Exploratory and computational analysis of huge volume images for connectomics

This diploma thesis has been carried out by Thorben Kröger at the Interdisciplinary Center for Scientific Computing under the supervision of Prof. Dr. Fred A. Hamprecht
Exploratory and computational analysis of huge volume images for connectomics

Obtaining the connectivity diagram of neural circuits is of great interest to understand how the brain works. Using electron microscopy, huge volume images of stained neural tissue can be acquired today, but manual analysis is prohibitively time consuming. In this work an automated segmentation procedure for surface-stained SBFSEM tissue data has been improved such that it can be applied to a dataset of $2000^3$ voxels for the first time. By employing MPI parallelization and efficient C++ data structures, the procedure can handle 64 times more data than before. First the contrast of the raw data is enhanced by means of supervised learning based on local voxel features and manual training data. Second a watershed transformation divides this data into supervoxels. In order to remove over-segmentation, classifier outputs based on contextual features for faces between supervoxels and their intersections are combined in an optimization problem.

For interactive exploration of the data, all intermediate steps and the final results, both in 2D and 3D, a performant visualization program has been written. Using caching and mesh reduction it can handle huge data volumes and enables a view on the segmentation never seen before.

A second dataset has higher resolution but uses a different staining technique which makes cell organelles visible. Contrary to first expectations it could be shown quantitatively that the segmentation procedure can be successfully applied in this setting. The influence of different ground-truth labels, sets of voxel features and watershed parameters on the segmentation result has been evaluated using objective error measures. Furthermore, the Radon-Like features specially developed for connectomics have been extended to 3D and their usefulness for segmentation has been evaluated.

To complete the wiring diagram, synapse detection is as important as segmentation. Towards automation, suitable geometric features on the discretized membrane surface are proposed and implemented: surface curvature, surface area and surface normal gray value statistics.
Diplomarbeit

im Studiengang Physik

eingereicht von

Thorben Kröger

geboren in Hannover

2010
Die Diplomarbeit wurde von Thorben Kröger ausgeführt am Interdisziplinären Zentrum für Wissenschaftliches Rechnen unter der Betreuung von Prof. Dr. Fred A. Hamprecht
Explorative und automatische Analyse riesiger Volumenbilder für Connectomics

Die Funktionsweise des Gehirns zu verstehen ist nicht möglich ohne Kenntnis der neuronalen Schaltkreise. Diese kann man prinzipiell aus elektronenmikroskopischen Bildern extrahieren, allerdings sind die riesigen Datenmengen in einem Menschenleben nicht manuell analysierbar.


Um Daten, Zwischenschritte und Endergebnisse in 2D und 3D interaktiv erkunden zu können, wurde ein neues performantes Visualisierungsprogramm entwickelt, das mittels Caching und Meshreduktion riesige Volumina handhabbar macht und neue Sichtweisen auf die Segmentierung ermöglicht.


Erklärung: Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 1. Oktober 2010
Acknowledgements

A year of hard work has been condensed into these few pages, a year full of new insights, templated code, build systems, computer clusters and late-night debugging.

I want to thank my supervisor Prof. Fred A. Hamprecht for giving me the opportunity to work on this wonderful project and for his guidance and Dr. Ullrich Köthe for his invaluable ideas and help.

Björn Andres has not only let me play with all his projects, but patiently kept explaining and helping me in every way. In return, I have introduced him to subversion and the CMake build system. With him as well as my fellow diploma students Rahul Nair and Nathan Hüsken there, working in the office remained being fun even when compilers refused to obey or results just would not become sensible.

I also want to thank all the other people in the HCI that made it worth going there every weekday. In particular, Christoph Straehle has endured my questions and feature requests regarding ILASTIK and within breaks, there were JIT Python compilers to discuss. Visiting Bernhard X. Kausler’s office for a short chat and an opinion on KDE is always a welcome distraction, and of course we owe him for organizing a wonderful group retreat. Anna Kreshuk worked on the same FIBSEM data as myself but does not want to admit how good her German actually is.

My thanks also go to my family, who have enabled me to study in Heidelberg and supported me in all my decisions and all my friends who kept me sane.

Most of all I want to thank my girlfriend Christine Hartmann, who has endured a year of too much work on my side, listened to all my problems and has still always been there for me. Your love means a lot.
1 Introduction
   1.1 Structural imaging techniques for neural science: an overview . . . . . . 1
   1.2 Neuron cytology: what to expect in EM images of neural tissue . . . . . 3
   1.3 The retina: an interesting model system for neural science . . . . . . . . 5

2 Segmentation of volume electron microscopy images for connectomics 7
   2.1 Overview of the segmentation procedure . . . . . . . . . . . . . . . . . . 7
   2.2 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.3 Voxel features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.3.1 Feature definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.3.2 Parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
   2.4 Learning to distinguish intra-cellular from extra-cellular tissue . . . . . 15
      2.4.1 Random Forests . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.4.2 Ground-truth voxel labels . . . . . . . . . . . . . . . . . . . . . . 16
      2.4.3 Training RF\textsubscript{1} . . . . . . . . . . . . . . . . . . . . . 18
      2.4.4 Block-wise parallel prediction . . . . . . . . . . . . . . . . . . . . 18
   2.5 Marker-based watershed segmentation . . . . . . . . . . . . . . . . . . . 19
      2.5.1 The watershed transform . . . . . . . . . . . . . . . . . . . . . . 19
      2.5.2 Choice of seeds and place in the segmentation procedure . . . . . 20
   2.6 Geometry extraction using CGP . . . . . . . . . . . . . . . . . . . . . . 21
      2.6.1 Theory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      2.6.2 From \textit{j}-components to polygons . . . . . . . . . . . . . . . . 26
      2.6.3 Classes for convenient access to extracted geometry and topology 27
      2.6.4 Improving the performance of topological grid construction . . . 33
      2.6.5 Running geometry and topology extraction as part of the processing
          chain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
   2.7 Learning to distinguish essential from excessive faces . . . . . . . . . . . 36
      2.7.1 Face features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
      2.7.2 Parallelizing the feature calculation . . . . . . . . . . . . . . . . . 37
      2.7.3 Labels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
      2.7.4 Training RF\textsubscript{2} . . . . . . . . . . . . . . . . . . . . . . . 39
      2.7.5 Parallelizing the face classification . . . . . . . . . . . . . . . . . 40
   2.8 Learning to predict the joint configuration of intersecting faces . . . . . 41
      2.8.1 Intersection features . . . . . . . . . . . . . . . . . . . . . . . . . 41
      2.8.2 Parallelizing the feature computation . . . . . . . . . . . . . . . . 42
      2.8.3 Training RF\textsubscript{3} . . . . . . . . . . . . . . . . . . . . . . . 43
      2.8.4 Parallelizing the intersection classification . . . . . . . . . . . . . 44
   2.9 Globally consistent face removal using a Graphical Model . . . . . . . . 45
      2.9.1 Graphical Models, energy function and optimization . . . . . . . . 45
      2.9.2 Enabling belief propagation for very large Graphical Models . . . . 51
      2.9.3 Constructing the Graphical Model . . . . . . . . . . . . . . . . . . 52
      2.9.4 Finding approximate solutions . . . . . . . . . . . . . . . . . . . . 53
## Contents

2.10 Running the segmentation pipeline .............................................. 54  
2.11 Validation .................................................................................. 55  
  2.11.1 Comparing segmentations ....................................................... 55  
  2.11.2 Comparing face labels .......................................................... 58  
  2.11.3 Sparse skeletons .................................................................... 59  
  2.11.4 Visual inspection of results on e1088 ..................................... 62  

3 Towards automatic synapse detection: geometric features 69  
  3.1 Membrane curvature ................................................................. 70  
    3.1.1 Definition of surface curvature ........................................... 70  
    3.1.2 Estimating the curvature .................................................... 71  
    3.1.3 Implementation of curvature estimation ............................... 73  
  3.2 Membrane size ........................................................................... 74  
  3.3 Surface normal gray value statistics .......................................... 74  
  3.4 Outlook ...................................................................................... 75  

4 Visualization of intermediate steps and results 77  
  4.1 Design considerations ............................................................... 77  
  4.2 Usage of SegmentationViewer .................................................... 79  
  4.3 Implementation details .............................................................. 84  
    4.3.1 Pre-processing ................................................................. 85  
    4.3.2 Classes .............................................................................. 88  

5 Applying the first steps of the segmentation procedure to FIBSEM volume images 91  
  5.1 ILASTIK ..................................................................................... 92  
    5.1.1 Towards segmentation: choices facing the user .................. 92  
    5.1.2 Extensions ......................................................................... 93  
  5.2 Ground-truth segmentation ....................................................... 93  
    5.2.1 What to label? ................................................................... 93  
    5.2.2 Dense manual voxel labels ............................................... 94  
    5.2.3 Skeletons .......................................................................... 95  
    5.2.4 Deriving dense voxel labels from skeletons ....................... 95  
  5.3 Labeling strategies for classifier training ..................................... 95  
    5.3.1 Membrane versus background .......................................... 96  
    5.3.2 Membrane, vesicles and mitochondria versus background .... 96  
  5.4 Feature selection ......................................................................... 98  
  5.5 Radon-Like features in 3D ........................................................... 99  
  5.6 Evaluation using VI and OCE error measures ............................ 100  
  5.7 Conclusions .............................................................................. 101  

6 Conclusion 107  

Bibliography 109
1 Introduction

1.1 Structural imaging techniques for neural science: an overview

In biological research, obtaining three-dimensional structural information on many different length scales is a prerequisite for understanding the purpose of complex functional units.

Unraveling the architecture of large nervous systems such as the human brain or retina has been the dream of scientists for generations. Already at the end of the nineteenth century, Camillo Golgi had developed a technique for selectively staining individual nerve cells or neurons (Golgi, 1906), and Ramón y Cajal (Andres-Barquin, 2001) used this method extensively for his studies of the citology of the nervous system. Both shared the Nobel Prize in Physiology or Medicine in 1906. Though significant insight can be gained about the structure of individual nerve cells with a selective staining, the collective function of all neurons as a complex network with axons and dendrites as “wires” and synapses as “contacts” between them cannot be studied.

On a large scale (encompassing the entire brain), techniques such as functional MRI, EEG and DTI have been used to get an idea of the network structure and information flow between anatomically distinct brain regions (Modha and Singh, 2010). The low resolution represents an averaging over the underlying neuronal structure, and thus only little information can be gained about the functional principle of the smallest units, the neuronal circuits themselves.

However, obtaining the complete connectivity pattern of all neurons would be of tremendous value for understanding neural computations (Briggman and Denk, 2006). Unfortunately, there is no hope of being able to assemble the connectivity information (termed the human connectome in Sporns et al., 2005) for the entire brain, as it would comprise an estimated $10^{11}$ neurons and $10^{15}$ synapses, which is several orders of magnitude larger than the size of the genome, which consists of only $3 \times 10^9$ base pairs with 20,000 to 30,000 genes (Sporns et al., 2005). However, having the complete connectivity information of only a small cube of neural tissue with lengths of some hundred μm in each dimension will already enable biologists to validate hypotheses about the geometrical statistics of neural wiring (Briggman and Denk, 2006), or the detection of motion in the retina (Denk and Horstmann, 2004).


Electron microscopy. To resolve even the narrowest neuronal processes (diameter approx. 100nm) and the thinnest necks of dendritic spines (diameter approx. 50nm) in the mammalian brain, electron microscopy (EM) techniques are needed as they offer the required imaging resolution (Briggman and Denk, 2006).

Until recently, 3D volumes of tissue have been imaged using transmission electron microscopy (TEM). A block of plastic-embedded tissue is consecutively cut into very...
thin slices, which are then transferred one by one into the imaging plane of an electron microscope (serial section transmission electron microscopy, SSTEM). A substantial fraction of the electrons pass through the slice and are focused onto film. The contrast is obtained by prior staining with heavy metals: the deflection of the charged beam depends on the electron density in the material. A drawback of this approach is that the thickness of the slices is limited and loss of sections, section folds and serious distortion artifacts occur frequently enough to make obtaining large stacks of aligned TEM images very difficult (Denk and Horstmann, 2004; Knott et al., 2008). Because non-uniform distortion cannot be prevented entirely, the imaged sections need to be aligned in a complicated postprocessing step (Kaynig et al., 2007; Mishchenko, 2009).

Block face imaging. A new approach is to image the block face before cutting. As transmission is not an option anymore, imaging techniques have to make use of secondary or back-scattered electrons. In scanning electron microscopy (SEM), a tightly focused beam of electrons is raster-scanned over the block face and the back-scattered electrons are detected.

Denk and Horstmann (2004) introduced the Serial Block-Face Scanning Electron Microscopy (SBFSEM). A custom-designed ultra-microtome with a diamond knife is used to reliably obtain stacks of thousands of images with a slice thickness of as low as 25nm. Imaging the block-face prior to cutting means that distortion is now negligible and the risk of losing sections is reduced.

The sample is placed in low vacuum conditions. Though the overwhelming fraction of electrons are able to reach the sample unscattered, the rest strike gas molecules and create ions, which are needed to prevent the accumulation of residual charge on the block surface (Helmstaedter et al., 2008).

The limited slice thickness of SBFSEM can be overcome by replacing the mechanical microtome with focused ion-beam (FIB) milling (see Knott et al., 2008). A focused beam of gallium ions interacts with the sample, ablating a thin layer from the block surface. To reduce milling artifacts like non-uniform thickness due to material differences, platinum solid is deposited on the surface of the sample by means of decomposition of a precursor gas in the presence of the gallium ions. The FIB approach is less sensitive to damage of the sample’s plastic embedding material (incurred by the large amounts of energy deposited by the electron beam), which complicates mechanical cutting. Therefore, higher electron doses are possible, allowing for higher resolution and better signal-to-noise ratio. However, the uniformity of the section thickness remains a problem (Helmstaedter et al., 2008).

Staining. Another important consideration is the staining used to provide the contrast of the electron microscopy images (Helmstaedter et al., 2008). The selection of an appropriate staining strategy depends on the objective of the data evaluation.

- **Conventional staining** with osmium, uranyl acetate and lead citrate stains the cell membranes and in addition different cell organelles like mitochondria, vesicles, ribosomes and microtubules. It also enhances the post-synaptic density. Fig. 1.2 shows an exemplary serial section TEM image with conventional staining where different cell parts have been labeled manually.

- **Surface staining** relies on the infiltration of horseradish peroxidase (HRP) into the living tissue. It enhances the extra-cellular space between individual neurons, but does not stain any cell organelles. This type of contrast enhancement is promising for
1.2 Neuron cytology: what to expect in EM images of neural tissue

automated tracing of all neurons in the volume as it greatly reduces the complexity of the data. However, it is yet unclear whether the detection of synapses, a prerequisite for reconstructing the neural circuit, is possible from the geometry of the neurons alone (chap. 3).

Datasets. The work presented here is mainly concerned with two electron microscopy volume images of neural tissue:

• A SBFSEM volume image (named “e1088”) prepared with surface staining shows part of the Inner Plexiform Layer (IPL) of rabbit retina. It has the almost isotropic resolution of $22 \times 22 \times 30 \text{ nm}^3$ per voxel, with a total size of $2048 \times 1892 \times 2048$ or approx. $7.5 \times 10^9$ voxels. The data was kindly provided by the group of Prof. W. Denk at the Max Planck Institute for Medical Research, Heidelberg (chap. 2 and 3).
• A FIBSEM volume image of $2048 \times 1536 \times 1435 \approx 4.5 \times 10^9$ voxels with conventional staining (osmium and uranyl acetate), kindly provided by the group of Dr. G. Knott at the École polytechnique fédérale de Lausanne. The data shows a region of the hippocampus of an adult rat brain. Image resolution is isotropic with 5nm per pixel in $x$- and $y$-direction, with consecutive slices between 5nm and 6nm apart. The tissue was embedded in epoxy resin hardened at 60°C for 48 hours (chap. 5).

Acquisition time. A limiting factor for all scanning electron microscopy techniques is the acquisition time. For a single voxel, the voxel dwell time depends on the desired signal-to-noise ratio, the electron beam current and the back-scattering coefficient of the imaged material, and is chosen in experimental setups to be in the order of tens of µs (Denk and Horstmann, 2004). Thus the imaging of a meaningful brain region, a cortical column of size $300 \times 300 \times 1000 \text{ µm}^3$, would take approx. two years. Analysis speed is the major bottleneck however. For a complete manual reconstruction, Helmstaedter et al. (2008) estimate a few thousand person-years of work. This time might be reducable up to a factor of 70 by not tracing every voxel in the volume, but only a skeleton for each neuron (Helmstaedter et al., 2010). Still it is clear that automated analysis is needed.

This thesis improves upon a fully automatic procedure (Andres et al., 2008b) for segmenting (that is partitioning into meaningful three dimensional parts) electron microscopy volume images (chap. 2 for surface stained tissue, chap. 5 for conventionally stained high-resolution data). Using algorithms that are at most log-linear in the number of voxels and fast parallel processing, the computation time for the automated analysis of the entire e1088 dataset is in the order of weeks. Though additional human post-processing will still be necessary, accurate reconstructions might soon be possible in acceptable time.

1.2 Neuron cytology: what to expect in EM images of neural tissue

Nerve cells. The Neuron doctrine put forward by Ramón y Cajal asserts that “the nervous system is made up of individual signaling elements, the neurons, which contact one another only at specialized points of interaction, called synapses” (Kandel et al., 1991, chapter 1) and his Principle of connectional specificity states that “each cell makes specific connections at precise and specialized points of synaptic contacts – with some
postsynaptic target cells but not with others” (Kandel et al., 1991, chapter 2).

A model neuron is made up of different components:

- The cell body or soma contains the cell nucleus.
- **Dendrites** are thin processes emerging from the soma, branching many times to form the **dendritic tree**, which can receive input from many other nerve cells.
- The soma also gives rise to one **axon**, a thin, cylindrical process which is responsible for sending the received and transformed input signal onwards to other nerve cells, sometimes over considerable length. While the signal received at the dendritic branches is analog, a region called the **axon hillock** is responsible for conversion into a binary representation with a frequency-based encoding scheme. The electrical signal is called an **action potential**.
- The action potential is borne to the end of the axon which splits into fine branches. Special sites called **presynaptic terminals** are responsible for transmitting the signal to the neighboring dendrite of another cell, called the **postsynaptic dendrite** in this context. The dendrites have small **dendritic spines** docking onto the presynaptic terminal. The separating extra-cellular space is called the **synaptic cleft**; together the whole signal transmission site is known as the synapse. At the presynaptic terminals, organelles called **vesicles** release a chemical transmitter which forms the output signal of the axon, binding to chemical receptors of the receiving cell.

Synaptic connections can either be **excitatory** or **inhibitory**. If the connection is inhibitory, the ability of the receiving cell to generate an action potential is decreased; while for excitatory connections this ability is increased upon receiving an incoming signal. In electron microscopy images, there is no hope of reliably differentiating between these two types; it will only be possible to say that there is a connection.

For a more involved treatment of neural science, Kandel et al. (1991) is an excellent textbook.

**Figure 1.1** - a) A presynaptic terminal interfaces via two synapses to two dendritic spines. The length of the scale bar is approx. 400nm. b) Tracing of (a), with labels for the major synaptic structures. Figure taken from Kennedy (2000).
1.3 The retina: an interesting model system for neural science

One of the datasets of interest in this thesis (see sec. 1.1) is a SBFSEM volume image ("e1088") of the inner plexiform layer of rabbit retina, which includes parts of the inner-nuclear and ganglion-cell layers (Helmstaedter et al., 2010, see also fig. 1.3).

The retina is responsible for pre-processing the visual information which has been focused by the lens of the eye onto photoreceptor cells (rods and cones). Information flows from the receptors over different types of interneurons (bipolar, horizontal and amacrine cells) to the ganglion cells, whose axons form the optical nerve going into the brain itself.

The retina is a particularly interesting model system. It is part of the central nervous system and originates from the same part of the embryo that later gives rise to the brain; thus they share similar organizational structure. In contrast to the brain however, the retina is quite simple, containing only five classes of neurons which are layered in an orderly fashion. All the complexity stems from the synaptic connections between the nerve cells. Fig. 1.3 shows this layered organization of the retina. With obtaining the connectivity matrix for parts of the retina, it may be possible to understand the hierarchical, parallel processing of the input image into pre-processed information containing first computations about movement, edgeness etc. (Kandel et al., 1991, chap. 28).
Figure 1.3 – Schematic drawing of a cross-section of the retinal layers; taken from Gray (1918).
2 Segmentation of volume electron microscopy images for connectomics

Reconstructing the neural circuit from electron microscopy volume data requires two steps:

(1) extracting the shape and location of all neurons and
(2) finding all synapses in the data.

The focus of this chapter is the automation of task (1) for surface stained SBFSEM volume data, while segmentation of conventionally stained high-resolution FIBSEM volume data is considered in chap. 5.

First steps towards task (2) are discussed in chap. 3.

As Jain et al. (2010b) note, mistakes in following neurites have disastrous consequences on the connectome because an erroneous connection of axon or dendrite with the soma will assign all associated synapses to the wrong neuron. The result is a global error, while failures in synapse classification constitute only a local error.

For segmentation of SBFSEM volume data with surface staining we have proposed a multi-step segmentation procedure (Andres et al., 2010d, building on previous work Andres et al., 2008b). Using an improved implementation of the processing chain which is presented in this chapter, I have been able to apply this segmentation procedure for the first time to the full e1088 dataset. Making use of MPI parallelization, memory-saving data structures and completely written in C++, total runtime is less than 2 weeks on a 4 × 4-core machine with 128GB RAM. Vast improvements in usability and speed compared to an earlier MATLAB implementation finally allow systematic parameter tuning on significant subsets of the data.

Sec. 2.1 gives an overview of the segmentation procedure and theory, implementation and experiments are discussed in detail for each step in sections 2.3 through 2.9. A short overview of related work is the subject of sec. 2.2. Runtime measurements for all steps necessary for segmentation of e1088 are reported in sec. 2.10, while validation, both on small subsets as well as on the complete e1088 dataset, is discussed in sec. 2.11.

2.1 Overview of the segmentation procedure

The segmentation procedure consists of the following steps:

(1) For each voxel in the raw data, 28 rotation invariant features are extracted describing its neighborhood (sec. 2.3).

(2) Using manually annotated training data, a Random Forest classifier RF is trained to distinguish extra-cellular space (stained) from intra-cellular space (unstained). By predicting the class label for each voxel – as represented by the features computed in step 1 – a probability map is obtained where each voxel contains the probability of it belonging to the extra-cellular class (sec. 2.4).

(3) The marker-based watershed algorithm is used to partition the volume image into an over-segmentation of |S| connected components, called supervoxels. The probability
map from step 2 serves as the elevation map, while seeds are selected based on size and confidence of them lying inside the cell (sec. 2.5).

(4) Two adjacent supervoxels from the over-segmentation (step 3) are separated by a two-dimensional surface (face). Two such faces, in turn, are separated by a 1-dimensional curve. A description of this geometry, including all neighborhood relations between the above entities, are extracted using the CGP algorithm, and stored in an efficient data structure (sec. 2.6).

(5) A second classifier, RF₂, is trained to distinguish essential and excessive faces. Ground-truth face annotations can either be obtained manually or derived from dense manual voxel annotations. The sizes of the adjacent supervoxels as well as the distribution of filter responses over the surface serve as features. The result is a probability for each face (extracted in step 4) to be essential (sec. 2.7).

(6) A third classifier, RF₃, is used to jointly predict the true configuration of all faces bounded by a particular curve based on local geometry and topology alone. As features, the angles between these faces and various statistics over them are used (sec. 2.8), reflecting the observation that in neural tissue, smooth continuations of supervoxel faces are much more likely than sharp edges. For training, the labels from the previous step are re-used.

(7) The predictions of RF₂ and RF₃ are combined in a unified Graphical Model. Minimization of the energy function results in a joint optimal decision to remove or preserve faces (sec. 2.9).

The final resulting segmentation is obtained by removing the faces discarded in step 7 and merging the bounded supervoxels.

In the following, these steps are described in detail. For each I first review the necessary theory and then go on to describe implementation, experiments and results. For all steps, C++ command line tools have been written. As an example of how to apply the procedure to a new SBFSEM data volume, assume that it is stored as an HDF5 file named D.h5 within a dataset called sbfsem (datatype: unsigned char). In the following, all program descriptions refer to this naming convention. Alternatively, the Python interface of the processing chain can be used by constructing a Pipeline object: p = denkc.Pipeline('D.h5').

2.2 Related work

Jain et al. (2010b) give a good overview of current work on automated segmentation for connectomics.

Many EM imaging projects produce serial-section transmission electron microscopy images prepared with a staining that highlights cell organelles such as mitochondria and vesicles in addition to cell membranes. Segmenting this data is challenging because of anisotropic resolution, distortion artifacts and clutter (see also chap. 5). Jurrus et al. (2010) detect membranes in 2D with a serial artificial neural network followed by a linking step. Kaynig et al. (2010) use a Random Forest classifier with Haar-like and histogram features as one term in an energy function to enforce perceptual grouping constraints on 2D slices which is solved by graph cut. Chklovskii et al. (2010) describe a semi-automatic reconstruction pipeline for ssTEM images.

Detecting clutter has been studied for example in Lucchi et al. (2010), who start from

---

1 http://www.hdfgroup.org/HDF5
a superpixel segmentation and use support vector machines with shape, texture and boundary features and a final graph-cut optimization to detect mitochondria. Kumar et al. (2010) propose new features for the detection of membranes, mitochondria and vesicle clusters (see also chap. 5).

Light microscopic imaging of neural tissue labeled with different fluorescent proteins may allow for easier tracking of individual neurons. Lu et al. (2009) use a semi-automated segmentation procedure relying on seeded region growing with simple stopping criteria.

On surface-stained data similar to e1088, Macke et al. (2008) use a 2D semi-automated approach. Seung and colleagues are working on the same e1088 dataset. They use convolutional neural networks (Turaga et al., 2010) to contrast-enhance the raw data, followed by a watershed segmentation. The features are selected implicitly during learning, thus there is no need for step 1 in their approach (at the cost of having to approximately solve a complicated optimization problem during learning). They have also proposed the warping error for comparing two segmentations (Jain et al., 2010a) which penalizes topological disagreements, an important property for evaluating automatic segmentation results.

2.3 Voxel features

The raw data \( d(x) \) is a function \( d : G \to [0, 2^8) \) where the voxel grid \( G \) has an extent of \( G = [0, n_1) \times [0, n_2) \times [0, n_3) \). For each voxel \( x \) a number of features are computed as a compact but discriminative description of its three-dimensional neighborhood (note that the resolution of both datasets used in this thesis is almost isotropic).

