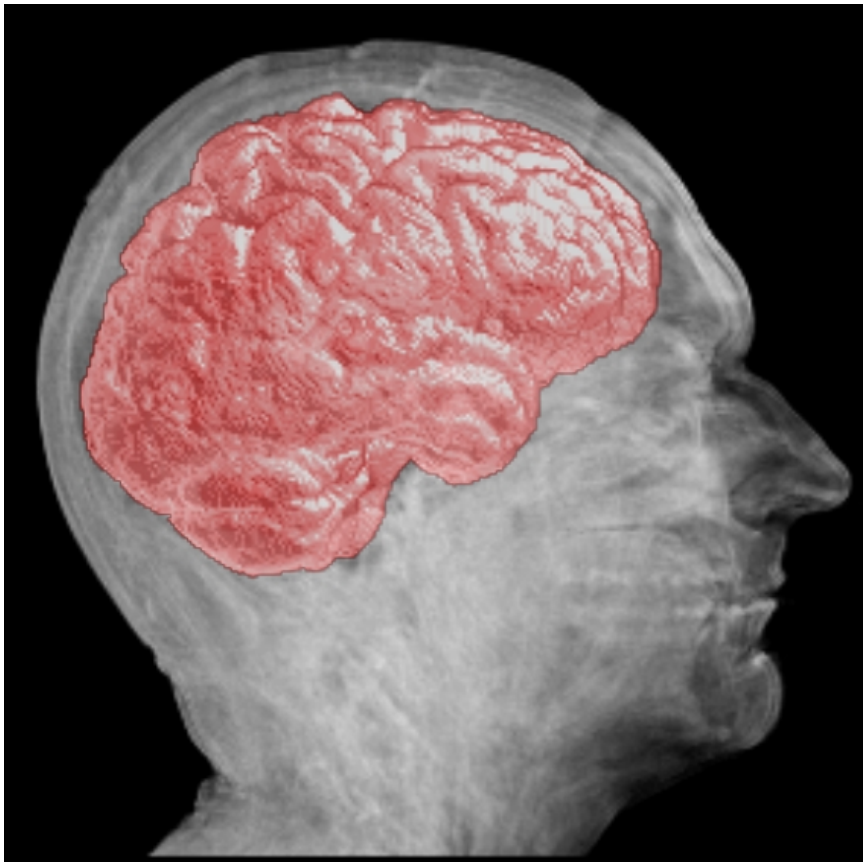
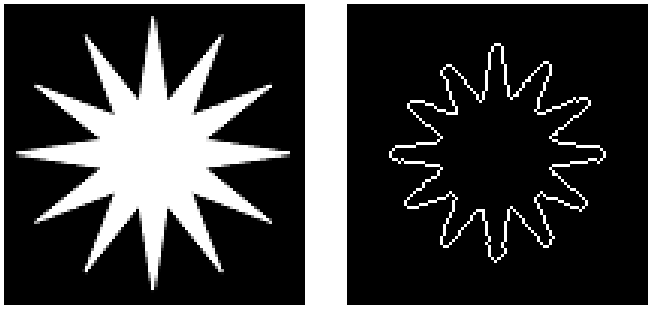


Figure 4.9: Original volume of head and segmented brain volume in red.



(a) (b)

Figure 4.10: Slices along the z-axis of the brain segmented volume, superimposed on the original volume.

it, making it difficult to prevent the interface from leaking into the surroundings. Although the curvature term will prevent 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. So the parameters has to be chosen wisely to get a good segmentation. Figure **??** (TODO: lag bildet) shows a slice in the z-axis of the volume. The liver is the big semi-uniform area that stretches from the left and over the middle of the stomach. As we can see the difference in the pixel values between the liver area and its surroundings are small. The dimensions of the image are 320x220x72. Next we see the segmented volume (figure ct with poor threshold) using values (insert values for liver CT) and (insert iterations) iterations. The threshold in this image (reference figure ct with poor threshold) is too high and so the

4.3 Performance

Testing environment

The tests were performed on a laptop PC with the specifications as described in table 4.1. The software related specifications are Windows 7 as operative 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 4.1: Specifications of the PC the tests were performed on.

TODO HELT til slutt i kapittelet: en tabell med antall iterasjoner og tid brukt p alle segmenteringene, bde 2D og 3D (eventuelt ogs CUDA). The results in this table were all aquired by using a single seed point set at an optimal location close to the center of the object being segmented, with a radius of 10. The variable values used were found to be the fastest ones,

given a specific object, while resulting in a correct and complete segmentation. The time was taken for only the segmentation process, thus not including the time used to read input, initialize and write result back to file.

Chapter 5

Discussion and future work

5.1 Modification of the speed function

As mentioned in the when discussing the speed function in chapter 3 the data function was modified a little. The reason for this modification is that the zero level set evolved very little from iteration to iteration using the originally defined speed function. This is true especially when using a low ϵ value is used, because the maximum value the data function can have (before the modification) is ϵ. So with a low value of ϵ and the data term weighted high (high α value) the resulting speed function would have a low value, even if the curvature function gives good result for expandation (high α means low weighting on curvature).

alpha har endret seg mye, fordi fr mtte alpha vektes veldig hy for f speed funksjonen for utvikle seg, men n som data termen er blit sterkere i seg selv, s trenger ikke alpha vektes s mye. denne nye modifiserte speed funksjonen vekter dermed

scaling of the curvature

clamping of the speed function result

ikke noe skikkelig program for visualisere volumer skikkelig

Bibliography

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

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

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

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

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

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

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

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

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

Source code - 2D sparse field

main.cpp

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "main.h"
5 #include <list>
6 #include "EasyBMP.h" //library for reading bmp files
7 #include "update.h" //levelset evolvement happens here
8 #include <cstdio> //to calculate runtime
9 #include <ctime>  //to calculate runtime
10
11 #include <sstream>
12 using namespace std;
13
14 //to calculate runtime
15 clock_t start;
16 double duration;
17
18 float image[HEIGHT][WIDTH] = { 0 }; //input -> image to be
       segmented
19 float phi[HEIGHT+BORDER][WIDTH+BORDER] = { 0 }; //level set
20 short init[HEIGHT+BORDER][WIDTH+BORDER] = { 0 }; //binary mask
       with seed points
21 int label[HEIGHT+BORDER][WIDTH+BORDER] = { 0 }; //contains info
       about the layers
22 int zeroLevelSet[HEIGHT][WIDTH] = { 0 }; //output
23
24 int iterations;
25 float threshold, alpha, epsilon;
26
27 list<Pixel> lz;
28 list<Pixel> lp1;
29 list<Pixel> ln1;
30 list<Pixel> lp2;
31 list<Pixel> ln2;
32
33 list<Pixel> sz;
34 list<Pixel> sp1;
35 list<Pixel> sn1;
36 list<Pixel> sp2;
37 list<Pixel> sn2;
38
39 //fills init with circular seed point, returns 1 if success
40 int fillSphere(int seedX, int seedY, int radius){
41   //(seedX, seedY) -> coordinates - center of seed point
42   if(seedX < 0 || seedX > HEIGHT || seedY < 0 || seedY > WIDTH){
```

