Extreme Image Segmentation

Research Project Report

Vincent Markert
TU Dresden

Supervisor: Prof. Björn Andres

Winter Term 2020/21

Contents

Chapter 1

Motivation

Image segmentation is a process in which an image is dissected into different meaningful regions, containing instances of objects. Therefore, it facilitates the possibility for a machine to understand the structure of a scene within a image. It is often used in many application domains, such as bioinformatics, medical analysis or autonomous driving.
By defining a decomposition of the image, it is implicitly decided for each pair of pixels in the image, whether they belong to the same region or different ones. As a result, two types of errors can be made: The joining of two pixels into one region, which in reality belong to different objects (referred to as "false join") and the splitting or cutting of two pixels into different regions, which in reality belong to the same object (referred to as "false cut").
Often, segmentation algorithms incorporate the possibility of setting parameters to tackle one of the two error classes, while potentially loosing performance on the other error type. For example when using a segmentation algorithm which tends to produce a oversegmented decomposition (high amount of false cuts), there is often a way to push the segmentation algorithm to do less separation of regions. In the course of doing that, however, one also highers the risk of joining two regions, which in fact do not belong together, therefore yielding false joins.
While in many applications images tend to mainly produce one of the two types of errors and segmentation algorithms are primarily required to have a high accuracy with respect to that error, there are application where both types of errors are occurring simultaneously and the segmentation algorithm needs to be assessed for both errors equally (Figure 1.1). This is for example the case, if in one part of the image, there are only thin borders dividing two objects, thus yielding a high risk of joining two objects together and in another part of the image having very thjn objects themselves, which tend to be cut into multiple parts.

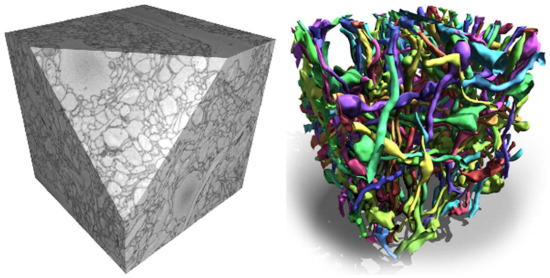


Figure 1.1: From [2]: Image segmentations in the field of Connectomics may tend to produce both false joins and false cuts.

The goal of this research project is the implementation of analysis for the two error

setting for a known segmentation algorithm in the three-dimensional domain. In this case the analysis is performed on a watershed segmentation, which is a classical algorithm where gray values are interpreted as height values and the decomposition accomplished based on the resulting topographic surface. To test the performance of watershed segmentation, a procedure to generate data which emphasizes on the specific error classes, is implemented as part of this project. Then the performance of the segmentation algorithm is evaluated with the help of two popular metrics, the Rand Index and the Variation of Information.
The implementations for this work are written in Python 3.

Chapter 2

Methodology

2.1 Data Generation

To test the segmentation algorithm, data in form of 3D gray scale images is synthesized, where the structures expose the aforementioned problems. For the false join problem class, images with thin membranes, and for the false cut problem class, images containing thin objects, are generated.
A Voronoi diagram is used for the first one and the resulting image set referred to as "False Joins Case". For the latter one, Bézier curves are used and it is referred to as "False Cuts Case".
While the structures are defined in a continuous space, it must be transferred to a discrete version to determine each individual pixel gray value. In this work, two approaches to make this transformation are explained. In the first version each pixel in the image is iterated and the gray value computed based on the distance to the nearest object. In the other one, samples are drawn along the objects bodies and the nearest pixel marked as belonging to an object. Then the distances to the nearest marked pixel within the image array are computed as a sequence of array operations.
Furthermore, to make it a non-deterministic segmentation problem, random noise is added to the image in both cases.

2.1.1 False Joins

Continuous Structures

The false joins task image set is based on the Voronoi diagram. In a Voronoi diagram, each point in space gets assigned the sample, out of a set of sample points, that is closest to that point, according to some metric.
The image shall contain the objects within the positive three-dimensional unit cube $[0, 1]^3$. Therefor, a number of samples is sampled uniformly inside the cube. It is also possible to sample points, that are slightly outside the cube, for example $s \in [-0.2, 1.2]^3$.
To avoid having to deal with special cases at the border of the diagram, where the edge planes and cells would go to infinity, additional points outside the volume can be added. Sample points at $s_{outside} = [i, j, k]\ for\ i, j, k = -1, 2$ were added in the implementation. Based on all sample points, the Voronoi diagram can be constructed in real space.

To make the segmentation more challenging, surfaces between neighboring Voronoi cells are removed at random, effectively joining the cells and forming potentially non-convex shapes. This can be done by randomly sampling indices and removing surfaces with the corresponding index.

Discrete

Iterating Pixels

In the first version all pixels are now iterated and each pixel value determined. The following formula was used to compute gray values g:

$$g(p) = -dist^{1/k} + u, \tag{2.1}$$

where $dist$ is the distance of the pixel p to the closest surface, k is a parameter to regulate the influence of the distance, $u \in N(0, s)$ is a Gaussian random number and s is a parameter to regulate the influence of noise. For the experiments $k = 10$ and $s = 0.02$ were chosen and the Euclidean Norm is used. While this leaves most of the values of $g \in [-1.06, 0.02]$ (with edge cases exceeding the boundaries), the values are adjusted in a later step when transforming the image to 8 Bit.
The main difficulty here is the computation of $dist$. The cell containing a pixel p can be found via the closest sample. The closest sample is identified by computing the distance of the pixel to all samples and taking the one with the minimal distance. This can be done with a convenient matrix operation.
The minimal sample also defines the ground truth of the pixel for the task. If the cell has none of it's surfaces removed, the distance can be computed as the minimal point-plane distance given by

$$\frac{ax_0 + by_0 + cz_0 + d}{a^2 + b^2 + c^2}, \tag{2.2}$$

where $p = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$ is the position of the pixel, $n = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ is the normal of the surface plane and d the distance of the plane to the origin.
However, if any of the cell's surfaces is removed, all surfaces of all connected neighboring cells have to be considered. Due to the potential non-convexity of the shape, a simple point-plane distance might lead to artifacts (Figure (2.1)). For that reason the distances of the pixel to the actual polygons have to be computed.
A possible way is given in [5]. All surface polygons are divided into triangles by fixating one vertex, iterating sets of other neighboring vertex pairs and defining a triangle on the triple. It is determined, on which part of the triangle

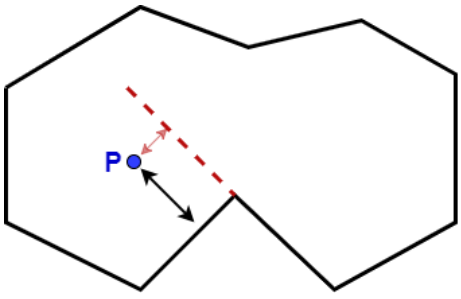


Figure 2.1: Only computing point-plane distances on non-convex cells can lead to artifacts, as point P is closer to the dotted red line continuation than to the actual border in black.

