Contents

1 Introduction **2**

1.1 Background 2

1.2 Project Goals 2

1.3 Outline 2

2 Background Theory **3**

2.1 Segmentation 3

2.1.1 Histogram-based segmentation methods 4

2.1.2 Region based segmentation 5

2.1.3 Edge based segmentation 7

2.2 Level Set Method 11

2.2.1 Introduction 11

2.2.2 Signed Distance Transform 14

2.2.3 Discretization by upwinding and difference of normals 15

2.3 Narrow Band - for lite? 18

2.3.1 Introduction 18

2.3.2 Overview of the Narrow Band method 19

2.4 Sparse Field 20

2.4.1 Introduction 20

2.4.2 Overwiew of the Sparse Field method 22

2.5 Parallel computing in GPU 23

2.5.1 Data and task parallelism 23

2.5.2 Central processing unit 24

2.5.3 Flynn's taxonomy of computer architectures 24

2.5.4 Graphics processing unit 27

2.5.5 OpenMP 27

2.5.6 General Purpose GPU 27

2.5.7 CUDA 28

3 Sparse Field - Implemented code 33
3.1 Introduction 33
3.2 TODO 33
3.3 Forskjeller fra vr kode og pseudokoden til lankton 35
3.4 Om koden vr 35
3.4.1 Datastructures and types used - elr noe lignende . . . 35
3.4.2 Code structure 42
3.4.3 Input and initialization 42
3.4.4 Speed function explained 42
3.5 Problems met 43
3.6 Performance 45

4 Results 48

5 Results 49

A Sparse Field - Pseudocode 52

B Appendix A - Source Code 53

List of Figures

2.1 (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 $T = 0.7686$. 6
2.2 (a): Image to be segmented, (b): LoG, (c): Sobel - highlighting horizontal edges, (d):Sobel - highlighting vertical edges . . 10
2.3 Interface of a moving surface. 11
2.4 Interface evolution difficult to represent parametrically. 12
2.5 (a): Circle with arrows pointing in direction of movement, (b): Corresponding level set function 13
2.6 (a): Binary image, (b): SDT based on city-block distance, (c): SDT based on euclidean distance 15
2.7 The narrow band extending out with a width of k from the level set. 20
2.8 Single Instruction Single Data. 25
2.9 Single Instruction Multiple Data. 26
2.10 Multiple Instruction Single Data. 26
2.11 Multiple Instruction Multiple Data. 27
2.12 CUDA programming model. 29
2.13 CUDA memory model. 30

3.1 Label image: image showing the different layers under segmentation. 34
3.2 A label image with pixels in places they should not be. 36
3.3 How the label image should have been. 36
3.4 *Label* image with the layers not continous in the corner 3.4. . 37
3.5 Zero level set corresponding to the label image in figure 3.2. . 44
3.6 Layer image with Lz two pixels wide. 45

Chapter 1

Introduction

1.1 Background

1.2 Project Goals

1.3 Outline

Chapter 2

Background Theory

2.1 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])

$$I = \bigcup_{i=1}^{S} I_i, \quad I_i \cap I_j = \emptyset, \quad i \neq j \tag{2.1}$$

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

2.1.1 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 2.1c and 2.1d (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.

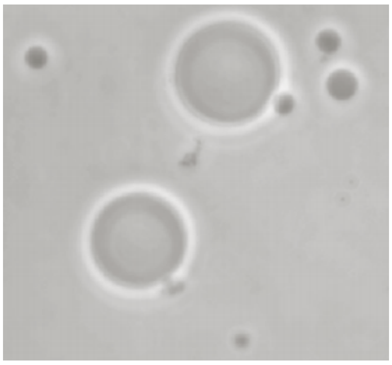
Otsu's thresholding method

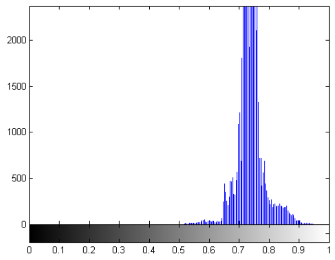
Otsu's thresholding method assumes that the image contains two regions with the values in each region creating a cluster. Otsu's method tries to make each cluster (or class) as tight as possible, thus minimizing their overlap. The goal then is to select the threshold that minimizes the combined spread. 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 2.1 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 2.1a, and its histogram in figure 2.1b. As can be seen from the histogram, it is not possible to select a near perfect threshold by just looking at it. Figure 2.1c illustrates the segmentation result from the iterative global thresholding method and figure 2.1d 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.

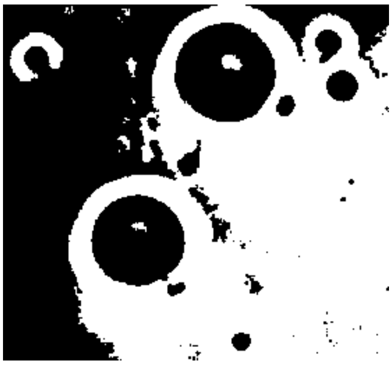
2.1.2 Region based segmentation

Region based segmentation methods tries to find homogenous regions based on gray-scale, color, texture or any other pixel based measure in an image.


(a)



(b)


(c)

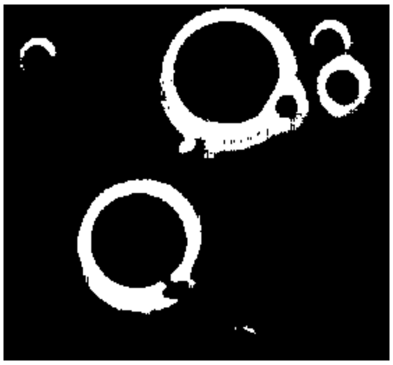

(d)

Figure 2.1: (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 $T = 0.7686$.

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 2.1, 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.

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.

2.1.3 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:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.2}$$

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

$$\nabla^2 f(x,y) = f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)-4f(x,y) \tag{2.3}$$

This Laplacian equation is represented by the mask in equation 2.4, and a variant of the equation that also takes into account diagonal elements is shown in 2.5.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{2.4}$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{2.5}$$

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 2.6 and 2.7 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 2.6 and 2.7 highlights horizontal edges.

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \quad (2.6) \qquad \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad (2.7)$$

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 2.2a illustrates a gray-scale image and 2.2b is the edge segmented image based on LoG. 2.2c is the edge image resulted from the Sobel mask in equation 2.7b and 2.2d is the result from segmentation after rotating the mask 90°. The segmentations resulted by using the Prewitt operator to segment the image in figure 2.2a had no significant difference from the Sobel segmented images.