```
43      printf("Wrong input to create a circular seed\n");
44      printf("Coordinates out of range\n");
45      return 0;
46    }
47    else if(radius < 1 || radius > HEIGHT/2 || radius > WIDTH/2){
48      printf("Wrong input to create a circular seed\n");
49      printf("Radius must be a positive integer less than min(
            width, height)/2\n");
50      return 0;
51    }
52    for(int i = seedX - radius; i < seedX + radius; i++){
53      for(int j = seedY - radius; j < seedY + radius; j++){
54        if(sqrt((float)((seedX-i)*(seedX-i)+(seedY-j)*(seedY-j)))
              < radius){
55          init[i][j] = 1;
56        }
57      }
58    }
59    return 1;
60  }
61
62  //can replace fillSphere() if a rectangular seed point is wanted
63  int fillRect(short minX, short minY, short maxX, short maxY){
64    //(minX, minY) -> upper left corner
65    //(maxX, maxY) -> lower right corner
66    if(maxX - minX <= 0 || maxY - minY <= 0){
67      printf("Wrong input to create a rectangular seed\n");
68      printf("Input must be in this order minX minY maxX maxY\n");
69      return 0;
70    }
71    else if(minX < 0 || maxX >= HEIGHT || minY < 0 || maxY >=
          WIDTH){
72      printf("Wrong input to create a rectangular seed\n");
73      printf("Input out of range\n");
74      return 0;
75    }
76    for (int i = minY+1; i<maxY+1; i++){
77      for (int j = minX+1; j<maxX+1; j++){
78        init[i][j] = 1;
79      }
80    }
81    return 1;
82  }
83
84  /* returns true if any neighbour of coordinates (i,j) in either
85     init[][] (id = 1) or label[][] (id = 2) equals res */
86  bool checkMaskNeighbours(short i, short j, int id, short res){
87    if(id == 1){ //id == 1 -> init
88      if(init[i+1][j] == res)
```

```
89        return true;
90      else if(init[i-1][j] == res)
91        return true;
92      else if(init[i][j+1] == res)
93        return true;
94      else if(init[i][j-1] == res)
95        return true;
96    }
97    else if(id == 2){ //id == 2 -> label
98      if(label[i+1][j] == res)
99        return true;
100     else if(label[i-1][j] == res)
101       return true;
102     else if(label[i][j+1] == res)
103       return true;
104     else if(label[i][j-1] == res)
105       return true;
106   }
107   return false;
108 }
109
110 //add pixels to lists according to their label
111 void assignLabel(Pixel p, short level){
112   switch(level){
113   case 1:
114     lp1.push_back(p);
115     label[p.x][p.y] = level;
116     phi[p.x][p.y] = level;
117     break;
118   case 2:
119     lp2.push_back(p);
120     label[p.x][p.y] = level;
121     phi[p.x][p.y] = level;
122     break;
123   case -1:
124     ln1.push_back(p);
125     label[p.x][p.y] = level;
126     phi[p.x][p.y] = level;
127     break;
128   case -2:
129     ln2.push_back(p);
130     label[p.x][p.y] = level;
131     phi[p.x][p.y] = level;
132     break;
133   }
134 }
135
136 void setLevels(Pixel p, short level){
137   if(label[p.x+1][p.y] == 3){
```

```
138       assignLabel(Pixel(p.x+1, p.y), level);
139     }
140     if(label[p.x][p.y+1] == 3){
141       assignLabel(Pixel(p.x, p.y+1), level);
142     }
143     if(label[p.x-1][p.y] == 3){
144       assignLabel(Pixel(p.x-1, p.y), level);
145     }
146     if(label[p.x][p.y-1] == 3){
147       assignLabel(Pixel(p.x, p.y-1), level);
148     }
149     if(label[p.x+1][p.y] == -3){
150       assignLabel(Pixel(p.x+1, p.y), -level);
151     }
152     if(label[p.x][p.y+1] == -3){
153       assignLabel(Pixel(p.x, p.y+1), -level);
154     }
155     if(label[p.x-1][p.y] == -3){
156       assignLabel(Pixel(p.x-1, p.y), -level);
157     }
158     if(label[p.x][p.y-1] == -3){
159       assignLabel(Pixel(p.x, p.y-1), -level);
160     }
161 }
162
163 //initializes Ln2, Ln1, Lz, Lp1, Lp2 based on seed point(s)
164 void initialization(){
165   list<Pixel>::iterator it;
166
167   for (int i = 0; i<HEIGHT+BORDER; i++){
168     for (int j = 0; j<WIDTH+BORDER; j++){
169       if(init[i][j] == 0){
170         label[i][j] = 3;
171         phi[i][j] = 3;
172       }
173       else{
174         label[i][j] = -3;
175         phi[i][j] = -3;
176       }
177     }
178   }
179   for (int i = 1; i<HEIGHT+1; i++){
180     for (int j = 1; j<WIDTH+1; j++){
181       if(init[i][j] == 1 && checkMaskNeighbours(i, j, 1, 0) ==
            true){
182         lz.push_back(Pixel(i,j));
183         label[i][j] = 0;
184         phi[i][j] = 0;
185       }
```

```
186       }
187     }
188     for (it = lz.begin(); it != lz.end(); it++){
189       setLevels(*it, 1); //add to either Lp1 or Ln1
190     }
191     for (it = lp1.begin(); it != lp1.end(); it++){
192       setLevels(*it, 2); //add to Lp2
193     }
194     for (it = ln1.begin(); it != ln1.end(); it++){
195       setLevels(*it, 2); //add to Ln2
196     }
197 }
198
199 //read input file
200 void readFile(BMP img){
201   //copy input data (img) to image[][] and normalize to [0, 1]
202   for (int i =0; i<HEIGHT; i++){
203     for (int j = 0; j<WIDTH; j++){
204       image[i][j] = (img(i,j)->Red + img(i,j)->Green + img(i,j)
              ->Blue) / 3;
205       image[i][j] /= 255;
206     }
207   }
208 }
209
210 //store output data to disk
211 void writeFile(BMP img, int id, int iter){
212   string name;
213   stringstream sstm;
214   if(id==1){
215     for (short i =0; i<HEIGHT; i++){
216       for (short j = 0; j<WIDTH; j++){
217         img(i,j)->Red = (label[i][j] +3)*42; //normalize to [0,
              255]
218         img(i,j)->Green = (label[i][j] +3)*42;
219         img(i,j)->Blue = (label[i][j] +3)*42;
220       }
221     }
222
223     sstm << iter << "label" <<".bmp";
224     name = sstm.str();
225     img.WriteToFile(name.c_str());
226     printf("\nlabel image stored");
227   }
228   else{ //zeroLevelSet
229     for (short i =0; i<HEIGHT; i++){
230       for (short j = 0; j<WIDTH; j++){
231         img(i,j)->Red = zeroLevelSet[i][j];
232         img(i,j)->Green = zeroLevelSet[i][j];
```