In the following, \([a, b]\) or \([a, b)\) usually denote a consecutive interval of integers and indices starting with 0 are used for easier comparison with the actual C++ implementations.

2.3.1 Feature definitions

Bilateral filter. In scanning electron microscopy, back-scattered electrons are imaged, resulting in a low signal-to-noise ratio. For segmentation purposes it is crucial that the extra-cellular space, which is mostly visible as a very thin boundary between cells, is easily recognizable in a local window. To denoise the raw data, a bilateral filter is applied, see fig. 2.1a.

While simple Gaussian smoothing has the advantage of speed (the filter is separable), this low-pass filtering blurs edges. Bilateral filtering uses a non-separable, locally adaptive filter mask to preserve edges (Tomasi and Manduchi 1998, relation to adaptive smoothing and anisotropic diffusion shown in Barash 2002). As boundaries are averaged along their main direction, it is to be hoped that small holes due to noise or staining problems become less of a problem.

In bilateral filtering, the input image \( d(x), x \in \mathbb{R}^3 \), is transformed into the output image \( \tilde{d}(x) \) by

\[
\tilde{d}(x) = \frac{\int \int d(x)c(x, x')s(d(x), I(x')) \, dx'}{\int \int c(x, x')s(I(x), d(x')) \, dx'},
\]

where \( c(\cdot, \cdot) \) is a function measuring closeness in the spatial domain, while \( s(\cdot, \cdot) \) measures similarity in the image’s range. The bilateral filter uses the following similarity
measures:

\[
c(x, x') = \exp \left( -\frac{\|x - x'\|^2}{2\sigma_s^2} \right) \quad (2.2)
\]

\[
s(v, v') = \frac{1}{1 + \frac{(v - v')^2}{\sigma_v^2}} . \quad (2.3)
\]

**Gaussian gradient magnitude.** The gradient magnitude is an indicator of local edge strength. First the components of the gradient vector \( \mathbf{D}_i(x) = (\nabla d(x))_i \) are computed by separable convolution. A Gaussian smoothing filter (standard deviation \( \sigma_m \)) is applied to all dimensions other than the one for which the partial derivative is calculated.

---

**Figure 2.1** – Examples of calculated voxel features. See fig. 2.7 and 2.26 for the next steps of the processing chain shown for the same slice.
2.3 Voxel features

<table>
<thead>
<tr>
<th>index</th>
<th>feature</th>
<th>scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>raw data</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>bilateral filter</td>
<td>$\sigma_s = 1, \sigma_v = 3; 4$ iterations</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian gradient magnitude</td>
<td>$\sigma_m = 1$</td>
</tr>
<tr>
<td>3-5</td>
<td>eigenvalues of Structure Tensor (descending)</td>
<td>$\sigma_i = 1, \sigma_o = 2\sigma_i$</td>
</tr>
<tr>
<td>6-8</td>
<td></td>
<td>$\sigma_i = 1.5, \sigma_o = 2\sigma_i$</td>
</tr>
<tr>
<td>9-11</td>
<td></td>
<td>$\sigma_i = 1$, $\sigma_o = 2\sigma_i$</td>
</tr>
<tr>
<td>12-14</td>
<td></td>
<td>$\sigma_h = 1$</td>
</tr>
<tr>
<td>15-27</td>
<td></td>
<td>$\sigma_h = 1.5$</td>
</tr>
</tbody>
</table>

Table 2.1 – Voxel features, as calculated by the program voxel-features-mpi, tab. 2.2 with default settings.

Finally the gradient magnitude is calculated as $|\nabla d(x)| = (\sum_{i=1}^{3} D_i \circ D_i)^{1/2}$, where $(\circ)$ denotes a point-wise product, see Jähne (2005, chap. 12).

**Structure Tensor.** A local descriptor for orientation in an image should treat opposite gradient directions as equal. One approach therefore is to find a normal direction $\mathbf{n}$ for which the average squared projection of the gradient vector over a window $w(x)$ is maximal:

$$\max_{\mathbf{n}} \left[ \int w(x - x') \left( \nabla g(x') \mathbf{n} \right)^2 d^3x' \right] = \max_{\mathbf{n}} \left[ \mathbf{n}^T \mathbf{J} \mathbf{n} \right]$$  \hspace{1cm} (2.4)

with

$$\mathbf{J} = \int w(x - x') \left( \nabla g(x') \nabla g(x')^T \right) d^3x'. \hspace{1cm} (2.5)$$

The eigenvector corresponding to the largest eigenvalue of $\mathbf{J}$ then gives the locally predominant orientation. All three eigenvalues provide a useful first-order descriptor of the local orientation structure, and are rotation invariant. The entries of the Structure Tensor, $J_{pq}$ can be computed by a pixel-wise multiplication ($\circ$) of the partial derivatives $D$ in $p$ and $q$ direction, followed by convolution ($*$) with a smoothing filter $B$ (Jähne, 2005, chapter 13):

$$\mathbf{J} = B * (D_p \circ D_q). \hspace{1cm} (2.6)$$

Again – as during the calculation of the gradient magnitude – all dimensions other than $p$ are smoothed with a Gaussian filter (standard deviation $\sigma_i$, “inner scale”), when computing $D_p$. $B$ is a Gaussian smoothing filter with standard deviation $\sigma_o$ (“outer scale”).

**Hessian matrix.** All possible partial derivatives of order two can be arranged to form the Hesse matrix:

$$H_{pq} = \frac{\partial^2}{\partial x_p \partial x_q}. \hspace{1cm} (2.7)$$

where Gaussian partial derivative filters are used (scale: $\sigma_h$). The three eigenvalues describe the curvature in the coordinate system spanned by the three eigenvectors and are – as all eigenvalues – rotation invariant.
2.3.2 Parallelization

Like many of the following steps, voxel feature calculation benefits from parallelization. Once different parallel execution units have been assigned independent sets of data to work on, there is little need for communication. This allows each unit to make maximal use of available resources without having to wait for others. It also makes the implementation easier.

Choice of parallelization platform. Parallelization can be implemented using one of two main approaches: (1) threads and (2) communicating processes. With usual operating systems, processes are scheduled independently. Each process has its own memory space. Threads are lightweight parallelized execution units contained within one process which share its memory space. Therefore a multi-threaded program can only be executed on one shared memory machine with a single address space. Processes on the other hand can be run on a distributed memory system; their communication from individual computer to individual computer can then be relaid over the network.

The Message Passing Interface (MPI) specifies an API and communication protocol for message-passing between processes, which has become the de facto standard for parallel programs running on computer clusters. For example, it is the only supported parallelization framework on the Heidelberg Linux Cluster System II (Helics II). I have therefore chosen to use MPI for any parallelization efforts. A good introductory book is Bauke and Mertens (2006), which inspired the following implementations.

Master-worker scheme. In all my implementations presented here, a master-worker communication scheme has been employed. One distinguished process is called the master and is responsible for distributing the work to a number of worker processes and collecting any resulting computations. Communication happens only between the master and any of its workers, but never among different workers.

The master divides the data to be processed into a set of blocks. Fig. 2.2 illustrates two possible design choices.

1. In 2.2a, the master process alone has access to the data file (for simplicity, it is assumed that the input data is overwritten with the output data). When a worker signals that it is free, the master reads a block of data from the file, serializes it and sends it over the network. After having performed some time-consuming calculations, the worker signals to its master that it has finished, serializes the result data structure, and waits for an OK to start sending. When the master has time, the communication is initiated. The received block of computed data is saved at the appropriate position in the data matrix by the master process.

2. In another variant (2.2b), no data is serialized or sent over the network via MPI (this happens implicitly by the underlying network file system). Instead, numbers identifying blocks of data are passed between master and worker. All workers have read-only access to the input data file. When their computations are finished, they request the global I/O lock to gain write access to the output file (again, input and output file are identical in the figure). Only one worker process can hold the lock at any time.
2.3 Voxel features

Blockwise-parallel feature computation. The program voxel-features-mpi implements the blockwise-parallel feature computation with the master-worker scheme 1. The Vigra image-processing library (Köthe, 2010) is used for all feature computations. As an image filter summarizes the neighborhood of a voxel, an additional margin around each block must be added so that even for voxels near the edge all necessary data is available. The worker process is sent the enlarged data block, calculates the features and sends the result back to the master process, which disregards the previously added margin and then integrates the remaining block into its proper position in the feature matrix for the complete data volume.

Fig. 2.3 shows an example execution timeline for calculating the features of a 127³ subvolume of e1088 on a laptop with two processor cores. Different colors indicate different features: bilateral filter (red), gradient (yellow), Structure Tensor (blue) and Hessian matrix (green). Execution timelines for e1088 with 9 workers running on a 4 × 4-core machine are shown in fig. 2.4. It is obvious that the parallelization does not scale very well. Worker 9 remains idle most of the time. This happens because the master handles all I/O operations (reading/writing) sequentially, and workers have to wait until the master has processed their I/O requests. Asynchronous I/O operations or master-worker scheme 2 (which necessitates a network file system providing parallel read access!) could be used to speed up the feature calculation. However, with an execution time of 1.5 days for e1088, the speed of voxel-features-mpi is already sufficient.

The time complexity of the feature calculation is \( O(1) \) per voxel.

Running the program. Tab. 2.2 summarizes the usage of voxel-features-mpi. As any MPI program, it needs to be launched through mpiexec with an appropriate number of processors (-np).
Figure 2.3 – Execution timeline of voxel-features-mpi on a 127³ data cube.

Figure 2.4 – Execution timeline of voxel-features-mpi on e1088.

mpiexec -np 3 voxel-features-mpi

INPUT

--hdf5data 'D.h5#sbfsem'  
input raw data volume, shape (n₁,n₂,n₃)

OUTPUT

--hdf5features 'voxel-features-D.h5#voxel-features' 
feature volume \( \chi \), shape (n₁,n₂,n₃,nVF) 
where \( nVF = 2(2 + 3|\Sigma_{ST}| + 3|\Sigma_{HM}|) \)

BLOCK-WISE PROCESSING

--margin 10  
needed block overlap in voxels

--blockSize 75 75 75  
block size (without overlap margin)

FEATURES TO CALCULATE

--gradientScale 1.0  
scale \( \sigma_m \) used by Derivative-of-Gaussian filter

--structureTensorInnerScales 1.0 1.5  
list of scales \( \sigma_i \in \Sigma_{ST} \); EV of Structure Tensor with inner scale \( \sigma_i \), outer scale \( \sigma_o = 2\sigma_i \) will be computed

--hessianScales 1.0 1.5  
list of scales \( \sigma_h \in \Sigma_{HM} \); EV of Hessian matrix with scale \( \sigma_h \) will be computed

--bilateralFilterSpatialScale 1.0  
spatial scale \( \sigma_s \) of bilateral filter

--bilateralFilterIntensityScale 3.0  
intensity scale \( \sigma_v \) of bilateral filter

--bilateralFilterDiffusionSteps 4  
number of iterations of bilateral filter

Table 2.2 – Command line options for the voxel-features-mpi program. The listed default parameters were used in all experiments reported here. From Python use denke.Pipeline.run.voxelFeatures().
2.4 Learning to distinguish intra-cellular from extra-cellular tissue

2.4.1 Random Forests

In supervised statistical learning, the goal is to estimate a prediction function given a training set $S = \{s_1, \ldots, s_n\}$ of training samples. Each sample $s$ consists of a number of features or inputs $x^{(s)} \in X_1 \times X_2 \times \cdots \times X_m = \mathcal{F}$, where $X_i$ is a real interval and one output $c^{(s)}$. In a regression setting the task is function approximation, and $c^{(s)} \in \mathbb{R}$. In a classification setting, the output is a class label $c \in [1, |C|] \subset \mathbb{N}$.

Supervised statistical learning can be used to enhance the contrast of the SBFSEM raw data (other uses follow later in this chapter). As the $e1088$ dataset was prepared with surface staining, cell organelles are not visible. Thus as a first approximation there are only two different classes visible: intra-cellular space (unstained) and extra-cellular space (stained). As image noise is due to random processes, deterministic simple methods like thresholding are bound to fail on such data. Therefore the idea is to calculate a set of features for each voxel, which describes its neighborhood. Using a set of voxels which have been manually annotated as belonging to either intra-cellular or extra-cellular space, the marginal probability distribution $p(\text{class} | \text{features})$ is to be learned.


A tree can be used to define a recursive partition of the feature space. Fig. 2.5 illustrates this with an example. Here the feature space $\mathcal{F}$ of all possible observations (2.5a) is spanned by $X_1$ and $X_2$. A binary decision tree (2.5b) describes the partition

(a) two-dimensional feature space partitioned into six regions

(b) corresponding binary decision tree

Figure 2.5 – Illustration of the idea behind Classification and Regression Trees, inspired by Hastie et al. (2009), chap. 9.
of $\mathcal{F}$ into six regions $R_1, \ldots, R_6$. In classification, each region is assigned a class label $c \in [1, |C|]$.

Beginning at the root of the tree, a decision is made at each node of the tree whether to descend to the left or the right child. In this way, a new – previously unseen – tuple $(x_1, x_2) \in \mathcal{F}$ can be passed down the tree until it reaches a leaf node and is assigned the class label associated with the leaf’s feature space region.

To automatically construct such a decision tree, the algorithm needs to find the optimal splitting variable $X_i$ and split point $\xi_i$ in each step, splitting leaves until some stopping criterion is fulfilled. Unfortunately, constructing an optimal tree is infeasible (Hastie et al., 2009, chap. 9), so a greedy procedure is used instead. The choice of splitting rule at a certain leaf is usually based on an objective function called the Gini index. As a stopping criterion, the minimum number of training samples $n_{\text{min}}$ in the leaf nodes is considered ($n_{\text{min}}$ is set to one if all these samples have the same label). Usually, trees are grown until purity ($n_{\text{min}} = 1$). This leads to deep trees with low bias but high variance.

The idea behind Random Forests is to employ both bagging and random feature selection to decrease the variance and correlation of multiple trees. Many trees (usually 255 in the presented segmentation procedure) are grown, and each sample is passed down each tree. The predicted class label is the majority vote of the prediction of every single tree, and the fraction of trees voting for a particular label can be interpreted as the class probability.

Each tree is constructed in the following way (Hastie et al., 2009, chap. 15):

1. A bootstrap sample $S'$, $|S'| = |S|$ is drawn from the training set $S$ at random with replacement. This technique, called bagging, helps with reducing the variance of the tree ensemble. $S'$ is associated with the root node of a tree.
2. Until the stopping criterion of only $n_{\text{min}}$ bootstrap samples in the leaves is met, repeat:
   i. Select $m_{\text{try}}$ variables from the set of $m$ input variables (features) at random, without replacement. Usually one chooses $m_{\text{try}} = \lfloor \sqrt{m} \rfloor$. This step serves to decrease the correlation among different trees.
   ii. Among those variables, find the best possible splitting variable and split point by optimizing the Gini index.
   iii. Split the leaf node into two child nodes.

All command line programs presented here use the excellent Random Forest implementation in C++ of Nair (2010), which implements the algorithm exactly as described in Breiman (2001). The default settings are left at $m_{\text{try}} = \lfloor \sqrt{m} \rfloor$. 255 trees are grown because this corresponds to the maximum precision of the probability that can be stored in an 8-bit data array.

2.4.2 Ground-truth voxel labels

**Training dataset 0.** For a block of $127^3$ voxels from $e1088$ (sbfsem-e1088-unaligned.h5) manual tracings of intra-cellular space were provided by the Max Planck Institute for Medical Research, Heidelberg. As the tracing was repeated multiple times, there is a confidence measure for each voxel (gray scale colormap in fig. 2.6a), which is used for conversion into binary voxel labels (fig. 2.6b). For training the Random Forest classifier RF$_1$, 20 000 samples per class were drawn without replacement from the set of all labeled voxels.
2.4 Learning to distinguish intra-cellular from extra-cellular tissue

![Image](a) manual tracings for e1088-unaligned  
![Image](b) derived binary labels for e1088-unaligned  
![Image](c) X1 with sparse labels  
![Image](d) X2 with sparse labels

**Figure 2.6** – Two different sets of voxel labels have been used with the *e1088* dataset.

a,b) Labels derived from dense manual tracings provided by the Max Planck Institute for Medical Research, Heidelberg.

c,d) Sparse manual labeling on subvolumes $X_1$ and $X_2$.

**Training dataset 1.** A second training set for *e1088* is only very sparsely populated with labels. Two blocks of $150^3$ voxels were selected, one from the dense inside (offset $o = (799, 799, 799)$, sbfsem-X1.h5), the other from the border of the IPL ($o = (99, 699, 999)$, sbfsem-X2.h5), paying tribute to the variability in the dataset. Björn Andres has used an incremental labeling strategy in which most obvious labels were placed first and care was taken to precisely center extra-cellular labels on the visible boundary and spacing individual labels at least a few voxels apart. After labeling roughly 500 voxels per class, these initial labels were used to train RF$_1$ and to predict on both volumes. Where the predicted probability maps needed improvement, additional labels were placed. This procedure was repeated multiple times until approx. 3200 voxels were labeled. The final labelings are voxel-labels-X1-andres-7.h5 and voxel-labels-...
train-boundary-classifier

**Input** (pass switches repeatedly for multiple training volumes)

- `--hdf5features` voxel-features-D.h5#features
- `--hdf5labels` voxel-labels.h5#labels

**Probability-map** \( \phi \) (probability-map-mpi, tab. 2.4)

**Ground-truth face labels** (unlabeled: 0, intra-cellular: 1, extra-cellular: 2)

**Output**

- `--randomForestHDF5File` RF1.h5

**Parameters**

- `--trees` 255 number of trees to be grown
- `--numSamplesPerClass` -1 number of samples per class, drawn at random without replacement, used for a balanced training set. -1 means all samples.

<table>
<thead>
<tr>
<th>Block-wise processing</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--blockSize</code> 75 75 75 block size</td>
</tr>
</tbody>
</table>

### Table 2.3 – Command line options for the train-boundary-classifier program.

mpiexec -np 3 probability-map-mpi

**Input**

- `--hdf5features` voxel-features-D.h5#voxel-features
- `--randomForestFile` rf1.h5

**Output**

- `--hdf5probabilityMap` probability-map-D.h5#probability-map

**Volume** \( \phi \) (unsigned char) with extra-cellular space probabilities \( \in [0, 255] \)

### Table 2.4 – Command line options for the probability-map-mpi program. From Python use denkc.Pipeline.run.classifyVoxels().

X2-andres-5.h5. Fig. 2.6c and 2.6d show representative slices through these two training volumes with given labels overlaid.

### 2.4.3 Training RF

The program train-boundary-classifier is used to train the random forest RF\(_1\) given voxel features and voxel labels. Multiple training datasets (like \(X_1, X_2\)) can be passed to the program. Its usage is summarized in tab. 2.3. The runtime is in the order of minutes.

### 2.4.4 Block-wise parallel prediction

The program probability-map-mpi (see tab. 2.4) implements block-wise parallel prediction using the master-worker scheme 1 from page 12. The output is the probability map \( \phi : [0, n_1] \times [0, n_2] \times [0, n_3] \rightarrow [0, 2^8] \) where \( \phi(x) \) gives the probability of voxel \( x \) being extra-cellular. Even when run on all cores of a 4 \( \times \) 4-core machine, there were no significant idle times. The total runtime depends on the used training dataset. With TD\(_0\), prediction took 4.5 days; with TD\(_1\) only 0.5 days. Time complexity is \( O(1)/\text{voxel.} \)
2.5 Marker-based watershed segmentation

2.5.1 The watershed transform

Figure 2.7 – Thresholding the voxel boundary probability (a) produces the seeds (b) that are used in the seeded watershed algorithm to produce an over-segmentation (c).

The watershed transform (good overview in Roerdink and Meijster, 2000) is a segmentation method from the field of gray scale mathematical morphology.

In image segmentation, the task is to partition the input image into a set of disjoint regions so that each region encompasses only similar values of image features such as gray value, texture or (binary) class probability.

The watershed algorithm has a very intuitive notion from which the name is derived: Imagine the (two-dimensional) image as a three-dimensional relief, with the gray scale values specifying the elevation. Let it rain onto this landscape and water begins to fill in deep valleys (the local minima of the image) first, forming catchment basins. While the water level rises, it may happen that the water in two basins flows together. This is prevented by building a dam called a watershed barrier where the water would flow together. The procedure stops when the highest peak is immersed in water. As a result, the image has been partitioned into a set of disjoint regions, the catchment basins, separated by watershed lines. Different variants of the watershed transform exist for the discrete case, representing watersheds either with image pixels or as intra-pixel edges. In the latter case, the watersheds are only implicitly defined.

What has been described above for 2D images can be generalized to 3D volume images with watershed lines becoming watershed surfaces (or short: faces) between 3D catchment basins called supervoxels or segments.

Each local minimum of the image eventually grows into one segment, thus finally the image is partitioned into exactly as many regions as there were local minima. Obviously local minima are abundant in any physically acquired image due to noise, resulting in a severe over-segmentation. This problem can be alleviated by using marker-based variants of the watershed transform. Essentially one modifies the input image such that only significant local minima appear in the image prior to running the algorithm. In practice, seeds are chosen such that the resulting segmentation will still be an over-segmentation (discussion in sec. 2.5.2). To make segments correspond to biologically meaningful objects (neurons), incorrect watershed boundaries have to be removed in subsequent postprocessing steps.
2 Segmentation of volume electron microscopy images for connectomics

**segment**

<table>
<thead>
<tr>
<th>INPUT</th>
<th>probability-map φ (output of probability-map-mpi, tab. 2.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--hdf5map 'probability-map-D.h5#probability-map'</td>
<td>probability-map φ (output of probability-map-mpi, tab. 2.4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>supervoxels σ (uint32) with dense indices in [1, Σ + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--hdf5seg 'seg-D.h5#seg'</td>
<td>supervoxels σ (uint32) with dense indices in [1, Σ + 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETERS</th>
<th>maximum number of trees t_{max} that can vote for a voxel to be extra-cellular space for it to be considered part of a seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>--seedPmapLimit 0</td>
<td>maximum number of trees t_{max} that can vote for a voxel to be extra-cellular space for it to be considered part of a seed</td>
</tr>
<tr>
<td>--minimumSeedSize 3</td>
<td>remove seeds smaller or equal to the given size s_{min}; a dense relabeling is performed</td>
</tr>
</tbody>
</table>

**Table 2.5** – Command line options for the segment program.

From Python use `denke.Pipeline.run.segment(s_{min}, t_{max});`.

The *turbo-watershed* (TWS) seeded region-growing marker-based watershed algorithm is used (implementation by Björn Andres, see the supplementary material of Andres et al., 2010d). The *elevation map* φ : G → [0, 2^8) maps from the voxel grid G into the domain of all 8-bit unsigned integers, while the *seed map* ˜σ : G → [0, Σ + 1) constitutes a dense connected component labeling of all Σ significant seeds with 0 denoting background. For segmentation of $e1088$, only 32-bit unsigned integers are large enough to represent all seeds and resulting segments. The output is the *segment label map* σ which replaces the input array ˜σ. Amongst all possible paths $p \rightsquigarrow q$ between two points, the overall lowest path still has a maximal elevation called the *min-max-distance*

$$d(p, q) = \min_{p \rightsquigarrow q} \left[ \max_{x \in p \rightsquigarrow q} \phi(x) \right].$$ (2.8)

Let $\tilde{K}_i^{(3)} = \{ p \in ˜σ | ˜σ(p) = i \}$ denote the set of voxels forming the connected-component of seed with index $i$. Define the distance

$$d(p, \tilde{K}_i^{(3)}) = \min_{q \in \tilde{K}_i^{(3)}} [d(p, q)].$$ (2.9)

The *watershed transform* σ = TWS(φ, ˜σ) is computed as

$$\sigma(x) = \arg \min_{i \in [1, Σ+1]} \left[ d(x, \tilde{K}_i^{(3)}) \right].$$ (2.10)

The six-neighborhood is used. Internally, the usage of 256 queues, one each for every possible gray value, limits the complexity to $O(1)$ per voxel.

**2.5.2 Choice of seeds and place in the segmentation procedure**

When choosing initial seeds, the following tradeoff must be considered:

- *Too many seeds* will slow down subsequent geometry extraction, higher-order feature calculation and energy minimization and increase the memory requirements for intermediate results such as segment geometry or Graphical Model representations.
As segments become very small, statistics over 1-, 2- or 3-components (see sec. 2.6) become less meaningful. This limits their discriminative power in correct excessive face removal.

- **Too few seeds** can lead to initial under-segmentation, where the seeds of two neighboring neurons are connected or no seed is placed into a small neuron at all. The resulting merging error will be irreversible.

Two parameters govern this tradeoff: minimum seed size and seed threshold.

- A voxel is still considered as part of a seed if the number of trees that voted against the intra-cellular label is less than or equal to $t_{\text{max}}$. This is a simple threshold on the probability map.
- After the above thresholding operation, very small seeds are eliminated (their vote is not supported by the seeds’ surrounding; for example an isolated seed pixel is most probably due to noise) by imposing a minimum size limit $s_{\text{min}}$ (in voxels). Surviving seeds are densely relabeled, as required by the TWS algorithm, p. 20.

The segment command-line program calls the TWS algorithm with seeds computed using parameters $t_{\text{max}}$ and $s_{\text{min}}$. Its usage is summarized in tab. 2.5.

### 2.6 Geometry extraction using CGP

#### 2.6.1 Theory

The volume labeling $\sigma$ as produced by the watershed algorithm (sec. 2.5.1) implicitly defines the geometry and topology of the segmentation.

- **Segments** $S \in \mathcal{S}$ are connected components of voxels with the same label.
- **Faces** $F \in \mathcal{F}$ are connected components of voxel faces separating the same two segments.
- Several faces can intersect to form a **curve** $C \in \mathcal{C}$, a connected set of voxel edges.
- The intersection of several curves yields a **point** $P \in \mathcal{P}$, a voxel corner.

![Illustration of the topological grid.](image)

*Left:* In 2D the shown connected components of pixels are separated by three connected components of pixel edges. Only the black pixel corner is active.
*Right:* In 3D the topological grid explicitly represent voxels, voxel faces, voxel edges and voxel corners.
Algorithm 1: normalOrientation

Input: topological 2-cell \( t \)
Result: index \( k \) so that \( n = 1 \) if \( j = k \), else \( 0 \mid j \in [0,3) \) \( \perp t \)

for \( i \in [0,3) \) do
  if \( \text{mod}(t_i, 2) \neq 0 \) then
    return \( i \)

Figure 2.9 – Definition of 0-, 1-, 2- and 3-cells.