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 connectivity). Then consider each pixel with magnitude > 0 as candidate edge pixels. For every candidate edge pixel look at the two neighboring pixels in edge-direction and the opposite direction. If the magnitude of the candidate edge pixel is not larger than the magnitude of these neighboring pixels, mark the pixel for deletion. When all candidate edge pixels are inspected, remove all the candidates that are marked for deletion. Now all the edges will contain a single response, but there still are lines/pixels that are not

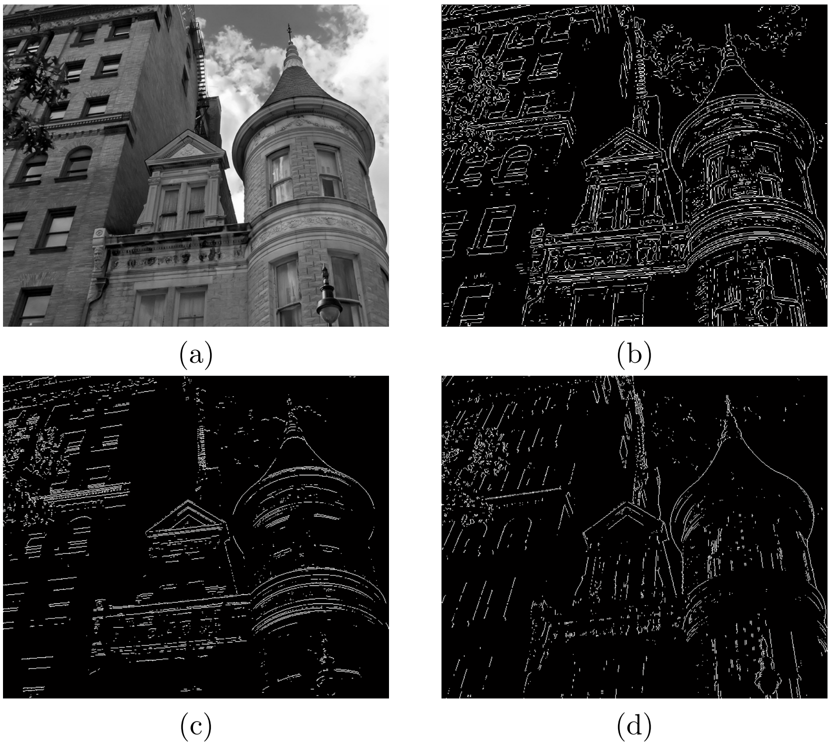


Figure 2.2: (a): Image to be segmented, (b): LoG, (c): Sobel - highlighting horizontal edges, (d):Sobel - highlighting vertical edges

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.

2.2 Level Set Method

2.2.1 Introduction

Surfaces that evolves over time can be difficult to represent. Taking the surface in figure 2.3 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 inerface has more y coordinates for a particular x coordinate (which is true for all closed interfaces), the intercane cannot be correctly represented using this notion. A better alternative is to use a parametric equation. The

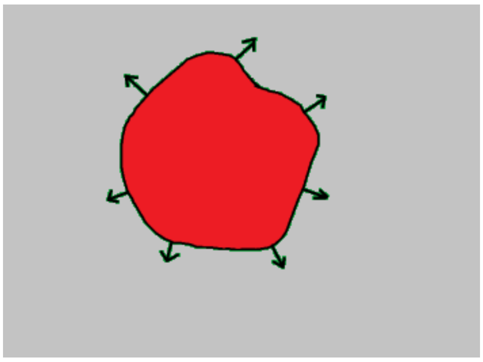


Figure 2.3: Interface of a moving surface.

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 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 2.4, 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.

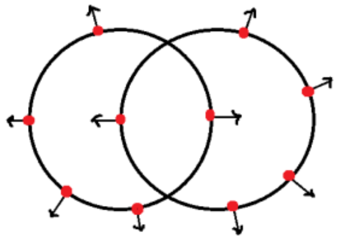


Figure 2.4: Interface evolution difficult to represent parametrically.

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

$$\phi(x, y, t) \tag{2.8}$$

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:

$$\Gamma(x, y, t) = \{\phi(x, y, t) = 0\}. \tag{2.9}$$

The initial curve is at the xy-plane, that is, at $\phi(x, y, 0)$. As an example, figure 2.5a depicts a circle with arrows pointing in the direction it is evolving, and figure 2.5b is the cone that represents the corresponding level set function with the start-position in red.

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

$$\frac{\partial \phi}{\partial t} = |\nabla \phi| F \tag{2.10}$$

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

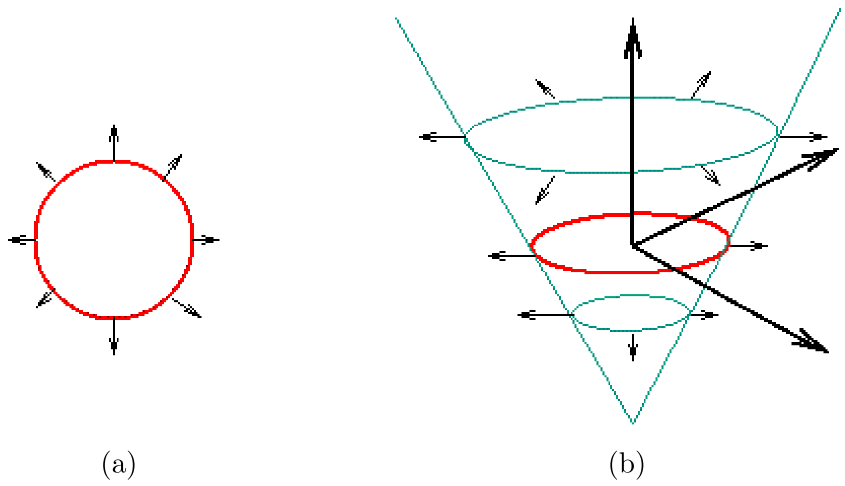


Figure 2.5: (a): Circle with arrows pointing in direction of movement, (b): Corresponding level set function

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]

$$F = \alpha D(I) + (1 - \alpha)\nabla \frac{\nabla \phi}{|\nabla \phi|} \tag{2.11}$$

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:

$$D(I) = \epsilon - |I - T| \tag{2.12}$$

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 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.