(Point/Edge/Plane) the closest point to
the pixel is located, by projecting the pixel onto the three edge lines and checking whether the projected pixels lie inside or outside the edge segments of the triangle (Figure (2.2)). To distinguish between a closest point inside the triangle or on one of the edges, a plane is constructed on the edge, that is orthogonal to the triangle and it is tested, whether the pixel is above (outside the triangle) or below the plane. If the closest point is within the triangle, again a simple point-plane distance (2.2) can be computed, else the pixel to point distance is calculated.

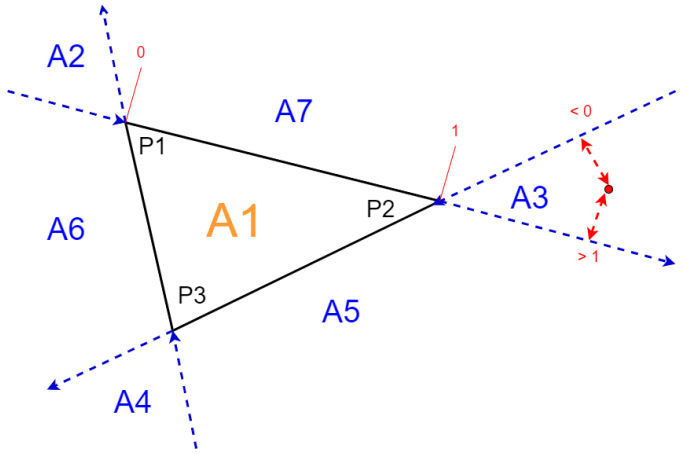


Figure 2.2: The closest point of the triangle to the red pixel is determined by projecting the pixel onto the directed triangle lines and determining if they are before, inside or after the triangle line segment.

The (n, d)-pairs, as well as the datastructures needed for the triangles, making up each polygon, can be computed for each surface in a preprocessing step.
As final step, the image is transformed to unsigned 8 Bit by interpolating the existing values $g \in [g_{min}, g_{max}]$ to the integer range of 0 to 255, with values near 0 indicating a high distance and values near 255 indicating a small distance to the next cell border.

Sampling Triangles

While working on this project, another approach came up, which was more promising in terms of time efficiency, compared to the above mentioned. Instead of iterating over all pixels and individually computing their distances, all objects are "drawn" into the image and then the distances to the nearest marked pixel determined.
Therefor, to represent a border splitting two cells, points are sampled on the plane surface and then the nearest pixels to that points are marked as belonging to a border. Different ways to sample on the plane would be e.g. on a regular grid or uniformly. To create the image in a non-deterministic way, the latter is preferential. To sample uniformly along the surface, the convex polygon is divided into triangles, similar to before. To sample within the triangle, the formula from [7] was used:
For a triangle $T = \{A, B, C\}$ firstly generate a uniformly distributed point on an arbitrary edge, for example $\overrightarrow{AB}$, according to

$$D = (1 - r) * A + r * B,$$

where $r \in U(0, 1)$. Secondly, sample a point non-uniformly on the line segment between C and D by,

$$E = (1 - \sqrt{s}) * C + \sqrt{s} * D,$$

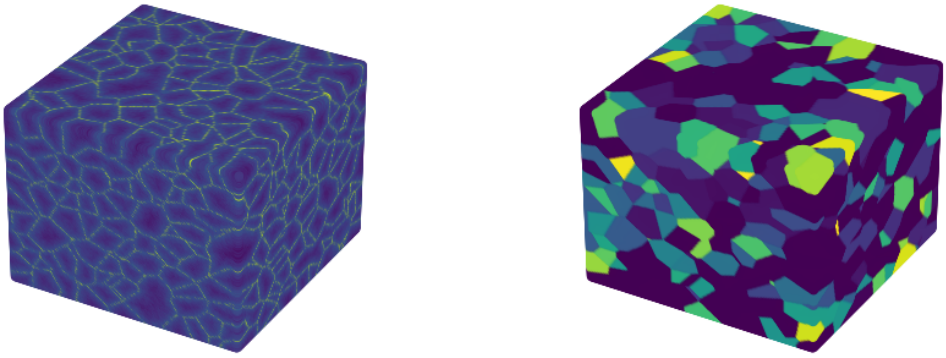


Image Ground Truth

Figure 2.3: Example of a false joins problem instance from the sampling method.

where $s \in U(0,1)$. This results in E being uniformly distributed over the triangle. The number of samples to create for a given triangle is based on the area of the triangle:

$$area = \frac{1}{2}\left\|\overrightarrow{AB} \times \overrightarrow{AC}\right\|.$$

The number of samples is then computed by multiplying the area with the square of the image resolution width and rounding to the next integer. The number could also be drawn proportionally to the triangle area.
The resulting sample coordinates are rounded to the next integer coordinate and the corresponding index marked in an image-sized array. This introduces an error by rounding the continuous sample to a discrete one.
Afterwards, to compute the distance of an index to the next marked index within the array, the Euclidean Distance Transform (EDT), included in the Python package *scipy.ndimage*, got applied. This makes uses of highly optimized array operations (BLAS) from the *numpy* package. It returns, for each array entry, the Euclidean distance to the nearest occurrence of a zero, hence all objects were labeled zero. To regulate the impact of the distance in the image, an approach similar to (2.1) was added. The distance values were divided by the image width and then the $k-th$ root was taken, with $k = 5$.
Finally, a noise filter is added onto the image. After interpolating the gray values, the complement of the image is computed, to make the cells have a gray value near 0 and the borders one near 255.
To construct the ground truth image, again we can use the EDT, which also provides the possibility to return the index of the nearest zero. Therefor, we round all sample point indices to the next integer coordinates and label the corresponding array entries as zero, for example in an array of ones. The transform is applied to determine the closest sample and storing the number of the sample as the ground truth value. This avoids having to iterate the pixels, as it was done in the other version. The rounding of sample indices before the distance computations might lead to small errors in the closest sample determination, however the error margin is only one pixel layer on the border between two cells and is thus rather insignificant for larger images.

2.1.2 False Cuts

Continuous Structures

The small tubular objects for the false cuts problem class are created with the help of splines, in this case Bézier curves are used. Bézier curves are parameterized functions $f(t)$ with $t \in [0,1]$, which are n-th grade polynomials with coefficients $a_i \in \mathbb{R}^m$ $(m = 3)$:

$$f(t) = a_0 + a_1 t + a_2 t^2 + ... + a_n t^n,$$

hence $f(t) \in \mathbb{R}^3$. Alternatively the curve can be represented in the Bernstein basis with so called control points $c_i \in \mathbb{R}^3, i = 0, ..., n$:

$$f(t) = \sum_{i=0}^{n} B_{n,i}(t) c_i.$$

The $B_{n,i}, i = 0, ..., n$ are the Bernstein polynomials defined as

$$B_{n,i}(t) = \frac{n!}{i!\,(n-i)!} t^i (1-t)^{n-i} = \binom{n}{i} t^i (1-t)^{n-i}.$$

The control points can be used to freely change the course of the curve. Therefore, a set of sample points can be randomly drawn and these points used as control points of the function. To make curves with different degrees, first a number $n \in [n_{min}, n_{max}]$ is randomly chosen, here $n_{min} = 4, n_{max} = 7$. Afterwards n control points $s_i \in [-1.5, 1.5]^3, i = 0, ..., n$ are uniformly sampled. From these a Bézier curve is defined.