It would be advantageous to have a data structure which can be queried about topographic relations such as: “Which segments are separated by a given face?” or “Which are the faces that a given curve bounds?”. It is also desirable to have constant time access to geometry information: “Return an explicit surface representation (such as a mesh) for the face with the given label”. One application is feature extraction for the eventual removal of over-segmentation (sec. 2.7 and 2.8), or for synapse detection (chap. 3); another visualization (chap. 4).

In Andres et al. (2010b), a new block-wise algorithm CGP (Cartesian Grid Partitioning) for extracting the above topology and geometry information from large volume labelings has been proposed. After reviewing the notation and basic concepts, the parallelization, visualization and new convenient classes to query the resulting data structures stored on disk are discussed.

Let the segment label map \( \sigma : G \rightarrow \mathbb{N} \) be defined on the voxel grid \( G \) with extent \( G = [0,n_1) \times [0,n_2) \times [0,n_3) \subset \mathbb{N}_0^3 \). To be able to explicitly represent voxel faces, voxel edges and voxel corners, a larger grid is needed, called the topological grid \( T = [0,2n_1 - 1) \times [0,2n_2 - 1) \times [0,2n_3 - 1) \subset \mathbb{N}_0^3 \). The topological label map \( \tau : T \rightarrow \mathbb{N}_0 \) is the representation of \( \sigma \) in this new space.

The relation between a coordinate in \( \sigma \) and one in \( \tau \) is simple. In the following, \( x = (x_1,x_2,x_3) \) denotes a coordinate in \( G \) and \( t = (t_1,t_2,t_3) \) a position in \( T \) (named cartesian coordinate and topological coordinate respectively).

**Topological 3-cell.** Each voxel \( x \in G \) has a corresponding voxel in \( T \).

\[
\forall x \in G : \sigma(x) = \tau(2x).
\] (2.11)

Clearly, all \( t \in T \) that represent voxels from \( \sigma \) have entries \( t_1,t_2,t_3 \) that are all even. The voxel \( t \) is called a 3-cell.
2.6 Geometry extraction using CGP

Algorithm 2: corners

**Input:** topological 2-cell $t$

**Result:** set $C$ containing the cartesian coordinates of the four corners of $t$

$n \leftarrow \text{normalOrientation}(t)$
$a \leftarrow -1, b \leftarrow -1$

for $i \in [0, 3)$ do

if $i \neq n$ then

if $a = -1$ then

$a \leftarrow i$
else

$b \leftarrow i$

end if

end if

$C \leftarrow \emptyset$

for $i \in [1, 4]$ do

if $n = 0 \lor n = 2$ then

$k \leftarrow \text{mod}([i/2], 2)$
$l \leftarrow 0$ if $i \leq 2$ else 0

else

$l \leftarrow \text{mod}([i/2], 2)$

$k \leftarrow 0$ if $i \leq 2$ else 0

end if

$c \leftarrow (0, 0, 0)$
$c_n \leftarrow (t_n + 1)/2 + k$
$c_a \leftarrow t_a/2 + k$
$c_b \leftarrow t_b/2 + k$

$C \leftarrow C \cup c$

end for

return $C$

Topological 2-cell. A voxel face $f$ shared by voxels $x^{(1)}, x^{(2)} \in G$ (fig. 2.9a) has a surface normal $f_n \in \{\hat{x} = (1, 0, 0), \hat{y} = (0, 1, 0), \hat{z} = (0, 0, 1)\}$, which is represented by the normal orientation index $OI(f_n)$ of 0, 1 or 2 (computed by alg. 1). Then $f$ has the coordinate $t = (t_1, t_2, t_3)$ with

\[
    t_i = \begin{cases} 
        x_i^{(1)} + x_i^{(2)} & \text{if } OI(f_n) = i, \\
        2x_i^{(1)} & \text{else}.
    \end{cases}
\]

(2.12)

The 2-cell $t$ has exactly two even coordinate entries. For a visual representation, the cartesian coordinates of the four corners must be computed, see alg. 2.

Topological 1-cell. A voxel edge $e$ shared by voxels $x^{(1)}, x^{(2)} \in G$ (fig. 2.9b) has an orientation again described by an orientation index $OI(e)$. In $T$, this edge has the coordinate $t = (t_1, t_2, t_3)$ with

\[
    t_i = \begin{cases} 
        2x_i^{(1)} = 2x_i^{(2)} & \text{if } OI(e) = i, \\
        x_i^{(1)} + x_i^{(2)} & \text{else}.
    \end{cases}
\]

(2.13)

The 1-cell $t$ has one even coordinate. To represent it visually, the two end points of the line segment must be computed (alg. 3).
Algorithm 3: endpoints

**Input:** topological 1-cell \( t \)

**Result:** cartesian end points \((c_1, c_2)\), of the voxel edge which represents \( t \)

\[
\begin{align*}
c_1 &= (t_i/2 + 1 \mid i \in [0,3)) \\
c_2 &= (t_i/2 + 1 - \text{mod}(t_i + 1, 2) \mid i \in [0,3))
\end{align*}
\]

return \((c_1, c_2)\)

---

**Figure 2.10** – Definition of \( \Gamma \)-neighborhood for topological cells.

**Topological 0-cell.** A voxel corner shared by voxels \( x^{(1)}, x^{(2)} \in G \) (fig. 2.9c) corresponds to the coordinate \( t = (t_1, t_2, t_3) \in T \) with

\[
t_i = x^{(1)}_i + x^{(2)}_i
\]

and thus has zero even coordinate entries. For visualization, the cartesian coordinate is \((t + (1, 1, 1))/2\).

**Topological cell order.** Given any \( t \in T \), the function

\[
\text{ord}(t) = 3 - \sum_{i=1}^{3} \text{mod}(t_i, 2)
\]

calculates the order of the cell.

**\( \Gamma \)-neighborhood.** Given a \( j \)-cell \( t \in T \), the \( \Gamma \)-neighborhood is the set of all voxels satisfying

\[
\Gamma(t) = \{ u \in T \mid u, t \text{ are 6-connected } \land \text{ord}(u) = \text{ord}(t) + 1 \}.
\]

A 0-cell has six neighboring 1-cells (fig. 2.10a) which each has four neighboring 2-cells (fig. 2.10b) which each have two neighboring voxels (fig. 2.10a). Thus \( |\Gamma(t)| = 6 - 2j \).

**Connectivity relation.** \( t_1, t_2 \in T \) are directly connected (\( \leftrightarrow \)) if and only if there exists a \( u \in T \) with \( t_1, t_2 \in \Gamma(u) \). Thus two voxels are connected via a voxel face, two faces via a voxel edge and two edges via a voxel corner. 0-cells cannot be connected, \( \Gamma(0\text{-cell}) = \emptyset \).
2.6 Geometry extraction using CGP

**Cell activity.** As the highest $j$-cell, a 3-cell is always active. The activity of 2-, 1- and 0-cells depends upon their $\Gamma$-neighborhood:

Let $t$ be a $j$-cell with $0 \leq j < 3$ and $\Gamma(t) = \{u_1, \ldots, u_{6-2j}\}$. Then define $\tau_k$ as the $\leftrightarrow$-connected component of $(j+1)$-cells encompassing the active cell $u_i$ or as $\emptyset$ when the cell is inactive. Furthermore, let

$$\theta = \{\tau_k | \tau_k \text{ occurs exactly once in } \{\tau_1, \ldots, \tau_{6-2j}\}\}.$$  \hfill (2.17)

Then $t$ is active $\iff \theta(t) \neq \emptyset$ and the connected components $\theta$ are bounded by $t$. In $\tau$, inactive cells are encoded with value 0.

**Topological $j$-components.** A maximal set $K_q^{(j)} \subseteq T$ with the following properties

$$\forall t \in K_q^{(j)} : \text{ord}(t) = j \land j \text{ is active}$$ \hfill (2.18)

$$\exists \Theta : \Theta(t) = \theta \forall t \in K_q^{(j)}$$ \hfill (2.19)

$$t_1, t_2 \in K_q^{(j)} \iff \text{there is a } \leftrightarrow\text{-path connecting both cells within } K_q^{(j)}$$ \hfill (2.20)

is called a $j$-component. All $j$-cells making up the $j$-component are assigned the label $q$ in the topological label map $\tau$. Specifically, these entities describe the following:

- **0-components** $\Leftrightarrow$ 0-cell, points $P \in \mathcal{P}$,
- **1-components** curves $C \in \mathcal{C}$,
- **2-components** faces $F \in \mathcal{F}$ and
- **3-components** segments $S \in \mathcal{S}$.

The definition of the $\Gamma$-neighborhood is extended for $j$-components to comprise all bounded $(j+1)$-components:

$$\Gamma(K_q^{(j)}) = \theta(t \in K_q^{(j)}).$$ \hfill (2.21)

The $\Gamma$-neighborhood is stored for each $j$-component, allowing for fast querying of topographic relations. Such a representation is called a cellular complex.

**cgpx algorithm.** In Andres et al. (2010b), a parallelized algorithm for the consistent construction of 0-, 1-, 2- and 3-components in $\tau$ is presented.

For the $e1088$ dataset, $\sigma$ is stored with 32-bit precision. As $\tau$ is about $2^3$ times larger, approx. 256GB of memory are needed. It is important therefore to allow for block-wise processing, keeping only parts of $\tau$ in RAM. Additionally this has the advantage that the algorithm can be sped up by parallelization. There are complications, however. After labeling the $j$-components of all individual blocks, the given labels have to be reconciled. Therefore, the blocks of $\sigma$ are chosen so that they have an overlap of 1 voxel. The 1-sets and 2-sets contained in the overlap region belong to two blocks and are labeled differently. Their labels are merged using a union-find data structure. One subtlety of the algorithm is that this is only correct for the merging of 2-component labels. To be entirely correct, all 0-cells have to be reconsidered in a last step, possibly merging bounded 1-components or changing their activity status.
Algorithm 4: mesh

**Input**: 2-component $C = \{c_1, \ldots, c_n\}$ as a list of topological 2-cells

**Result**: triangle mesh $M = (V, F)$ representing $C$

```plaintext
if $n = 0$ then
  return $M \leftarrow (\emptyset, \emptyset)$
K $\leftarrow \emptyset$, $V \leftarrow \emptyset$, $F \leftarrow \emptyset$
i $\leftarrow 0$
for $c \in C$ do
  $d \leftarrow$ corners($c$)
  for $j \in [0, 4)$ do
    $K \leftarrow K \cup (d_j, i)$
i $\leftarrow i + 1$
  $F \leftarrow F \cup (4i, 4i + 1, 4i + 2) \cup (4i, 4i + 2, 4i + 3)$

sort $K$ by a lexicographic comparison of the triplets $K_{i0}$, $i \in [0, |K|)$

$U \leftarrow$ UnionFind($|K|$)
for $j \in [1, |K|)$ do
  if $K_{j0} = K_{(j-1),0}$ then
    $U \leftarrow$ Merge($U$, $K_{j1}, K_{j2}$)

$R \leftarrow$ DenseRelabeling($U$)
for $f \in F$ do
  for $j \in [0, 3)$ do
    $f_j \leftarrow R[j]$
for $k \in K$ do
  $k_1 \leftarrow R[k_1]$

$n \leftarrow$ NumberOfSets($U$)
resize $V$ to size $n$
$V[0] \leftarrow K_{00}$
for $j \in [1, n)$ do
  if $K_{j1} = K_{(j-1),1}$ then
    $V[K_{j1}] \leftarrow K_{j0}$
return $M = (V, F)$
```

2.6.2 From j-components to polygons

Alg. 4 shows how a 2-component, as represented by its list of topological coordinates, is transformed into a cartesian triangle mesh, represented by vertices (positions in 3D space) and faces (triples of indices into the vertices array) defining triangles. It is implemented as `cgp::TwoSet::toMesh`. By using a union-find data structure, vertices are shared by as many faces as possible. This reduces the size of the mesh data structure and ensures connectivity.

Alg. 5 can be used to convert a 1-component into a polygon line, represented by vertices $V$ and edges $E$, pairs of indices into the vertex array. Again, for efficiency, each vertex position is made to appear only once in $V$. 
Algorithm 5: pathsort

Input: 2-component \( C^{(u)} = \{c_1, \ldots, c_n\} \) as a list of topological 2-cells 

Result: 2-component \( C^{(s)} \) such that the elements, in order, form a connected path when represented in cartesian space

\[
C^{(s)} \leftarrow \left\{ C_0^{(u)} \right\} \\
k \leftarrow 1 \\
\text{while } |C^{(s)}| \neq |C^{(u)}| \text{ do} \\
\quad p \leftarrow \text{false} \\
\quad \text{for } i \in [k, |C^{(u)}|) \text{ do} \\
\quad \quad \text{if isConnectedTo}(C_i^{(u)}, C_0^{(s)}) \text{ then} \\
\quad \quad \quad C^{(s)} \leftarrow \left\{ C_i^{(u)}, C^{(s)} \right\}, \ p \leftarrow \text{true} \\
\quad \quad \quad \text{swap}(C_i^{(u)}, C_k^{(u)}) \\
\quad \quad \quad k \leftarrow k + 1 \\
\quad \quad \text{else if isConnectedTo}(C_i^{(u)}, C_{|C^{(s)}|-1}^{(s)}) \text{ then} \\
\quad \quad \quad C^{(s)} \leftarrow \left\{ C^{(s)}, C_i^{(u)} \right\}, \ p \leftarrow \text{true} \\
\quad \quad \quad \text{swap}(C_i^{(u)}, C_k^{(u)}) \\
\quad \quad \quad k \leftarrow k + 1 \\
\quad \quad \text{if } \neg p \text{ then} \\
\quad \quad \quad \text{Precondition that } C^{(u)} \text{ is a 2-component does not hold.} \\
\quad \text{return } C^{(s)} \\
\]

2.6.3 Classes for convenient access to extracted geometry and topology

The command line tools \texttt{cgpx} and \texttt{cgpr} (initial implementation: Björn Andres) produce the topological grid representation \( \tau \) of the input volume labeling \( \sigma \) (file: \texttt{topology-D.h5}) and for each \( j \)-component the list of its constituent \( j \)-cells (file: \texttt{geometry-D.h5}). Working with these HDF5 files is not straightforward however. The user has to deal with the facts that the topological grid is stored block-wise (including overlap), that the label-reconciliation has to be performed manually by using a separate conversion table and that \( j \)-components may be split into multiple parts if they cross block boundaries. For performance reasons, these parts are stored in a number of binning groups.

As a high-level interface, I have written the two classes \texttt{cgp::hdf5::TopologyReader} and \texttt{cgp::hdf5::GeometryReader} which take the respective HDF5 file as input and offer convenience functions which hide most complexities from the user. These classes have also been wrapped with \texttt{boost::python}\(^\text{\footnotemark} \) for easy scripting.

Additionally, a class hierarchy represents the concept of \( j \)-cells with various topological point classes, while \( j \)-components are represented as topological point sets (an alternative name for \( j \)-component is \( j \)-set). To differentiate between coordinates in \( T \) and \( G \), a \texttt{cgp::CartesianCoordinate} is introduced.

\footnotetext{\url{http://www.boost.org/doc/libs/release/libs/python/doc}}
label_type oneSetLabel = 42;

//get adjacency information
marray::vector<label_type> bds, bby, nbs;
GeometryReader r("geometry.h5");
  r.bounds(1,onesetlabel, bds);
  r.boundedBy(1,onesetlabel, bby);
  r.adjacent (1, onesetlabel , nbs);

//plot
w = new Window("find/uni2423neighbors/uni24231");
  w->plot(r.oneSet(oneSetLabel).toVtkTube(0.3), 1.0,0,0/
    *color*/ , 1.0/
    *alpha*/ ;
  for(size_t i=0; i<bds.size(); ++i) {
    w->plot(r.twoSet(bds(i)).toVtkPolyData(), 0,0,1, 0.5);
  }
  for(size_t i=0; i<bby.size(); ++i) {
    w->plot(r.zeroSet(bby(i)).toVtkSphere(1.0), 0,0,0, 1.0);
  }
  for(size_t i=0; i<nbs.size(); ++i) {
    w->plot(r.oneSet(nbs(i)).toVtkTube(0.3), 0,1,0, 1.0);
  }

Figure 2.11 – Finding and visualizing the neighbors of a given 1-set with cgp.

TopologyReader.  cgpx writes a topology-D.h5 file. The input volume labeling \( \sigma \) had a total extent of \( S = \{n_0,n_1,n_2\} \) as recorded in the entry segmentation-shape.

The entry block-shape records the extent of blocks \( B = \{B_0,B_1,B_2\} \) in \( \sigma \) that were processed (without accounting for overlap). Thus there are \( N^{(B)} = \{[S_0/B_0], [S_1/B_1], [S_2/B_2]\} \) blocks in each direction. In \( \tau \), these blocks have extents between \( 2B_i - 1 \) and \( 2B_i + 1 \); they are stored in groups blocks/k/topological-grid, where \( k = b_0 + b_1N_0^{(B)} + b_2N_1^{(B)}N_2^{(B)} \) (\( b \) denotes the blocks’ coordinate).

The labels given to 1-components and 2-components in a block are not final. The top-level datasets relabeling-1 and relabeling-2 (\( r_1 \) and \( r_2 \) in the following) and the dataset blocks/k/label-offsets (lo) must be used for relabeling a \( j \)-cell \( t \):

\[
\tau^{new}(t) = r_j \left( \tau^{old}(t) \right) + lo(j). \tag{2.22}
\]

Finally, when stitching blocks together, care must be taken to account for possible overlap.

TopologyReader’s constructor takes the topology-D.h5 filename as only input. Then a block \( \tau_B \subseteq \tau \) can be loaded by specifying its offset \( o \) and extent \( s \) in terms of the volume segmentation \( \sigma \):

\[
\text{loadBlock}(o, s, \tau_B).
\]

In Python, cgp.TopologyReader.loadBlock returns a Numpy\(^3\) array containing \( \tau_B \) instead.

\(^3\)http://numpy.org
2.6 Geometry extraction using CGP

using vtkMath::Random;
label_type twoSetLabel = 42;
vtkPolyData ∗p = 0;

// get adjacency information
marray::Vector<label_type> neighbors;
GeometryReader r("geometry.h5");
r.adjacent(2, twoSetLabel, neighbors);

// plot
p = r.twoSet(twoSetLabel).toVtkPolyData();
w->plot(p, 1.0, 0.0, 1.0);
for(int i = 0; i < neighbors.size(); ++i){
p = r.twoSet(neighbors(i)).toVtkPolyData();
w->plot(p, Random(0,1), Random(0,1), Random(0,1), 0.5 /α∗/α);
}

Figure 2.12 – Finding and visualizing the neighbors of a given 2-set with cgp.

GeometryReader. cgp writes a geometry-D.h5 file allowing constant time access to the lists of topological coordinates making up each 1-, 2- and 3-component. max-labels, $l^\text{max}$, stores the number of consecutive labels given to $j$-components; their labels $q$ are always in the range $[1, l^\text{max}_j]$.

$j$-components are sorted into bins for performance reasons; this way, too many entries in one hierarchy level are avoided. The maximum number of bins is available as number-of-bins, $N_{\text{bin}}$. Additionally, the $j$-components may be broken up into multiple parts, how many is recorded in the entries parts-counters-$j$, $p_{c(j)}$. Then, given an order $j$ and a label $q \in [1, l^\text{max}_j]$, the list of topological coordinates $K_q^{(j)}$ can be accessed with

$$K_q^{(j)} = \{"j\text{-sets/bin-b/q-p}" | p \in [0, p_{c(j-1)}] \land b = \text{mod}(q, N_{\text{bin}})\}. \tag{2.23}$$

Finally, the geometry-D.h5 file stores the $\Gamma$-neighbors for each $j$-component as the matrices neighborhood-$j$, $nb^{(j)}$. They have extents $[0, 6 - 2j] \times [0, l^\text{max}_j]$ and each row gives the $\Gamma$-neighborhood of a $j$-component:

$$\Gamma(K_q^{(j)}) = \left\{K_{q'}^{(j+1)} \mid q' \in nb^{(j)}_{q-1} \land q' \neq 0\right\}. \tag{2.24}$$

GeometryReader’s constructor takes the geometry-D.h5 filename as a required argument. An optional EnableFlag specifies which features should be enabled. By default, all functionality is enabled, at the cost of memory overhead and computation time on constructing the reader object.

Given a $j$-component $K_q^{(j)}$, the function topologicalPointSet($j$, $q$) returns the set of topological coordinates $\{t \in T | \text{ord}(t) = j \land \tau(t) = q\}$ as a generic container object cgp::TopologicalPointSet. Alternatively, the functions zeroSet, oneSet, twoSet and threeSet return specialized topological coordinate lists, namely ZeroSet, OneSet, TwoSet and ThreeSet.
Algorithm 6: isConnectedTo

Input: topological 1-cell \(t^{(1)}, t^{(2)}\)

Result: whether \(t^{(1)}, t^{(2)}\), as represented by the voxel edges they describe, are connected via a voxel corner

\[
f \leftarrow (0, 0, 0)
q \leftarrow 0
e \leftarrow (-1, -1)
\]

for \(i \in [0, 3)\) do
  \[
d \leftarrow |t_i^{(1)} - t_i^{(2)}|
  \]
  if \(d < 3\) then \(f_d \leftarrow f_d + 1\)
  if \(t_i^{(1)} = t_i^{(2)}\) then \(q \leftarrow q + 1\)
  if \(\text{mod}(t_i^{(1)}, 2) = 0\) then \(e_0 \leftarrow i\)
  if \(\text{mod}(t_i^{(2)}, 2) = 0\) then \(e_1 \leftarrow i\)

return \((f_0 = 1 \land f_1 = 2 \land q = 1) \lor (f_2 = 1 \land q = 2 \land |e_0 - e_1|)\)

For querying topographic relations, \(\text{GeometryReader}\) offers the following interface:

- The function \(\text{bounds}(j, q, \Gamma(K^{(j)}))\) finds the \(\Gamma\)-neighbors of a given \(j\)-component
  \[
  \text{bounds}(K^{(j)}) = \Gamma(K^{(j)}).
  \]  
  \(2.25\)

- \(\text{boundedBy}\) computes the inverse of the \(\text{bounds}\)-function: for a given \(j\)-component, it returns all \((j - 1)\)-components that bound it:
  \[
  \text{boundedBy}(K^{(j)}) = \{ K^{(j-1)} \mid K^{(j)} \in \Gamma \left( K^{(j-1)} \right) \}.
  \]  
  \(2.26\)

The inverse function is constructed on instantiation. Because this takes a few seconds for the \(e1088\) dataset and consumes some 100MB of memory, this functionality can be disabled using the \(\text{EnableFlag}\).

- \(\text{adjacent}\) finds all neighbors of a given \(j\)-component that have the same order, that is
  \[
  \text{adjacent}(K^{(j)}) = \{ K^{(j)} \mid K^{(j)} \in \Gamma^{-1} \left( K^{(j-1)} \right) \}.
  \]  
  \(2.27\)

Fig. 2.14, 2.12 and 2.11 demonstrate the API of the \(\text{CGP}\) library for visualization purposes and the usage of the \(\text{bounds}, \text{boundedBy},\) and \(\text{adjacent}\) functions.

From Python this class is available as \(\text{cgp.GeometryReader}\). Instead of return-by-reference, all functions return a Python object. Otherwise the API is unchanged.

Classes representing \(j\)-cells. Any three-dimensional coordinate \(x \in C^3\) can be represented by the template class \(\text{cgp::Point}<C>\).

A topological coordinate \(t \in T\) is represented by the derived \(\text{cgp::TopologicalPoint}<C>\) class. A good choice is \(C = \text{short}\), with which the extent of the topological grid of the \(e1088\) dataset can be represented.

The classes \(\text{cgp::TopologicalPoint0}, \text{cgp::TopologicalPoint1}, \text{cgp::TopologicalPoint2}\) and \(\text{cgp::TopologicalPoint3}\) are further specializations and represent \(j\)-cells \(t\). They were created for the two purposes of (1) explicit type-checking for \(\text{ord}(t) = j\) and (2) to provide \(j\)-specific functionality, such as conversion to a representation in cartesian space. Thus

- A 0-cell’s representation as a point in \(\mathbb{R}^3\) is available via \(\text{toCartesian}\).
- A 1-cell’s representation as a line segment in \(\mathbb{R}^3\) is available via \(\text{endpoints}\), see alg. 3.
Algorithm 7: isEdgeConnectedTo

Input: topological 2-cell \(t^{(1)}, t^{(2)}\)

Result: whether \(t^{(1)}, t^{(2)}\), as represented by the voxel faces they describe, are connected via a voxel edge

\[
\begin{align*}
\text{Is} & \leftarrow (\emptyset, \emptyset, \emptyset) \\
\text{for } i & \in [0, 3] \text{ do} \\
\quad \text{Is} & \leftarrow |t_i^{(1)} - t_i^{(2)}|
\end{align*}
\]

\[
\begin{align*}
\text{s} & \leftarrow d_0 + d_1 + d_2 \\
\text{n} & \leftarrow \text{normalOrientation}(t^{(1)})
\end{align*}
\]

if \(\text{s} = 2\) then

\[
\begin{align*}
\text{if } d_n = 1 & \text{ then} \\
\text{return } d_{\text{mod}(n+1,3)} = 1 \lor d_{\text{mod}(n+2,3)} = 1
\end{align*}
\]

else if \(d_n = 0 \land (d_{\text{mod}(n+1,3)} = 0 \lor d_{\text{mod}(n+2,3)} = 0)\) then

\[
\text{return true}
\]

return false

whether the 1-cell is connected to another can be determined via \isConnectedTo, see alg. 6.

• A 2-cell’s representation as a square in \(\mathbb{R}^3\) is available via \cartesianCorners, see alg. 2; whether the 2-cell is connected to another can be determined via \isEdgeConnectedTo, alg. 7. In some applications, the sign of the cell’s normal vector matters (see sec. 3.1.3). All 2-cells of a 2-component bound the same two 3-components with labels \(l_1^{(3)}\) and \(l_2^{(3)}\). An arbitrary, but globally consistent criterion for the sign is \(+1\) if \(l_1^{(3)} < l_2^{(3)}\) and else \(-1\). The method \normalSign returns this sign.