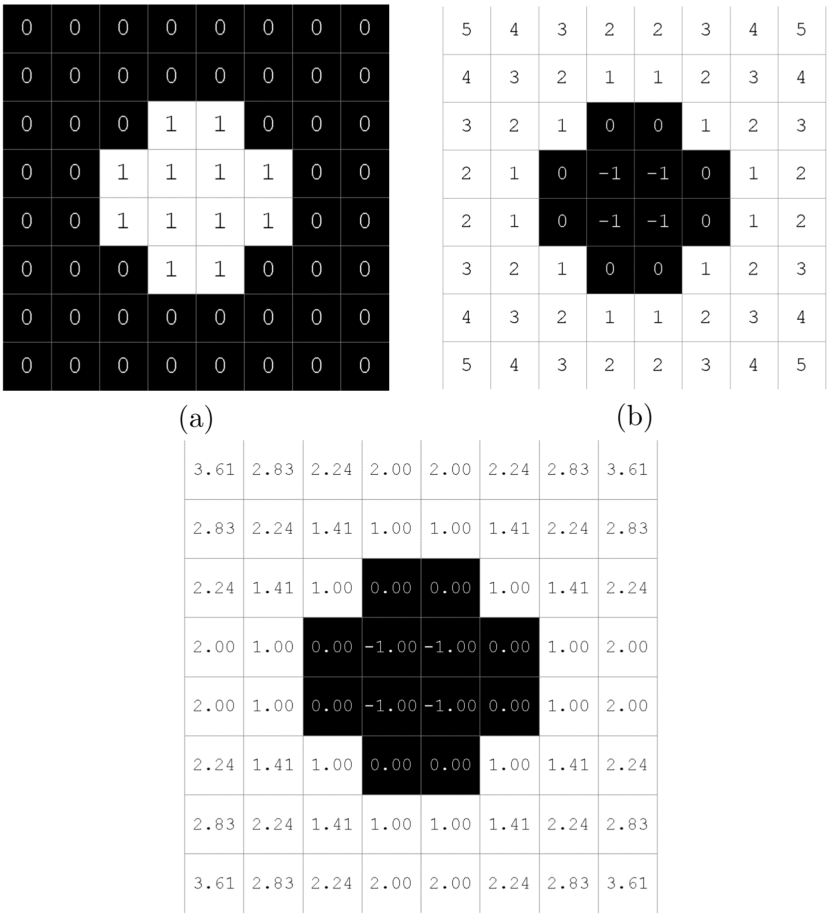
2.2.2 Signed Distance Transform

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

$$D(r, S) = min(r - S) \quad for\ all\ r \in \mathbb{R}^3 \tag{2.13}$$

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 2.6a (the numbers in this image represent intensity values). Figure 2.6b is the signed distance transform of 2.6a where city-block (manhattan) distance have been used, and figure 2.6c 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 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.



(a)

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b)

5	4	3	2	2	3	4	5
4	3	2	1	1	2	3	4
3	2	1	0	0	1	2	3
2	1	0	-1	-1	0	1	2
2	1	0	-1	-1	0	1	2
3	2	1	0	0	1	2	3
4	3	2	1	1	2	3	4
5	4	3	2	2	3	4	5

(c)

3.61	2.83	2.24	2.00	2.00	2.24	2.83	3.61
2.83	2.24	1.41	1.00	1.00	1.41	2.24	2.83
2.24	1.41	1.00	0.00	0.00	1.00	1.41	2.24
2.00	1.00	0.00	-1.00	-1.00	0.00	1.00	2.00
2.00	1.00	0.00	-1.00	-1.00	0.00	1.00	2.00
2.24	1.41	1.00	0.00	0.00	1.00	1.41	2.24
2.83	2.24	1.41	1.00	1.00	1.41	2.24	2.83
3.61	2.83	2.24	2.00	2.00	2.24	2.83	3.61

Figure 2.6: (a): Binary image, (b): SDT based on city-block distance, (c): SDT based on euclidean distance

2.2.3 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 2.10, given by (from [4])

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + F^n \cdot \nabla \phi^n = 0 \tag{2.14}$$

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):

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u^n \phi^n_x + v^n \phi^n_y + w^n \phi^n_z = 0, \tag{2.15}$$

where the techniques used to approximate the $u^n \phi^n_x$, $v^n \phi^n_y$ and $w^n \phi^n_z$ 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 2.15 can be written as

$$\frac{\phi^{n+1}_i - \phi^n_i}{\Delta t} + u^n_i (\phi_x)^n_i = 0, \tag{2.16}$$

where $(\phi_x)^n_i$ 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

$$\begin{array}{lll} D_x = \dfrac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{2} & D_y = \dfrac{\phi_{i,j+1,k} - \phi_{i,j-1,k}}{2} & D_z = \dfrac{\phi_{i,j,k+1} - \phi_{i,j,k-1}}{2} \\ D^+_x = \phi_{i+1,j,k} - \phi_{i,j,k} & D^+_y = \phi_{i,j+1,k} - \phi_{i,j,k} & D^+_z = \phi_{i,j,k+1} - \phi_{i,j,k} \\ D^-_x = \phi_{i,j,k} - \phi_{i-1,j,k} & D^+_y = \phi_{i,j,k} - \phi_{i,j-1,k} & D^+_z = \phi_{i,j,k} - \phi_{i,j,k-1} \end{array} \tag{2.17}$$

which is taken from the appendix of [7]. This is a *consistent* finite difference approximation to the level set equation in 2.10, 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 $|u|$,

$$\Delta t = \frac{\Delta x}{max\{|u|\}}, \tag{2.18}$$

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,

$$\nabla\phi = \begin{cases} ||\nabla\phi_{max}||_2 & F_{i,j,k} > 0 \\ ||\nabla\phi_{min}||_2 & F_{i,j,k} < 0 \end{cases} \tag{2.19}$$

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

$$\nabla\phi_{max} = \begin{bmatrix} \sqrt{max(D_x^+,0)^2 + max(-D_x^-,0)^2} \\ \\ \sqrt{max(D_y^+,0)^2 + max(-D_y^-,0)^2} \\ \\ \sqrt{max(D_z^+,0)^2 + max(-D_z^-,0)^2} \end{bmatrix} \tag{2.20}$$

$$\nabla\phi_{min} = \begin{bmatrix} \sqrt{min(D_x^+,0)^2 + min(-D_x^-,0)^2} \\ \\ \sqrt{min(D_y^+,0)^2 + min(-D_y^-,0)^2} \\ \\ \sqrt{min(D_z^+,0)^2 + min(-D_z^-,0)^2} \end{bmatrix} \tag{2.21}$$

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:

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

$$\begin{aligned}D_z^{+x} &= (\phi_{i+1,j,k+1} - \phi_{i+1,j,k-1})/2 \quad D_z^{-x} = (\phi_{i-1,j,k+1} - \phi_{i-1,j,k-1})/2 \\ D_z^{+y} &= (\phi_{i,j+1,k+1} - \phi_{i,j+1,k-1})/2 \quad D_z^{-y} = (\phi_{i,j-1,k+1} - \phi_{i,j-1,k-1})/2\end{aligned} \tag{2.22}$$

Then these derivatives are used to compute the normals n^+ and n^- in equation 2.23, which is used to compute the mean curvature H in equation 2.24 taken from [7].