```
233           img(i,j)->Blue = zeroLevelSet[i][j];
234         }
235       }
236       sstm << iter << "zero" <<".bmp";
237       name = sstm.str();
238       img.WriteToFile(name.c_str());
239       printf("\nzero image stored\n");
240     }
241 }
242
243 bool getAndVerifyInput(int argc, char *argv[]){
244   if(argc != 5){
245     printf("Need four inputs: iterations, threshold, epsilon,
          alpha \n");
246     return false;
247   }
248   if(sscanf (argv[1], "%i", & iterations)!=1 || iterations<0) {
249     printf("Need four inputs: iterations, threshold, epsilon,
          alpha \n");
250     return false;
251   }
252   if(sscanf(argv[2], "%f", &threshold)!=1 || threshold >1){
253     printf("Need four inputs: iterations, threshold, epsilon,
          alpha \n");
254     return false;
255   }
256   if(sscanf(argv[3], "%f", &epsilon)!=1 || epsilon >1){
257     printf("Need four inputs: iterations, threshold, epsilon,
          alpha \n");
258     return false;
259   }
260   if(sscanf(argv[4], "%f", &alpha)!=1 || alpha>1){
261     printf("Need four inputs: iterations, threshold, epsilon,
          alpha \n");
262     return false;
263   }
264   return true;
265 }
266
267 int main(int argc, char *argv[]){
268   //verify input
269   if(!getAndVerifyInput(argc, argv)){
270     system("pause");
271     return 0;
272   }
273
274   //read input file
275   BMP img;
276   img.ReadFromFile("inputImage1.bmp");
```

```
277   readFile(img);
278
279   //set seed point (can do multiple calls to set multiple seed
          points)
280   if(fillSphere(250, 250, 10) == 0){
281     system("pause");
282     return 0;
283   }
284
285   initialization();
286
287   calculateMu(); //only needed if the Chan Vese speed function
          is used
288
289   list<Pixel>::iterator itt;
290   printf("starting main loop\n");
291   start = std::clock();
292   for(int i=1; i<iterations; i++){
293     prepareUpdates();
294     updateLevelSets();
295     if(i%100 == 0){
296       printf("\niteration: %i\n", i);
297     }
298   }
299   duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
300   printf("\nmain loop finished\n");
301   printf("\ntime used: %f\n", duration);
302
303   for(itt = lz.begin(); itt != lz.end(); itt++){
304     zeroLevelSet[itt->x][itt->y] = 255;
305   }
306   writeFile(img, 1, iterations);
307   writeFile(img, 2, iterations);
308   system("pause");
309 }
```

main.h

```
1  #ifndef MAIN_H
2  #define MAIN_H
3
4  using namespace std;
5
6  #define HEIGHT 512
7  #define WIDTH 512
8  #define BORDER 2
9
10 struct Pixel{
```

```
11   short x, y;
12   Pixel(short k, short g):x(k), y(g){};
13 };
14
15 bool checkMaskNeighbours(short i, short j, int id, short res);
16
17 #endif /* MAIN_H */
```

update.cpp

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <cmath>
4  #include <list>
5  #include <exception>
6  #include <algorithm>
7  #include "main.h"
8  #include "update.h"
9  using namespace std;
10
11 /*average values of back- and foreground,
12 only used if the Chan-Vese speedfunciton is used*/
13 float muOutside;
14 float muInside;
15
16 float fResult;
17 //Returns either max or min (based on greaterOrLess) of the
       neighbours, with values less or greater than checkAgainst
18 float follow(Pixel p, short greaterOrLess, short checkAgainst){
19   fResult = checkAgainst;
20   if(greaterOrLess == 1){
21     if(label[p.x+1][p.y] >= fResult){
22       fResult = phi[p.x+1][p.y];
23     }
24     if(label[p.x][p.y+1] >= fResult){
25       fResult = phi[p.x][p.y+1];
26     }
27     if(label[p.x-1][p.y] >= fResult){
28       fResult = phi[p.x-1][p.y];
29     }
30     if(label[p.x][p.y-1] >= fResult){
31       fResult = phi[p.x][p.y-1];
32     }
33   }
34   else if(greaterOrLess == -1){
35     if(label[p.x+1][p.y] <= fResult){
36       fResult = phi[p.x+1][p.y];
37     }
```

```
38      if(label[p.x][p.y+1] <= fResult){
39        fResult = phi[p.x][p.y+1];
40      }
41      if(label[p.x-1][p.y] <= fResult){
42        fResult = phi[p.x-1][p.y];
43      }
44      if(label[p.x][p.y-1] <= fResult){
45        fResult = phi[p.x][p.y-1];
46      }
47    }
48    return fResult;
49  }
50
51  /*calculates the average back- and foreground values
52  only used if the Chan-Vese speedfunciton is used*/
53  void calculateMu(){
54    float muTempInside = 0;
55    int numInside = 0;
56    float muTempOutside = 0;
57    int numOutside = 0;
58    double threshold = 0.5;
59    for(int i = 0; i<HEIGHT; i++){
60      for(int j = 0; j<WIDTH; j++){
61        if(image[i][j] > threshold){
62          muTempInside += image[i][j];
63          numInside++;
64        }
65        else if(image[i][j] < threshold){
66          muTempOutside += image[i][j];
67          numOutside++;
68        }
69      }
70    }
71    muOutside = muTempOutside / numOutside;
72    muInside = muTempInside / numInside;
73    if(numInside == 0){
74      muInside = 1;
75    }
76    if(numOutside == 0){
77      muOutside = 0;
78    }
79  }
80
81  float speedFunctionChanVese(int x, int y){
82    return (((image[x][y] - muInside)*(image[x][y] - muInside)) -
          ((image[x][y] - muOutside)*(image[x][y] - muOutside)))/2;
83  }
84
85  float curvature;
```

```
86 float speedFunction(short i, short j){
87   float data = epsilon - abs(image[i][j] - threshold); //the
         data term (based on pixel intensity)
88   D1 d1 = D1(i, j); //calculates the first order derivatives
89   D2 d2 = D2(i, j); //calculates the second order derivatives
90   Normal n = Normal(d1, d2); //calculates the normals
91   curvature = (n.nPlusX - n.nMinusX) + (n.nPlusY - n.nMinusY);
         //the curvature
92   float speed = -alpha*data + (1-alpha)*(curvature/4);
93   if(speed > 1){
94     speed = 1;
95   }
96   if(speed < -1){
97     speed = -1;
98   }
99   return speed;
100 }
101
102 list<Pixel>::iterator it;
103 float M = 0;
104 void prepareUpdates(){
105   for(it = lz.begin(); it != lz.end();){ //Lz
106     phi[it->x][it->y] += speedFunction(it->x, it->y);
107     if(phi[it->x][it->y] >= 0.5){
108       sp1.push_back(*it);
109       it = lz.erase(it);
110     }
111     else if(phi[it->x][it->y] < -0.5){
112       sn1.push_back(*it);
113       it = lz.erase(it);
114     }
115     else{
116       it++;
117     }
118   }
119
120   for(it = ln1.begin(); it != ln1.end();){ //pixels moving out
         of Ln1
121     if(checkMaskNeighbours(it->x,it->y, 2, 0) == false){
122     //if none of the neighbors are in Lz
123       sn2.push_back(*it);
124       it = ln1.erase(it);
125     }
126     else{
127       M = follow(*it, 1, 0);
128       phi[it->x][it->y] = M-1;
129       if(phi[it->x][it->y] >= -0.5){ //moving from ln1 to sz
130         sz.push_back(*it);
131         it = ln1.erase(it);
```

```
132         }
133         else if(phi[it->x][it->y] < -1.5){
134           sn2.push_back(*it);
135           it = ln1.erase(it);
136         }
137         else{
138           it++;
139         }
140       }
141     }
142     for(it = lp1.begin(); it != lp1.end();){ //pixels moving out
          of Lp1
143       if(checkMaskNeighbours(it->x,it->y, 2, 0) == false){
144       //if none of the neighbors are in Lz
145         sp2.push_back(*it);
146         it = lp1.erase(it);
147       }
148       else{
149         M = follow(*it, -1, 0);
150         phi[it->x][it->y] = M+1;
151         if(phi[it->x][it->y] < 0.5){
152           sz.push_back(*it);
153           it = lp1.erase(it);
154         }
155         else if(phi[it->x][it->y] >= 1.5){
156           sp2.push_back(*it);
157           it = lp1.erase(it);
158         }
159         else{
160           it++;
161         }
162       }
163     }
164     for(it = ln2.begin(); it != ln2.end();){
165       if(checkMaskNeighbours(it->x,it->y, 2, -1) == false){
166       //if none of the neighbors are in Ln1
167         label[it->x][it->y] = -3;
168         phi[it->x][it->y] = -3;
169         it = ln2.erase(it);
170       }
171       else{
172         M = follow(*it, 1, -1);
173         phi[it->x][it->y] = M-1;
174         if(phi[it->x][it->y] >= -1.5){
175           sn1.push_back(*it);
176           it = ln2.erase(it);
177         }
178         else if(phi[it->x][it->y] < -2.5){
179           label[it->x][it->y] = -3;
```

```
180           phi[it->x][it->y] = -3;
181           it = ln2.erase(it);
182         }
183         else{
184           it++;
185         }
186       }
187     }
188     for(it = lp2.begin(); it != lp2.end();){
189       if(checkMaskNeighbours(it->x,it->y, 2, 1) == false){
190       //if none of the neighbors are in Lp1
191         label[it->x][it->y] = 3;
192         phi[it->x][it->y] = 3;
193         it = lp2.erase(it);
194       }
195       else{
196         M = follow(*it, -1, 1);
197         phi[it->x][it->y] = M+1;
198         if(phi[it->x][it->y] < 1.5){
199           sp1.push_back(*it);
200           it = lp2.erase(it);
201         }
202         else if(phi[it->x][it->y] >= 2.5){
203           label[it->x][it->y] = 3;
204           phi[it->x][it->y] = 3;
205           it = lp2.erase(it);
206         }
207         else{
208           it++;
209         }
210       }
211     }
212 }
213
214 void updateLevelSets(){
215     for (it = sz.begin(); it != sz.end(); it++){
216       label[it->x][it->y] = 0;
217       lz.push_back(*it);
218     }
219     sz.clear();
220
221     for (it = sn1.begin(); it != sn1.end(); it++){
222       label[it->x][it->y] = -1;
223       ln1.push_back(*it);
224       if (phi[it->x+1][it->y] == -3){
225         phi[it->x+1][it->y] = phi[it->x][it->y]-1;
226         it->x++;                    //[x+1,y]
227         sn2.push_back(*it);
```

```
228         it->x--;                    //Siden vi bruker if, og ikke if
                else setninger m verdien settes tilbake til x
229       }
230       if (phi[it->x][it->y+1] == -3){
231         phi[it->x][it->y+1] = phi[it->x][it->y]-1;
232         it->y++;                    //[x,y+1]
233         sn2.push_back(*it);
234         it->y--;
235       }
236       if (phi[it->x-1][it->y] == -3){
237         phi[it->x-1][it->y] = phi[it->x][it->y]-1;
238         it->x--;                    //[x-1,y]
239         sn2.push_back(*it);
240         it->x++;
241       }
242       if (phi[it->x][it->y-1] == -3){
243         phi[it->x][it->y-1] = phi[it->x][it->y]-1;
244         it->y--;                    //[x,y-1]
245         sn2.push_back(*it);
246         it->y++;
247       }
248     }
249     sn1.clear();
250
251     for (it = sp1.begin(); it != sp1.end(); it++){
252       label[it->x][it->y] = 1;
253       lp1.push_back(*it);
254       if (phi[it->x+1][it->y] == 3){
255         phi[it->x+1][it->y] = phi[it->x][it->y]+1;
256         it->x++;                     //[x+1,y]
257         sp2.push_back(*it);
258         it->x--;
259       }
260       if (phi[it->x][it->y+1] == 3){
261         phi[it->x][it->y+1] = phi[it->x][it->y]+1;
262         it->y++;                     //[x,y+1]
263         sp2.push_back(*it);
264         it->y--;
265       }
266       if (phi[it->x-1][it->y] == 3){
267         phi[it->x-1][it->y] = phi[it->x][it->y]+1;
268         it->x--;
269         sp2.push_back(*it);
270         it->x++;
271       }
272       if (phi[it->x][it->y-1] == 3){
273         phi[it->x][it->y-1] = phi[it->x][it->y]+1;
274         it->y--;
275         sp2.push_back(*it);
```

```
276         it->y++;
277       }
278     }
279     sp1.clear();
280
281     for (it = sn2.begin(); it != sn2.end(); it++){
282       label[it->x][it->y] = -2;
283       ln2.push_back(*it);
284     }
285     sn2.clear();
286
287     for (it = sp2.begin(); it != sp2.end(); it++){
288       label[it->x][it->y] = 2;
289       lp2.push_back(*it);
290     }
291     sp2.clear();
292 }
```