• To determine if a 3-cell is connected via a face to another 3-cell, the function \isConnectedTo is provided.

See fig. 2.13 for a class diagram of CGP’s coordinate classes.

Classes representing \(j\)-components. Sets of coordinates are represented in CGP by classes derived from the generic \PointSet class, a convenience wrapper of the container \(\text{std::vector<cpp::Point<\text{C}}>\).

Again – to avoid confusion – the difference between coordinates in \(T\) and \(G\) is made explicit by the classes \TopologicalPointSet and \CartesianPointSet. \(j\)-components can either be represented by the more generic \TopologicalPointSet class or by specialized classes \ZeroSet, \OneSet, \TwoSet and \ThreeSet, which offer convenient \(j\)-specific visualization functions when VTK (Visualization Tool Kit\(^4\)) is available.

• A 0-component can be visualized as a sphere at the position of the voxel corner it represents via \toVtkSphere.

• A 1-component can be visualized as the connected set of voxel edges it represents: \toPolyline returns the vertices and edges for the connected line, while \toVtkPolyData returns a VTK polygon data structure. \toVtkRibbon and \toVtkTube allow for easier visibility due to larger line widths.

\(^4\text{http://www.vtk.org}\)
Figure 2.13 – Class Diagram for the coordinate classes in the CGP library.
2.6 Geometry extraction using CGP

- A 2-component can be visualized as a mesh: *toBeMesh* returns a simple *cgp::Mesh<C>* structure, *toVtkPolyData* a VTK mesh. Furthermore, it will be convenient to have the set of all adjacent voxels (the \( \Gamma \)-neighbors for each 2-cell) available when computing features in sec. 2.7: function *adjacentVoxels*.

See fig. 2.15 for a class diagram of CGP’s coordinate classes.

### 2.6.4 Improving the performance of topological grid construction

Early implementations of the topological grid construction in the CGP software package (Andres et al., 2010b) suffered from some severe limitations for very large volumes such as the \( \epsilon 1088 \) dataset: (i) though in principal the labeling of \( j \)-components could be done in parallel, the initial implementation was still sequential (ii) storing the geometry as coordinate lists using the HDF5 file format results in hundreds of millions of dataset. For HDF5 to be able to handle this many datasets performantly, special settings were

```cpp
using vtkMath::Random;
GeometryReader r("geometry.h5");

w = new Window("all,0−sets,,1−sets");
for (label_type z=1; z<r.maxLabel(0); ++z) {
    w->plot(r.zeroSet(z).toVtkSphere(4, 8, 8), 0,0,0, 1);
}
for (label_type o=1; o<r.maxLabel(1); ++o) {
    w->plot(r.oneSet(o).toVtkTube(0.1), Random(0,1),Random(0,1), Random(0,1), 1.0);
}

w = new Window("all,2−sets");
for (label_type t=1; t<r.maxLabel(2); ++t) {
    w->plot(r.twoSet(t).toVtkPolyData(), Random(0,1),Random(0,1) Random(0,1), 1.0);
}
```

Figure 2.14 – For a subvolume of the \( \epsilon 1088 \) dataset, size \( 50 \times 50 \times 50 \), the geometry extraction was run on the initial watershed segmentation. The code to generate figures a) and b) is listed below.
Figure 2.15 - Class Diagram for the set classes in the CGP library.
Making use of HDF5 performance improvements. In order to make use of the latest HDF5 file format, which has much improved handling of huge amounts of datasets (as in the case of extracting the geometry for all of e1088), the following lines of C++ code are needed:

```c++
hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_libver_bounds(fapl, H5F_LIBVER_LATEST, H5F_LIBVER_LATEST);
hid_t file = H5Fcreate("name.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl);
```

Because of HDF5’s compatibility requirements, the new and improved file format is not enabled by default. As downward compatibility to previous HDF5 implementations is not needed, it is advisable to always create HDF5 files like this.

Parallelizing cgpx. Using the Message Passing Interface (MPI), two design choices for parallelization were explored. Both use a master-worker scheme.

(1) As a first implementation, already existing code from the parallelized blockwise calculation of the voxel features $\chi$ and of the probability map $\phi$ was reused. The master process opens the input file in read-only mode, reads it in a blockwise fashion and sends the complete data over the network. The worker receives it as the new workload, computes and sends the result as a serialized data structure back over the network to the master. After deserialization, the master can then finally integrate the newly arrived partial result into the output file. This corresponds to master-worker scheme 1 from p. 12.

Though easy to implement at the time, this first implementation had serious drawbacks: A complex result data structure, the Topology class, takes time to serialize and deserialize. In addition, a lot of burden is placed on the master because it has to handle all I/O operations. While writing the result to disk, it cannot hand out new jobs to the workers, which will then remain idle for too long. An execution timeline for this approach is shown in fig. 2.16.

(2) A second implementation aimed for an even simpler approach: the master hands out unique job identification numbers, and the workers themselves open the input file (in read only mode) and read the block corresponding to the unique job ID from it. This can be done in parallel. When their job is finished, the workers themselves need to integrate the result into the output file. However, HDF5 does not allow to write to different datasets from multiple program instances at the same time. A simple solution is a global read/write lock which is controlled by the master process.
and given in turn to all of the workers who ask for it. The worker who currently holds the global lock can then reopen the output file and write out its current partial result (master-worker scheme 2 from p. 12).

Running the parallelized cgpx program on the watershed segmentation $\sigma$ of the e1088 dataset on 16 CPU cores took 1 hour 38 minutes. For the first 52 minutes, the program execution was parallel, and only 7.6% of the time was spent on I/O lock contention. The rest of the execution time cannot be parallelized as the results obtained in a blockwise fashion need to be post-processed to become globally consistent.

### 2.6.5 Running geometry and topology extraction as part of the processing chain

First, the parallelization enabled cgpx tool builds the topological grid. When MPI is available, it can be started on multiple CPUs:

```
mpiexec -np 3 cgpx 'seg-D.h5' 'seg' 75 75 75 'topological-grid-D.h5'
```

Then, the cgpr tool builds the data structure for constant-time access to 0-, 1- and 2-component geometry. As the list of all voxels a 3-component is made up of is not needed anywhere, a considerable amount of space and I/O time can be saved by disabling the extraction of 3-component geometry:

```
cgpr 'topological-grid-D.h5' 'geometry-sbfsem-e1088-aligned.h5' SUPPRESS_03_SETS.
```

### 2.7 Learning to distinguish essential from excessive faces

Towards the goal of removing over-segmentation, features for each face $F \in \mathcal{F}$ are extracted (sec. 2.7.1, 2.7.2) and a classifier trained (sec. 2.7.4) to distinguish excessive faces from essential faces: too many local minima (seeds) lead to many faces $F$ that do not correspond to any edge in the elevation map $\phi$ at all.

By going from voxels to supervoxels – which should capture the local structure of the data – statistics over local voxel features become more meaningful.

#### 2.7.1 Face features

For each 2-component extracted in sec. 2.6.5, a set of features that characterize it are calculated next:

- A first approximation of the *surface area* is the number of voxel faces it is made up of.
- The voxel count of a 3-component is a first approximation of its *volume*. Each face has two neighboring segments; the sum and absolute difference of their volumes are calculated and the cube roots (a measure of extent) are taken as features.
- $F$ is a set of voxel faces each separating two voxels in $G$, the set of all separated voxels $F^{(G)}$ can be regarded as a symmetrical representation of $F$ on the voxel grid $G$. Statistics over the distribution of various voxel features from $\chi$ and the voxel probability map $\phi$ at the positions $F^{(G)}$ serve as features for the face $F$. This is motivated by the fact that good supervoxels should capture the geometry of the elevation map.

Tab. 2.6 summarizes the extracted features for each face.
2.7 Learning to distinguish essential from excessive faces

<table>
<thead>
<tr>
<th>index</th>
<th>feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component sizes</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>surface area (number of voxel faces)</td>
</tr>
<tr>
<td>1</td>
<td>((v_1 + v_2)^{1/3}) ((v_1, v_2) are the sizes of the neighboring supervoxels)</td>
</tr>
<tr>
<td>2</td>
<td>(</td>
</tr>
<tr>
<td><strong>Bilateral filter over Face</strong></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mean</td>
</tr>
<tr>
<td>4</td>
<td>standard deviation</td>
</tr>
<tr>
<td>5</td>
<td>minimum</td>
</tr>
<tr>
<td>6</td>
<td>0.25 quantile</td>
</tr>
<tr>
<td>7</td>
<td>median</td>
</tr>
<tr>
<td>8</td>
<td>0.75 quantile</td>
</tr>
<tr>
<td>9</td>
<td>maximum</td>
</tr>
<tr>
<td><strong>Other Statistics</strong></td>
<td></td>
</tr>
<tr>
<td>10-16</td>
<td>Gaussian gradient magnitude of bilateral filter over face, with same statistics</td>
</tr>
<tr>
<td>17-23</td>
<td>Greatest EV of Hessian matrix of bilateral filter over face, with same statistics</td>
</tr>
<tr>
<td>24-30</td>
<td>probability predicted by RF over face, with same statistics</td>
</tr>
</tbody>
</table>

Table 2.6 – Face features as calculated by the program face-feature-extraction-mpi, tab. 2.7.

2.7.2 Parallelizing the feature calculation

**Experiments.** For each 2-component \(F\) of the segment label map \(\sigma\), random access to the probability map \(\phi\) and the voxel features \(\chi\) is needed. The access is confined to a tight bounding box \(BB_F\) around the face \(F\) in which all voxels \(F^{(G)} = \{x \in G \mid x \in \Gamma(f) \land f \in F\}\) are contained. Thus all voxels that are bounded by one of \(F\)’s 2-cells can be accessed.

Loading \(\chi\) into RAM is not an option: For \(e1088\), 28 features stored with 4-byte float precision amount to 785GB! I have experimented with the following implementations:

1. A first implementation used the following naive approach:
   
   For each \(F \in [1, l_2^{\text{max}}]\), find the bounding box \(BB_F\), then load the corresponding data blocks from both the probability map and the voxel features file which are stored on disk. Then, using random access, accumulate voxel features and voxel probabilities over the surface \(F\) to calculate face features.
   
   First experiments showed that this approach is much too slow. With \(l_2^{\text{max}} = 20052088\), features calculation took 9.6 days. Clearly, a faster implementation was needed.

2. A second implementation employs a simple parallelization scheme: The list of faces for which features need to be calculated were partitioned into \(n_{\text{proc}}\) equally large subsets, with the number of processes set to \(n_{\text{proc}} = 16\) (one for each core of a 4 × 4-core processor). The data files were stored on a fast RAID attached to the computer. In practice, the I/O performance was the bottleneck. Calculating the features amounts to a few simple, fast operations; but now 16 processes compete to get new data blocks. As the requests are unordered, these blocks may lie anywhere within \(\phi\) and \(\chi\), which means frequent disk seeks. Using the `top` program to monitor CPU usage and `iotop` (Chazarain, 2010) confirmed my suspicions.
A third implementation incorporates additional block-wise processing. As a pre-processing step, the bounding boxes $BB_F$ for all faces $F \in \mathcal{F}$ are calculated. Then, the data volumes are partitioned into a number of blocks based on a user-specified block size. Each process is assigned a (roughly equally sized) subset of blocks. For each of these blocks $B$, a process first loads the corresponding data into RAM, then checks for all faces $F \in \mathcal{F}$ if $BB_F$ is wholly contained within $B$. If yes, all data is available and the process proceeds to calculate the feature values. If no, $F$’s features are marked as not-yet-calculated. When all faces have been checked, the next block is loaded. Finally, when a process has finished handling all its assigned blocks, it requests the global I/O lock. This lock protects the file in which the calculated features for all faces are stored. After having acquired the lock, the process merges the features it has calculated into the file (which might already contain output from other processes). When all blocks have been handled, all faces crossing block boundaries still await feature computation. I have considered doing a second round of block-wise processing, with blocks shifted by half their extent in each dimension, but found the following post-processing to be performant enough: all remaining faces’ features are calculated by the master process sequentially as in the first approach. One $\sigma$ of $e1088$ had 15 082 034 2-components, of which 860 856 or 5.7% remained to be processed after the block-wise calculation. The complete runtime was 2.6 days.

**Command line tools.** The bounding boxes $BB_F$ (as defined above) are pre-computed by the `export-twoset-bbox` program. The block-wise parallel feature computation (approach 3) is available as `face-feature-extraction-mpi`. See tab. 2.7 for the usage of these two programs.

### 2.7.3 Labels

**Training data version 0 (TD₀).** On the training dataset, the segmentation pipeline is run up to the geometry extraction step. There are two segmentations: $\sigma^g$, defined via the ground-truth voxel labels and $\sigma^a$, the automatic watershed segmentation. Voxels that do not belong to any segment have a background label 0.

The overlap matrix $\mathcal{O}$ has as many rows as $\sigma^a$ has segments and as many columns as the number of segments in $\sigma^g$:

$$\mathcal{O}_{ij} = \left| \{ \mathbf{x} \in \sigma | \sigma^a(\mathbf{x}) = i \land \sigma^g(\mathbf{x}) = j \} \right|. \quad (2.28)$$

$\mathcal{O}$ can be used to derive labels for all faces in $\sigma^a$: for each row $j$ of $\mathcal{O}$, the automatic segment $j$ has best overlap with the ground-truth segment $\text{arg max}(\text{row } i)$. This is recorded in the vector $B$:

$$B_i = \text{arg max}_k \mathcal{O}_{ik}. \quad (2.29)$$

For each face $F$, select any voxel face $f \in F$. Its $\Gamma$-neighbors are two voxels $\mathbf{x}_1, \mathbf{x}_2$ belonging to the two segments that $F$ bounds.

- If either $B[\sigma^a(\mathbf{x}_1)]$ or $B[\sigma^a(\mathbf{x}_2)]$ is zero, face $F$ is not labeled.
- If $B[\sigma^a(\mathbf{x}_1)] = B[\sigma^a(\mathbf{x}_2)]$, the bounded segments both have best overlap with the same ground-truth segment, thus $F$ is assigned an excessive label.
- If $B[\sigma^a(\mathbf{x}_1)] \neq B[\sigma^a(\mathbf{x}_1)]$, $F$ is essential.

This is implemented in the program `face-labeling`.  

38
### 2.7 Learning to distinguish essential from excessive faces

#### mpirun -np 3 export-twoset-bbox

**INPUT**

- `--geometry 'geometry-D.h5'`  
  output of cgpr, representing the geometry of the watershed segmentation $\sigma$

**OUTPUT**

- `--out 'twoset-bbox-D.h5#seg'`  
  bounding boxes $BB_F$ for each face in $F \in \mathcal{F}$

#### mpirun -np 3 face-feature-extraction-mpi

**INPUT**

- `--hdf5geometry 'geometry-D.h5'`  
  output of cgpr, representing the geometry of $\sigma$
- `--hdf5features 'voxel-features-D.h5#voxel-features'`  
  voxel features $\chi$ (output from voxel-features-mpi, tab. 2.2)
- `--hdf5probabilityMap 'probability-map-D.h5#probability-map'`  
  voxel probability $\phi$ to be extra-cellular space (output from tab. 2.4)
- `--twosetBbox 'twoset-bbox-D.h5'`  
  bounding boxes $BB_F$ for each face in $\sigma$, computed by the above program
- `--segmentSizes 'seg-D.h5#segment-sizes'`  
  sizes (in voxels) of each 3-component of $G$, computed by tab. 2.5

**OUTPUT**

- `--hdf5faceFeatures 'face-features-D.h5#face-features'`  
  array $[0, l_{2\text{max}}^{(2)}] \times [0, 31) \rightarrow \mathbb{R}$ with calculated features, for face $K_q^{(2)}$ row $q - 1$ (see tab. 2.6)

**Block-wise processing**

- `--blockSize 75 75 75`  
  block size $B$

---

**Table 2.7** – Command line options for the export-twoset-bbox (above) and face-feature-extraction-mpi (below) programs. From Python use denkc.Pipeline.run.twosetBbox() and denkc.Pipeline.run.faceFeatures().

---

**Training data version 1 (TD1).** sbfsem-Y.h5 contains a $250^3$ subset, starting at offset $o = (700, 700, 700)$, of the $e1088$ dataset. One watershed segmentation, seg-sbfsem-Y.h5, was loaded into the visualization tool (sec. 4.2) for which I have implemented an additional face labeling mode. Giving correct labels turns out to be quite difficult, as the configuration of faces in 3D space is not immediately apparent when viewing only the three slice views. Labeling has a lot in common with putting a puzzle together: faces are turned on and off in a neighborhood until everything seems to fit. Björn Andres and I have labeled many of the faces in the segmentation of $Y$, and finally the labeling by B. Andres was chosen as ground-truth (face-labels-sbfsem-Y-andres-1.h5, TD1), which is comprised of 53645 unlabeled, 874 excessive and 2692 essential faces.

---

**2.7.4 Training RF$_2$**

Tab. 2.8 summarizes the usage of train-face-classifier program to obtain a serialized Random Forest RF$_2$. From the input labels ($l_1$ essential, $l_2$ excessive labels) a balanced training set is constructed, consisting of $\min(l_1, l_2)$ samples per class drawn at random without replacement.
train-face-classifier

**Input**

--hdf5features 'face-features-D.h5#face-features'

face features, output from face-feature-extraction-mpi, tab. 2.7

--hdf5labels 'face-labels-D.h5#face-labels'

ground-truth face labels (see sec. 2.7.3)

**Output**

--randomForestHDF5File 'RF2.h5'

trained Random Forest RF2

**Parameters**

--trees 255

number of trees to be grown

Table 2.8 – Command line options for the train-face-classifier program.

![Figure 2.17](image)

Figure 2.17 – Distribution of the face probabilities $p_F$ for $e1088$ (parameters: see tab. 2.18).

2.7.5 Parallelizing the face classification

Using RF2 to classify all faces of $e1088$ is not a runtime bottleneck. The simplest parallelization, giving each process an equally large workload on application start and merging all results into one file using a global I/O lock (master-worker scheme 2 from p. 12) is already sufficient to achieve a runtime of 55s for 15,082,034 faces. Tab. 2.9 summarizes the usage of the classify-faces-mpi program.

Fig. 2.17 shows the distribution of predicted face probabilities $p_F$ for $e1088$. The classifier is quite certain of its decision for most faces; this indicates the strength of the chosen features.
2.8 Learning to predict the joint configuration of intersecting faces

Due to the discrete nature of the voxel grid $G$, a 1-component always bounds either three or four 2-components (the 1-component is called an intersection in this context, as it arises due to the intersection of the bounded faces). Typically, over 99% of all 2-components are intersections of three faces.

In 2D slice views of the data, supervoxel faces appear as 1D inter-pixel boundary lines. Often, humans can recognize excessive faces by looking only at the geometry of these boundaries without viewing the raw data at all. In neural tissue, most membranes are gently curved; sharp edges are improbable.

With this prior knowledge, one can see in fig. 2.19 that the neurons running from lower left to upper right are interrupted by excessive, ladder-step-like faces almost perpendicular to the true membrane boundaries.

### 2.8.1 Intersection features

How can this information be distilled into a set of features? The angle between two adjacent faces can indicate if one is a smooth continuation of the other or not. More precisely, angles are estimated as follows:

Let $K_q^{(1)} = \{t_1^{(1)}, \ldots, t_n^{(1)}\}$ be the set of 1-cells making up the 1-component labeled $q$. Each $t^{(1)}$ represents a voxel edge with center

$$x^{(1)} = \left(\frac{t_0^{(1)}}{2}, \frac{t_1^{(1)}}{2}, \frac{t_2^{(1)}}{2}\right). \quad (2.30)$$

Let $A, B \in \Gamma(K_q^{(1)})$ be two bounded 2-components. $t^{(2)} \in A, B$ represents a voxel surface with the center coordinate

$$x^{(2)} = \left(\frac{t_0^{(2)}}{2}, \frac{t_1^{(2)}}{2}, \frac{t_2^{(2)}}{2}\right). \quad (2.31)$$

All $x^{(2)} \in A, B$ with $\|x^{(2)} - x^{(1)}\| < d_{\text{max}}$ are averaged as $\bar{a}, \bar{b}$. Finally, the angle between $A$ and $B$ is taken as $\angle(\bar{a}, x^{(1)}, \bar{b})$. 

---

**mpirun -np 3 classify-faces-mpi**

**INPUT**

--hdf5faceFeatures 'face-features-D.h5#face-features'

--faceRandomForestFile 'RF2.h5'

**Output**

--hdf5predictedFaceLabels 'predicted-face-labels-D.h5#face-probabilities'

<table>
<thead>
<tr>
<th>Table 2.9 – Command line options for the classify-faces-mpi program. From Python use <em>denkc.Pipeline.classifyFaces()</em>.</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>--hdf5faceFeatures</td>
<td>'face-features-D.h5#face-features'</td>
<td>face features</td>
<td>(face-feature-extraction-mpi, tab. 2.7)</td>
</tr>
<tr>
<td></td>
<td>--faceRandomForestFile 'RF2.h5'</td>
<td>RF2 (train-face-classifier, tab. 2.8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>--hdf5predictedFaceLabels</td>
<td>'predicted-face-labels-D.h5#face-probabilities'</td>
<td>matrix $[0, t_{\text{max}}^{(2)}] \times [0, 2) \rightarrow [0, 1] \subset \mathbb{R}$, where the probabilities for face $K_q^{(2)}$ to be essential are stored in $(q-1, 0)$ and to be excessive in $(q-1, 1)$</td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.18 – Illustration of angle estimation: a) the two-dimensional case. b) A 1-set (gray) bounds three 2-sets (translucent red, green, blue). Given one point of the 1-set (black), the center of mass (pink) is calculated for each 2-set using all voxel face centers that are in a certain range (these are shown as red, green and blue spheres).

The distribution of the angles estimated for all positions $x^{(1)}$ of the 1-component is then described by various statistics. Fig. 2.18 illustrates the angle estimation.

Distributions of angles are only calculated for 1-components bounding exactly three 2-components $(A, B, C)$: angles $\angle(A, B)$, $\angle(B, C)$ and $\angle(C, A)$. The various statistics over these distributions that serve as features are summarized in tab. 2.10.

2.8.2 Parallelizing the feature computation

As noted in sec. 2.6.4 using the latest HDF5 file format may bring huge performance gains. This was particularly obvious when running the intersection feature computation.

Using the geometry representation of an 1088 volume labeling, runtime was 3.5 days when using the HDF5 1.6 file format ($l_{\text{max}}^1 \approx 3 \times 10^7$). After switching to the 1.8 file format and implementing a simple parallelization where the work is divided into equally sized chunks on startup and written back to one resulting file at the end using a global I/O lock (master-worker scheme 2 from p. 12), the runtime could be reduced to 3.5 hours.
2.8 Learning to predict the joint configuration of intersecting faces

<table>
<thead>
<tr>
<th>index</th>
<th>feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mean</td>
</tr>
<tr>
<td>1</td>
<td>standard deviation</td>
</tr>
<tr>
<td>2</td>
<td>minimum</td>
</tr>
<tr>
<td>3</td>
<td>0.25 quantile</td>
</tr>
<tr>
<td>4</td>
<td>median</td>
</tr>
<tr>
<td>5</td>
<td>0.75 quantile</td>
</tr>
<tr>
<td>6</td>
<td>maximum</td>
</tr>
<tr>
<td>7-13</td>
<td>same statistics over angles between 2. and 3. face</td>
</tr>
<tr>
<td>14-20</td>
<td>same statistics over angles between 3. and 1. face</td>
</tr>
</tbody>
</table>

Table 2.10 – Intersection features, as calculated by the program intersection-feature-extraction-mpi, tab. 2.7.

![Figure 2.19](image)

Figure 2.19 – In this slice of the watershed segmentation $\sigma$, only the inter-voxel faces have been drawn. Humans can find many excessive boundaries by considering the depicted geometry.

**Command line tools.** The program intersection-feature-extraction-mpi takes $d_{\text{max}}$ as a parameter as well as a list of quantiles to describe the angle distribution. In all experiments, $d_{\text{min}} = 7$ was chosen and the minimum, 0.25 quantile, median, 0.75 quantile and maximum were used as features, see tab. 2.10.

2.8.3 Training RF₃

Tab. 2.11 summarizes the usage of the train-intersection-classifier program to obtain a serialized Random Forest RF₃.

In neural tissue data, the configurations $(0,0,1)$, $(0,1,0)$ and $(0,0,1)$ should *not* occur as they indicate non-closed cell membranes; five classes representing non-forbidden configurations remain. Let the smallest number of training samples per class be $n$. Then a balanced training set is constructed by drawing at random without replacement $n$ samples from each class.
2 Segmentation of volume electron microscopy images for connectomics

**train-intersection-classifier**

**Input**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--hdf5intersectionFeatures</td>
<td>face features (face-feature-extraction-mpi, tab. 2.7)</td>
</tr>
<tr>
<td>'intersection-features-D.h5#features'</td>
<td></td>
</tr>
<tr>
<td>--hdf5intersectionLabels</td>
<td>ground-truth face labels $L : [0, l_{\text{max}}^1) \rightarrow [0, 9)$</td>
</tr>
<tr>
<td>'intersection-labels-D.h5'</td>
<td>where $L_q$ gives the label for $K^1_q$</td>
</tr>
<tr>
<td>#intersection-labels</td>
<td></td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--randomForestHDF5File 'RF3.h5'</td>
<td>trained Random Forest RF$_3$</td>
</tr>
</tbody>
</table>

**Parameters**

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--trees</td>
<td>255</td>
<td>number of trees to be grown</td>
</tr>
</tbody>
</table>

(* CGP defines an ordering for the faces bounded by curve $K^1_q$ as returned by `GeometryReader::bounds(1,q)`. Then, the configurations $(0, 0, 0)$ lexicographically through $(1, 1, 1)$ have labels 0 to 7; label 8 denotes unlabeled curves.

Table 2.11 – Command line options for the train-intersection-classifier program.

**Figure 2.20** – Distribution of the probability values for the likeliest configuration of three faces for $e1088$ (parameters: see tab. 2.18).

**2.8.4 Parallelizing the intersection classification**

With classify-intersections-mpi (tab. 2.13) the previously trained RF$_3$ is used together with the calculated features to predict the configuration of all intersections of three faces. Again a simple parallelization scheme is sufficient to reduce the runtime to about 5 minutes for $e1088$.