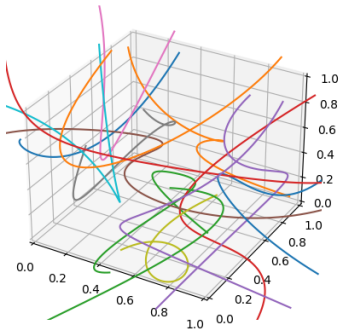


Figure 2.4: A set of curves intersecting the unit cube.

To make different curves distinguishable in the resulting image, it has to be assured that two curves aren't too close to each other. Therefor, if a newly created curve is closer than some distance d to any other existing curve, it is rejected. While there are different ways to compute the curve-to-curve distance, a simple way is to sample a number of k points along both curves and take the minimal distance between any two points, where one points is on the first and the other point is on the second curve. While this may not be a very efficient/accurate solution to the problem, it is fair enough since the parameter d is free to chose. Furthermore the solution converges to an exact solution for $k \to \infty$, but is computational quite expensive (k^2 point-to-point distances) for high k, although the distance computation for the samples of a pair of curves can be computed in parallel with the help of matrix operations.

A number of *num_spline_tries* curves were created, tested for their distance and rejected if they were closer than d to any other curve.

Discrete

Iterating Pixels

As for the false joins task, in the first version all pixels in the image are iterated and the gray value computed according to the same formula as in (2.1). Again the same parameters k and s were chosen.
Once more, the distance of each pixel to the closest tube is computed. While the naive way would be again to sample points along the curve and compute point-to-pixel distances, it is computational inefficient and inaccurate.
Another possibility, as described in [1], is to define a distance function $d(t)$ returning the distance of the curve $f(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$ at parameter t to the pixel $p = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$ and minimizing this function with respect to t.

$$d(t) = (x(t) - x_0)^2 + (y(t) - y_0)^2 + (z(t) - z_0)^2 = \sum_k (f_k(t) - p_k)^2$$

is the squared norm of the vector from $f(t)$ to p. We are interested in those

$$t^* \in \operatorname*{argmin}_{t \in [0,1]} d(t).$$

To find a minimum, different optimizers can be used. Due to it's simplicity, Newton's Method was implemented. The method is an iterative process, where starting from an initial parameter value t_0, the parameter is updated based on the slope of the function at the current parameter t_i, pushing it towards a nearby root ($f(t)$ and therefore $d(t)$ are n-th order polynomials, thus smooth and indefinitely often differentiable). The iteration performs the following update:

$$t_{i+1} = t_i - \frac{d(t_i)}{d'(t_i)}.$$

Since we are interested in a minimum, we are looking for a root in the first derivative $f'(t)$ resulting in the iteration:

$$t_{i+1} = t_i - \frac{d'(t_i)}{d''(t_i)}.$$

The first and second derivatives of $d(t)$ compute to:

$$\begin{aligned} \frac{\delta d(t)}{\delta t} &= \frac{\delta (x(t)^2 - 2x(t)x_0 + x_0^2 + y(t)^2 - 2y(t)y_0 + y_0^2 + z(t)^2 - 2z(t)z_0 + z_0^2)}{\delta t} \\ &= 2x(t)x'(t) - 2x'(t)x_0 + 2y(t)y'(t) - 2y'(t)y_0 + 2z(t)z'(t) - 2z'(t)z_0 \\ &= 2\sum_k (f_k(t)f_k'(t) - f_k'(t)p_k) \\ &= 2\sum_k f_k'(t)(f_k(t) - p_k). \end{aligned} \tag{2.3}$$

$$\begin{aligned}\frac{\delta^2 d(t)}{\delta t^2} &= \frac{\delta 2(x(t)x'(t) - x'(t)x_0 + y(t)y'(t) - y'(t)y_0 + z(t)z'(t) - z'(t)z_0)}{\delta t} \\ &= 2(x'(t)^2 + x(t)x''(t) - x''(t)x_0 + y'(t)^2 + y(t)y''(t) - y''(t)y_0 \\ &+ z'(t)^2 + z(t)z''(t) - z''(t)z_0) \\ &= 2\sum_k (f_k'(t)^2 + f_k(t)f_k''(t) - f_k''(t)p_k) \\ &= 2\sum_k (f_k'(t)^2 + f_k''(t)(f_k(t) - p_k)).\end{aligned} \tag{2.4}$$

Hence, the first and second order derivatives of the spline function are required [6]. Given the spline function $f(t) = \sum_{i=0}^{n} B_{n,i}(t)c_i$ in the Bernstein basis the derivatives of the Bernstein polynomials $B_{n,i}(t) = \frac{n!}{i!(n-i)!}t^i(1-t)^{n-i}$ compute to:

$$B'_{n,i}(t) = n(B_{n-1,i-1}(t) - B_{n-1,i}(t))$$

Since in $f(t)$ only the Bernstein polynomials are dependent on t we come to:

$$\begin{aligned}\frac{\delta f(t)}{\delta t} &= \sum_{i=0}^{n} B'_{n,i}(t)c_i \\ &= \sum_{i=0}^{n} n(B_{n-1,i-1}(t) - B_{n-1,i}(t))c_i \\ &= n(B_{n-1,-1}(t) - B_{n-1,0}(t))c_0 + n(B_{n-1,0}(t) - B_{n-1,1}(t))c_1 \\ &+ n(B_{n-1,1}(t) - B_{n-1,2}(t))c_2 + ... + n(B_{n-1,n-1}(t) - B_{n-1,n}(t))c_n \\ &= \sum_{i=0}^{n-1} B_{n-1,i}(t)(n(c_{i+1} - c_i)) \\ &= \sum_{i=0}^{n-1} B_{n-1,i}(t)q_i\end{aligned} \tag{2.5}$$

where $B_{n-1,-1} = B_{n-1,n} \coloneqq 0$ and $q_i = n(c_{i+1} - c_i)$. Similarly for the second derivative:

$$\begin{aligned}\frac{\delta^2 f(t)}{\delta t^2} &= \frac{\delta \sum_{i=0}^{n-1} B_{n-1,i}(t)(n(c_{i+1} - c_i))}{\delta t} \\ &= \sum_{i=0}^{n-2} B_{n-2,i}(t)(n(n-1)(c_{i+2} - 2c_{i+1} + c_i)) \\ &= \sum_{i=0}^{n-2} B_{n-2,i}(t)(n-1)(q_{i+1} - q_i)\end{aligned} \tag{2.6}$$

While the Bernstein functions have to be evaluated during run time, the scaled vectors q_i from p_i to p_{i+1}, as well as the factorials used for the Bernstein polynomials, can be computed in a preprocessing step.
Since Newton's method is prone to get stuck in local minima or not converge at all,