update.h

```
1  extern float image[HEIGHT][WIDTH]; //image to be segmented
2  extern short init[HEIGHT+BORDER][WIDTH+BORDER]; //mask with seed
       points
3  extern float phi[HEIGHT+BORDER][WIDTH+BORDER]; //representation
       of the zero level set interface
4  extern int label[HEIGHT+BORDER][WIDTH+BORDER];//contains only
       integer values between -3 and 3
5  extern float F[HEIGHT][WIDTH];
6
7  extern list<Pixel> lz; // zero level set
8  extern list<Pixel> lp1;
9  extern list<Pixel> ln1;
10 extern list<Pixel> lp2;
11 extern list<Pixel> ln2;
12
13 extern float threshold, alpha, epsilon;
14
15 //temp values
16 extern list<Pixel> sz; //temp for lz
17 extern list<Pixel> sp1;
18 extern list<Pixel> sn1;
19 extern list<Pixel> sp2;
20 extern list<Pixel> sn2;
21
22 void prepareUpdates();
23 void updateLevelSets();
24 void calculateMu();
25
```

```
26 struct D1{ //first order derivative
27   float dx, dy, dxPlus, dyPlus, dxMinus, dyMinus;
28   D1(short i, short j){
29     dx = (phi[i+1][j] - phi[i-1][j]) / 2;
30     dy = (phi[i][j+1] - phi[i][j-1]) / 2;
31     dxPlus = phi[i+1][j] - phi[i][j];
32     dyPlus = phi[i][j+1] - phi[i][j];
33     dxMinus = phi[i][j] - phi[i-1][j];
34     dyMinus = phi[i][j] - phi[i][j-1];
35   }
36 };
37
38 struct D2{ //second order derivatives
39   float dxPlusY, dxMinusY, dyPlusX, dyMinusX;
40   D2(short i, short j){
41     dxPlusY = (phi[i+1][j+1] - phi[i-1][j+1])/2;
42     dxMinusY = (phi[i+1][j-1] - phi[i-1][j-1])/2;
43     dyPlusX = (phi[i+1][j+1] - phi[i+1][j-1])/2;
44     dyMinusX = (phi[i-1][j+1] - phi[i-1][j-1])/2;
45   }
46 };
47
48 struct Normal{ //normals
49   float nPlusX, nPlusY, nMinusX, nMinusY;
50   Normal(D1 d1, D2 d2){
51     nPlusX = d1.dxPlus / sqrt(d1.dxPlus*d1.dxPlus + pow((d2.
         dyPlusX + d1.dy) / 2, 2));
52     nPlusY = d1.dyPlus / sqrt(d1.dyPlus*d1.dyPlus + pow((d2.
         dxPlusY + d1.dx) / 2, 2));
53     nMinusX = d1.dxMinus / sqrt(d1.dxMinus * d1.dxMinus + pow((
         d2.dyMinusX + d1.dy) / 2, 2));
54     nMinusY = d1.dyMinus / sqrt(d1.dyMinus * d1.dyMinus + pow((
         d2.dxMinusY + d1.dx) / 2, 2));
55   }
56 };
```