Fig. 2.20 shows the distribution of the probabilities for the likeliest configuration.
intersections-feature-extraction-mpi

**INPUT**

--hdf5geometry 'geometry-D.h5'
output of cgpx, representing the geometry of the watershed segmentation \( \sigma \)

**OUTPUT**

--hdf5intersectionFeatures 'intersection-features-D.h5#features'
intersection features, tab. 2.10

**PARAMETERS**

--distance 7 \( d_{\text{max}} \)
--angleQuantiles 0 0.25 0.5 0.75 1.0 list of quantiles to use as features; see tab. 2.10

Table 2.12 – Command line options for the *intersection-feature-extraction-mpi* program. From Python use *denke.Pipeline.intersectionFeatures()*.

classify-intersections-mpi

**INPUT**

--hdf5intersectionFeatures 'intersection-features-D.h5#features'
intersection features from *intersection-feature-extraction-mpi*, tab. 2.12

--intersectionsRandomForestFile 'RF3.h5'
RF3 from *train-intersection-classifier*, tab. 2.11

**OUTPUT**

--hdf5predictedIntersectionLabels 'predicted-intersection-labels-D.h5#intersection-labels'
matrix \( L : [0, l_{\text{max}}] \rightarrow [0, 8) \); if \( K_q^{(1)} \) is a 3-junction, \( L_{q-1,i} \) gives the probability for the configuration \( i (\ast) \), undefined otherwise

Table 2.13 – Command line options for the *classify-intersections-mpi* program. From Python use *denke.Pipeline.run.classifyIntersections()*.

\( \ast \) CGP defines an ordering for the faces bounded by curve \( K_q^{(1)} \) as returned by *GeometryReader::bounds(1,q)*. Then, the configurations \( (0,0,0) \) lexicographically through \( (1,1,1) \) have labels \( i = 0 \ldots 7 \).

### 2.9 Globally consistent face removal using a Graphical Model

A 2-component is usually bounded by many 1-components. The decision whether to classify that particular face \( F \) as essential or excessive depends (1) on the local decision of the Random Forest RF2 (2) on the decisions of bounded curves with regard to the face \( F \). Therefore a global optimization problem needs to be solved.

The *Graphical Models* framework allows to formulate this problem with the help of an energy function; by exploiting its mathematical structure a configuration of all faces that approximately minimizes the energy can be found in acceptable time.

#### 2.9.1 Graphical Models, energy function and optimization

**Graphical Models.** Graphical Models (Bishop 2006, chap. 8) describe the factorization of a multi-variate function by means of graph theory.

*Directed Graphical Models* or *Bayesian Networks* are used to describe probability distributions \( p(x_1, \ldots, x_K) \), which can always be written as a product over the marginal
probability distributions. Each variable $x_j$ is represented as a node in a graph. For each factor, directed edges are introduced pointing from each variable on which the distribution is conditioned to the node representing the argument of the marginal. The absence of edges is key: Whenever a link $x_i \rightarrow x_j$ is missing, this indicates that variable $x_j$ does not have a direct statistical dependence on $x_i$.

Given any directed, acyclic graph, the probability distribution it represents can be written as (pa denotes the parents of node $x_i$):

$$p(x_1, \ldots, x_K) = \prod_{i=1}^{K} p(x_i | \text{pa}(x_i)).$$  \hfill(2.32)

Often, the goal is to compute the marginal probabilities for all variables (inference). For each $x_i$, this involves calculating the sum

$$p(x_i) = \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_K} p(x_1, \ldots, x_K).$$  \hfill(2.33)

**Undirected Graphical Models** can describe the factorization of any function. Instead of conditional probabilities $p(x_j | x_i)$ involving hard dependencies compatibility functions $p(x_i, x_j)$ can be used, which serve as soft constraints between variables. In the context of reducing over-segmentation in the presented segmentation procedure, each face $F \in \mathcal{F}$ corresponds to a binary variable $x_F \in [0, 1]$, where 0 means that the face is excessive and should be removed and 1 that it is essential.

In the following variables corresponding to the $\Gamma$-neighbors of a curve $C$ are denoted with $\Gamma(C) = \{x_{c1}, x_{c2}, \ldots \}$. One way to define the joint probability of all binary face variables $x_F$ is (compare eq. 2.32)

$$p(\{x_F\}) = \frac{1}{Z} \cdot (1 - \gamma) \prod_{F \in \mathcal{F}} \psi^{(1)}(x_F) \cdot \gamma \prod_{C \in \mathcal{C}, |\Gamma(C)|=3} \psi^{(3)}(x_{c1}, x_{c2}, x_{c3})$$

$$\cdot \gamma \prod_{C \in \mathcal{C}, |\Gamma(C)|=4} \psi^{(4)}(x_{c1}, x_{c2}, x_{c3}, x_{c4}).$$  \hfill(2.34)

$$\hfill(2.35)

The expressions involved will be discussed in detail later. With a substitution $E^{(i)} = -\ln \psi^{(i)}$ this can be written as

$$p(\{x_F\}) = \frac{1}{Z} e^{-E(\{x_F\})/T},$$  \hfill(2.36)

which is Boltzmann’s law from statistical physics (here $T = 1$, and $Z$ is the partition function). This motivates the usage of an energy function in inference problems: Implicitly, $E(\{x_F\})$ defines a probability distribution via eq. 2.36. A maximum likelihood solution $p_{\text{max}} = \max_{F \in \mathcal{F}} \{p(\{x_F\})\}$ corresponds to the lowest energy $E_{\text{min}}$ among all possible configurations of $x_1, \ldots, x_{|\mathcal{F}|}$.

**Energy function.** The energy function $E(\{x_F\})$ therefore is

$$E(\{x_F\}) = (1 - \gamma) \sum_{F \in \mathcal{F}} E^{(1)}(x_F) + \gamma \sum_{C \in \mathcal{C}, |\Gamma(C)|=3} E^{(3)}(x_{c1}, x_{c2}, x_{c3})$$

$$+ \gamma \sum_{C \in \mathcal{C}, |\Gamma(C)|=4} E^{(4)}(x_{c1}, x_{c2}, x_{c3}, x_{c4}).$$  \hfill(2.37)
2.9 Globally consistent face removal using a Graphical Model

There are three contributions to the total energy (again in analogy to physics, individual addends are called potentials):

1. $E^{(1)}(x_F)$ is called a first order potential as it only depends on one variable, the binary state of face $F$. There are only $2^1 = 2$ possible values. The higher the energy $E^{(1)}(0)$, the easier it will be in any low energy configuration to set $x_F = 0$.

2. Let $C \subseteq \mathcal{C}$ be any curve and $\Gamma(C)$ the set of faces it bounds. Either $|\Gamma(c)| = 3$ or $|\Gamma(c)| = 4$. For each case a separate potential is introduced, depending on variables $x_{c1}, x_{c2}, x_{c3}$ and possibly $x_{c4}$. $E^{(3)}(x_{c1}, x_{c2}, x_{c3})$ is called a third order potential. There are $2^3 = 8$ possible values, which should be chosen to encourage the most probable states of the three adjacent faces.

3. The fourth order potential $E^{(4)}(x_{c1}, x_{c2}, x_{c3}, x_{c4})$ has $2^4 = 16$ possible values.

For the potentials $E^{(3)}$, the probabilities from sec. 2.8 (RF$_3$) are used, except for the three configurations $(0,0,1)$, $(0,1,0)$ and $(1,0,0)$, which are assigned a high energy penalty $E_h$. This reflects the fact that all cell membranes in neural tissue should be closed.

As four intersecting faces are scarce, learning their configurations and predicting the 16 possible states is replaced by a deterministic potential. $E_h$ is assigned to the forbidden configurations (which have only one essential face); for all other configurations the decision is left up to the other potentials.

Let $p_F$ be the probability of face $F$ to be essential. Then the unary potential is defined as $E^{(1)}(0) = 1 - p_F'$ and $E^{(1)}(1) = p_F'$. A modification $p_F'$ of the probabilities $p_F$ with

$$
p_F' = \begin{cases} 
1 & p \geq 1 - \alpha \\
0 & p \leq 1 - \beta \\
\frac{1}{\beta - \alpha} \cdot p_F - (1 - \beta) & \text{else}
\end{cases}
$$

has been used in the following experiments. Fig. 2.21 illustrates $p_F'$. A larger $\alpha$ ensures that probabilities less close to absolute certainty are still clamped to 1, expressing a distrust in the votes of RF$_2$. A larger $\alpha$ thus makes it harder to remove faces, resulting in more over- but less under-segmentation.

$\beta$ has the opposite effect; in all following experiments $\beta = 1$. 

![Figure 2.21](image-url) - The parameters $\alpha$ and $\beta$ are used to modify the RF$_2$ probabilities $p_F$ for a face to be essential. See eq. 2.38.
Segmentation of volume electron microscopy images for connectomics

1\textsuperscript{st}-order

<table>
<thead>
<tr>
<th>(x_F)</th>
<th>(E(x_F))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1 - p_F')</td>
</tr>
<tr>
<td>1</td>
<td>(p_F')</td>
</tr>
</tbody>
</table>

2\textsuperscript{nd} - and 3\textsuperscript{rd}-order potentials

| Label | \(x_{c1}\) | \(x_{c2}\) | \(x_{c3}\) | \(x_{c4}\) | \(p_C(\{x|x \in \Gamma(C)\})\) | \(E_C(\{x|x \in \Gamma(C)\})\) |
|---|---|---|---|---|---|---|
| Curve \(k\) bounds three faces |
| 0 | 0 | 0 | 0 | – | \(p_C(0,0,0)\) | \(1 - p_C(0,0,0)\) (\(\ast\)) |
| 1 | 0 | 0 | 1 | – | 0 | \(E_h\) |
| 2 | 0 | 1 | 0 | – | 0 | \(E_h\) |
| 3 | 0 | 1 | 1 | – | \(p_C(0,1,1)\) | \(1 - p_C(0,1,1)\) (\(\ast\)) |
| 4 | 1 | 0 | 0 | – | 0 | \(E_h\) |
| 5 | 1 | 0 | 1 | – | \(p_C(1,0,1)\) | \(1 - p_C(1,0,1)\) (\(\ast\)) |
| 6 | 1 | 1 | 0 | – | \(p_C(1,1,0)\) | \(1 - p_C(1,1,0)\) (\(\ast\)) |
| 7 | 1 | 1 | 1 | – | \(p_C(1,1,1)\) | \(1 - p_C(1,1,1)\) (\(\ast\)) |

Curve \(k\) bounds four faces

| \(x_{c1}\) | \(x_{c2}\) | \(x_{c3}\) | \(x_{c4}\) | \(p_C(\{x|x \in \Gamma(C)\})\) |
|---|---|---|---|---|
| 9 | 0 | 0 | 0 | 1 | \(E_h\) |
| 9 | 0 | 0 | 1 | 0 | \(E_h\) |
| 9 | 0 | 1 | 0 | 0 | \(E_h\) |
| 9 | 1 | 0 | 0 | 0 | \(E_h\) |
| 9 | else | – | – | 0 |

\(p_C(\{x_{c1}, x_{c2}, x_{c3}\})\) denotes the joint probability function of the three faces bounded by curve \(C\), as predicted by RF

Tab.2.14 summarizes the definition of energies \(E^{(1)}\), \(E^{(3)}\) and \(E^{(4)}\). \(p_C(\{x_{c1}, x_{c2}, x_{c3}\})\) denotes the joint probability function of the three faces bounded by curve \(C\), as predicted by RF

Typical segmentations \(\sigma\) lead to a complicated topology where one face \(F\) can have a large number of directly adjacent faces. Fig. 2.22 shows the distribution of the neighbor count for a segmentation of \(e1088\). In the Graphical Model, all adjacent faces have a direct influence on the decision for \(F\) via third- and fourth-order potentials. This may be helpful for belief propagation (see below): the state of \(F\) is influenced by a lot of adjacent faces which can all contribute evidence.

On the other hand, fig. 2.23 shows how many faces \(F\) bound a given segment \(S \in S\). There is one segment that is bounded by 437,970 faces (this object probably incorporates a glia cell, a support structure of the neural tissue). When two large objects touch – involving many shared faces – only one wrong excessive label will merge them. This fate will become increasingly certain the larger the touching objects become.

Factor graphs. Any function that factorizes with regard to a commutative and associative operation \(\otimes\) (usually either multiplication \(\otimes = \prod\) or addition \(\otimes = \sum\)) can be represented by a bipartite graph, a generalization of the previous Graphical Models where factors are represented explicitly. Each variable \(x_i\) has a corresponding variable
2.9 Globally consistent face removal using a Graphical Model

Figure 2.22 – Distribution of the number of faces directly adjacent to a given face (segmentation of e1088, parameters: see tab. 2.18).

node in the graph and each factor $f_j$ by a corresponding factor node. There is an undirected edge $f_j \leftrightarrow x_i$ if and only if $f_j$ is a function of $x_i$. Thus the factor graph expresses the "is an argument of" relation between variables and local functions (Kschischang et al., 2001).

Fig. 2.24 shows the factor graph for the Graphical Model used for removing oversegmentation (yielding eq. 2.35 with $\otimes = \prod$ and eq. 2.37 with $\otimes = \sum$) with a simple toy example. The three types of factors correspond to first order, third order and fourth order potential functions.

For inference, the goal is to either compute the marginals (eq. 2.33) for each variable or simply to find the most likely state of $x_i$ (MAP estimate) by computing the arg max of the marginal probabilities (or arg min when working with energies). It can be shown that the sum-product, max-product and min-sum algorithms can find these solutions exactly when the factor graph has tree structure. Exploiting the factorization property, these algorithms are linear in the number of factor and variable nodes, while a naive evaluation of the sum in eq. 2.33 requires an exponential number of operations. Sum-product, max-product and min-sum algorithms are special cases of belief propagation.

Belief propagation. In the belief propagation algorithm, messages are sent between factors and variables. Node $n$ receives messages from all its neighbors, the nodes $\mathcal{N}(n)$. Summarizing these messages, node $n$ sends a message to neighboring node $n' \in \mathcal{N}(n)$ by placing it on the directed edge $e = (n, n')$. The content: $n$ tells $n'$ what it believes –
2 Segmentation of volume electron microscopy images for connectomics

Figure 2.23 – Distribution of the number of faces bounding a given segment (segmentation of e1088, parameters: see tab. 2.18).

(a) A configuration of six faces $f_1 \ldots f_6$, creating intersections of both three and four faces.
(b) Factor graph of the corresponding Graphical Model.

Figure 2.24 – From topology of the over-segmentation $\sigma$ to a factor graph representation.
based on the information it has available – which state the receiving node should be in.

There are two types of messages: messages $\mu_{x\rightarrow f}$ from a variable to a function and messages $\mu_{f\rightarrow x}$ from a function to a variable. The update rules for these messages are:

$$\mu_{x\rightarrow f} = \prod_{h \in N(f) \setminus \{f\}} \mu_{h\rightarrow f}(x)$$  \hspace{1cm} (2.39)$$

$$\mu_{f\rightarrow x} = \text{acc} \left( f(X) \otimes \prod_{y \in N(f) \setminus \{x\}} \mu_{y\rightarrow f}(y), x \right),$$  \hspace{1cm} (2.40)$$

where acc($f, x_i$) is the accumulation operation. Given a function $f(X)$, it eliminates all variables $X \setminus \{x_i\} = \{x_1, \ldots, x_{i-1}, x_{i+1}, x_{|X|}\}$ from $f$, so that the resulting function only depends on $x_i$.

- **Marginalization:**
  - sum-product algorithm with $\otimes = \sum$, acc($f, x$) = $\sum_{X \setminus \{x_i\}} f$.
- **Optimization:**
  - min-sum algorithm with $\otimes = \sum$, acc($f, x$) = $\min_{X \setminus \{x_i\}} f$,
  - max-product algorithm with $\otimes = \prod$, acc($f, x$) = $\max_{X \setminus \{x_i\}} f$.

For tree-structured factor graphs, it suffices to pass, for every undirected edge, one message in each direction to compute the correct beliefs for each variable.

**Loopy belief propagation.** One can apply the update rules of the belief propagation algorithm to a graph with loops, where in each iteration all messages are updated (synchronous variant). There is no guarantee of convergence, and the resulting beliefs may or may not be a good approximation of the true beliefs. For many applications however, applying loopy belief propagation has worked surprisingly well (Yedidia et al. (2003), for more background on the relation to physics see Yedidia 2000).

**Search based optimization.** A second class of popular algorithms for optimizations of Graphical Models are search-based (Andres et al., 2010a). The Lazy Flipper constrains the search to all possible subsets of (binary) variables with a user-specified maximum size $lf_{\text{max}}$ (with $lf_{\text{max}} = 3$ being reasonable for the Graphical Models in the e1088 segmentation). For each subset, the best possible configurations of its variables can be found by exhaustive search. With $lf_{\text{max}} = 1$, the Iterated Conditional Modes (ICM) algorithm is recovered. The storage requirements necessary to keep track of already treated subsets are the main limitation.

**2.9.2 Enabling belief propagation for very large Graphical Models**

OpenGM (née PGM, tech report Andres et al. 2008a) is a C++ template library for inference in factor graphs with arbitrary-order functions of discrete variables (Andres et al., 2010a). Different variants of belief propagation and search-based algorithms (such as the Lazy Flipper) are implemented.

Initially, the Graphical Model for the e1088 segmentation could not be optimized with belief propagation because its representation and all messages did not fit into memory (using 128GB RAM).

A function $f(X)$ of discrete variables $X = \{x_1, \ldots, x_{|X|}\}$ maps from $S_1 \times \cdots \times S_{|X|}$ to (usually) $\mathbb{R}$, where $S_i$ are sets of variable states. Such a function can always be represented by its function table, an $|X|$-dimensional matrix of real values.
It quickly turned out that the data structures used in the \textit{ExplicitFactor} class were too general for the purpose of minimizing eq. 2.37. Variables $x_F$ represent the binary state of face $F$, therefore $\forall i: |S_i| = 2$. This means that the number of states does not have to be explicitly stored for each variable.

Additionally, a generic dynamic matrix class (\textit{Marray} in this case, Andres et al. 2010c) stores for each dimension of the function table the extent. This is unnecessary as all extents are known to be two. From the total size $s$ of a particular block of memory alone the number of dimensions can be computed as $\log_2 s$.

\textit{Marray} is derived from the class \textit{marray::View}, which defines the following member variables (in brackets: size in bytes on a 64-bit machine)

- \texttt{unsigned short dimension\_} (2B), \texttt{std::vector<size\_t> shape\_} (empty: 24B) describe the array’s dimension and its extent in each dimension. \texttt{size\_t size\_} (8B) stores the pre-computed total number of array elements.
- \texttt{std::vector<size\_t> strides\_} and \texttt{std::vector<size\_t> shapeStrides\_} (empty: 24B) holds the strides, once memory-order independent, once for the chosen memory order.

The enum \texttt{CoordinateOrder coordinateOrder} (4B) will resolve to a compiler-dependent type (see ISO, 2010, p. 152), 4 bytes for 64-bit g++. Additionally, \textit{Marray} stores a \texttt{bool isSimple\_} (1B) to keep track of special situations.

It is clear that \textit{Marray} was optimized for speed rather than memory footprint. In total, all members sum to 103 bytes. Due to memory alignment (which depends on the order in which the variables are declared, still room for optimization!) this is blown up to 120 bytes, just for an empty array. For a factor $f(X)$ the required storage (mem) is

$$\text{mem}(f(X)) = 120\text{byte} + 3|X| \cdot 8\text{byte} + 2^{|X|} \cdot 4\text{byte}. \tag{2.41}$$

One Graphical Model (see tab. 2.18) had $45\,202\,771$ factors (first order: $15\,082\,034$, third order: $25\,274\,156$, fourth order: $4\,846\,581$) and $15\,082\,034$ variables. For both types of messages alone, this sums up to approx. 110GB \textit{only} for the value tables!

An additional template parameter was introduced for \texttt{opengm::ExplicitFactor} which specifies the type used for the value table. I have written highly optimized \texttt{ArrayPow2} array class specifically for multi-dimensional arrays of extent 2 in each dimension (their size $s$ is always a power of 2, thus the name). This class implements a \textit{Marray} compatible interface, and uses fast bit-shift operations wherever possible. As members, only the data pointer (8-byte) and the number of dimensions $\log_2 s$ (1 byte) need to be stored. Due to alignment, this sums up to 16B for one value table, or 2.2GB for all factors and variables in the above Graphical Model. A positive side effect of this change was the vast reduction in calls to \texttt{malloc} and \texttt{dealloc} (profiling with \texttt{valgrind}, dynamic allocations in the \texttt{std::vector} members of \textit{Marray} had a large impact), speeding up the complete belief propagation procedure (a Graphical Model for a $130 \times 130 \times 130$ sized $\sigma$ with 100 iterations took 1.9min compared to 5.2min).

### 2.9.3 Constructing the Graphical Model

The program \texttt{build-graphical-model} (tab. 2.15) builds the Graphical Model according to eq. 2.37 and tab. 2.14 and writes a serialized representation to an HDF5 file, which can then be used in conjunction with the command line program \texttt{opengm} (distributed with the openGM library) to do inference. The parameters $\alpha$, $\beta$ (altering the RF$_2$ probabil-
2.9 Globally consistent face removal using a Graphical Model

### build-graphical-model

**Input**

- `--faceProbabilities 'predicted-face-labels-D.h5#face-probabilities'`
  - face probabilities (classify-faces-mpi, tab. 2.9)

- `--intersectionProbabilities 'predicted-intersection-labels-D.h5#intersection-labels'`
  - intersection configuration probabilities (classify-intersections-mpi, tab. 2.13)

**Output**

- `--graphicalModelFile 'graphical-model-D.h5'`
  - serialized Graphical Model (see. eq 2.37)

**Parameters**

- `--alpha 0`
  - Parameters used to modify the RF

- `--beta 1.0`
  - Penalties for forbidden configurations

- `--gamma 0.5`
  - Mixture parameter weighing unary potentials against higher-order ones

- `--energyHigh 1.5`
  - Penalty for forbidden configurations

Table 2.15 – Command line options for the build-graphical-model program. From Python use `denke.pipeline.buildGM(\alpha, \beta, \gamma)`.

Saving each factor as a separate HDF5 group and storing the value tables as individual datasets turned out to be rather slow. Additionally, the extra meta-information needed increases the file size. Therefore, the complete Graphical Model is now encoded within a single floating point vector. Care must be taken that the floating point precision is large enough: some entries in the vector represent indices into the set of variables or factors which can be as large as $1 \times 10^8$. This means that `float` is no option (a nasty bug to find, as only very large indices get rounded to the wrong integer) and `double` has to be used in `opengm::serialize`.

2.9.4 Finding approximate solutions

A combination of belief propagation and search-based algorithms is used to find a (hopefully good) local minimum of the energy function, eq. 2.37. First, belief propagation (as implemented in openGM) is run for $bp_{\text{steps}}$. In each step, the current energy is evaluated. The configuration that achieved the lowest energy is then taken as the initial variable states for the Lazy Flipper algorithm, which runs up to a search-depth of $lf_{\text{max}}$. In fig. 2.25, $bp_{\text{steps}} = 10$, then (transition indicated by red vertical line) each new, lower energy found by the Lazy Flipper is taken as another step. The Variation of Information (sec. 2.11.1) for the test dataset is calculated for each energy. As expected, the VI (circles) reduces with falling energy.

The program `evaluate-graphical-model` (tab. 2.16) performs the above energy minimization. In a recent release of openGM, the `opengm` program was made available, which can be used to experiment with the other energy minimization algorithms that openGM offers.
evaluate-graphical-model

Input

--graphicalModel 'graphical-model-D.h5' Graphical Model
(build-graphical-model, tab. 2.15)

Output

--faceStates 'face-states-D.h5#face-states' vector $[0, l_{\text{max}}] \rightarrow \{0, 1\}$,
entry $q-1$ gives the inferred state for face $K_q^{(2)}$

Parameters

--bpSteps 50 number of belief propagation steps $b_p^{\text{steps}}$
--lfMax 3 maximum search depth for Lazy Flipper $l_f^{\text{max}}$

Table 2.16 – Command line options for the evaluate-graphical-model program. From Python use `denkc.Pipeline.run.evaluateGM()`

---

Figure 2.25 – Evolution of the energy $E(\{x_F\})$ (solid line) with successive optimizations steps. Additionally, the Variation of Information (VI) is shown (circles).

2.10 Running the segmentation pipeline

Sections 2.3 to 2.9 have shown how to run the segmentation pipeline. I have implemented each step in C++ (using existing libraries were possible) and different parallelization strategies, based on the Messaging Passing Interface, have been explored where execution time was critical. Additionally, data structures have been optimized to cope with very large datasets. All this work has made applying the segmentation algorithm from Andres et al. (2010d) to the entire $e1088$ dataset possible for the first time. With $2048 \times 1892 \times 2048$ voxels, it is roughly 64 times larger than the $512 \times 512 \times 512$ datasets that the older MATLAB implementation of the algorithm could handle. For debugging intermediate steps and exploring the final result – both in 2D and 3D – I have written a visualization tool, the subject of chap. 4.

Runtimes for segmentation of $e1088$ are summarized in tab. 2.17 (training data TD$_0$)
2.11 Validation

Validation was performed on a separate test set, a subset of $130 \times 130 \times 130$ voxels of \textit{e1088} (\texttt{sbfsem-e1088-unaligned}). Similarly to the training dataset, dense ground-truth voxel labels (\texttt{voxel-labels-e1088-aligned.h5}) were provided by the Max Planck Institute for Medical Research, Heidelberg (compare sec. 2.4.2).

Additionally, starting from a $\sigma$ produced with TD$_0$ (\texttt{seg-sbfsem-e1088-aligned-rf-0.h5}) Björn Andres has labeled 561 faces as excessive, 2243 as essential and left 1193 unlabeled.

2.11.1 Comparing segmentations

A ground-truth segmentation $\sigma^{gt}$ and an automatic segmentation $\sigma^a$ of the same data volume represent two different \textit{clusterings} of the voxel grid $G$. A distance metric from

### Table 2.17 - Runtime of the segmentation procedure on the $2048 \times 1892 \times 2048$ \textit{e1088} dataset using a shared $4 \times 4$-core machine with 128GB of RAM. The \textit{training data version 0} (TD$_0$) was used, resulting in 20,517,310 0-components, 37,473,318 1-components, 20,052,088 2-components, and 3,030,299 3-components. (*unparallelized version.