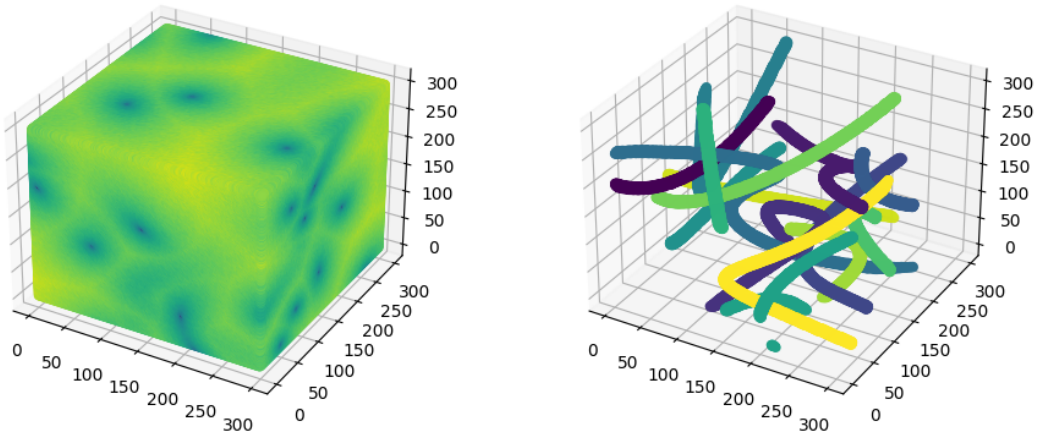


Image Ground Truth

Figure 2.5: Example of a false cuts problem instance from the sampling method.

depending on the initial parameter, multiple intializations are needed. A polynom of order n has at most $n-1$ minima. Initializing t_0 equidistantly at $n-1$ points along the spline guarantees for the method to visit the minimum in $[0,1]$ for the right parameters. A reduced set of initial points can be considered as well.
Unfortunately, the Newton's Method itself does not solely produce variables in the interval of $[0,1]$, which is why the image may contain artifacts at pixels where the iteration update would leave the boundary of $[0,1]$, but is instead rejected. A root finding algorithm that is bounded in the interval would need to be implemented, but was dropped due to the results in section 3.1, where we find that the sampling method clearly wins out in run time.
The ground truth is computed the same way as in the Sampling Curves section.

Sampling Curves

Similarly to the false joins problem case the alternative to iterating all pixels, is to first mark the curves inside the image and then compute the distances with an Euclidean Distance Transform. Sampling along a curve is more straightforward when compared to a polygon.
The number of samples can be calculated proportional to the length of the curve. The length is computed by

$$\int_{B([0,1])} 1dx = \int_0^1 \|B'(s)\|ds.$$

The number of samples is determined by multiplying the length times the image width and rounding to an integer. To track curves that are slightly outside the image, but are still closer to the pixels at the edge of the image than other curves, pixels slightly outside the image should be tracked as well. Pixels within a given distance of $extension\_rate$ to the image borders are considered.
Based on the resulting marks from the curve samples, the EDT can be applied again to get an distance image. A random noise filter is then added to make the task non-deterministic. After the interpolation, pixels nearer to curves end up closer to gray value 0.

For the ground truth, the markers from the samples can be used aswell. However, with only the markers, the curves appear very thin and not always seemingly continuous. To compensate for that, all pixels within a given radius r of an marked pixel can be considered as belonging to the object (once again using the distance image). This widens the thickness of curves and makes them consistent. For r, $r = ceil(image\_width/100)$ was used. The indices additionally provided by the transform, are used to label each distinct curve.

2.2 Segmentation

2.2.1 Watershed Segmentation

As segmentation algorithm, to test the both problem instances with, the watershed segmentation was chosen. The watershed interprets the gray values of the image as heigth values and thereby forms a topological surface over the image. Then this elevation surface is, figuratively speaking, flooded, beginning from a set of given markers (typically the local minima in gray values). The segmentation can be defined by the ridge lines of this process, i.e. the lines (or faces in 3D) along which the flooding of two different markers meet. Thus each initial marker defines one component in the resulting decomposition.
Normally, to segment a gray scale image the first step is to divide the image into foreground and background by thresholding. Then the distance of each pixel to the background is computed, once again with the help of the EDT. Finally, the watershed segmentation is performed on the negative of this distance image, where the minima will lie in the geometrical middle of objects, i.e. the points where the distance is maximal to the background.
Due to the construction of the problem cases in this work, the preceding steps can be omitted, except for the minima computation, as the gray values naturally come as a distance map. The watershed segmentation is directly applied on the input images.
Although one could determine the local gray value minima by searching in a neighborhood of n neighboring layers in each dimension, this tends to produce a higher number of minima than desired for smaller neighborhoods, cause the values locally deviate due to the noise.
A way to circumvent this, is to use h_minima. These are all minima with a depth $\geq h$. A local minimum with depth h is a minimum, that has at least one path to a deeper minimum along which the values aren't increasing by more than h with respect to the minimums value and no path with the given restrictions to a deeper minimum for a smaller h.
Therefore this prevents to find multiple minima that are only separated by "hills" with height less than h and hence reduces oversegmentation. Consequently, it is import that the gray values of borders between objects need to increase by more than h over the objects minima.

2.2.2 NL-LMP

On a side node, in the course of this project, experimentations with solving algorithms for the Node Labeling and Lifted Multicut Problem (NL-LMP) were tried

out. The NL-LMP is a problem, whose solutions are defined by both a valid node labeling x and lifted multicut y. The labeling x assigns to each pixel in the image exactly one class $l \in L$. The lifted multicut y is defined on a graph $G = (V, E)$ containing all pixels as nodes V and a set of edges E. A multicut now defines for each edge in the edge set, if the two connected vertices belong to the same component or different ones, under restrictions to keep the component structures non-empty, node-induced (i.e. the edges of a component are those of the total graph, that connect pairs of vertices in the component) and connected (i.e. there is no isolated vertex inside a component). Thus, a multicut defines a decomposition of the graph or equivalently a partition of the node set. In fact, there is a bijection between multicuts and decompositions ([4], Lemma 2). A lifted multicut y now defines a multicut on an edge set E' possibly larger than the original set E, again under a set of restrictions ([4], Lemma 5).
To solve the NL-LMP, different solving algorithms have been proposed [3]. These involve local search processes, that improve on a valid solution in each step, by minimizing

$$\psi(x,y) = \sum_{v\in V}\sum_{l\in L} c_{vl}x_{vl} + \sum_{vw\in A}\sum_{ll'\in L^2} c^{\sim}_{vw,ll'}x_{vl}x_{wl}(1-y_{\{v,w\}}) + \sum_{vw\in A}\sum_{ll'\in L^2} c^{\nsim}_{vw,ll'}x_{vl}x_{wl}y_{\{v,w\}}$$

for an arbitrary orientation A and cost functions $c, c^{\sim}$ and $c^{\nsim}$ that can be explicitly defined or learned from data.
For example, alternating improvements, to the labeling and to the multicut, can be searched for and applied. The alternating algorithm in [3] has been tried out for different graph structures during the project. While this algorithm manages to find a very good segmentation, especially for comprehensive graphs (e.g. complete graph), the complexity of the multicut update increases quickly for increasing numbers of edges and thus requires a very efficient implementation of the graph data structures. This includes a very quick lookup time for edges between specific vertices. Suboptimal data structures were used during the project and therefore, the segmentation wasn't efficient in time. The run time already became very high for rather small images (< 50 image width), also missing potential parallelization. A more efficient implementation is provided in the supplement of [3].