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

$$n^- = \begin{bmatrix} \frac{D_x^-}{\sqrt{(D_x^-)^2 + (\frac{D_y^{-x} + D_y}{2})^2 + (\frac{D_z^{-x} + D_z}{2})^2}} \\ \frac{D_y^-}{\sqrt{(D_y^-)^2 + (\frac{D_x^{-y} + D_x}{2})^2 + (\frac{D_z^{-y} + D_z}{2})^2}} \\ \frac{D_z^-}{\sqrt{(D_z^-)^2 + (\frac{D_x^{-z} + D_x}{2})^2 + (\frac{D_y^{-z} + D_y}{2})^2}} \end{bmatrix} \tag{2.23}$$

$$H = \frac{1}{2}\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} = \frac{1}{2}[(n_x^+ - n_x^-) + (n_y^+ - n_y^-) + (n_z^+ - n_z^-)] \tag{2.24}$$

Finally, the level set equation is updated as

$$\phi(t + \Delta t) = \phi(t) + \Delta t F|\nabla\phi|. \tag{2.25}$$

2.3 Narrow Band - for lite?

2.3.1 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.

2.3.2 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 2.7), 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

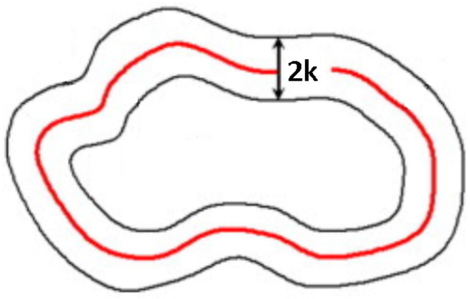


Figure 2.7: The narrow band extending out with a width of k from the level set.

position of the zero level set and ϕ must be reinitialized. Reinitializing ϕ 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.

2.4 Sparse Field

2.4.1 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.

2.4.2 Overwiew of the Sparse Field method

Like described in section 2.2.3 (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.

2.5 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.

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

2.5.2 Central processing unit

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

2.5.3 Flynn's taxonomy of computer architectures

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

Single Instruction Single Data - SISD

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

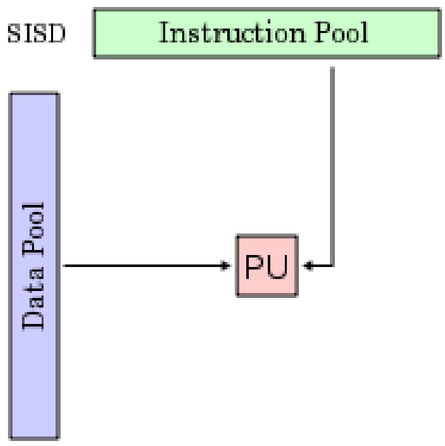


Figure 2.8: Single Instruction Single Data.

Single Instruction Multiple Data - SIMD

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

Multiple Instruction Single Data - MISD

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

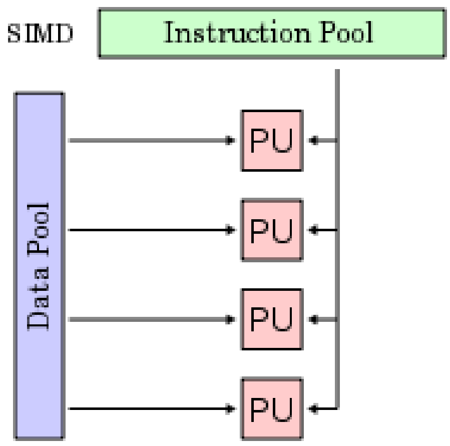


Figure 2.9: Single Instruction Multiple Data.

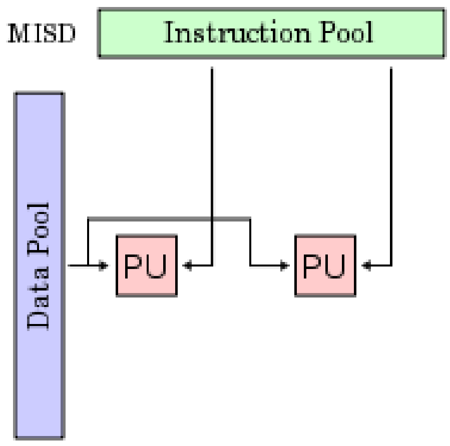


Figure 2.10: Multiple Instruction Single Data.

Multiple Instruction Multiple Data - MIMD

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

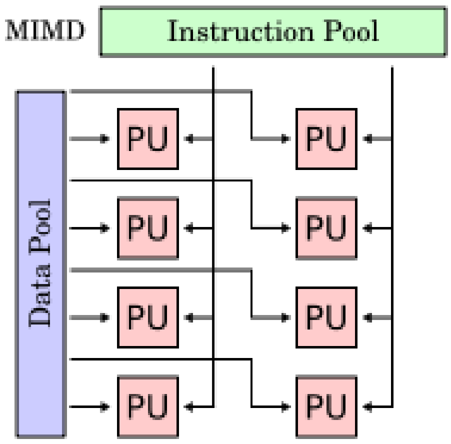


Figure 2.11: Multiple Instruction Multiple Data.

2.5.4 Graphics processing unit

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

2.5.5 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.

2.5.6 General Purpose GPU

TODO

2.5.7 CUDA

CUDA (Compute Unied Device Architecture) is a program development environment by NVIDIA 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 host. 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. But 32 threads will always execute the same code (even if less than 32 executions is necassary), and such a group of 32 threads is called a warp. Figure 2.12 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 2.13. Registers are the smallest, but also the fastest, and the per-thread register limit for compute capability (version) 3.0 is 63 registers per thread. If a thread needs more than 63 registers the shared memory is used (L1 cache) which is much slower. And if even more is needed the even slower global is also used.