<table>
<thead>
<tr>
<th>program</th>
<th>time $TD_0$</th>
<th>CPUs</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>voxel-features-mpi</td>
<td>1d12h</td>
<td>10</td>
<td>$B = (150, 150, 150)$, defaults</td>
</tr>
<tr>
<td>probability-map-mpi</td>
<td>4d5h</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>segment</td>
<td>3h39min</td>
<td>1</td>
<td>$t_{\text{max}} = 0$, $s_{\text{min}} = 3$</td>
</tr>
<tr>
<td>cgpx</td>
<td>1h48min</td>
<td>16</td>
<td>$B = (150, 150, 150)$</td>
</tr>
<tr>
<td>cgpr</td>
<td>21h48min</td>
<td>1</td>
<td>SUPPRESS_03_SETS</td>
</tr>
<tr>
<td>face-feature-extraction *</td>
<td>9d14h</td>
<td>1</td>
<td>default</td>
</tr>
<tr>
<td>intersection-feature-extraction *</td>
<td>3d11h</td>
<td>1</td>
<td>default</td>
</tr>
<tr>
<td>classify-faces *</td>
<td>25min</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>classify-intersections *</td>
<td>41min</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>build-graphical-model</td>
<td>4h9min</td>
<td>1</td>
<td>$\alpha = 0$, $\beta = 1$, $\gamma = 0.5$, $E_h = 100$</td>
</tr>
<tr>
<td>evaluate-graphical-model</td>
<td>1d10h</td>
<td>1</td>
<td>$bp^{\text{steps}} = 100$, $l_f^{\text{max}} = 0$</td>
</tr>
</tbody>
</table>

and tab. 2.18 (TD$_1$). Note that with a different $\phi$ and slightly changed parameters for the seeded watershed, the number of 3-components varies by a factor of two!

Starting from the volume segmentation $\sigma$ shown in fig. 2.7c, the following steps of the algorithm are illustrated in fig. 2.26. Using the CGP algorithms, geometry and topology has been extracted from $\sigma$. For each face $F$, the Random Forest RF$_2$ predicts a probability $p_F$ for it to be essential (2.26b). A third Random Forest, RF$_3$, predicts for all 3-junctions the most probable configuration (2.26c). Often, the maximum-likelihood configurations of two 1-components $C$ that bound the same face $F$ disagree as to what label $F$ should get. In the figure, these are “lines” that change their color at places other than at 1-cells. Similarly, one can find 1-components in 2.26b where a hard thresholding of $p_F$ would result in forbidden configurations. This motivates the use of the Graphical Model which combines the \textit{probabilities} for individual faces and face configurations. Finally, the best (automatic) guess for a segmentation is 2.26d. Black boundaries signal a transitive merging error, this is discussed in sec. 2.11.4 (p. 63).
Figure 2.26 – Improving the initial over-segmentation.

a) Raw data (see fig. 2.7 and 2.1).
b) Face probabilities \( p_F \) are color coded.
c) Blue circles indicate 1-cells \( C \). Where \( |\Gamma(C)| = 3 \), the most probable configuration is indicated by coloring the three bounded faces either red or green. Where \( |\Gamma(C)| = 4 \), faces are colored black. In this 2D slice, faces appear as “lines” \( l_i = (C \leadsto C'_i) \), which each connect \( C \) to another 1-cell. Along these \( l_i \), color spreads only up to half their lengths.
d) Face states after energy minimization. Green faces are essential, red faces excessive; black faces indicate transitive deletion of an essential face.

The field of information theory, called the Variation of Information (VI, Meila, 2007), combines entropies and conditional entropy of two clusterings.

The entropy of a random variable \( X \) with support \( \mathcal{X} \),

\[
H(X) = - \sum_{x \in \mathcal{X}} p(x) \log p(x),
\]

is a measure of uncertainty.
2.11 Validation

**Figure 2.27** – Runtime of prediction pipeline on e1088 (using training data version 1).

<table>
<thead>
<tr>
<th>program</th>
<th>time</th>
<th>TD1</th>
<th>CPUs</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>voxel-features-mpi</td>
<td>1d12h1min</td>
<td>10</td>
<td></td>
<td>B = (150, 150, 150), defaults</td>
</tr>
<tr>
<td>probability-map-mpi</td>
<td>11h39min</td>
<td>16</td>
<td></td>
<td>B = (150, 150, 150)</td>
</tr>
<tr>
<td>segment</td>
<td>2h13min</td>
<td>1</td>
<td></td>
<td>(t_{\text{max}} = 1, s_{\text{min}} = 10)</td>
</tr>
<tr>
<td>cgpx</td>
<td>1h38min</td>
<td>16</td>
<td></td>
<td>B = (150, 150, 150)</td>
</tr>
<tr>
<td>cgpr</td>
<td>17h44min</td>
<td>1</td>
<td></td>
<td>SUPPRESS_03_SETS</td>
</tr>
<tr>
<td>twoset-bbox</td>
<td>15min</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>face-feature-extraction-mpi</td>
<td>2d14h29min</td>
<td>16</td>
<td></td>
<td>B = (256, 256, 256)</td>
</tr>
<tr>
<td>intersection-feature-extraction-mpi</td>
<td>1h25min</td>
<td>16</td>
<td></td>
<td>default</td>
</tr>
<tr>
<td>classify-faces-mpi</td>
<td>1min</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>classify-intersections-mpi</td>
<td>5min</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>build-graphical-model</td>
<td>2h30min</td>
<td>1</td>
<td></td>
<td>(\alpha = 0.9, \beta = 1, \gamma = 0.5, E_h = 100)</td>
</tr>
<tr>
<td>evaluate-graphical-model</td>
<td>1d10h28min</td>
<td>1</td>
<td></td>
<td>(b_{\text{steps}} = 50, I_{f_{\text{max}}} = 1)</td>
</tr>
</tbody>
</table>

**Table 2.18** – Runtime of the segmentation procedure on the 2048 × 1892 × 2048 e1088 dataset using a shared 4 × 4-core machine with 128GB of RAM. The training data version 1 (TD1) was used, resulting in 16 824 483 0-components, 30 120 737 1-components, 15 082 034 2-components, and 1 776 920 3-components.

Furthermore, one can define the conditional entropy as

\[
H(X|Y) = \sum_x p(x)H(Y|X = x)
\]

\[
= - \sum_x \sum_y p(y|x) \log p(y|x),
\]

which quantifies the remaining uncertainty in \(X\) after \(Y\) is known. If \(X\) and \(Y\) are not correlated, this reduces to the entropy of \(X\).
Then, the Variation of Information is defined as
\[ \text{VI}(X, Y) = H(X|Y) + H(Y|X). \]  
(2.45)

As a metric, VI is always non-negative, is 0 iff \( X = Y \), is symmetric and satisfies the triangle inequality.

For comparing segmentations, the discrete random variable \( X \) has as many states as the ground-truth segmentation has segments, namely \(|\mathcal{S}(\sigma_{\text{gt}})|\). Similarly, \( Y \) refers to the \(|\mathcal{S}(\sigma_{\text{a}})|\) segments of the automatic segmentation. Assuming any point \( x \in G \) is picked with equal probability, the probability of it belonging to segment \( \mathcal{K}_q^3 \) is the simple ratio of voxel counts
\[ p(\mathcal{K}_q^{(3)}) = \frac{|\mathcal{K}_q^{(3)}|}{|G|}. \]  
(2.46)

The probability that \( x \in G \) belongs to segment \( q \) in \( \sigma_{\text{gt}} \) and segment \( l \) in \( \sigma_{\text{a}} \) is the joint probability distribution
\[ p(\mathcal{K}_q^{(3)}, \mathcal{K}_l^{(3)}) = \frac{|\mathcal{K}_q^{(3)} \cap \mathcal{K}_l^{(3)}|}{|G|}, \]  
(2.47)
which is easily calculated from the confusion matrix of the segments.

In fig. 2.28, two approaches for reducing over-segmentation and their dependence on key parameters are compared:
1. A threshold \( t_{\text{rf}} \) on the face probabilities \( p_F \) results in an excessive label if \( p_F < t_{\text{rf}} \). Face \( F \) is then deleted and its bounded segments are merged.
2. Energy minimization of the Graphical Model yields a label for each \( F \) and thus a final segmentation. The mixture parameter \( \gamma \) weighs first-order against higher-order potentials.

As VI is a metric, smaller values denote more similar segmentations. It can be seen that approach 1 can be tuned to perform slightly better than the more complicated approach 2 involving higher-order potentials and difficult approximate energy minimization. Before any conclusions can be drawn however, there are other measures of similarity left to explore.

### 2.11.2 Comparing face labels

Another possibility for comparing two segmentations is to manually label the faces of the automatic segmentation \( \sigma_{\text{a}} \), and to compare the results of the later steps of the segmentation procedure with these labels. This assumes that \( \sigma_{\text{a}} \) exhibits no severe under-segmentation.

Again, the two approaches for reducing over-segmentation from above are compared in the following experiments.

As a quality measure the fraction of faces that have wrongly been classified as excessive (false removals, FR), the fraction of faces that have wrongly been assigned the essential label (false preservations, FP), and the total fraction of misclassified faces are computed. Fig. 2.29 and 2.30 show the results for the two different training datasets TD\(_0\) and TD\(_1\).
2.11 Validation

Fig. 2.28 – Variation of Information depending on $t_{rf}$ or $\gamma$.

Fig. 2.29a compares strategies 1 (left) and 2 (right). TD$_0$ (dense voxel labels, derived face labels) has been used. Ground-truth face labels have been derived from the dense voxel ground-truth as described in sec. 2.7.3. Similarly to the previous findings from sec. 2.11.1, both approaches can be tuned to perform equally well.

In fig. 2.29b only the ground-truth labels have been changed: Instead of deriving labels, faces were manually annotated directly. Interestingly, employing the Graphical Model now brings a real benefit.

To investigate the dependence on the training data, fig. 2.30 compares the two approaches when TD$_1$ (sparse voxel labels, manual face labels) has been used for training. Unfortunately no manual ground-truth face labels have been created yet for this setting; comparison to derived labels is shown. Again, the Graphical Model approach is the winner.

2.11.3 Sparse skeletons

Another form of ground-truth was provided by the Max Planck Institute for Medical Research, Heidelberg: sparse skeletonizations of neurons. A number of human tracers had the following instructions: start at the same cell, find the slice view to which the tubular structure is orthogonal and mark the middle of the cell with a vertex; go a few slices up or down the stack and repeat, connecting the vertices together. Wherever branching occurs, be sure to follow all branches. In this way, a skeleton of the neuron is constructed. Helmstaedter et al. (2010) describe how skeletons from multiple tracers for the same neuron can be consolidated into one. Compared to dense labeling of the intracellular space (as in sec. 2.4.2), tracing sparse skeletons is reported to be considerably faster, allowing for the creation of larger ground-truth volumes.

The 98 skeletons shown in fig. 2.31b have been used for validation of the segmentation procedure. Most skeleton vertices are contained within a subvolume “BNB” of $e1088$ at offset $o = (576, 576, 976)$, size $s = (242, 242, 242)$. Only a small fraction of all cells in this volume contain a skeleton. Thus the ground-truth is sparse in two senses: (1) for an individual cell, not all of its cell-interior is labeled and (2) not all cells are labeled.

For validating the final segmentation $\sigma^a$, the skeletons can be used to find errors. Let $\Omega = \{\omega_1, \ldots, \omega_{|\Omega|}\}$ be the set of all skeletons. Then a segment $S \in S(\sigma^a)$ arose due to
Segmentation of volume electron microscopy images for connectomics

(a) comparing with face labels derived from dense voxel labels

(b) comparing with manual face labels

Figure 2.29 – False Removals/False Preservations depending on key parameters. Segmentation pipeline run with training data version 0.

A wrong merger, when multiple skeletons intersect the segment:

\[ \text{wrong merger at segment } S \iff \left| \{ \omega \mid \omega \in \Omega \land \omega \cap S \neq \emptyset \} \right| > 1. \quad (2.48) \]

Similarly, when multiple segments intersect the same skeletons \( \omega \in \Omega \), the remaining over-segmentation is due to a number of false splits:

\[ \text{false splits at skeleton } \omega \iff \left| S \mid S \in S \land S \cap \omega \neq \emptyset \right| > 1. \quad (2.49) \]

In first experiments, I have implemented the above union operation by linear interpolation of every skeleton edge \( e = (x_1, x_2), x_i \in \mathbb{R}^3 \): on the voxel grid \( G \), a discrete 6-connected line is traversed (using the algorithm in Liu et al. (2008); see fig. 2.32b). Then for each voxel on this line, the union operation can be performed.

It turns out, however, that this approach is too strict: apparently the tracers sometimes spaced individual vertices \( x_1, x_2 \) too far apart (see fig. 2.32a), such that some bends in the neuron cause linear interpolation to intersect another, unrelated, segment. Therefore, only the vertices can be relied upon.

Using the interactive viewer (chap. 4) the reports of false splits and false mergers for the watershed segmentation \( \sigma \) can be examined. For BNB, the latter could be classified into the following cases:
2.11 Validation

Figure 2.30 – False Removals/False Preservations depending on key parameters. Segmentation pipeline run with training data version 1. Comparison with derived face labels is shown.

Figure 2.31 – Skeleton tracings, kindly provided by the Max Planck Institute for Medical Research, Heidelberg. See Helmstaedter et al. (2010) for a detailed description of the creation process, involving consolidation of skeletons from multiple tracers for the same cell.

- Skeletons extend a few voxels too far along a very thin branch, where it is very difficult – even for a human – to figure out the correct segmentation. The resolution limit is noticeable.
- Individual skeleton vertices are misplaced such that they just cross a watershed boundary.
- One neuron has two skeletons describing it, these should be merged.
- Under-segmentation is due to missing seeds or holes in the probability map; in some cases there appear to be staining problems in the data, and a human can just guess that a membrane should be there, in other cases, the used filter sizes are obviously not large enough to collect enough evidence for poorly visible membranes.

For BNB, there were about 50 such wrong mergers in the initial over-segmentation, of which only 3 to 5 turned out to be actual segmentation errors. It is feasible to show these cases to a human expert to improve the skeletons.
2 Segmentation of volume electron microscopy images for connectomics

(a) part of a skeleton, edges are shown in black, vertices in red
(b) a 6-connected 3D line representing the edge \( e = (x_1, x_2) \) on the voxel grid \( G \)

Figure 2.32 – Using linear interpolation on skeleton edges.

Because ground-truth is not perfect and sometimes the raw data is not interpretable with considerably more prior knowledge that a computer algorithm does not have, it is very difficult to quantify any errors the algorithm makes.

Using just the number of false mergers and false splits as an automatic quality measure is not an option: as not every neuron in the volume contains a skeleton, there are merging errors that cannot be detected. In the extreme case, the whole volume is made up of only one segment: this leads to one wrong merger (in which all skeletons are involved) but no wrong splits. On the other hand, if each voxel were an individual segment, there would be no wrong mergers, but a vast amount of wrong splits. Furthermore, it is unclear how the two types of errors should be weighed against each other. However, it might be possible to use measures from information theory for validation (see sec. 5.6 for evaluation with skeleton ground-truth for each cell in a different data volume). Whether the fact that only a fraction of all cells in BNB contains a skeleton would diminish these measures’ usefulness will have to be investigated.

Nevertheless, while tuning the parameters \( s_{\text{min}}, t_{\text{max}}, \alpha, \gamma \) and the training set for BNB, the number of false mergers and false splits have been helpful coupled with a visual inspection of the results. As BNB is more than eight times larger than the volume with dense voxel labels, the chosen parameters are more likely to be useful for segmentation of the whole \( e1088 \) dataset.

2.11.4 Visual inspection of results on \( e1088 \)

The training and test datasets for which manual labels are available are very small compared to the total size of the \( e1088 \) dataset. Can one be sure that they are representative of all data? Does parameter-tuning on the test dataset result in over-fitting? As neurons extend over the whole dataset, increasing the size of the dataset increases the number of faces that bound a certain neuron. Any wrongly labeled face merges two adjacent neurons. Does this mean one has to be more careful with removing faces on large datasets? For answers, one can turn to visual explorations of the results.

In fig. 2.33 and 2.34, a number of neurons from the final segmentation have been selected. Meshes were exported as text files containing the lists of vertices and faces,
Figure 2.33 – Reconstructed cells contained in the $e1088$ dataset of the inner plexiform layer of rabbit retina. Parameters from tab. 2.18 have been used; cells were manually selected based on size and position. See also fig. 2.34.

imported into Blender\textsuperscript{5}, and given random colors. In contrast, fig. 2.35 shows one such neuron from the final segmentation, where each $K_q^{(3)} \in S(\sigma)$ has been assigned a different color. Without examining the data in detail, it can be assumed from the picture that the over-segmentation could be reduced significantly. Under-segmentation is not visible: one would expect large deviations from the tree-like, thinly branched structure.

Finally, fig. 2.36 and 2.37 show results for the full $e1088$ dataset on one representative 2D slice. For this experiment, the parameters from tab. 2.17 have been used.

Boundaries drawn in black indicate faces $F$ for which $x_F = 1$ after energy minimization (that is the face is determined to be essential), but $\Gamma(F) = S_1, S_2$ yields two neighboring segments that have been merged (by deleting another separating face $F'$ somewhere else in the dataset). In many cases, this indicates an error.

For $\alpha = 0, \beta = 1$, the shown slice is awash with these transitive merging errors. Being a lot more conservative and setting $\alpha = 0.96$ (consider fig. 2.17) appears to fix many problems.

\textsuperscript{5}http://www.blender.org
Figure 2.34 – Reconstructed cells contained in the e1088 dataset of the inner plexiform layer of rabbit retina. Parameters from tab. 2.18 have been used; cells were manually selected based on size and position. Fig. 2.33 shows the same data, but from another point of view.
Figure 2.35 – A reconstructed neuron from \textit{e1088}. Original watershed segments are colored randomly. For used parameters see tab. 2.18.
Figure 2.36 – original data (above) and probability map (below)
Figure 2.37 – final result for $\alpha = 0$, $\beta = 1$ (above) and $\alpha = 0.96$, $\beta = 1$ (below). In both cases $\gamma = 0.5$. 
3 Towards automatic synapse detection: geometric features

While the focus of chap. 2 has been the correct segmentation of all neurons in large electron microscopy volume images, the focus now turns to the detection of synapses, an equally important topic as a circuit diagram with only disconnected “wires” is of little use.

Chap. 1 provides a brief overview of the biology of the nervous system; in particular one focus has been how synapses look like in electron microscopy images (sec. 1.2). Their appearance heavily depends on the type of staining used:

- With traditional staining, the postsynaptic density (PSD) is visible as a thickening of the postsynaptic membrane. Even the untrained eye can find many synapses by scanning 2D slices for elongated, diffuse sites where the intensity falls of perpendicular to the membrane direction. Vesicles, appearing as small spherical objects near the PSD, give additional clues.
- Surface staining is ideal for tracing neurons, as almost all “clutter” inside the cells is rendered invisible, and the segmentation problem can be approached as a two-class problem (classes extra-cellular and intra-cellular, sec. 2.4.1). The downside is that synapse detection becomes more difficult. Vesicles are not visible at all, and the postsynaptic density is much less pronounced.

![Figure 3.1 – Synapse candidate: cells at $x_1 = (155, 101, 108)$ and $x_2 = (164, 101, 138)$ in the BNB dataset. In the shown $x$-slice of the raw data (position indicated by semi-transparent plane in 3D), the two shown objects are visible in the center.](image)
It might, however, be possible to find other discriminative features for synapse detection. In this chapter, I will explore three such candidate features. Hopefully these features could be used in a supervised learning setting to find synapse candidates. This is left for future work.

3.1 Membrane curvature

In Ziff (1997), the postsynaptic density is attributed to be “gently curved” and to “often resemble a concave disk” as it follows the contour of the spine head. This observation motivates the computation of geometric features from a surface representation of cell membranes. In particular, measures of surface curvature might be able to capture the “gently curved” and “concave” properties described above.

3.1.1 Definition of surface curvature

Local curvature of a surface. Let $S$ be a 2-dimensional surface embedded into $\mathbb{R}^3$. The curvature at $p \in S$ can be described as follows. The surface normal vector $U(p)$ and any tangential vector $u_p$ at $p$ span a plane $\Pi(u_p, U(p))$. The intersection of this plane with the surface $\Pi(u_p, U(p)) \cap S$ yields a 1-dimensional curve. Its curvature $k(u_p)$ is just the second derivative of the curve evaluated at $p$. Among all directions $u_p$ on the tangential plane, the ones with smallest and largest $k(u_p)$ are called the principal directions $u_1$ and $u_2$ with the principal curvatures $k_1$ and $k_2$. An important result from differential geometry (Gray, 1994, chap. 14) is that $u_1$ and $u_2$ can be computed as the eigenvectors of the Weingarten map, and thus $u_1 \perp u_2$. The eigenvalues are $k_1$ and $k_2$.

Gaussian and mean curvature. Measures of the local curvature at $p \in S$ are the Gaussian curvature $K = k_1 k_2$ and the mean curvature $H = 1/2(k_1 + k_2)$. Fig. 3.2 shows a saddle-like surface with negative Gaussian curvature. Both measures are rotation invariant and therefore commend themselves as local surface features.

Figure 3.2 – Illustration of principal curvature. The function $f(x, y) = x^2 - 1/2 \cdot y^2$ has principal curvatures $k_1 = -1$ and $k_2 = 2$ at $(x, y) = (0, 0)$. 
3.1 Membrane curvature

![Gaussian curvature](image-a)

![Mean curvature](image-b)

![Mean curvature from a different perspective](image-c)

**Figure 3.3** – Synapse candidate: cells at \( \mathbf{x}_1 = (162, 100, 126) \) and \( \mathbf{x}_2 = (156, 101, 109) \) in the BNB dataset. The range of the colormap is from the 0.05 to the 0.95 quantile.

3.1.2 Estimating the curvature

Given a segmentation \( \sigma \), the goal is to estimate the surface curvature for all points on all surfaces \( F \in \mathcal{F} \). An important consideration is the representation of the membrane surface. Using CGP, the triangle mesh for any 2-component can easily be obtained with the algorithm presented in sec. 2.6.2. However, the segmentation \( \sigma \) is constrained to the voxel grid, resulting in a staircasing artifact which adds high-frequency noise to the true surface. One approach is to go back to the original data and re-fit the mesh to subpixel precision, another to disregard the underlying data and smooth the mesh, for example by relaxation. Alternatively, if the curvature is estimated from a large enough neighborhood, the voxel-resolution mesh might already be sufficient. The last approach is used in the following.

Lachaud and Vialard (2003) and Fourey and Malgouyres (2009) describe how to estimate surface normals directly from the discrete voxel-resolution mesh called a *digital surface* by 2D contour analysis or fast on-surface convolution. Further work by Fourey and Malgoures (2008) shows that the surface curvature can be estimated similarly, but low accuracy and inconsistency in near-by values are reported. An advantage of these methods is speed due to the discrete nature of the digital surface, but they would have to be abandoned if sub-voxel accurate surfaces are considered.

Cazals and Pouget (2005) describe how local differential quantities can be estimated at a point \( p \) on a surface by fitting a truncated Taylor expansion. The surface can either be represented by a mesh or by a point cloud. There is no restriction to the voxel grid. In a coordinate system (origin \( p \)) with its \( z \)-direction not belonging to the tangent space,
the surface can be locally written as a height function $f(x, y)$ with Taylor expansion up to order $n$:

$$f(x, y) = \sum_{k=0}^{n} \sum_{j=0}^{k} B_{k-j, j} x^{k-j} y^j + \mathcal{O} \left( \|(x, y)\|^{n+1} \right).$$

(3.1)

A truncated Taylor expansion up to order $n$ is called a $n$-jet. When the linear terms vanish, the $z$-direction corresponds to the normal direction of the surface, and the coordinate system is said to be principal. When, in addition, the $x$- and $y$-axes correspond to the principal directions $u_1$, $u_2$ of the surface at $p$, the Taylor expansion has the simple form

$$f(x, y) = \frac{1}{2} (k_1 x^2 + k_2 y^2) + \mathcal{O} \left( \|(x, y)\|^3 \right),$$

(3.2)

where $k_1$ and $k_2$ are the principal curvatures. Such a coordinate system is called the Monge frame.

The algorithm in Cazals and Pouget (2005) works as follows. First $N$ points around $p$ have to be collected. If a mesh representation of the surface $S$ is available, these can be gathered by visiting neighboring vertices by going in rings of increasing radius around $p$. If a point cloud describes $S$, some nearest neighbor variant can be used.

A principal component analysis of the gathered points $P$ is used to obtain a first approximation of the surface normal. A local coordinate with the $z$-axis in the direction of the eigenvector with the smallest eigenvalue is called the fitting basis $F$. The points $P$ are expressed with respect to $F$. A least-squares approximation estimates the coefficients $B_{i, j}$ for an $n$-jet. Finally, the fitted height function can be transferred into the Monge frame to give the estimated local differential properties such as surface normal, principal directions and principal curvatures for $n = 1$ and $n = 2$, respectively.
3.1 Membrane curvature

3.1.3 Implementation of curvature estimation

I have used the implementation of curvature estimation via polynomial fitting as implemented (Cazals and Pouget, 2007) in the Computational Geometry Algorithms Library\(^1\) (CGAL).

The proof-of-concept implementation is based on the Jet\_Fitting\_3 example shipped with CGAL which uses a half-edge data structure to quickly gather points in rings around a given \(p\). The prerequisite for using such a data structure is that the mesh represents an orientable surface (a well-known example of a non-orientable surface is the Möbius strip, see Gray 1994, chap. 12). Each face (a triangle in our case) is described by vertices and edges connecting vertices. In the half-edge data structure, the face normal defines a direction of circulating around the face. All edges connecting two vertices are directed; the direction is aligned with the circulating direction of the face. There are at most two edges connecting any two vertices, and both have opposite directions (fig. 3.6a).

With digital surfaces, however, the orientable property does not always hold as is illustrated in fig. 3.6b. A related property for binary volume images is well-composedness. In a well-composed image (Latecki, 1997) the 6-neighborhood and 26-neighborhood are equal and configurations such as fig. 3.6b where two 6-connected components are connected only via a voxel edge or the configuration where two 6-connected components are connected only via a voxel corner do not occur. For a well-composed binary image

![Image](image.png)

**Figure 3.5** – The standard deviation of the gray values along the surface normal is shown (symmetric). The range of the colormap is from the 0.05 to the 0.95 quantile. The same objects as in fig. 3.3 are shown.

\(^1\)http://www.cgal.org
Towards automatic synapse detection: geometric features

Figure 3.6 – Problems when using a half-edge data structure to represent digital surfaces.

of an object, the surface is orientable and can thus be represented with a half-edge data structure.