2.3 Metrics

The performance of the segmentation procedure is assessed with the help of two metrics, the Rand Index and the Variation of Information. While there are variants of the two that are "adjusted for chance", the plain versions are used here.

2.3.1 Rand Index

The Rand Index is a measure, that evaluates the similarity of two given partitions. This is done by taking all pairs of elements in the partitions and computing the relative amount of agreements, i.e. the two elements are in the same or different components in both clusterings. If the two elements are in the same component in one partition and in different components in the other partitions, it is a disagreement.

The Rand Index for n-element clusterings A and B hence computes to:

$$R(A,B) = \frac{s+d}{\binom{n}{2}} \in [0,1],$$

where s is the number of occurrences where the two elements in a pair are in the same component in both clusterings and d the number of occurrences where they are in different ones in both clusterings. The total amount of pairs of an n-element set is $\binom{n}{2}$.
As only the condition, whether two elements are in the same or different components, is evaluated, it avoids the need for the component indices to be aligned in both clusterings, i.e. the Rand Index is invariant under the permutation of the clusterings. A Rand Score of 1 corresponds to a perfect matching, while a Rand Score of 0 implies no agreements between the clusterings at all.

2.3.2 Variation of Information

The Variation of Information measures the distance of two partitions A and B. It can be computed as

$$VI(A,B) = H(A) + H(B) - 2I(A,B),$$

where $H(A)$ and $H(B)$ are the Entropies of of A and B and $I(A,B)$ is the Mutual Information Score.
In turn, the Entropy follows

$$H(A) = -\sum_i p_i \, log(p_i),$$

where $p_i, i = 1, ..., k_A$ (with k_A being the number of components in A) are the probabilities of component i appearing in the labels. These are simply the counts of occurrences of a component index i divided by the length of the labeling $n = len(A)$. The Entropy quantifies the "amount of information" held in A.
On the other side, the Mutual Information Score is a measure of the dependencies between two (random) variables A and B. It measures the "amount of information" of A that can be obtained through the observation of B or vice versa. It compares the difference between the joint distribution of $P_{(A,B)}$ and each marginal distribution P_A, P_B. It is calculated as

$$I(A,B) = \sum_{i=1}^{k_A} \sum_{j=1}^{k_B} \frac{|A_i \cap B_j|}{n} log\left(\frac{n|A_i \cap B_j|}{|A_i||B_j|}\right).$$

The resulting Variation of Information $VI(A,B)$ is bounded by $[0, log(n)]$ in the number of elements and $[0, 2\,log(max\{k_A, k_B\})]$ in the number of components. Thereby a value near 0 indicates a small distance between the two partitions and a higher value implies a larger difference. The variation is invariant under permutations of the clusterings as well.

Chapter 3

Results

3.1 Iterating Pixels vs. Sampling

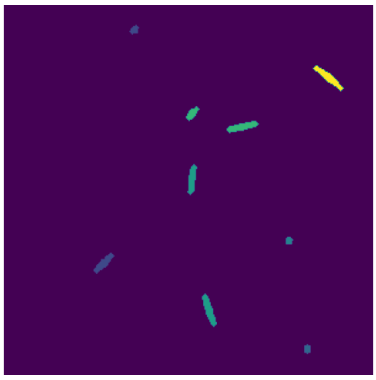
To chose which version of the discretization scheme should be used, a small test was made to compare the run times. A set of 5 images with image dimensions of $(50, 50, 50)$ was created for both tasks and both discretizations and the required time recorded. The parameters were set as follows:

PARAMETERS			
False Joins		False Cuts	
image_width	50	image_width	50
num_samples	50	d	0.05
num_removed_ surfaces	20	num_spline_ tries	15
		extension_rate	0.1

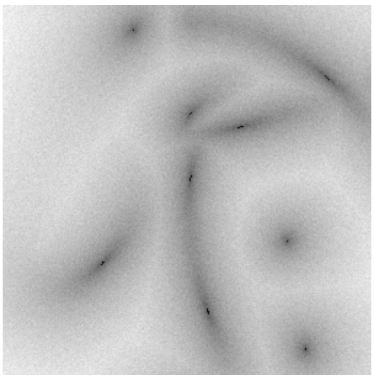
The experimentations were run on 5 CPU cores and 16 GB RAM, but were not run on isolated hardware. Therefor the times might deviate slightly from run to run.
The resulting times are shown in the table below and represent the time for the creation of an instance of both the image and the ground truth:

Average Time in s	False Joins	False Cuts
Iterating Pixels	155.09	2041.02
Sampling	0.44	0.08

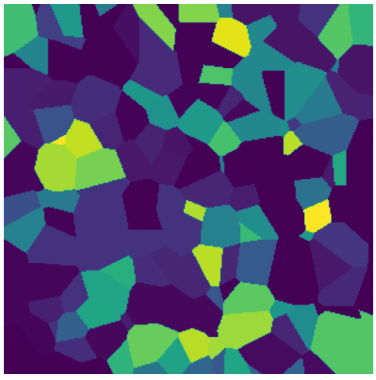
One can see, that the sampling method wins out significantly over the iteration for both tasks. Especially the false cut iteration is slow, due to many Newton's Method solving runs. This trend only increases for larger and more complex images. While the first version iterates every pixel and makes multiple computations for each one, the second version is able to generate the image within a couple of matrix operations. While there might be ways to further parallelize the iteration approach, the sampling approach provides a much more straightforward solution and would most likely still result in a faster run time.
Consequently, in the following experimentations, only the sampling version is considered.



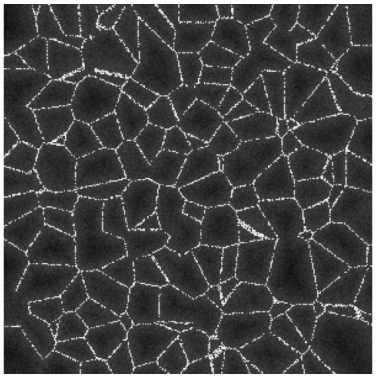
False Cut Ground Truth



False Cut Image



False Joins Ground Truth



False Joins Image

Figure 3.1: Slices of segmentation examples from both synthesized tasks.

3.2 Dataset

To test the performance of watershed segmentation, additional datasets with 25 examples were created for each task. Emphasis was laid on displaying a fairly complex scene. Therefor, both the resolution and the number of geometrical attributes were increased a lot. This way, the efficiency of the methods is presented as well. The noise filter produces a random number $u \in N(0, 0.02)$ for each pixel, as before. As a remark, $300^3 = 27,000,000$ pixels have to be processed for both the image and the ground truth.