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 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

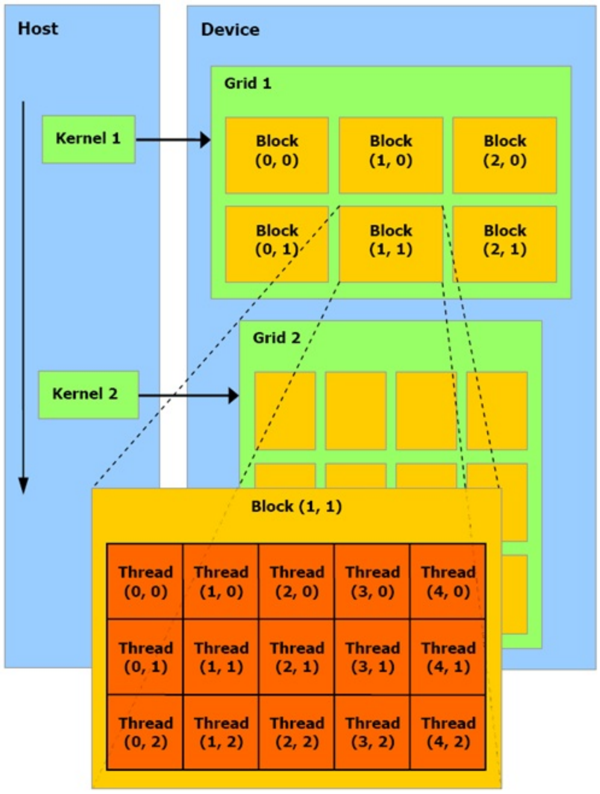


Figure 2.12: CUDA programming model.

indicating transfer from host to device and device to host, respectively. On line 26 the kernel is launched. The syntax for calling a kernel function is kernelName<<<numBlocks, numThreadsPerBlock>>>(arg1, arg2, ..., argN). The triple angle brackets indicate that it is a kernel launch, i.e. a call from host code to device code. The first number within these brackets is dimension of the grid, which also is the number of blocks in that grid, which can be one or two dimensional. The second is the block dimension, i.e. the numbers of threads in a block which can be up to three dimensions. Calling a kernel with X number of blocks and Y number of threads per block results in $X * Y$ parallel executions of that kernel. The kernel function is in line 38-43 and starts with *__global__* before *void kernelName*(...), which imply that the function runs on a device and is called from host code. The code within kernels 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),

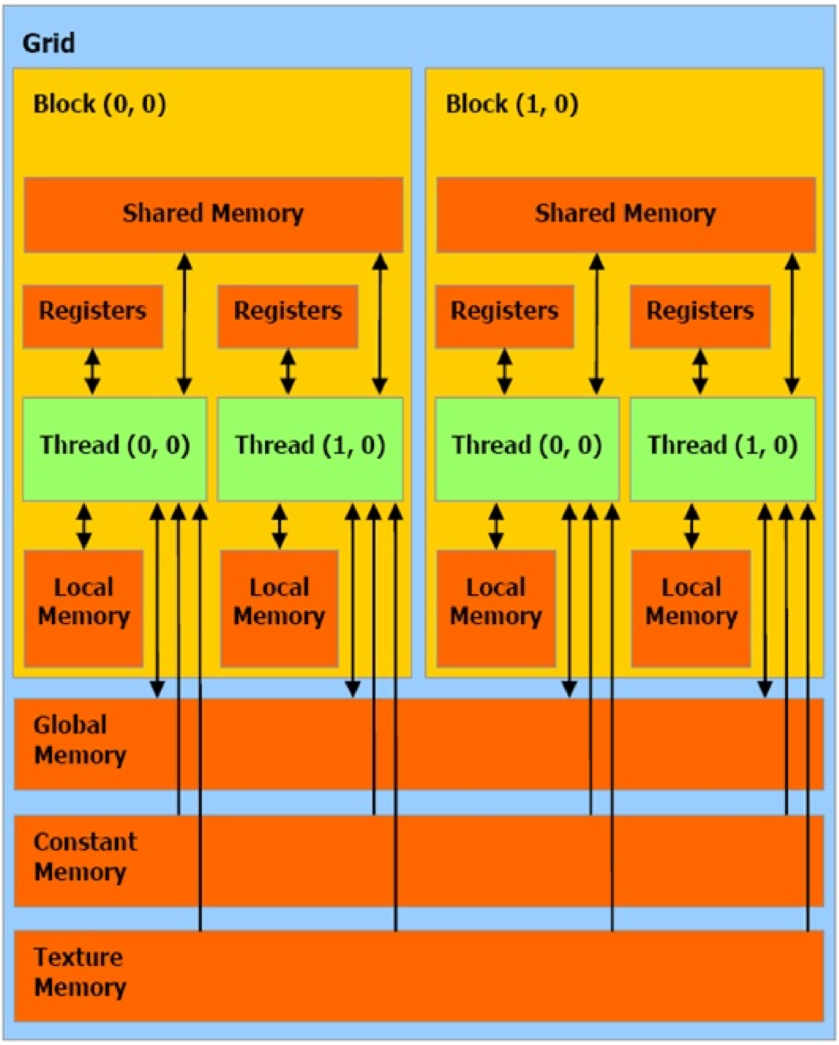


Figure 2.13: CUDA memory model.

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. NVIDIAs own compiler named nvcc splits up the code into host and device components, compiles the kernel code itself, and lets the standard host compiler compile the host code.

Listing 2.1: Simple CUDA program

```
#define N (2048*2048) //number of threads
#define M 512 //number of threads per block
int main( void ) {
    int *a, *b, *c; // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int arraySize = N * sizeof( int ); // need space for N intege

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, arraySize );
    cudaMalloc( (void**)&dev_b, arraySize );
    cudaMalloc( (void**)&dev_c, arraySize );

    a = (int*)malloc( arraySize );
    b = (int*)malloc( arraySize );
    c = (int*)malloc( arraySize );

        //fill the a and b arrays with randon integers
    random_ints( a, N );
    random_ints( b, N );

    // copy inputs to device
    cudaMemcpy( dev_a, a, arraySize, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, arraySize, cudaMemcpyHostToDevice );

    // launch add() kernel with blocks and threads
    add<<< N/M, M >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, arraySize, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