Tustison et al. (2007) offer a randomized algorithm for repairing multi-label volume images as an extension to the Insight Tool Kit\(^2\). On a 242\(^3\) subset of \(e1088\), this algorithm needed 2.8 days to make all \(F \in \mathcal{F}\) of a given watershed segmentation \(\sigma\) well-composed. Additionally, CGP has been extended (p. 2.6.3) to also report a consistent sign of the normal vector for each voxel face \(f \in F\), so that the half-edge data structure can be built correctly.

3.2 Membrane size

The area of contact at a synapse might make for a discriminative feature. As feature that is rapidly computable, it could be used to exclude surfaces that are either too small or too large from the set of synapse candidates. Other features, such as the more expensive curvature (sec. 3.1) or surface normal gray value (sec. 3.3) features would then only be calculated for the remaining candidates.

A first approximation of the surface area of \(F \in \mathcal{F}\) is the number of voxel faces it consists of. For a more accurate estimate, one can turn again to digital geometry (Lachaud and Vialard, 2003; Windreich et al., 2003). If a sub-voxel accurate mesh is available, it is easiest and sufficiently accurate to just sum the areas of all triangles.

However, it is unclear if surfaces from a watershed segmentation can be relied upon to represent neither more nor less than the area of “contact” at the synapse. The error in surface estimation is likely irrelevant in this context.

3.3 Surface normal gray value statistics

In traditionally stained electron microscopy images, the postsynaptic density is visible as a thickening of the postsynaptic membrane with added diffusiveness. Even though in surface-stained samples like \(e1088\) this effect is much less pronounced, still, there is hope that a classifier like the Random Forest could make use of an appropriate feature.

\(^2\)http://www.itk.org
To capture the change in membrane thickness, I propose to make use of the vertex normals estimated in sec. 3.1.2. Starting from a vertex, interpolated gray values are sampled along the normal direction $\mathbf{n}$ in equidistant steps up to a maximum distance $d_{\text{max}}$ from the surface. The same is done for the opposite direction $-\mathbf{n}$ for a symmetric measure. Fig. 3.7 illustrates the above sampling range.

A number of statistics are computed for the picked-up gray values: minimum, 0.25 quantile, median, mean, 0.75 quantile, maximum, standard deviation and variance. Fig. 3.5 shows the standard deviation for a segment possibly exhibiting a synapse site. Especially striking are the red “borders” surrounding individual surfaces $F \in \mathcal{F}$ (which together make up the shown mesh). This is understandable, as the normal vectors happen to lie within another membrane which connects to the object at that location.

The extraction of these surface normal gray value features uses a spline interpolation of the volume containing the surface (class $\text{SplineVolumeView}$, Ullrich Köthe, private communication).

3.4 Outlook

For visual exploration, I have extended the $\text{SegmentationViewer}$ program (chap. 4) to be able to show Gaussian curvature, mean curvature and all statistics of the surface normal gray values. Hovering over any object shown in the 3D view and pressing $p$ brings up a context menu as shown in fig. 3.8.

Unfortunately, there are no ground-truth labels for synapses in the $e1088$ dataset, thus it was not possible to investigate the usefulness of the above geometric features. In future work, it would be interesting to calculate these features for an EM volume image of neural tissue with traditional staining, such as the dataset from chap. 5. The surface normal gray value features in particular might be more powerful in such a setting, and even non-experts could generate ground-truth labels.
Towards automatic synapse detection: geometric features

Figure 3.7 – Above: one surface extracted by CGP. The colormap shows a gray-level statistic along the estimated surface normal at each vertex. For a one-dimensional cut, the normal vectors are shown as green line segments, their extent visualizes the range in which gray values are picked up.

Below: Along the one-dimensional cut, gray values along the normal direction are color- and elevation-coded (left); another view of the same information (right) reveals variations in the surface thickness (visible in the middle of the cut).

Figure 3.8 – Synapse features can be displayed using the SegmentationViewer program by hovering over any displayed segment and pressing p.
4 Visualization of intermediate steps and results

Visualizing different aspects of any algorithm can provide important insights into its inner workings (in which cases does the algorithm succeed, in which cases does it fail?), help with debugging (do the results look reasonable?), proofreading (where does the result differ from what a human would do?) and conveying its functioning principles to others.

This chapter describes the motivation, implementation and usage of the tool SegmentationViewer, an interactive volume viewer I have written to visualize almost every step of the segmentation pipeline presented in chapter 2.

4.1 Design considerations

2D interaction. After obtaining a large three dimensional volume dataset, the first thing to do is viewing the raw data. Software for this purpose usually presents the user with three slice views, orthogonal views of the image data, showing the $yz$, $xz$ and $xy$ image planes with normal directions of $x$, $y$ and $z$, respectively. Navigating works by going up or down these stacks one or several images at a time. When pinpointing a specific position $p = (x, y, z)$ given in image coordinates, each slice view is updated to show the respective image. By considering all three views at once, the user can get an idea of the 3D geometry around $p$.

When applying image processing algorithms to the raw data, intermediate results such as voxel features, the response of a classifier for each voxel or the final segmentation result are represented as data volumes with the same extent as the original volume. They can be displayed by overlaying the raw data with a grayscale or color representation. Toggling between raw data and overlay allows to quickly grasp how the raw data gave rise to these (intermediate) results in the processing chain.

3D interaction. In the setting of a segmentation problem, one also wants to examine a selected subset of the resulting objects in 3D to get a better understanding of their geometry and relative position amongst each other. In this way, it is for example possible to find synapse candidates. Desirable features include free rotation and zooming ability and the indication and modification of the current position $p$ by displaying or dragging orthogonal planes representing the slice views.

Challenges. Displaying large SBFSEM or FIBSEM volumes is challenging because of their size. Each voxel in the $e1088$ dataset is stored with 8-bit precision, summing up to approx. 8GB. With only a moderate increase in lateral resolution or extent of the imaged specimen region, or a reduction in slice thickness the size of these datasets increases with the third power. Already today, acquired EM volume datasets reach several hundred gigabytes in size and while an increased resolution of up to $4 \times 4 \times 4$ nm$^3$
(G. Knott, personal communication) will possibly allow for improved automatic segmentation, the explosion in storage space required will pose an even greater challenge in the feature.

Size is an important issue for displaying segmented 3D objects as well. There are two main ways to interactively render 3D objects: volume rendering or surface rendering.

In **volume rendering** the segmented object is indirectly represented by the set of voxels with the object’s index in a volume labeling $\sigma$. Every voxel not belonging to any object is assigned a 0. A technique called **ray-casting** simulates light rays falling onto $\sigma$, passing through transparent 0-valued voxels, while being attenuated, reflected or otherwise changed when hitting or passing through an object. The required calculation can be performed on the GPU, provided that $\sigma$ fits into memory, or on the CPU, which is prohibitively slow. In the V3D software, (Peng et al., 2010), volume rendering of a 3GB dataset on the GPU leads to a framerate of 5 frames per second. Given their performance graphs, it is doubtful that e1088 could be easily visualized this way using off-the-shelf software components (compression and level-of-detail representations might solve the problem, but take a long time to program).

Before the recent trend of putting hundreds of **general purpose** processing cores on a GPU, the graphics card’s main purpose was to render large number of triangles forming the surface representation (**mesh**) of objects. The mesh representation of the segmented objects can be built in a preprocessing step, for example by the **Marching Cubes** algorithm (used for example in the software Omni, Warne, 2009) or by using **CGP** (Andres et al., 2010b, see also chap. 2.6). The main performance concern is the number of triangles produced. Both Marching Cubes and CGP will generate two triangles per voxel, which is too much for highly branched neurons with a large surface area exhibiting additional noise. **Mesh reduction** can be employed to reduce the mesh size at a trade-off in quality and correctness.

It is desirable for any visualization software to be able to run on an average office machine or laptop. These machines are distinguished by a low amount of random access memory (RAM) of approx. 2GB and a graphics card with an upper limit of a hundred MB memory. As the complete dataset does not fit into main memory, some form of **caching** will be required for the program to stay responsive, and mesh reduction must be used to limit mesh sizes.

**Related Work.** Of course this research project is not the first in need of visualization of large volume images. Peng et al. (2010) have written V3D, a general-purpose 3D visualization tool for biological image datasets. Unfortunately, the closed source nature of this program would have made it difficult to introduce the changes necessary to visualize all intermediate steps of our segmentation procedure.

Researchers at MIT, working on the same e1088 SBFSEM data, have written **Omni**, specifically designed to allow scalable visualization and editing of connectome data, both in 2D and 3D (Shearer, 2009; Warne, 2009). Omni is very similar to **SegmentationViewer**, with a higher emphasis on scalability and performance, employing image pyramids and depth of field techniques to improve rendering times and reduce triangle count. However, the program is not available for general use.

**TrakEM2** is a plugin for **Fiji**, written for manual segmentation of EM image stacks (Cardona, 2010b).

Taking into consideration that any existing tool would need considerable customiza-
tion, and the lack of open-source alternatives, I decided to implement a custom-designed visualization application based on open source toolkits. The goal was to be able to

(i) interactively explore the full e1088 dataset
(ii) visualize intermediate steps of the segmentation algorithm from chap. 2
(iii) explore the segmentation by interactively selecting objects and showing them in 3D.

When datasets become even bigger in the future, the performance of SegmentationViewer will scale worse than for example Omni. Therefore, its functionality should be merged into superior tools once they become available.

### 4.2 Usage of SegmentationViewer

<table>
<thead>
<tr>
<th>Table 4.1 – keyboard shortcuts for SegmentationViewer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slice Views:</strong></td>
</tr>
<tr>
<td><strong>Navigating the Slice View</strong></td>
</tr>
<tr>
<td>wheel</td>
</tr>
<tr>
<td>wheel drag</td>
</tr>
<tr>
<td>left click + f</td>
</tr>
<tr>
<td>↑</td>
</tr>
<tr>
<td>↓</td>
</tr>
<tr>
<td>←</td>
</tr>
<tr>
<td>→</td>
</tr>
<tr>
<td>Alt+(↑ or ↓)</td>
</tr>
<tr>
<td>Ctrl+(↑ or ↓)</td>
</tr>
<tr>
<td>Ctrl+Alt+(↑ or ↓)</td>
</tr>
<tr>
<td>right click</td>
</tr>
<tr>
<td><strong>Working with Overlays</strong></td>
</tr>
<tr>
<td>right click</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>left click</td>
</tr>
<tr>
<td>n,m</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>Ctrl+d</td>
</tr>
<tr>
<td>g</td>
</tr>
</tbody>
</table>

*Continued on next page*
Visualization of intermediate steps and results

Manual Face Labeling

- right click cycle through face states (essential, excessive, unlabeled)
- l save user face labels

3D View:

Navigating the 3D View

- wheel zoom
- wheel drag pan
- Ctrl + left drag rotate
- x,y,z toggle slice plane visibility
- a cycle through pre-computed mesh decimation strategies

Working with shown Objects

- n,m decrease/increase opacity of 3D meshes
- d toggle between coloring each watershed segment individually or giving the merged supervoxels the same color
- Ctrl+d generate new random colors for segments
- g generate new random colors for skeletons
- a display different mesh simplification strategies
- Ctrl+s export view to a set of x_y.neuro files for importing into Blender
- hover + p pick an object and show its context menu

Starting. SegmentationViewer assumes that the complete segmentation pipeline (chapter 2) has been run as described in section 2.10. Additional pre-processing is used for offline generation of meshes and optimization of volume data layout for faster slice access. The resulting HDF5 files follow a fixed naming scheme, which can be derived from the original raw data file that the pipeline was run on. The path to this file is a required argument (`--path`); additionally the maximum amount of memory that should be used for caching chunks of raw data and different overlays needs to be specified (`--GBcache`). All other parameters are optional, see tab. 4.2.

User Interface. Fig. 4.1 shows a screenshot of the running program. Each slice view as well as the 3D view are shown in a separate window.

The current coordinate \( p \) is displayed in the upper left corner of the slice view, while the 2D position \( q \) of the mouse pointer is shown in the lower left corner (or “Off Image” if the cursor is outside the image). The locking mode (toggable via the context menu that appears on right-click over any slice view) determines whether the coordinate \( p \) should immediately reflect any changes in \( q \) (unlocked) or not (locked). Without locking, trying to understand 3D geometry is easier, as all slice views update on moving the mouse around. However, for large datasets, data transfer speeds may become too slow, resulting in an unresponsive application. Therefore, it is advisable to use the locked mode in these circumstances. Changing the position \( p \) is still possible by using the context menu.

Most of the functionality is driven by keyboard shortcuts and context menus. Apart from coordinate synchronization, the slice views do not share a common state; overlays are toggled and manipulated for each view separately. It is therefore important to give focus to the correct window. All keyboard shortcuts are listed in tab. 4.1.
4.2 Usage of *SegmentationViewer*

Figure 4.1 – Screenshot of *SegmentationViewer* showing the original data (x view, upper left), the voxel boundary probability (z view, upper right) and seeds with a random color per connected component for the marker-based watershed algorithm on the lower left. The 3D view indicates the position of the slice views within the $130 \times 130 \times 130$ SBFSEM volume.

**Slice Views.** Modifying the 2D view is possible by mouse wheel scroll (*zooming*) and mouse wheel drag (*panning*). To move a left-clicked position to the center of the window, press $\text{[Enter]}$. Use the arrow keys to go *up or down* the image stacks in steps of $1, 5, 10, 100$ or $500$ slices.

The user has the choice of two different *data views*

(i) the raw data
(ii) the voxel intra-cellular probability (toggle: key $\text{[F]}$).

The data view always has full opacity and is below any additional overlays. It can be *overlaid* by

(1) the seeds for the marker-based watershed segmentation, displayed with a random
Visualization of intermediate steps and results

(2) the intermediate watershed segmentation or final segmentation after merging, displayed with a random colormap, showing all segments
(3) same as 2, but only coloring selected objects.

The opacity of these overlays is adjustable by keys $\text{[1]}$ and $\text{[2]}$. Often two adjacent seeds or segments receive similar random colors, making it hard to make out whether they are merged or not. It is therefore possible to generate a new random colormap by pressing $\text{Ctrl} + \text{d}$.

The boundary overlay indicates the faces between the watershed superpixels, which appear as one-dimensional region boundaries in 2D. $\text{[y]}$ toggles the boundary overlay. There are three possible types:

(i) confidence of the Random Forest RF$_2$ in the classification of faces as essential or excessive is color-coded as green over black to red
(ii) the binary face states after minimization of the Graphical Model energy function are color-coded as green (essential) and red (excessive)
(iii) the user can interactively label the faces as essential or excessive. See “Labeling faces” below.

3D view. The $xyz$ axes indicator in the lower left corner of the 3D view window in fig. 4.1 shows the normal vectors of the $yz$, $xz$ and $xy$ slice planes, respectively.

Navigating inside the 3D view can be performed with a mouse wheel scroll (zooming) and mouse wheel drag (panning). Dragging while pressing the left mouse button rotates the objects, pressing $\text{[z]}$ in addition rotates with the rotation axis normal to the viewing window. To fly to a left-clicked position press $\text{f}$.

The current position of the three slice views is represented by semi-transparent slice planes, which can be dragged through the volume. All views are updated accordingly. The planes can be toggled by $\text{x}$, $\text{y}$ or $\text{z}$ for better visibility.

In fig. 4.2, the 3D view shows three objects. Their opacity is changed by the $\text{[1]}$ and $\text{[2]}$ keys, which is particularly useful when working with skeletons. $\text{Ctrl} + \text{m}$ exports all currently shown objects as a set of custom text files. One file is exported for each watershed segment $S \in S$. The filename $x\_y$.neuro indicates the label $x$ of the merged objects that $S$ belongs to and the label $y$ of the segment $K_y^{(3)} \in S(\sigma)$. The file content gives the object’s color and the lists of all vertex position and triangle defining index triples. A custom import script was written to recreate the shown 3D scene from this set of files in the 3D animation program Blender.

Merging superpixels. Removing excessive faces from the over-segmentation produced by the marker-based watershed transform is an essential step of the segmentation procedure. The merging strategy to be examined is selected on startup (tab. 4.2) and neighboring watershed segments are merged together if the face separating them was removed.

The three possible merging strategies are:
1. do not remove any faces (to examine the watershed segmentation)
2. remove those faces for which the RF$_2$ probability to be essential is less than a given threshold
3. remove those faces that are labeled as excessive after energy minimization of the Graphical Model.
4.2 Usage of **SegmentationViewer**

**Figure 4.2** – **SegmentationViewer** shows the same data as in fig. 4.1, but here with boundary overlays. From left to right: RF\(_2\) predictions, binary face states after GM energy minimization and face labeling mode. Three selected objects are shown in the 3D view.

**Figure 4.3** – Using **SegmentationViewer** together with sparse ground-truth in the form of skeletons. For one skeleton, all intersected watershed superpixels are shown. See also sec. 2.11.3.

Showing selected objects in 3D or displaying a segmentation overlay in 2D always refers to the *merged objects*. By pressing \(\text{d}\), the user can toggle between two coloring modes:

(i) give each merged object a random color

(ii) give each original watershed segment a random color.

This way it is easy to find merging errors. The coloring mode applies to all 2D and 3D views.

**Labeling faces.** To obtain reliable ground-truth for the faces between watershed segments, **SegmentationViewer** contains a *labeling mode*, accessible by enabling the boundary overlay \(\text{c}\), and switching to the labeling mode by repeatedly pressing \(\text{x}\). Boundaries are colored: black indicates an unlabeled boundary, while the labels essential boundary and excessive boundary are shown in green and red. Right clicking on a boundary cycles through the two possible labels and the unlabeled state.

With the switch --userLabels an initial uint8 vector of face states can be loaded on startup (1: unlabeled, 2: excessive, 3: essential). The current face labels are saved with key \(\text{l}\) as user-face-labels.h5, dataset face-labels with the same encoding.

**Labeling intersections.** To train the Random Forest classifier RF\(_3\), intersection labels can be derived from the face labels given above. When a 3-junction has only one face labeled as essential and the other two as excessive, all three faces and the intersection
4 Visualization of intermediate steps and results

<table>
<thead>
<tr>
<th>Required options</th>
<th>URL</th>
<th>path to the raw data $d$ on which the segmentation pipeline was run</th>
</tr>
</thead>
<tbody>
<tr>
<td>--path</td>
<td>URL</td>
<td>path to the raw data $d$ on which the segmentation pipeline was run</td>
</tr>
<tr>
<td>--GBcache</td>
<td>int</td>
<td>maximum amount of memory used for caching raw data and 2D overlays (in GB)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manipulating the view</th>
<th>URL</th>
<th>file giving labels of segments to be shown initially; one per line</th>
</tr>
</thead>
<tbody>
<tr>
<td>--showSegments</td>
<td>URL</td>
<td>file giving labels of segments to be shown initially; one per line</td>
</tr>
<tr>
<td>--initialCoordinate</td>
<td>int[3]</td>
<td>coordinate on startup</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skeletons</th>
<th>URL</th>
<th>path to skeleton file</th>
</tr>
</thead>
<tbody>
<tr>
<td>--skeletons</td>
<td>URL</td>
<td>path to skeleton file</td>
</tr>
<tr>
<td>--showSkeletons</td>
<td>int[n]</td>
<td>initially show skeletons given by an index list</td>
</tr>
<tr>
<td>--dataBbox</td>
<td>int[6]</td>
<td>specify the offset (first triple) and extent (second triple) of $d$ with respect to the origin of the skeletons</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Merging of supervoxels</th>
<th>string</th>
<th>one of mergeNone, mergeWhenFacesOffGM, mergeRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mergeMode</td>
<td>string</td>
<td>one of mergeNone, mergeWhenFacesOffGM, mergeRF</td>
</tr>
<tr>
<td>--thresh</td>
<td>float</td>
<td>threshold value $t_{RF}$ for mergeRF mode (see p. 58)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Face labeling</th>
<th>URL</th>
<th>path to HDF5 file containing labels for each face in the watershed segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>--userLabels</td>
<td>URL</td>
<td>path to HDF5 file containing labels for each face in the watershed segmentation</td>
</tr>
</tbody>
</table>

Table 4.2 – Command line options for SegmentationViewer. ‘int[n]’ denotes a space-separated list of $m$ entries of type int, enclosed by quotes.

are set to unlabeled, because isolated edges cannot appear in images of neural tissue. The derived intersection labels are saved as user-intersection-labels.h5, uint8 dataset intersection-labels (0: (000), 1: (001), 2: (010), 3: (011), 4: (100), 5: (101), 6: (110), 7: (111), 8: unlabeled, 9: curve has four adjacent faces).

Skeletons. When sparse ground-truth in the form of traced skeletons is available, they can be displayed in 3D (see fig. 4.3) and 2D (points of intersection of skeleton and slice plane). The skeleton data file and relative position relative to the data volume is specified on startup (tab. 4.2). Each skeleton is assigned a new random color by pressing $g$.

4.3 Implementation details

SegmentationViewer was implemented in C++ using the open source VTK (Schroeder et al., 2006) and Qt\(^1\) cross-platform toolkits.

As seen in sec. 4.1, memory requirements necessitate partial loading of volume datasets, and application responsiveness requires a caching strategy:

(i) a redundant copy of all volume datasets is made for each slice view, with optimized memory layout

\(^1\)http://doc.qt.nokia.com
4.3 Implementation details

Figure 4.4 – Illustration of possible memory layouts of 3D volume data. The $2 \times 2 \times 2$ cube is represented as 16 consecutive values in linearly addressable memory. Depending on the orientation of the $2 \times 2$ slice to extract, memory access patterns are good (green) or bad (red) with regard to hard disk seek times and CPU cache misses.

(ii) every slice view maintains its own cache
(iii) when a particular slice is requested, additional slices from above and below in the stack are loaded.

For the fast display of 3D objects and to alleviate the burden on the graphics card
(i) meshes for all watershed superpixels are pre-computed using the CGP library
(ii) meshes are decimated
(iii) meshes are smoothed for better visual quality.

4.3.1 Pre-processing

Volume Data. Fig. 4.4 illustrates possible memory layouts of a 3D data volume that is represented as a linear array in address space. It is clear that slice access in the three different dimensions cannot be equally efficient: It is crucial that the data to be loaded is consecutive in memory. When stored on disk, this reduces the disk seek time, and both disk and RAM use high-speed caches that build on the assumption of data locality. With three redundant copies of the same data it can be ensured that the three slice views can load slices as fast as possible. Writing an according permutation of raw data, voxel boundary probability map, watershed seeds and watershed segmentation is done as a pre-processing step. In Python, the function `denkc.Pipeline.run.xyzViews()` writes corresponding HDF5 files with filenames postfixed with $x$, $y$ or $z$ to indicate for which slice normal the memory layout is optimal.

Collecting 2-sets. In the next step, 2-sets are collected in order to be able to build meshes for each individual watershed segment. Usually, such a 3D segment $s$ is bounded by hundreds of individual faces (fig. 2.23), which can be found by querying the CGP geometry data structure with `GeometryReader::boundedBy`. Let $F = \{ f | f \text{ bounds } s \}$ be the set of all bounding faces and $T = \{ t \in \tau | t \in f \land f \in F \}$ the set of all topological 2-cells that bound $s$. `MeshReader::meshThreeSet` first constructs the set $F$, then loads the 2-component for each $f \in F$, and finally builds $T$ by union of all loaded 2-component cell lists.

The command line program `export-threesets` writes an HDF5 file with a dataset for each individual watershed segment containing the set $T$. From Python, the function `denkc.Pipeline.run.exportThreesets()` can be used. The program uses simple MPI parallelization (master-worker scheme 2, p. 12).
Constructing Meshes. After having obtained the sets $T$ for each segment in the watershed segmentation $\sigma$, the command line tool `export-meshes` finally builds a mesh representation.

Using alg. 4 (p. 26), a mesh $M$ is constructed from $T$ as a tuple $M = (V, F)$ of vertices (position in 3D) and faces (indices into the vertex list).

For each $M$, the aim is to reduce the size of the mesh as well as make its appearance less blocky. Making use of VTK’s in-built mesh processing algorithms, the following strategies are evaluated:

1. leave the mesh unchanged
2. use a `vtkSmoothPolyDataFilter` (10 iterations, feature angle 120°, edge angle 160°, relaxation factor 0.25), but no mesh simplification
3. on top of the smoothing filter of strategy 2, apply a `vtkQuadricDecimation` decimation filter
4. use a `vtkDecimatePro` filter (target reduction 1%, but preserve topology)
5. use a `vtkDecimatePro` filter (target reduction 5%, but no topology preservation)
6. use a `vtkQuadricClustering` filter (cluster bins set to $5 \times 5 \times 5$ voxel cubes).

Fig. 4.6 compares these strategies visually for a small object. Care should be taken with the interpretation: it is not clear how these methods scale with size, and of course the amount of mesh geometry and topology degradation the user is prepared to accept depends on the current distance of the object from the viewer. Therefore, different methods should be used depending on the required amount of detail, for example in a framework such as presented by Warne (2009). `export-meshes` computes all of the above decimation strategies and leaves the choice of finding the “best” one to the user. In `SegmentationViewer`, key $\mathbb{E}$ inside the 3D view window cycles through all above strategies (the total triangle count is displayed in the lower left corner of the window). The decimation strategy can also be changed for each object individually by using the object’s context menu that appears on mouse hover and $\mathbb{E}$ within the 3D view.
4.3 Implementation details

(a) strategy 1 (2676 triangles)
(b) strategy 2 (2676 triangles)
(c) strategy 3 (266 triangles)
(d) strategy 4 (1436 triangles)
(e) strategy 5 (133 triangles)
(f) strategy 6 (48 triangles)

Figure 4.6 – Visual comparison of the six mesh simplification strategies.
4 Visualization of intermediate steps and results

4.3.2 Classes

Fig. 4.7 shows a simplified collaboration diagram of SegmentationViewer. In the following I will briefly describe the main classes used and their relationship.

**SegmentationViewerController.** The three slice views and the 3D view hold a pointer to this central controller object. Whenever global properties change in one of the views (for example the 3D coordinate), this information is propagated via the controller object to all other views.

The controller can be used to show or hide a merged object containing position \( p \) given a watershed segment index \( \sigma(p) \) (showHideObjectByLabel), to show or hide a skeleton (showHideSkeleton), to assign new random segment or skeleton colors, (assignNewSegmentColors and assignNewSkeletonColors) and to toggle between showing the watershed segmentation and the merged segmentation (toggleMergeSegmentsMode).