PARAMETERS			
False Joins		False Cuts	
image_width	300	image_width	300
num_samples	1500	d	0.05
num_removed_surfaces	1000	num_spline_tries	40
		extension_rate	0.1
AVERAGE TIMES IN S			
41.4		21.5	

3.3 Assessment

3.3.1 Masked vs. Unmasked Watershed

The watershed was now applied to the images from both datasets. The performance, with respect to the mentioned metrics, on the false joins task ends up very high, as the images naturally form their minima, and therefore the watershed flooding basins, at the middle of each cell (Figure 3.2 (a), (c)). From there on the flooding moves towards the cell borders and meet with the floodings of neighboring cells to form a watershed ridge line.
While the false joins task includes a very convenient topology, the false cuts problem can not be solved efficiently by the watershed algorithm without any preprocessing. Since the false cut images have gray values near 0, thus minima, along the curves, the resulting flooding will end up meeting somewhere in empty space between two curves (Figure 3.2 (b)). The segmentation image looks somewhat similar to the joins task, and consequently does not reflect the strings inside the scene. Changing the minima to maxima or taking the complement of the image does not help that fact. A possibility to improve the capability, is to threshold the image before segmenting it. That way, the curves are extracted from the scene before the actual segmentation. The minima are still computed on the actual images, however the watershed segmentation done only with respect to the threshold mask.
While we can get very good results with using a mask for the false cut task and very good results without a mask for the false join task, the important question is, whether the algorithm is able to handle pictures, where thin objects as well as thin borders occur simultaneously, for example in different regions. Thus, it has to be decided if either masks are applied in both cases or no masks are applied.
In the following table the performances for the masked and unmasked approaches are shown. As parameters, threshold $\tau = 130$ and extremum-depth value $h = 20$ are used.

	False Joins			False Cuts		
	Time	Rand	VoI	Time	Rand	VoI
Unmasked	70.6	0.95	2.525	66.3	0.02	4.93
Masked	68.4	0.946	2.737	32.1	0.987	0.088
Masked (Foreground)				25.8	0.63	4.67

The Variation of Information (VoI) is bounded, in the number of elements, by

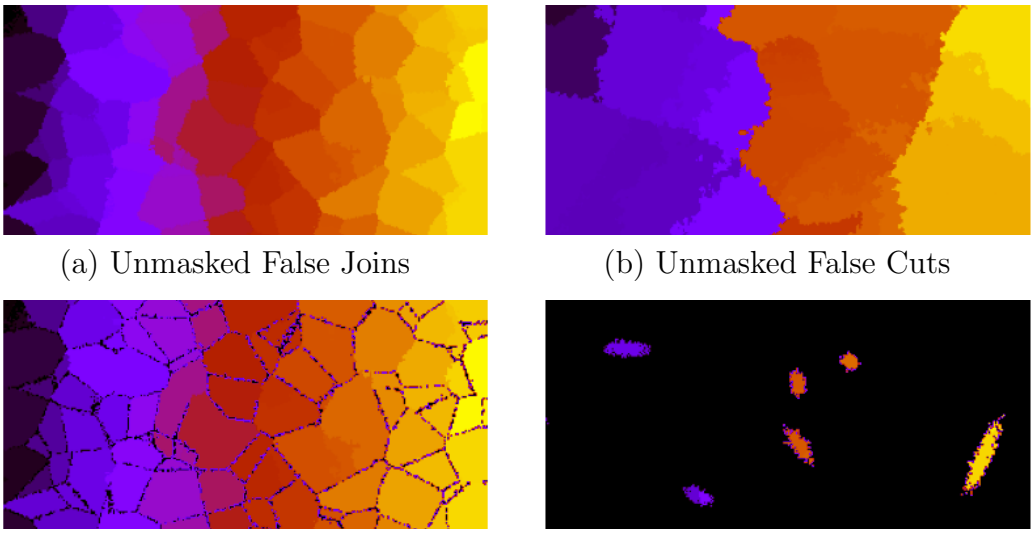
$$VI \leq log(27,000,000) \approx 17.1.$$

The bound from the number of classes varies from case to case. For example, the mean number of classes for the ground truth of masked joins segmentation is 581 and the mean number of classes in the prediction is 1616, hence

$$VI \leq 2log(1616) \approx 17.8.$$

For the cuts task, the numbers are 13 and 351 respectively. Therefore

$$VI \leq 2log(347) \approx 11.7.$$



(a) Unmasked False Joins (b) Unmasked False Cuts

(c) Masked False Joins (d) Masked False Cuts

Figure 3.2: Slices of examples from both synthesized tasks.

From these numbers we also see, that the false cuts task does in fact produce a number of false cuts, as the number of components in the prediction is much higher as in the ground truth.
For the false joins task the number in the prediction is actually higher as well, as the segmentation is more prone to split one of the non-convex cells than join two neighboring cells. This issue is addressed in subsection 3.3.3.
It shows, that the masked version wins out overall, due to the bad result from unmasked false cuts segmentation.
Although all pixels near cell surfaces in the masked false joins segmentation are classified as background and then not further considered for the watershed function, the performance remains reasonably good. Since the majority of the wrong classifications is made on the cell borders in the unmasked case as well, the significance of the error is not as high.
The performance of the masked cuts case will typically end up very high, as large parts of the image are going to belong to the background and the classification for these will mostly be correct from the thresholding already. Thus, to assess the accuracy of the actual segmentation, the Rand Index and Variation of Information could only be computed on the foreground. Hence only pixels, for which one of the two decompositions is non-zero (zero is background), are considered. The corresponding values are given in the third row of the table.

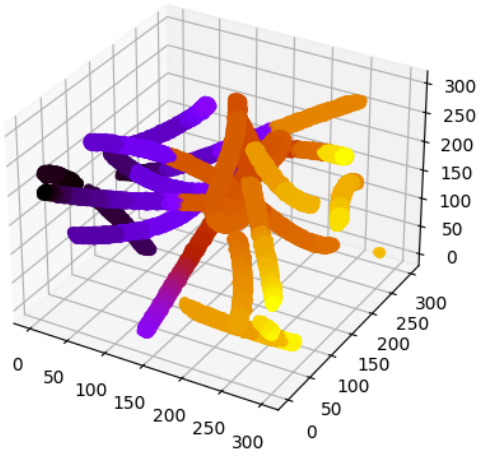


Figure 3.3: False Cuts Prediction.