```
__global__ void add( int *a, int *b, int *c ) {
    int threadId = threadIdx.x + blockDim.x*blockIdx.x;
        if(threadId < arraySize){
            c[threadId] = a[threadId] + b[threadId];
        }
}
```

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 is executed in the exact same was as the 2D version. Both the 2D and 3D versions of the implemented code can be found in appendix B, and the pseudocode (only in 2D) can be found in appendix A. In this chapter the pseudocode in [11] will be explained first, along with how it works. Secondly, the differences between Lankton's pseudocodes in [11] and the implemented code will be described. And finally there will be a detailed explanation of the implemented code.

3.2 TODO

As previously mentioned, the sparse field method can be implemented using linked lists to hold the pixels being used in the calculations. Pixels in this context does not mean the pixels in the original or segmented image, but the points in the matrix that represents the ϕ. 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 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.

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]

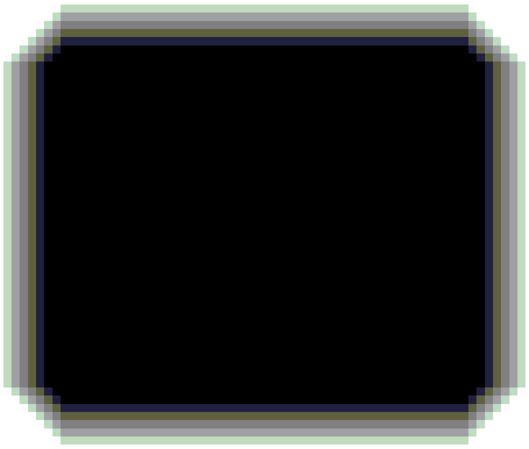


Figure 3.1: Label image: image showing the different layers under segmentation.

Figure 3.1 represents the 5 different layers with different colors. The black colored part is defined to be inside the object being segmented, the white part as outside, and these two parts are not used in the computation for the current iteration. The dark blue pixels around the dark part are the pixels contained in Ln2, and the brown pixels are those on Ln1. The dark-purple colored pixels are Lz elements, light-purple are Lp1 and light-blue are pixels in Lp2. This type of image will henceforth be referred to as the *label* image, because it shows the labels of the image being segmented.

3.3 Forskjeller fra vr kode og pseudokoden til lankton

Some improvements and changes from Lankton's pseudocode were made By looking closer at table 3.1 it can be seen that Lz has a slightly wider ramge 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 are shown in table 3.2, and even though the change seems insignificant it improves the result significantly (as will be discussed in 3.5).

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

Table 3.2: Range of lists used in the implementation

3.4 Om koden vr

3.4.1 Datastructures and types used - elr noe lignende

In addition to the five lists representing the as 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) that pixel is a member of, a simple lookup to the *label* is enough. Another excellent feature of the *label* array is that it can be used to verify that all the layers are correclty aligned and if there are any pixels of any layers that are poorly placed. The *label* image can thus be used to find artifacts that might have resulted from code errors. An example of a *label* image (zoomed in to be able to clearly see the artifacts) which clearly states that there is something wrong with how the layers are handled in the code is shown in figure 3.2. How the *label* image actually should have been is illustrated in figure 3.3.

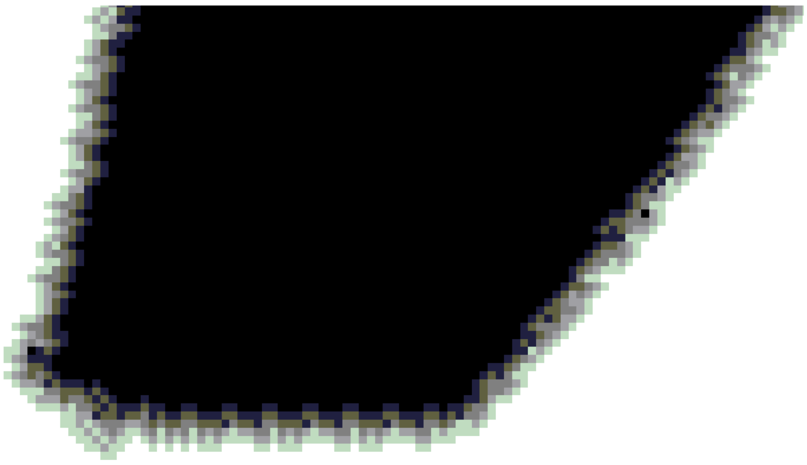


Figure 3.2: A label image with pixels in places they should not be.

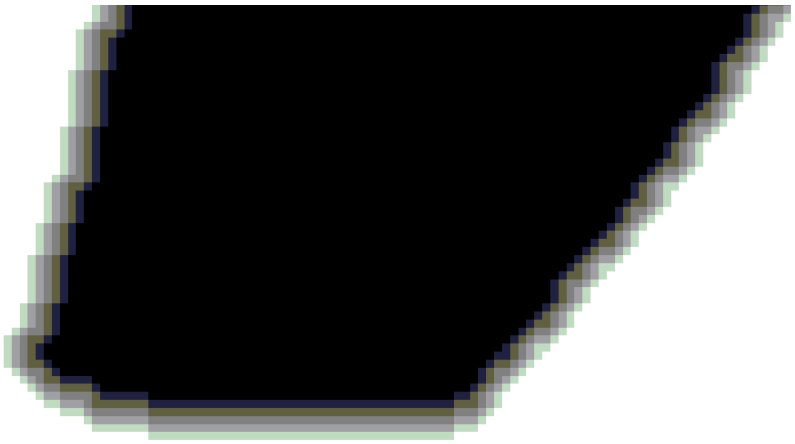


Figure 3.3: How the label image should have been.

The other array used is the ϕ - array, which contains the actual ϕ values of each pixel in the domain. The range of the values is exactly the same as in the *label* image, but while the *label* image only contains integer values describing which layer a pixel is part of, the ϕ image contains the actual values (floating point numbers) of the level set. The images represented by the *label* and ϕ arrays would thus be very similar (though small differences may be seen) when looking at, but they do have different tasks. The *label* is as mentioned used as a lookup table, while the ϕ array 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 Ln1, Ln1, Lz, Lp1 and Lp2 lists are prevented from being moved in the wrong direction or more than one step at a single iteration. All pixels that are moving into Lz are kept in Sz, (and similarly for the other lists), then all pixels that are neighbours to Lz but not still in Sz is added to Sz, before finally adding all pixels in Sz to Lz. So the main reason to use these temporary lists is to keep the layers continuous when evolving. An example of how a *layer* image would look like if this did not happen is illustrated in figure 3.4

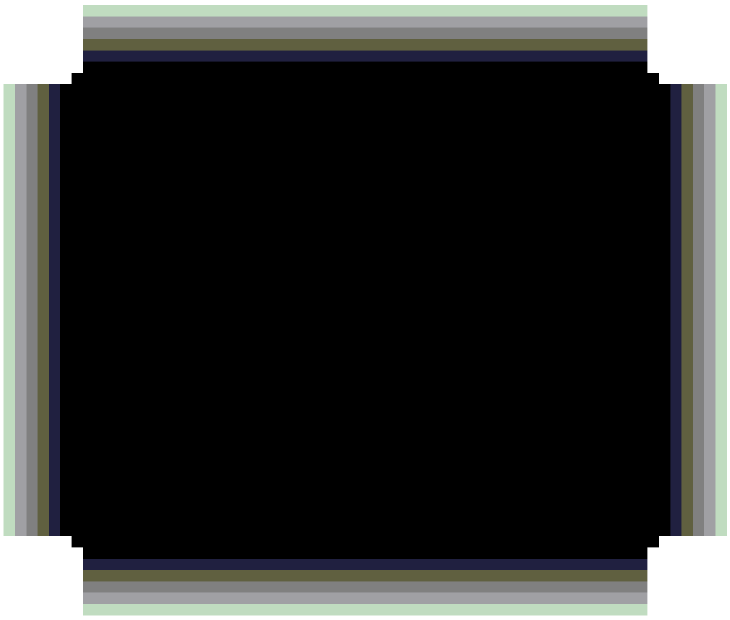


Figure 3.4: *Label* image with the layers not continous in the corner 3.4.

Figure 3.4 illustrates the problem with not adding neighbouring pixels of pixel A ($A \in$ any of the five layers) that is not part of any layer to the (correct) neighbouring layer of A's layer. In that case the segmentation process would go completely wrong. Figure 3.4 is not an actual *label* image output, but a manually edited version of figure 3.1 that illustrates this problem. Algorithm 1 shows how Lz is updated by the speed function and how the $Ln2$, $Ln1$, $Lp1$ and $Lp2$ reflects this change. It also shows how the temporary lists are filled by pixels that are moving from one layer to another. How the temporary lists are filled further by pixels that previously not were in any layers can be seen in algorithm 3, along with how the *label* and ϕ images are updated. Before taking a look at the pseudocode a brief explanation of how the speed function affect the different layers is needed. First the speed function is calculated for each pixel in Lz (will be explained in more detail in LINK TIL DER SPEED FUNCTION BLIR FORKLART). Then the four remaining lists are updated, but not using the speed function as with the Lz pixels. Since Ln1 and Lp1 are defined to be neighbours to Lz on each side, and Lp2 and Ln2 as neighbour to Lp1 Ln1 respectively, there is no need to use computation time to calculate their speed function. They will follow Lz, e.g. if Lz moves one step to the right, then its neighbours at each side (Ln1 and Lp1) will also move a step to the right. Similarly Ln2 and Lp2 will follwow Ln1 and Lp1 respectively. How this actually is implemented can be seen in algorithm 1. Alogrithm 1 and 3 is the pseudocode of the actions executed in each iteration of the segmentation process.

Algorithm 1 TODO Finn p et passende navn.