Source code - 2D CUDA sparse field

main.cu

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "main.h"
#include "update.h" //levelset evolvement happens here
#include "EasyBMP.h" //library for reading bmp files
#include "IO.h" //handles input and stores output
#include <cstdio> //to calculate runtime
#include <ctime>  //to calculate runtime
using namespace std;

float image[HEIGHT][WIDTH] = { 0 }; //input -> image to be
    segmented
float phi[HEIGHT][WIDTH] = { 0 };
int init[HEIGHT][WIDTH] = { 0 };
int label[HEIGHT][WIDTH] = { 0 };
int zeroLevelSet[HEIGHT][WIDTH] = { 0 }; //output
int layer[HEIGHT][WIDTH]; //-> see main.h for details

int iterations;
float threshold, alpha, epsilon;

//to calculate runtime
clock_t start;
double duration;

//device arrays
float *phiD;
int *labelD;
int *layerD;
float *imageD;

__device__ float thresholdD, alphaD, epsilonD;

//fills init with circular seed point, returns 1 if success
int fillSphere(int seedX, int seedY, int radius){
  if(seedX < 0 || seedX > HEIGHT || seedY < 0 || seedY > WIDTH){
    printf("Wrong input to create a circular seed\n");
    printf("Coordinates out of range\n");
    return 0;
  }
  else if(radius < 1 || radius > HEIGHT/2 || radius > WIDTH/2){
    printf("Wrong input to create a circular seed\n");
```

```
44      printf("Radius must be a positive integer less than min(
            width, height)/2\n");
45      return 0;
46    }
47    for(int i = seedX - radius; i < seedX + radius; i++){
48      for(int j = seedY - radius; j < seedY + radius; j++){
49        if(sqrt((float)((seedX-i)*(seedX-i)+(seedY-j)*(seedY-j)))
              < radius){
50          init[i][j] = 1;
51        }
52      }
53    }
54    return 1;
55  }
56
57  /* returns true if any neighbour of coordinates (i,j) in either
58     init[][] (id = 1) or label[][] (id = 2) equals res */
59  bool checkMaskNeighbours(int i, int j, int res){
60    if(init[i+1][j] == res)
61      return true;
62    else if(init[i-1][j] == res)
63      return true;
64    else if(init[i][j+1] == res)
65      return true;
66    else if(init[i][j-1] == res)
67      return true;
68    return false;
69  }
70
71  //add pixels to lists according to their label
72  void assignLabel(int i, int j, int level){
73    switch(level){
74    case 1:
75      layer[i][j] = 16; //add to lp1
76      label[i][j] = level;
77      phi[i][j] = level;
78      break;
79    case 2:
80      layer[i][j] = 17; //add to lp2
81      label[i][j] = level;
82      phi[i][j] = level;
83      break;
84    case -1:
85      layer[i][j] = 14; //add to ln1
86      label[i][j] = level;
87      phi[i][j] = level;
88      break;
89    case -2:
90      layer[i][j] = 13; //add to ln2
```

```
91      label[i][j] = level;
92      phi[i][j] = level;
93      break;
94    }
95  }
96
97  void setLevels(int i, int j, int level){
98    if(label[i+1][j] == 3){
99      assignLabel(i+1, j, level);
100   }
101   if(label[i][j+1] == 3){
102     assignLabel(i, j+1, level);
103   }
104   if(label[i-1][j] == 3){
105     assignLabel(i-1, j, level);
106   }
107   if(label[i][j-1] == 3){
108     assignLabel(i, j-1, level);
109   }
110
111   if(label[i+1][j] == -3){
112     assignLabel(i+1, j, -level);
113   }
114   if(label[i][j+1] == -3){
115     assignLabel(i, j+1, -level);
116   }
117   if(label[i-1][j] == -3){
118     assignLabel(i-1, j, -level);
119   }
120   if(label[i][j-1] == -3){
121     assignLabel(i, j-1, -level);
122   }
123 }
124
125 //initializes Ln2, Ln1, Lz, Lp1, Lp2 based on seed point(s)
126 void initialization(){
127   for (int i = 0; i<HEIGHT; i++){
128     for (int j = 0; j<WIDTH; j++){
129       if(init[i][j] == 0){
130         label[i][j] = 3;
131         phi[i][j] = 3;
132       }
133       else{
134         label[i][j] = -3;
135         phi[i][j] = -3;
136       }
137     }
138   }
139   for (int i = 0; i<HEIGHT; i++){
```

```
140       for (int j = 0; j<WIDTH; j++){
141         if(init[i][j] == 1 && checkMaskNeighbours(i, j, 0) == true
                ){
142           layer[i][j] = 15; //lz
143           label[i][j] = 0;
144           phi[i][j] = 0;
145         }
146       }
147     }
148     for (int i = 0; i<HEIGHT; i++){
149       for (int j = 0; j<WIDTH; j++){
150         if(layer[i][j] == 15){ //add to lz
151           setLevels(i, j, 1);
152         }
153       }
154     }
155     for (int i = 0; i<HEIGHT; i++){
156       for (int j = 0; j<WIDTH; j++){
157         if(layer[i][j] == 16){ //add to  lp1
158           setLevels(i, j, 2);
159         }
160       }
161     }
162     for (int i = 0; i<HEIGHT; i++){
163       for (int j = 0; j<WIDTH; j++){
164         if(layer[i][j] == 14){ //add to ln1
165           setLevels(i, j, 2);
166         }
167       }
168     }
169 }
170
171 //allocate and copy data to device
172 void setUpDeviceArrays(){
173     int err;
174     const size_t arrSize = size_t(HEIGHT*WIDTH);
175     err = cudaMalloc((void**)&phiD, sizeof(float)*arrSize);
176     if(err != cudaSuccess){
177       printf("phiD cudaMalloc error: %d\n", err);
178     }
179     err = cudaMalloc((void**)&labelD, sizeof(int)*arrSize);
180     if(err != cudaSuccess){
181       printf("labelD cudaMalloc error: %d\n", err);
182     }
183     err = cudaMalloc((void**)&layerD, sizeof(int)*arrSize);
184     if(err != cudaSuccess){
185       printf("layerD cudaMalloc error: %d\n", err);
186     }
187     err = cudaMalloc((void**)&imageD, sizeof(float)*arrSize);
```