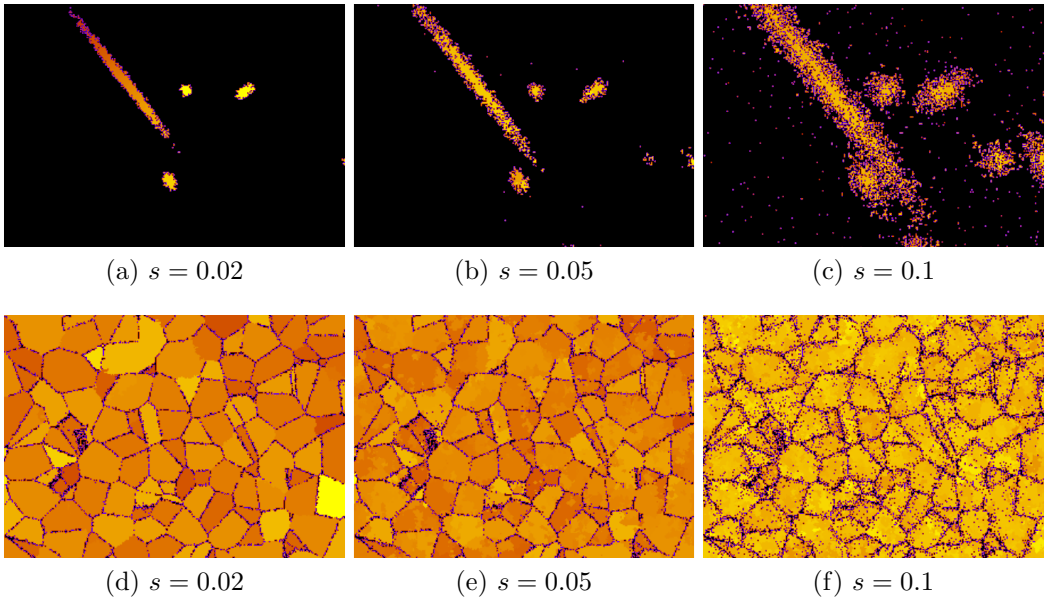
One sees that the performance here is much lower. The strings tend to be cut into multiple segments by the watershed

segmentation, i.e. producing false cuts.
In the following, the masked watershed segmentation is used and the false cut case evaluated with respect to the total and the decomposition foreground.

3.3.2 Increasing Noise

The algorithm is able to handle the problem images quite well. This is also due to the fact, that the borders and curves are visually good distinguishable. To test, how well it performs on more indistinct images, the noise $u \in N(0, s)$ was increased in the image creation, by fixating the geometry (a different one from the section before) and using different values for s. Then the accuracies were assessed again.

	False Joins			False Cuts		
	Time	Rand	VoI	Time	Rand	VoI
$s = 0.02$	76.2	0.941	2.793	30.4	0.987	0.088
$s = 0.05$	73.6	0.937	4.339	32.3	0.974	0.175
$s = 0.1$	70.9	0.922	6.268	34.8	0.906	0.651
$s = 0.02$	Foreground			24.5	0.612	4.657
$s = 0.05$	Foreground			26.9	0.432	6.221
$s = 0.1$	Foreground			30.4	0.172	8.759



(a) $s = 0.02$ (b) $s = 0.05$ (c) $s = 0.1$

(d) $s = 0.02$ (e) $s = 0.05$ (f) $s = 0.1$

Figure 3.4: (a)-(c) false cuts watershed segmentations for different standard deviations s: the number of components in the prediction increases from 347 ($s = 0.02$) to 112082 ($s = 0.1$) for 13 actual components (ground truth),
(d)-(f) false joins watershed segmentations for different standard deviations s: the number of components in the prediction increases from 1617 ($s = 0.02$) to 123300 ($s = 0.1$) for 581 actual components (ground truth)

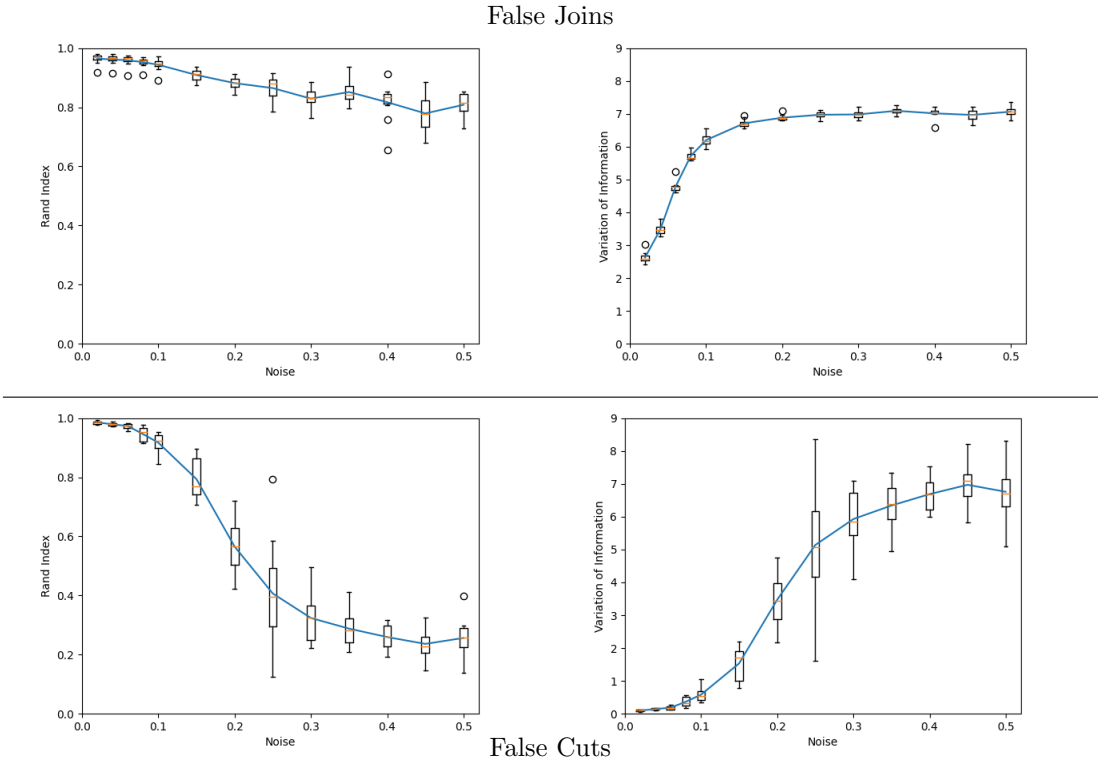


Figure 3.5: Boxplots of the performances for both tasks for varying levels of noise, parameterized by s on the x-axis, on datasets of 10 examples for each task. The blue line describes the mean of the according metrics over the datasets.

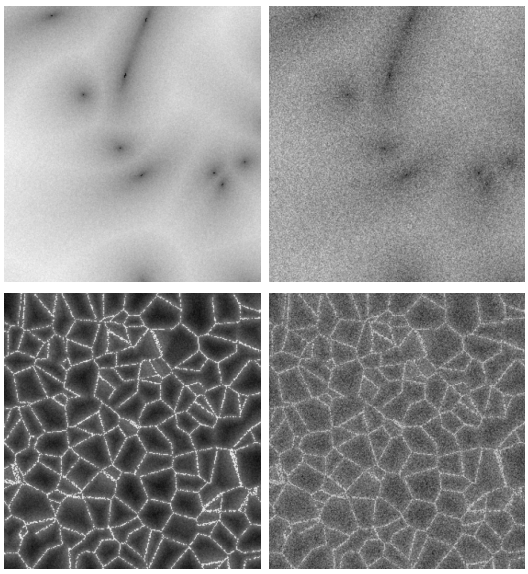


Table 3.1: Images with different noise impact.

As expected, the segmentation performance decreases for higher noise. Especially, the Variation of Information increases for increasing noise. This is due to the fact, that the algorithm tends to produces much more components as depicted in Figure 3.4. In a more noisy image, the h_minima function returns more and more minima due to local gray value fluctuations and the thresholding finds more outliers, which ends in more components.
The Rand Index is not changing as much, atleast for the false joins case. The increasing oversegmentation only produces disagreements for cell parts, which would otherwise be combined in one cell.
It also shows, that the false cuts instance is more prone to noise than the other. A more detailed evaluation of the metrics for varying noise levels is given in Figure 3.5.