```
for all $e \in Lz)$ do
    $e = e + speedFucntion(e.x, e.y)$
    if $phi(e.x, e.y) \geq 0.5$ then                                ▷ $phi = \phi$
        sp1.add(e)
        Lz.remove(e)
    else if $phi(e.x, e.y) < -0.5$ then
        Sn1.add(e)
        Lz.remove(e)
    end if
end for
for all $e \in Ln1)$ do
    if $e$ has no neighbour that is part of Lz then
        Sn2.add(e)
        Ln1.remove(e)
    else
        $M$ = Greatest value from $N(e)$ with $N(e) \geq 0$
                                        ▷ $N(e)$ = *neighbour* pixels of e in *label*
        $phi(e.x, e.y) = M - 1$
        if $phi(e.x, e.y) \geq -0.5$ then
            Sz.add(e)
            Ln1.remove(e)
        else if $phi(e.x, e.y) < -1.5$ then
            Sn2.add(e)
            Ln1.remove(e)
        end if
    end if
end for
for all $e \in Lp1)$ do
    if $e$ has no neighbour that is part of Lz then
        Sp2.add(e)
        Lp1.remove(e)
    else
        $M$ = Smallest value from $N(e)$ with $N(e) \leq 0$
        $phi(e.x, e.y) = M + 1$
        if $phi(e.x, e.y) < 0.5$ then
            Sz.add(e)
            Lp1.remove(e)
        else if $phi(e.x, e.y) \geq 1.5$ then
            Sp2.add(e)
            Lp1.remove(e)
        end if
    end if
end for
```

Algorithm 2 TODO Finn p et passende navn (continued).

```
for all $e \in Ln2)$ do
    if $e$ has no neighbour that is part of Ln1 then
        $label(e.x, e.y) = -3$
        $phi(e.x, e.y) = -3$
        Ln2.remove(e)
    else
        $M$ = Greatest value from $N(e)$ with $N(e) \geq -1$
        $phi(e.x, e.y) = M - 1$
        if $phi(e.x, e.y) \geq -1.5$ then
            Sn1.add(e)
            Ln2.remove(e)
        else if $phi(e.x, e.y) < -2.5$ then
            $label(e.x, e.y) = -3$
            $phi(e.x, e.y) = -3$
            Ln2.remove(e)
        end if
    end if
end for
for all $e \in Lp2)$ do
    if $e$ has no neighbour that is part of Lp1 then
        $label(e.x, e.y) = 3$
        $phi(e.x, e.y) = 3$
        Lp2.remove(e)
    else
        $M$ = Smallest value from $N(e)$ with $N(e) \leq 1$
        $phi(e.x, e.y) = M + 1$
        if $phi(e.x, e.y) < 1.5$ then
            Sp1.add(e)
            Lp2.remove(e)
        else if $phi(e.x, e.y) \geq 2.5$ then
            $label(e.x, e.y) = 3$
            $phi(e.x, e.y) = 3$
            Lp2.remove(e)
        end if
    end if
end for
```

Algorithm 3 TODO Finn p et passende navn her ogs.