```
188   if(err != cudaSuccess){
189     printf("imageD cudaMalloc error: %d\n", err);
190   }
191
192   err = cudaMemcpy(phiD, phi, sizeof(float)*arrSize,
          cudaMemcpyHostToDevice);
193   if(err != cudaSuccess){
194     printf("phiD cudaMemcpy error: %d\n", err);
195   }
196   err = cudaMemcpy(labelD, label, sizeof(int)*arrSize,
          cudaMemcpyHostToDevice);
197   if(err != cudaSuccess){
198     printf("labelD cudaMemcpy error: %d\n", err);
199   }
200   err = cudaMemcpy(layerD, layer, sizeof(int)*arrSize,
          cudaMemcpyHostToDevice);
201   if(err != cudaSuccess){
202     printf("layerD cudaMemcpy error: %d\n", err);
203   }
204   err = cudaMemcpy(imageD, image, sizeof(float)*arrSize,
          cudaMemcpyHostToDevice);
205   if(err != cudaSuccess){
206     printf("imageD cudaMemcpy error: %d\n", err);
207   }
208 }
209
210 int main(int argc, char *argv[]){ printf("1");
211   if(!getAndVerifyInput(argc, argv)){
212     system("pause");
213     return 0;
214   }
215
216   //read file
217   BMP img;
218   img.ReadFromFile("inputImage1.bmp");
219   readFile(img);
220
221   if(fillSphere(250, 255, 10) == 0){
222     system("pause");
223     return 0;
224   }
225
226   initialization();
227   setUpDeviceArrays(); //copy over data to device
228   setVariablesInDevice<<<1,1>>>(threshold, epsilon, alpha, image
          );
229
230   const dim3 BlockDim(16,16);
231     dim3 GridDim;
```

```
232     GridDim.x = (WIDTH + BlockDim.x - 1) / BlockDim.x;
233     GridDim.y = (HEIGHT + BlockDim.y - 1) / BlockDim.y;
234
235   printf("starting main loop\n");
236   start = std::clock();
237   for(int i=0; i<iterations+1; i++){
238     if(i%100 == 0){
239       printf("iteration: %i\n", i);
240     }
241     prepareUpdates1<<<GridDim, BlockDim>>>(phiD, layerD, imageD)
          ;
242     prepareUpdates2<<<GridDim, BlockDim>>>(phiD, layerD, labelD)
          ;
243     prepareUpdates3<<<GridDim, BlockDim>>>(phiD, layerD, labelD)
          ;
244     prepareUpdates4<<<GridDim, BlockDim>>>(phiD, layerD, labelD)
          ;
245     updateLevelSets1<<<GridDim, BlockDim>>>(phiD, layerD, labelD
          );
246     updateLevelSets2<<<GridDim, BlockDim>>>(layerD, labelD);
247   }
248   duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
249   printf("\nmain loop finished\n");
250   printf("\ntime used: %f\n", duration);
251
252   int err = cudaMemcpy(label, labelD, sizeof(int)*(HEIGHT)*(
        WIDTH), cudaMemcpyDeviceToHost);
253   if(err != cudaSuccess){
254     printf("cudaMemcpy error when writing to zeroLevelset: %d\n"
          , err);
255   }
256   for (int i = 1; i<HEIGHT; i++){
257     for (int j = 1; j<WIDTH; j++){
258       if(label[i][j] == 0){ //lz
259         zeroLevelSet[i][j] = 255;
260       }
261     }
262   }
263
264   writeFile(img, 1, iterations); //store label as image
265   writeFile(img, 2, iterations); //store zerolevel set as image
266
267   system("pause");
268 }
```

main.h