3.3.3 Adjusting Minima Depth

We have seen that the segmentation for the false joins task inadvertently produced more false cuts then false joins. A way to address this property, is to change the depth value h of the minima. By increasing it's value, we increase the height of the "hill" by which two minima need to be separated, effectively reducing the number of minima and hence components. Thereby, we decrease the extent of oversegmentation and push the algorithm to have a higher risk of producing false joins instead of false cuts.
We see that h is a parameter to adjust the balance between tendencies to the two error classes.
The results of the performed experiments for different s and h values are shown in the tables below. The last two columns shown the mean number of components in both the ground truth (Comp_GT) and in the prediction (Comp_Pred). A higher number of components in the prediction than in the ground truth gives hint to a tendency towards false cuts. The other way round, it indicates false joins. The data can also be compared to the results of subsection 3.3.2. The tab of "Foreground" again shows the metrics evaluated on the foreground of the masked cuts case.

$s = 0.02$						
Mode	h	Time	Rand	VoI	Comp_GT	Comp_Pred
False Joins	45	73.4	0.941	2.386	581	1140
	65	60.6	0.94	2.602	581	438
	85	46.4	0.714	4.487	581	19
False Cuts	45	32.2	0.987	0.08	13	298
	65	29.5	0.987	0.08	13	295
	85	27.6	0.987	0.076	13	122
Foreground	45	25.4	0.614	3.81	13	298
	65	23.7	0.614	3.808	13	295
	85	21.7	0.614	3.49	13	122

$s = 0.05$						
Mode	h	Time	Rand	VoI	Comp_GT	Comp_Pred
False Joins	45	71.4	0.938	2.639	581	1109
	65	60	0.934	3.138	581	255
	85	51.9	0.383	5.073	581	4
False Cuts	45	33.3	0.975	0.144	13	282
	65	32.6	0.975	0.131	13	129
	85	30.5	0.975	0.118	13	28
Foreground	45	27.6	0.442	4.49	13	282
	65	27	0.448	3.737	13	129
	85	24.9	0.455	2.941	13	28

$s = 0.1$						
Mode	h	Time	Rand	VoI	Comp_GT	Comp_Pred
False Joins	45	66.9	0.923	3.166	581	1860
	65	59.6	0.917	3.681	581	185
	85	55.1	0.376	5.26	581	3
False Cuts	45	35	0.92	0.404	13	731
	65	34.8	0.92	0.338	13	59
	85	33.8	0.92	0.286	13	16
Foreground	45	30	0.208	5.285	13	731
	65	29.8	0.223	3.832	13	59
	85	28.8	0.262	2.646	13	16

We see that with increasing the depth h, the number of components in the prediction is reduced in all cases, pushing the segmentation towards false joins.
Indeed, the performance with respect to the two metrics is also decreasing for the false joins task with higher values of h. The performance of the false cuts task is slightly increasing for higher h.
This shows that the two tasks do in fact highlight the two specific errors. If the segmentation algorithm is pushed towards producing less false cuts, the performance for the false cuts image set increases and vice versa. The same goes for the false joins class.
An alternative way to balance the error classes would be to adjust the gray value difference between objects and the borders separating them in the data creation phase. This has a similar effect to the h value changes.

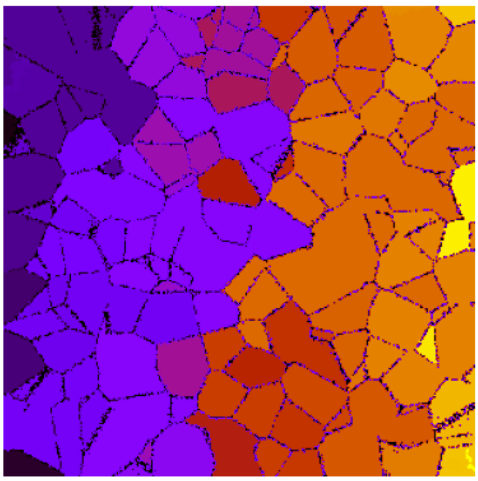


Figure 3.6: The segmentation includes more joined cells for higher depth values h. $(s = 0.02, h = 65)$

Chapter 4

Conclusion

In this research project, a pipeline was presented, that allows to test segmentation algorithms with respect to two specific types of classification errors. Therefor, a way to simulate data, which is used to assess the performance, was presented. There are two data types, that elevate each respective error class.
The data creation process is composed of a definition of the geometric structures in continuous space and a transfer of these structures on a distance based grid image. For the latter part, two variants were brought up and compared with respect to their run time. The version, that draws the objects in the image array first and then computes the distances as a sequence of matrix operations clearly won out, in terms of time efficiency, over the pixel iteration.
Two possible segmentation algorithms are described and the experimentations performed on the example of watershed segmentation. While there are different ways to apply a watershed function to an image, two alternatives, in a masked and a unmasked form, were tested.
We saw that thresholding the gray value images before the watershed and performing the segmentation only inside the found foreground mask greatly improved the accuracy for the false cuts image class, while the performance for the false joins class only decreased by a small amount.
The effect of noise, that is present in the synthesized data, on the segmentation was analyzed by generating variously noisy data and assessing the impact on the performance metrics.
Lastly, we found that the minima depth h is a parameter to balance the amount of false cuts and false joins, that the watershed segmentation produces and that each of the synthesized data types highlights the respective error classes.

Bibliography

[1] K. Atkinson, J. Kearney, and H. Wang. "Robust and Efficient Computation of the Closest Point on a Spline Curve". In: (2002). URL: http://homepage.cs.uiowa.edu/~kearney/pubs/CurvesAndSufacesClosestPoint.pdf.

[2] B. Andres et al. "3D segmentation of SBFSEM images of neuropil by a graphical model over supervoxel boundaries". In: *Medical Image Analysis* (2012). URL: https://brain.mpg.de/fileadmin/user_upload/Documents/Papers/Papers_Helmstaedter/2012_MedImAnalysis.pdf.

[3] E. Levinkov et al. "Joint Graph Decomposition and Node Labeling: Problem, Algorithms, Applications". In: (2016). URL: https://openaccess.thecvf.com/content_cvpr_2017/papers/Levinkov_Joint_Graph_Decomposition_CVPR_2017_paper.pdf.

[4] A. Horňáková, J.-H. Lange, and B. Andres. "Analysis and Optimization of Graph Decompositions by Lifted Multicuts". In: (2017). URL: http://proceedings.mlr.press/v70/hornakova17a/hornakova17a.pdf.

[5] bradgonesurfing. *Point-Triangle Distance*. URL: https://stackoverflow.com/a/47505833. (accessed: 02.03.2021).

[6] *Derivatives of a Bézier Curve*. URL: https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html. (accessed: 02.03.2021).

[7] *Uniformly sample over a Triangle*. URL: https://stackoverflow.com/a/11178935. (accessed: 02.03.2021).