```
1: for all $e \in Sz)$ do
2:     $label(e.x, e.y) = 0$
3:     Lz.add(e)
4: end for
5: Reset Sz
6: for all $e \in Sn1)$ do
7:     $label(e.x, e.y) = -1$
8:     Ln1.add(e)
9:     for all $ne \in N(e)$ do    ▷ The four neighbouring pixels: over, under, left and right
10:        if $phi(ne.x, ne.y) == -3$ then
11:            Sn2.add(ne)
12:        end if
13:    end for
14: end for
15: Reset Sn1
16: for all $e \in Sp1)$ do
17:     $label(e.x, e.y) = 1$
18:     Lp1.add(e)
19:     for all $ne \in N(e)$ do
20:         if $phi(ne.x, ne.y) == 3$ then
21:             Sp2.add(ne)
22:         end if
23:     end for
24: end for
25: Reset Sp1
26: for all $e \in Sn2)$ do
27:     $label(e.x, e.y) = -2$
28:     Ln2.add(e)
29: end for
30: Reset Sn2
31: for all $e \in Sp2)$ do
32:     $label(e.x, e.y) = 2$
33:     Lp2.add(e)
34: end for
35: Reset Sp2
```

3.4.2 Code structure

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.

3.4.3 Input and initialization

TODO skriv om input
An array of same size as *label* and the ϕ - arrays is used to initialize *label*, the ϕ - array 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 along with a radius given as input. These input values creates a circle (or sphere if 3D) of 1's in the init array that represents the seed points. Multiple coodrinates and radiuses can be given as input to create multiple seed points. Based on the values in init the two arrays *label* and ϕ are initialized. All pixels in *label* and ϕ corresponding with the ones in init with value 1 are set to -3 to indicate that they are inside the segmentation object. All other pixels in *label* and ϕ are set to 3 indicating that they are outside the object. Then the corresponding pixels to all values in init that are 1 but have 0 valued neighbours are set to 0, indicating that they are part of the zero level set. Then these pixels are added to Lz as initial zero level set values. Then $Ln1$, $Lp1$, $Ln2$ and $Lp2$ are filled according to their definitions, and the *label* and ϕ arrays are updated to refelct these changes. After these initializing actions are finished the segmentation process can start. The ϕ - array is initialized by the using the x,y (and z in 3D) coordinates along with the radius to create a circle. This circle is

3.4.4 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 one explained in chapter and a much more simple one based on the Chan-Vese energy. Simply put, the Chan-Vese energy (E^{CV}) is defined as

$$E^{CV}(c_1, c_2, C) = \int_{inside(C)} (\mu(x,y)-c_1)^2 \, dx\, dy + \int_{outside(C)} (\mu(x,y)-c_2)^2 \, dx\, dy \; [12] \tag{3.1}$$

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

$$E^{CV}(x,y) = (\mu(x,y) - c_1)^2 - (\mu(x,y) - c_2)^2 \tag{3.2}$$

TODO forklar litt om hvordan denne speed funksjonen virker
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.
TODO: Husk skrive om hvorfor vi clamper verdiene mellom -1 og 1, istedenfor normalisere (alts bare dele alt p dn hyeste mulige returnverdien)

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 3.5 which shows only the zero level set. The zero level set in figure 3.5 is the segmentation result (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

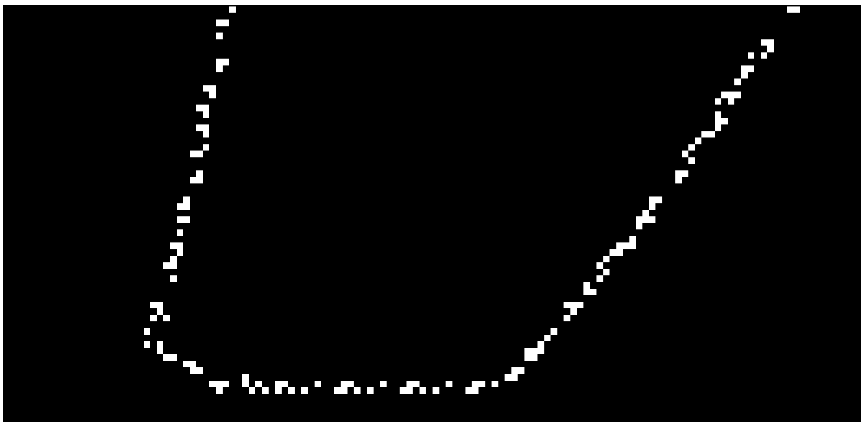


Figure 3.5: Zero level set corresponding to the label image in figure 3.2.

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 3.6. 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

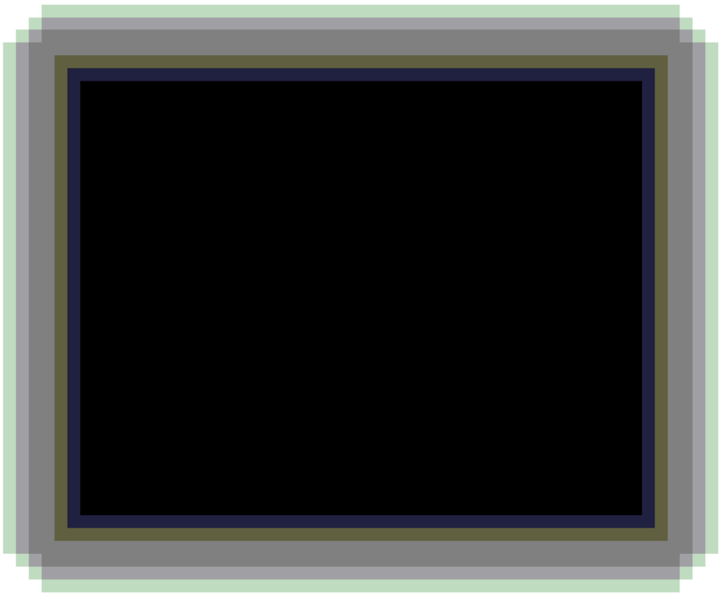


Figure 3.6: Layer image with Lz two pixels wide.

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 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. 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 256^3 and several linked-lists with hundreds of pixels being pushed and popped each itereation, the change reduced the runtime by ???? in 2D and ???

in 3D. (TODO finn ut hvor raskere segmenteringen kjrte etter double til float endringen) A chance that imroved 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 using *list* instead of *vector* was the second significant change. 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 because of the overhead when using *vector* associated with inserteion and erasure of elements not at the end. 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, which improves performance and speed. 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.

To speed up the process even more some parts of the code were run in parallel by using OpenMP. OpenMP is an API that supports shared memory multiprocessing in C, C++ and Fortran. OpenMP was used even before *vector* was replaced with *list*, and was chosen only because it reguires a few well placed lines of code. OpenMP was implemented right after the first 3D version was completed, and was used to speed up the process when different datasets were used as input to compare the results against each other (TODO: Dette avsnittet m forbedres). 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, so to being able to use single linked lists a complete makeover of the code is necessary, which due to the scope of time was not an option.

Chapter 4

Results

TODO

Chapter 5

Results

TODO

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.

Appendix A

Sparse Field - Pseudocode

The code written for this thesis (which can be found in appendix B) is based on the following pseudocode. This pseudocode originates from [11], with a different speed function and some other modifications.

TODO: skriv pseudocoden for koden vr her (ikke skriv direkte av lankton09)

Appendix B

Appendix A - Source Code

TODO: legg inn kode filene i Appendix mappen og uncomment det som str i tex filen.