```
1 #define HEIGHT 512
```

```
2 #define WIDTH 512
3 /*int layer[][]:
4   - array with two digit numbers
5   - first digit:
6     - 1: corresponding to being in any of the layers Ln2, Ln1,
          Lz, Lp1, Lp2
7     - 2: corresponding to being in any of the layers Sn2, Sn1,
          Sz, Sp1, Sp2
8   - second digit:
9     - 3: Ln2 or Sn2
10    - 4: Ln1 or Sn1
11    - 5: Lz or Sz
12    - 6: Lp1 or Sp1
13    - 7: Lp2 or Sp2
14  - Could have used two seperate integer values, but then a lot
        of additional
15    checks in the code would be needed.
16 */
```

update.cu

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <cmath>
4 #include <cuda.h>
5 #include <exception>
6 #include <algorithm>
7 #include "main.h"
8 #include "update.h"
9 using namespace std;
10
11 extern float *phiD;
12 extern int *labelD;
13 extern int *layerD;
14 extern float *imageD;
15 extern __device__ float thresholdD, epsilonD, alphaD;
16
17 void __global__ setVariablesInDevice(float threshold, float
       epsilon, float alpha, float image[HEIGHT][WIDTH]){
18   thresholdD = threshold;
19   epsilonD = epsilon;
20   alphaD = alpha;
21 }
22
23 //nvcc --machine 32 -arch sm_20 main.cu update.cu IO.cu EasyBMP.
       cpp
24
```

```
25 //Returns either max or min (based on greaterOrLess) of the
       neighbours, with values less or greater than checkAgainst
26 __device__ float follow(int i, int j, int greaterOrLess, int
       checkAgainst, float *phiD, int *labelD){
27   float fResult = checkAgainst;
28   if(greaterOrLess == 1){
29     if(labelD[(i+1)*WIDTH+j] >= fResult){
30       fResult = phiD[(i+1)*WIDTH+j];
31     }
32     if(labelD[i*WIDTH+(j+1)] >= fResult){
33       fResult = phiD[i*WIDTH + (j+1)];
34     }
35     if(labelD[(i-1)*WIDTH + j] >= fResult){
36       fResult = phiD[(i-1)*WIDTH+j];
37     }
38     if(labelD[i*WIDTH+(j-1)] >= fResult){
39       fResult = phiD[i*WIDTH+(j-1)];
40     }
41   }
42   else if(greaterOrLess == -1){
43     if(labelD[(i+1)*WIDTH+j] <= fResult){
44       fResult = phiD[(i+1)*WIDTH+j];
45     }
46     if(labelD[i*WIDTH+(j+1)] <= fResult){
47       fResult = phiD[i*WIDTH+(j+1)];
48     }
49     if(labelD[(i-1)*WIDTH+j] <= fResult){
50       fResult = phiD[(i-1)*WIDTH+j];
51     }
52     if(labelD[i*WIDTH+(j-1)] <= fResult){
53       fResult= phiD[i*WIDTH+(j-1)];
54     }
55   }
56   return fResult;
57 }
58
59 __device__ bool checkMaskNeighbours2(int i, int j, short res,
       int *labelD){
60   if(labelD[(i+1)*WIDTH+j] == res)
61     return true;
62   else if(labelD[(i-1)*WIDTH+j] == res)
63     return true;
64   else if(labelD[i*WIDTH+(j+1)] == res)
65     return true;
66   else if(labelD[i*WIDTH+(j-1)] == res)
67     return true;
68   return false;
69 }
70
```

```
71 __device__ float speedFunction(int i, int j, float *phiD, float
       *imageD){
72   //calculate data term
73   float data = epsilonD - abs(imageD[i*WIDTH+j] - thresholdD);
         //the data term (based on pixel intensity)
74   //calculate first order derivatives
75   float dx = (phiD[(i+1)*WIDTH+j] - phiD[(i-1)*WIDTH+ j]) / 2;
76   float dy = (phiD[i*WIDTH+(j+1)] - phiD[i*WIDTH+(j-1)]) / 2;
77   float dxPlus = phiD[(i+1)*WIDTH+j] - phiD[i*WIDTH+j];
78   float dyPlus = phiD[i*WIDTH+(j+1)] - phiD[i*WIDTH+j];
79   float dxMinus = phiD[i*WIDTH+j] - phiD[(i-1)*WIDTH+ j];
80   float dyMinus = phiD[i*WIDTH+j] - phiD[i*WIDTH+(j-1)];
81   //calculate second order derivatives
82   float dxPlusY = (phiD[(i+1)*WIDTH+(j+1)] - phiD[(i-1)*WIDTH+(j
         +1)])/2;
83   float dxMinusY = (phiD[(i+1)*WIDTH+(j-1)] - phiD[(i-1)*WIDTH+(
         j-1)])/2;
84   float dyPlusX = (phiD[(i+1)*WIDTH+(j+1)] - phiD[(i+1)*WIDTH+(j
         -1)])/2;
85   float dyMinusX = (phiD[(i-1)*WIDTH+(j+1)] - phiD[(i-1)*WIDTH+(
         j-1)])/2;
86   //calculate normals
87   float nPlusX = dxPlus / sqrt(dxPlus*dxPlus + pow((dyPlusX + dy
         ) / 2, 2));
88   float nPlusY = dyPlus / sqrt(dyPlus*dyPlus + pow((dxPlusY + dx
         ) / 2, 2));
89   float nMinusX = dxMinus / sqrt(dxMinus * dxMinus + pow((
         dyMinusX + dy) / 2, 2));
90   float nMinusY = dyMinus / sqrt(dyMinus * dyMinus + pow((
         dxMinusY + dx) / 2, 2));
91   //calculate curvature
92   float curvature = (nPlusX - nMinusX) + (nPlusY - nMinusY);
93   //calculate the speeed
94   float speed = -alphaD*data + (1.0f-alphaD)*(curvature/4.0f);
         //divided by 4 to narmalize (max(curvature) = 4)
95   //clamp speed
96   if(speed > 1.0f){
97     speed = 1.0f;
98   }
99   if(speed < -1.0f){
100    speed = -1.0f;
101  }
102  return speed;
103 }
104
105 __global__ void prepareUpdates1(float *phiD, int *layerD, float
       *imageD){
106  int i = threadIdx.x + blockDim.x * blockIdx.x;
107  int j = threadIdx.y + blockDim.y * blockIdx.y;
```

```
108   if(layerD[i*WIDTH+j] == 15){ //lz
109     phiD[i*WIDTH+j] += speedFunction(i,j, phiD, imageD);
110     if(phiD[i*WIDTH+j] >= 0.5){
111       layerD[i*WIDTH+j] = 26; //add to sp1
112
113     }
114     else if(phiD[i*WIDTH+j] < -0.5){
115       layerD[i*WIDTH+j] = 24; //add to sn1
116     }
117   }
118 }
119
120 __global__ void prepareUpdates2(float *phiD, int *layerD, int *
      labelD){
121   int i = threadIdx.x + blockDim.x * blockIdx.x;
122   int j = threadIdx.y + blockDim.y * blockIdx.y;
123   if(i !=0 && j !=0 && i < HEIGHT-1 && j < WIDTH-1){
124     float M = 0;
125     if(layerD[i*WIDTH+j] == 14){ //ln1
126       if(checkMaskNeighbours2(i, j, 0, labelD) == false){
127         layerD[i*WIDTH+j] = 23; //add to sn2
128       }
129       else{
130         M = follow(i, j, 1, 0, phiD, labelD);
131         phiD[i*WIDTH+j] = M-1;
132         if(phiD[i*WIDTH+j] >= -0.5){
133           layerD[i*WIDTH+j] = 25; //add to sz
134         }
135         else if(phiD[i*WIDTH+j] < -1.5){
136           layerD[i*WIDTH+j] = 23; //add to sn2
137         }
138       }
139     }
140   }
141 }
142
143 __global__ void prepareUpdates3(float *phiD, int *layerD, int *
      labelD){
144   int i = threadIdx.x + blockDim.x * blockIdx.x;
145   int j = threadIdx.y + blockDim.y * blockIdx.y;
146   if(i !=0 && j !=0 && i < HEIGHT-1 && j < WIDTH-1){
147     float M = 0;
148     if(layerD[i*WIDTH+j] == 16){ //lp1
149       if(checkMaskNeighbours2(i, j, 0, labelD) == false){
150         layerD[i*WIDTH+j] = 27; //add to sp2
151       }
152       else{
153         M = follow(i, j, -1, 0, phiD, labelD);
154         phiD[i*WIDTH+j] = M+1;
```

```
155           if(phiD[i*WIDTH+j] < 0.5){
156             layerD[i*WIDTH+j] = 25; //add to sz
157           }
158           else if(phiD[i*WIDTH+j] >= 1.5){
159             layerD[i*WIDTH+j] = 27; //add to sp2
160           }
161         }
162       }
163     }
164 }
165
166 __global__ void prepareUpdates4(float *phiD, int *layerD, int *
      labelD){
167   int i = threadIdx.x + blockDim.x * blockIdx.x;
168   int j = threadIdx.y + blockDim.y * blockIdx.y;
169   if(i !=0 && j !=0 && i < HEIGHT-1 && j < WIDTH-1){
170     float M = 0;
171     if(layerD[i*WIDTH+j] == 13){ //ln2
172       if(checkMaskNeighbours2(i, j, -1, labelD) == false){
173         labelD[i*WIDTH+j] = -3;
174         phiD[i*WIDTH+j] = -3;
175         layerD[i*WIDTH+j] = 0; //no longer part of ln2
176       }
177       else{
178         M = follow(i, j, 1, -1, phiD, labelD);
179         phiD[i*WIDTH+j] = M-1;
180         if(phiD[i*WIDTH+j] >= -1.5){
181           layerD[i*WIDTH+j] = 24; //add to sn1
182         }
183         else if(phiD[i*WIDTH+j] < -2.5){
184           labelD[i*WIDTH+j] = -3;
185           phiD[i*WIDTH+j] = -3;
186           layerD[i*WIDTH+j] = 0; //no longer part of ln2
187         }
188       }
189     }
190
191     if(layerD[i*WIDTH+j] == 17){ //lp2
192       if(checkMaskNeighbours2(i, j, 1, labelD) == false){
193         labelD[i*WIDTH+j] = 3;
194         phiD[i*WIDTH+j] = 3;
195         layerD[i*WIDTH+j] = 0; //no longer part of lp2
196       }
197       else{
198         M = follow(i, j, -1, 1, phiD, labelD);
199         phiD[i*WIDTH+j] = M+1;
200         if(phiD[i*WIDTH+j] < 1.5){
201           layerD[i*WIDTH+j] = 26; //add to sp1
202         }
```

```
203           else if(phiD[i*WIDTH+j] >= 2.5){
204             labelD[i*WIDTH+j] = 3;
205             phiD[i*WIDTH+j] = 3;
206             layerD[i*WIDTH+j] = 0; //no longer part of lp2
207           }
208         }
209       }
210     }
211 }
212
213 __global__ void updateLevelSets1(float *phiD, int *layerD, int *
      labelD){
214   int i = threadIdx.x + blockDim.x * blockIdx.x;
215   int j = threadIdx.y + blockDim.y * blockIdx.y;
216   if(i !=0 && j !=0 && i < HEIGHT-1 && j < WIDTH-1){
217     if(layerD[i*WIDTH+j] == 25){ //sz
218       labelD[i*WIDTH+j] = 0;
219       layerD[i*WIDTH+j] = 15; //add to lz
220     }
221     if(layerD[i*WIDTH+j] == 24){ //sn1
222       labelD[i*WIDTH+j] = -1;
223       layerD[i*WIDTH+j] = 14; //add to ln1
224       if(phiD[(i+1)*WIDTH+ j] == -3){
225         phiD[(i+1)*WIDTH+ j] = phiD[i*WIDTH+j] - 1;
226         layerD[(i+1)*WIDTH+ j] = 23; //add to sn2
227       }
228       if(phiD[i*WIDTH+ (j+1)] == -3){
229         phiD[i*WIDTH+ (j+1)] = phiD[i*WIDTH+j] - 1;
230         layerD[i*WIDTH+ (j+1)] = 23; //add to sn2
231       }
232       if(phiD[(i-1)*WIDTH+ j] == -3){
233         phiD[(i-1)*WIDTH+ j] = phiD[i*WIDTH+j] - 1;
234         layerD[(i-1)*WIDTH+ j] = 23; //add to sn2
235       }
236       if(phiD[i*WIDTH+ (j-1)] == -3){
237         phiD[i*WIDTH+ (j-1)] = phiD[i*WIDTH+j] - 1;
238         layerD[i*WIDTH+ (j-1)] = 23; //add to sn2
239       }
240     }
241     if(layerD[i*WIDTH+j] == 26){ //sp1
242       labelD[i*WIDTH+j] = 1;
243       layerD[i*WIDTH+j] = 16; ////add to lp1
244       if(phiD[(i+1)*WIDTH+ j] == 3){
245         phiD[(i+1)*WIDTH+ j] = phiD[i*WIDTH+j] + 1;
246         layerD[(i+1)*WIDTH+ j] = 27; //add to sp2
247       }
248       if(phiD[i*WIDTH+ (j+1)] == 3){
249         phiD[i*WIDTH+ (j+1)] = phiD[i*WIDTH+j] + 1;
250         layerD[i*WIDTH+ (j+1)] = 27; //add to sp2
```

```
251         }
252         if(phiD[(i-1)*WIDTH+ j] == 3){
253           phiD[(i-1)*WIDTH+ j] = phiD[i*WIDTH+j] + 1;
254           layerD[(i-1)*WIDTH+ j] = 27; //add to sp2
255         }
256         if(phiD[i*WIDTH+ (j-1)] == 3){
257           phiD[i*WIDTH+ (j-1)] = phiD[i*WIDTH+j] + 1;
258           layerD[i*WIDTH+ (j-1)] = 27; //add to sp2
259         }
260       }
261     }
262 }
263
264 __global__ void updateLevelSets2(int *layerD, int *labelD){
265   int i = threadIdx.x + blockDim.x * blockIdx.x;
266   int j = threadIdx.y + blockDim.y * blockIdx.y;
267   //no need to check if i and j are within range here
268   if(layerD[i*WIDTH+j] == 23){ //sn2
269     labelD[i*WIDTH+j] = -2;
270     layerD[i*WIDTH+j] = 13;  //add to ln2
271   }
272   if(layerD[i*WIDTH+j] == 27){ //sp2
273     labelD[i*WIDTH+j] = 2;
274     layerD[i*WIDTH+j] = 17; //add to lp2
275   }
276 }
```

update.h

```
1  extern float image[HEIGHT][WIDTH]; //image to be segmented
2  extern float phi[HEIGHT][WIDTH]; //representation of the zero
       level set interface
3  extern int label[HEIGHT][WIDTH];//contains only integer values
       between -3 and 3
4
5  void __global__ prepareUpdates1(float *phiD, int *layerD, float
       *imageD);
6  void __global__ prepareUpdates2(float *phiD, int *layerD, int *
       labelD);
7  void __global__ prepareUpdates3(float *phiD, int *layerD, int *
       labelD);
8  void __global__ prepareUpdates4(float *phiD, int *layerD, int *
       labelD);
9
10 void __global__ updateLevelSets1(float *phiD, int *layerD, int *
       labelD);
11 void __global__ updateLevelSets2(int *layerD, int *labelD);
```

```
12 void __global__ setVariablesInDevice(float threshold, float
       epsilon, float alpha, float image[HEIGHT][WIDTH]);
```

IO.cu

```
1  #include "IO.h"
2  #include "main.h"
3  #include <sstream>
4  using namespace std;
5
6  extern int iterations;
7  extern float threshold, alpha, epsilon;
8  extern float image[HEIGHT][WIDTH];
9  extern int label[HEIGHT][WIDTH];
10 extern int zeroLevelSet[HEIGHT][WIDTH];
11
12 //read input file
13 void readFile(BMP img){
14   //copy input data (img) to image[][] and normalize to [0, 1]
15   for (int i =0; i<HEIGHT; i++){
16     for (int j = 0; j<WIDTH; j++){
17       image[i][j] = (img(i,j)->Red + img(i,j)->Green + img(i,j)
             ->Blue) / 3;
18       image[i][j] /= 255;
19     }
20   }
21 }
22
23 void writeFile(BMP img, int id, int iterations){
24   string name;
25   stringstream sstm;
26   if(id==1){
27     for (short i =0; i<HEIGHT; i++){
28       for (short j = 0; j<WIDTH; j++){
29         img(i,j)->Red = (label[i][j] +3)*42; //normalize to [0,
               255]
30         img(i,j)->Green = (label[i][j] +3)*42;
31         img(i,j)->Blue = (label[i][j] +3)*42;
32       }
33     }
34
35     sstm << iterations << "label" <<".bmp";
36     name = sstm.str();
37     img.WriteToFile(name.c_str());
38     printf("\nlabel image stored");
39   }
40   else{ //zeroLevelSet
41     for (short i =0; i<HEIGHT; i++){
```

```
42        for (short j = 0; j<WIDTH; j++){
43          img(i,j)->Red = zeroLevelSet[i][j];
44          img(i,j)->Green = zeroLevelSet[i][j];
45          img(i,j)->Blue = zeroLevelSet[i][j];
46        }
47      }
48      sstm << iterations << "zero" <<".bmp";
49      name = sstm.str();
50      img.WriteToFile(name.c_str());
51      printf("\nzero image stored\n");
52    }
53  }
54
55  bool getAndVerifyInput(int argc, char *argv[]){
56    if(argc != 5){
57      printf("Need four inputs: iterations, threshold, epsilon,
           alpha \n");
58      return false;
59    }
60    if(sscanf (argv[1], "%i", & iterations)!=1 || iterations<0) {
61      printf("Need four inputs: iterations, threshold, epsilon,
           alpha \n");
62      return false;
63    }
64    if(sscanf(argv[2], "%f", &threshold)!=1 || threshold >1){
65      printf("Need four inputs: iterations, threshold, epsilon,
           alpha \n");
66      return false;
67    }
68    if(sscanf(argv[3], "%f", &epsilon)!=1 || epsilon >1){
69      printf("Need four inputs: iterations, threshold, epsilon,
           alpha \n");
70      return false;
71    }
72    if(sscanf(argv[4], "%f", &alpha)!=1 || alpha>1){
73      printf("Need four inputs: iterations, threshold, epsilon,
           alpha \n");
74      return false;
75    }
76    return true;
77  }
```

IO.h

```
1 #include "EasyBMP.h" //library for reading bmp files
2 void readFile(BMP img);
3 void writeFile(BMP img, int id, int iterations);
4 bool getAndVerifyInput(int argc, char *argv[]);
```