Additionally, other objects can query the controller for information such as the filenames of all intermediate results of the segmentation pipeline, the current color tables, RF\(_2\) predictions and Graphical Model energy minimization results for specific faces as well as objects needed globally such as access to the CGP geometry information via a GeometryReader instance or skeletons.

**SliceView.** All three slice views are instances of the SliceView class with different SliceView::SliceOrientation passed to the constructor. SliceView inherits from QMain-Window and sets a QVTKWidget as its central widget. The view is set fixed and the projection mode to parallel so as to enable true 2D interaction with the image.

Every slice view maintains its own cache with a fixed maximum size of \( c_{\text{max}} \) GB. Volume datasets that may need to be loaded or held in memory are

1. the raw data (8-bit),
2. the voxel extra cellular space probability \( \phi \) (8-bit),
3. watershed seeds \( \tilde{\sigma} \) (32-bit),
4. watershed segmentation \( \sigma \) (32-bit),
5. additionally, for each 32-bit volume a second volume with 8-bit per voxel is needed to easily apply a color table.

Let \((W, H)\) be the shape of a slice and \(T\) the number of slices. The maximum number of

\[
s_{\text{max}} = \min \left( \left\lfloor \frac{c_{\text{max}} \cdot 10^9}{W \cdot H (4 \cdot 1 \text{byte} + 2 \cdot 4 \text{byte})} \right\rfloor, T \right)
\]  

(4.1)

slices can be cached. Given a position \( t \) in the image stack that the slice view should display, the strategy now is to load the range of slices

\[
slices \text{ to load} = \begin{cases} [0, s_{\text{max}}) & \text{if } t < s_{\text{max}} \\
[T - s_{\text{max}}, T) & \text{if } t \geq T - s_{\text{max}} \\
[t - \lfloor s_{\text{max}}/2 \rfloor, t + \lceil s_{\text{max}}/2 \rceil) & \text{else.}
\end{cases}
\]  

(4.2)

This is motivated by the fact that users often want to examine the spatial neighborhood of the current position \( p \), and thus need to go only a few slices up or down in the stack. Whenever \( p \) changes, ensureInCache is called to make sure that the new position is in the cache.
Each volume dataset is stored as an HDF5 file with optimal memory order depending on the slice orientation (see sec. 4.3.1). Internally, the Marray (Andres et al., 2010c) classes are used to load the data and vtkImageImport instances act as 3D VTK image views on the data pointer. A lookup table (vtkLookupTable, applied by a vtkImageMapToColors object) is used to provide either a gray scale colormap (raw data, probability map) or a random colormap (seeds, segments). The opaque data view and semi-transparent overlays are combined by means of a vtkImageBlend object.

Interaction is possible by registering a SliceView::Callback object as an observer for key and mouse events. It uses a vtkPropPicker to obtain the current mouse position.

The boundary overlay is built by buildBoundaryOverlay using the topological grid slice corresponding to $p$ (read via TopologyReader) as a vtkPolyData object and placed in front of the image slice. Again a lookup table is used to color the overlay, allowing to switch between the different boundary overlay modes quickly.

Skeletons are represented as 3D vtkPolyData objects (see SegmentationViewerController::skeletonPolyData) and intersected with the plane as represented by the current image slice by means of the SkeletonCutter object. The cut operations results in a set of points which are placed in front of the image slice.

**ObjectView3D.** The 3D view is an instance of the ObjectView3D class, which inherits from QMainWindow and uses a QVTKWidget as its central widget.

The axes indicator is displayed via a vtkAxesActor as a vtkOrientationMarkerWidget, while the slice planes (vtkImplicitPlaneWidget2) show the current position (vtkImplicitPlaneRepresentation).

Interaction is possible by the two event-handlers ObjectView3D::callback which listens for key and mouse events and ObjectView3D::PlaneCallback, which handles any user slice plane movements and the key events to toggle plane visibility.

Skeletons are displayed as Skeleton3D objects, a simple aggregation of VTK classes necessary to display the colored polygon data.

Segments are represented by Object3D instances. It is important to remember that there is a one-to-one mapping of Object3D pointer and watershed segmentation indices. When the user toggles between showing the watershed and merged segmentation, the controller’s assignNewSegmentColors method is called, which just colors Object3Ds belonging to the same merged object identically.
4 Visualization of intermediate steps and results

Figure 4.7 – Simplified collaboration diagram for the classes in SegmentationViewer.
5 Applying the first steps of the segmentation procedure to FIBSEM volume images

In this chapter, the first steps of the segmentation pipeline from chap. 2 are applied to a different dataset: a high-resolution FIBSEM volume image of neural tissue with traditional staining, enhancing – in addition to cell membranes – mitochondria, vesicles, microtubules and the postsynaptic density (see sec. 1.1 under “datasets”). The goal, again, is the segmentation of neurons; how much the additional “clutter” complicates this task is to be investigated.

I use the ILASTIK program (sec. 5.1) for labeling, voxel feature computation, classifier training and prediction to arrive at a probability map $\phi$, from which a segmentation is derived using the segment program. Different labeling strategies (Should one use two or four labels? Should one give only very few, precise labels or label entire regions with a wide brush?) are investigated as well as different selections of features (Are many features better?) and different values of the segmentation parameters $t_{\text{max}}$ and $s_{\text{min}}$ (see tab. 2.5).

For an objective evaluation, I have created manual ground-truth in the form of skeleton tracings (sec. 5.2), from which additional dense tracings are derived. The Variation of Information metric and object-level consistency error (sec. 5.6) are used to assess the quality of the automatic segmentations.

![Figure 5.1 – Drawing skeletons in ILASTIK.](image)
5 Applying the first steps of the segmentation procedure to FIBSEM volume images

<table>
<thead>
<tr>
<th>group</th>
<th>used filters</th>
<th>#features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Gaussian Smoothing</td>
<td>1</td>
</tr>
<tr>
<td>Texture</td>
<td>Eigenvalues of Hessian of Gaussian</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Eigenvalues of Structure Tensor</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Gaussian Gradient Magnitude</td>
<td>1</td>
</tr>
<tr>
<td>Edge</td>
<td>Laplacian of Gaussian</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Difference of Gaussians</td>
<td>1</td>
</tr>
<tr>
<td>Radon-Like</td>
<td>sec. 5.5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.1 – Composition of feature sets in ILASTIK.

5.1 ILASTIK

The Interactive Learning and Segmentation Tool Kit (ILASTIK, Sommer et al. 2010) provides an intuitive graphical user interface (GUI) for labeling, feature selection and computation, classification and segmentation of 3D volume data. It builds on the same methodology used in the segmentation procedure from chap. 2 (sections 2.3 through 2.5). Implemented in Python it utilizes the Qt framework, numpy\(^1\) and vigranumpy (Köthe, 2010), thus it shares the same code for feature computations and classification as used in sec. 2.3.

5.1.1 Towards segmentation: choices facing the user

After loading the raw data, ILASTIK allows browsing through the volume via three slice views, similar to SegmentationViewer from chap. 4. On the way towards segmentation, the first choice the user has to make is the number of classes; each is assigned a unique color. The volume data can then be labeled by painting on any slice view. Different brush sizes allow for either precise and sparse or extensive labeling.

The second choice concerns the selection of voxel features used to train a Random Forest classifier\(^2\). For simplicity, features are sorted into the categories orientation, color, edge and texture; for each such feature group, one can select one or more scales that govern filter mask sizes.

After feature computation the classifier can be (re)-trained at any time, and the prediction – visualized using overlays – is updated accordingly. This allows for interactive refinement of the labeling similar to the incremental labeling strategy from sec. 2.4.2. In the “Live prediction” mode re-training and prediction on the currently shown slice views are run continuously in the background, enabling the user to get a feeling for what constitutes “good” and “bad” labels by watching the prediction overlay change almost in real time. It is then easy to spot areas where the classifier still makes mistakes and to correct them by placing additional labels.

Finally, the probability map \(\phi\) can be used to generate a segmentation, for example using a watershed algorithm as in sec. 2.5.

\(^1\)http://numpy.scipy.org

\(^2\)ILASTIK has a plug-in framework to replace classifiers and other parts of the software; in the following, however, only the plug-ins that were used in the experiments will be described
5.1.2 Extensions

For the purpose of skeleton-tracing, feature calculation and batch-processing I have created four extensions:

- For fast and precise labeling, a graphics tablet can be used for input.
- Neuron skeletons can be traced by drawing with one class color; a button press performs a connected-component labeling with a random color map showing the connectivity and another a watershed segmentation with the skeletons taken as seeds.
- A generalization of the Radon-Like features from Kumar et al. (2010) to 3D was added to the list of features.
- A batch processing script allows to run ILASTIK without user interaction. Training (100 trees) is performed on a given labeled dataset using the specified features; the trained Random Forest is then used to predict on a separate test dataset.

5.2 Ground-truth segmentation

Ground-truth was traced on a $300 \times 300 \times 300$ subvolume (validation dataset VD; offset $o = (512, 512, 80)$ in the second part of the data).

5.2.1 What to label?

The different staining used in this dataset as opposed to e1088 introduces more ambiguity into the labeling process. Firstly, the cell organelles sometimes get so close to the cell membrane that correct tracing becomes difficult. More severely, extra-cellular space now represents a third class besides intra-cellular space and membrane (in e1088 everything outside the cell is stained). In this dataset, individual neurons usually do not touch but leave a small gap between their membranes, which looks exactly like intra-cellular space. Should this space be labeled differently? Without considering geometry or topology, there is no way of distinguishing. Therefore, I have chosen not to draw skeletons inside these areas (sec. 5.2.3). When skeletons are used as seeds for a watershed segmentation, this also means that the extra-cellular space is somewhat arbitrarily assigned to neighboring segments representing the intra-cellular space class. However, as these gaps are quite small, this error should not be severe.
5 Applying the first steps of the segmentation procedure to FIBSEM volume images

Figure 5.3 – Sparse ground-truth labeling on VD in the form of skeletons.

5.2.2 Dense manual voxel labels

A first attempt was to completely label all cell membranes in VD manually using ILASTIK. A computer mouse makes drawing difficult and slow: its movement is often jerky especially when working on fine details. Using a Wacom Bamboo tablet, a pen can be used for drawing instead; tablet support has been integrated into ILASTIK so that the brush width can be varied with the pressure on the pen’s tip.

Membranes were labeled in a suitable slice view first by tracing part of a cell’s outline. Going up and down the stack, the same structure was labeled on each slice individually. Often, this resulted in holes visible in one of the other two slice views, because the position of the drawn outlines varied more than the pen widths over two successive slices (particularly visible in fig. 5.2c). Thus most membranes have to be re-traced in all slice views.

Fig. 5.2 shows representative slices after about one working week of labeling effort. Even though most membranes were traced, it was clear that fixing all “holes” in the drawn boundaries would be a massive effort; furthermore, the boundaries became quite inaccurate due to jitter and large pen widths. This type of labeling was therefore abandoned.
5.3 Labeling strategies for classifier training

<table>
<thead>
<tr>
<th></th>
<th>offset o</th>
<th>size s</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD1</td>
<td>450,400,100</td>
<td>250,250,70</td>
</tr>
<tr>
<td>TD2</td>
<td>300,300,75</td>
<td>400,400,100</td>
</tr>
<tr>
<td>TD3</td>
<td>500,500,100</td>
<td>200,200,80</td>
</tr>
<tr>
<td>TD4</td>
<td>512,512,128</td>
<td>300,300,80</td>
</tr>
</tbody>
</table>

Table 5.2 – Four training datasets (TD) were selected from the first part of the FIBSEM dataset to experiment with different labeling strategies.

5.2.3 Skeletons

Inspired by Helmstaedter et al. (2010), skeleton tracings have been used in a second approach. A class “skeleton” was created and labeling performed roughly as follows:

Focus on one slice view, say $\hat{z}$, and start at $z = 0$. Choose one clearly visible cell outline and mark its extent on the slice roughly by drawing a connected criss-cross pattern. Starting from a labeled pixel in the center of the cell go 10 slices down the stack (while continuing to label) and mark the visible cell outline as before. Make sure that the outlines at $z$ and $z' = z + 10$ are connected. Proceed down the stack until the cell cannot be traced reliably anymore. Repeat for all cells.

To help with this procedure, two visualizations have been added to ILASTIK (fig. 5.1). On the left, skeletons have first been enlarged using a binary dilation operation, then the connected components are shown with a random colormap. On the right, a watershed segmentation is performed with the connected components of the skeleton labeling as seeds (in this case, the raw data serves as elevation map). Updating these views once in a while, one can find connectivity errors.

Fig. 5.3 gives an impression of the final skeletons obtained after about a day of tracing. The first row shows the three $\hat{x}$, $\hat{y}$ and $\hat{z}$ slice views; the second row the single largest skeleton (converted into a graph by connecting voxel centers according to the 26-neighborhood, then displayed using the SegmentationViewer program). It is clearly visible that the $\hat{z}$ stack was used most of the time during tracing.

5.2.4 Deriving dense voxel labels from skeletons

It is yet unclear whether skeletons (as a sparse voxel labeling) are equally suitable for segmentation evaluation as are dense labelings. To enable a comparison, the connected components of the drawn skeletons were used as seeds in a watershed segmentation to create dense voxel labels. A probability map created with ILASTIK served as the elevation map. Several rounds of refinement were performed until the final segmentation (referred to as grown skeletons in the following) looked reasonable. Fig. 5.4 shows a representative slice from the two types of ground-truth.

5.3 Labeling strategies for classifier training

While the classes “intra-cellular” and “extra-cellular” are natural choices for e1088, which classes should be used for a dataset with different staining? What is the tradeoff between accuracy and quantity of labels? In the following, different labeling strategies on four training volumes (tab. 5.2) are presented. The number of labels varies, see tab. 5.3.
Applying the first steps of the segmentation procedure to FIBSEM volume images

Figure 5.4 – One slice from the manual ground-truth. a) Sparse skeletons, binary dilation operation applied for easier visibility. b) Sparse skeletons used as seeds for a watershed segmentation on a suitable elevation map.

5.3.1 Membrane versus background

Five different sets of training labels have been created for the case of two classes: “membrane” and “background” (\(C = 2\)). They fall into three different categories (fig. 5.5).

- **Aggressive Strategy.** Labeling was performed with a wide brush, quickly labeling membranes and labeling mitochondria, vesicles and cell plasma regardless as background.
- **Conservative Strategy.** Use a medium-sized brush when labeling and do not label over mitochondria, as their texture can easily be confused with clusters of vesicles close to a membrane or junctions of membranes. The resulting uncertainties in these places lead to under-segmentation. In one variant (5.5b) vesicles were labeled as background, in another (5.5c) they were not labeled.
- **Sparse Strategy.** Similar to TD\(_1\) from sec. 2.4.2 individual labels are centered precisely on membranes and spaced several voxels apart. Rather than rapidly increasing the label quantity, the goal was to interactively improve the prediction with as few labels as possible.

5.3.2 Membrane, vesicles and mitochondria versus background

While “playing” in ILASTIK, it became clear that more than two classes tend to give inferior results for the purpose of neuron segmentation (as elevation map \(\phi\) the probability for class membrane was used exclusively). Therefore, only two sets of training labels which use four classes (\(C = 4\)) have been included. Both were labeled with a conservative strategy and are shown in fig. 5.6.
5.3 Labeling strategies for classifier training

Figure 5.5 – Labeling strategies with two classes “membrane” and “background”.

(a) 2 classes, aggressive (TD$_1$)

(b) 2 classes, conservative (TD$_1$)

(c) 2 classes, conservative (TD$_2$)

(d) 2 classes, sparse (TD$_3$)

(e) 2 classes, sparse (TD$_4$)
5 Applying the first steps of the segmentation procedure to FIBSEM volume images

(a) 4 classes, conservative (TD₁)
(b) 4 classes, conservative (TD₃)

Figure 5.6 – Labeling strategies with the four classes “membrane”, “mitochondria”, “vesicles” and “background”.

<table>
<thead>
<tr>
<th>label set LS</th>
<th>background</th>
<th>membrane</th>
<th>mitochondria</th>
<th>vesicles</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = 2, aggressive on TD₁</td>
<td>58989</td>
<td>22857</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C = 2, conservative on TD₁</td>
<td>15221</td>
<td>11629</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C = 2, conservative on TD₂</td>
<td>9803</td>
<td>7150</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C = 2, sparse on TD₃</td>
<td>178</td>
<td>322</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C = 2, sparse on TD₄</td>
<td>109</td>
<td>196</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C = 4, conservative on TD₁</td>
<td>2754</td>
<td>5674</td>
<td>3952</td>
<td>1739</td>
</tr>
<tr>
<td>C = 4, conservative on TD₃</td>
<td>427</td>
<td>1057</td>
<td>765</td>
<td>593</td>
</tr>
</tbody>
</table>

Table 5.3 – The different sets of labels used for training the Random Forest classifier.

5.4 Feature selection

Selecting a set of discriminative voxel features is essential for a good classification result. As the incremental labeling technique has been employed for all labeling strategies described above, the labels implicitly depend on the chosen features that were used during labeling. However in the following, independence is assumed and for a given labeling strategy the best set of features among a predefined list of feature sets is found.

Tab. 5.4 summarizes the 8 feature sets FS₀ ... FS₇, where FS₁ to FS₇ are identical to the first four sets, except that the Radon-Like features (sec. 5.5) were added. All used features are rotation invariant; this excludes the usage of orientation features offered by ILASTIK. One feature set consists of only one color feature and represents thresholding. The other sets incorporate promising color, texture and edge features. In future work, automated assessment of the variable importance should be used as another objective criterion for selecting the best possible set of features, see for example Nair (2010).
5.5 Radon-Like features in 3D

Kumar et al. (2010) proposed a new class of features especially designed for application to connectomics. These features are called Radon-Like, because their method builds on the Radon Transform (Jähne, 2005, chap. 8)

\[ R(r, \vartheta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(x) \delta(x_1 \cos \vartheta + x_2 \sin \vartheta - r) d^2x \]

which integrates the values of the two-dimensional function \( I(x, y) \) along a line \( l \) defined by angle \( \vartheta \) and offset \( r \).

Radon-Like features arise by letting the limits of the integral depend on the underlying data \( I(x, y) \) and position \( p = (x, y) \). Here \( I \) is a 2D electron-microscopy image of neural tissue. Let a point \( p \) on \( l \) be parameterized by \( t \). Then each \( l \) is divided into a number of segments by a set of knots \( (t_1, \ldots, t_n) \) along \( l \).

The Radon-Like feature at a point \( p \) between \( I(t_i) \) and \( I(t_{i+1}) \) is defined as

\[ \Psi(p, l, t_i, t_{i+1})[I(x, y)] = T(I, l(t)), \ t \in [t_i, t_{i+1}] \]  \hspace{1cm} (5.2)

The extraction function \( T \) could be the integral from \( t_i \) to \( t_{i+1} \) from above.

For a fixed \( \vartheta \) but varied offset \( r \), the output is another image \( I'(x, y) \); when additionally varying the angle the output is a vector for each image pixel.

The division of lines into line segments is obtained by intersecting each \( l \) with the knot map, the output of a canny edge filter. For boundary enhancement, Kumar et al. (2010) suggest to use an extraction function \( T_2 \) that computes the average gray value along a line segment on a boundary-enhancing edge map of \( I \). In all their experiments, only the pixel-wise mean of the computed feature values is considered.

I have implemented these Radon-Like features for 3D volume images. To be fast enough, ideas from the field of volume rendering (Lacroute and Levoy, 1994) have been used. Rays are traced as discrete lines through the image volume. First, three permuted copies of the input data (both edge and knot map) are created. In each copy, only lines with \( \hat{l} \cdot \hat{n} \leq 1 \) are traced, where \( \hat{n} \) is the normal vector of the 2D slices defined by the memory order. This allows to sweep through the image volumes with a minimum number of cache misses. To save memory while computing \( T_2 \), image values are accumulated between successive knots and the intermediate results are written back to the current pixel in a first pass through the data. Another volume array saves the

<table>
<thead>
<tr>
<th>FS</th>
<th>Color</th>
<th>Texture</th>
<th>Edge</th>
<th>Radon-Like</th>
<th>#features</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS0</td>
<td>0.7</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>FS1</td>
<td>0.7, 1.6, 3.5</td>
<td>0.7, 1.6, 3.5, 5.0, 10.0</td>
<td>—</td>
<td>—</td>
<td>38</td>
</tr>
<tr>
<td>FS2</td>
<td>0.3, 1.0, 3.5, 10.0</td>
<td>0.3, 1.0, 3.5, 10.0</td>
<td>—</td>
<td>—</td>
<td>32</td>
</tr>
<tr>
<td>FS3</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>—</td>
<td>40</td>
</tr>
<tr>
<td>FS4</td>
<td>0.7</td>
<td>—</td>
<td>—</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>FS5</td>
<td>0.7, 1.6, 3.5</td>
<td>0.7, 1.6, 3.5, 5.0, 10.0</td>
<td>—</td>
<td>✓</td>
<td>44</td>
</tr>
<tr>
<td>FS6</td>
<td>0.3, 1.0, 3.5, 10.0</td>
<td>0.3, 1.0, 3.5, 10.0</td>
<td>—</td>
<td>✓</td>
<td>38</td>
</tr>
<tr>
<td>FS7</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>10.0</td>
<td>0.7, 1.0, 1.6, 3.5</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.4 – Used feature sets FS. For color, texture and edge filter groups, the numbers give the filter mask sizes \( \sigma \) that were selected.
Applying the first steps of the segmentation procedure to FIBSEM volume images

Figure 5.7 – Probability-map after training with $C = 2$, conservative on TD$_1$. a) No Radon-Like features. b) With Radon-Like features.

current distance from the last visited knot. In a second pass, lines are traversed in reversed direction, writing the average gray value for a line segment (computed from final accumulation value and distance map) back to each voxel lying on that line. The result is a feature image for all lines with a given direction. Feature images for all directions (directions are discretized into a sufficient number) are averaged to give the voxel-wise mean for all rays passing through that voxel.

A 3D canny edge detector (Ullrich Köthe, priv. comm.) is used for the knot map (parameters: threshold $c_t$ and standard deviation of Gaussian filter $c_{\sigma}$), the edge map is either taken to be the Laplacian of Gaussian filtered raw data (scale parameter $e_{\sigma}$) or unfiltered.

Six features were computed: $(c_t, c_{\sigma}) \in \{(1,10),(2,5),(3,1)\}$ combined with the raw data as edge map and $(c_t, c_{\sigma}, e_{\sigma}) \in \{(1,10,1),(2,5,2),(3,1,3)\}$ when the Laplacian of Gaussian filter was used.

5.6 Evaluation using VI and OCE error measures

Error measures. The Variation of Information metric (sec. 2.11.1) is used to compare automatic and ground-truth segmentations. For the sparse skeletons, only voxels lying on the skeletons are used in the computation, while the grown skeletons partition the whole test dataset.

Polak et al. (2009) proposed a new error measure called the object-level consistency error (OCE) for comparing two segmentations $I_g$ (ground-truth) and $I_s$ (automatic segmentation), which penalizes both over- and under-segmentation and is sensitive to the location of boundaries and sizes of segments. In the following, $M$ and $N$ denote the number of segments in $I_g$ and $I_s$, respectively; $\delta(\cdot)$ is the inverse delta function which
5.7 Conclusions

is zero if the argument equals 0 and one otherwise.

\[
E_{g,s}(I_g, I_s) = \sum_{j=1}^{M} \left[ 1 - \sum_{i=1}^{N} \frac{|A_j \cap B_i|}{|A_j \cup B_i|} \cdot W_{ji} \right] W_j
\]

(5.3)

\[
W_{ji} = \frac{\delta(|A_j \cap B_i|) |B_i|}{\sum_{k=1}^{N} \delta(|A_j \cap B_k|) |B_k|}
\]

(5.4)

\[
W_j = \frac{|A_j|}{\sum_{l=1}^{M} |A_l|}
\]

(5.5)

Then the object-level consistency error (OCE) is defined as

\[
OCE(I_g, I_s) = \min(E_{g,s}, E_{s,g}).
\]

(5.6)

For evaluation both the Variation of Information and object-level consistency error are computed for both the sparse skeletons and grown skeletons ground-truth.

Segmentation parameters. For segmentation, the segment program (tab. 2.5) has been used. For each feature set \(FS_i\) and each labeling strategy \(LS_j\) a grid search over the segmentation parameters \(t_{\text{max}}\) and \(s_{\text{min}}\) (see sec. 2.5.2) is performed, where \(t_{\text{max}} \in \{0, 1, 2, 5, 10, 20\}\) and \(s_{\text{min}} \in \{1, 3, 5, 7, 11, 15, 19\}\).

Evaluation. Tab. 5.6 shows for each pair \((FS_i, LS_j)\) the smallest values of VI(sparse), VI(grown), OCE(sparse) and OCE(grown) over the performed grid-search. The segmentation parameters \(t_{\text{max}}\) and \(s_{\text{min}}\) that lead to these smallest values are listed in tab. 5.7. To put things into perspective, tab. 5.5 lists error measure values for some artificially wrong segmentations.

5.7 Conclusions

From the experiments the following conclusions can be drawn:

- The inclusion of Radon-Like features (\(FS_4\) to \(FS_7\)) does not help. In fig. 5.7b close membranes become blurred and holes appear in membranes. However, it should be noted that the parameters for these features have not been systematically optimized, but rather chosen some after visual exploration.

- Over all labeling strategies, at least one error measure finds that \(FS_3\) gives the best results. With sparse skeleton ground-truth, 10 of 14 error measures vote for \(FS_3\), with grown skeleton ground-truth 9 of 14. This feature set includes color and texture features and is the only one to additionally include edge features. Apparently, in this application, more features are better.

- Best agreement is reported for labeling strategies with \(C = 2\); for \(FS_3\) a conservative labeling strategy is chosen by VI(grown), OCE(sparse) and OCE(grown) to be optimal and a sparse voxel labeling strategy by VI(sparse). This means that difficulties in the dataset due to “clutter” should be left to the classifier to resolve; its uncertainty in these regions manifests itself in probability values that do not represent seeds and are flooded in the watershed segmentation.

- For \(FS_3\) and a conservative labeling strategy, fig. 5.8 qualitatively compares the agreement of the different error measures by visualizing results for the grid-search over
Applying the first steps of the segmentation procedure to FIBSEM volume images

\[ t_{\text{max}} \] and \[ s_{\text{min}} \]. All but OCE(sparse) agree that a low threshold on the probability map and a large seed size should be used for optimal results. As discussed in 5.2.1, volumes formed by the extra-cellular space are quite small; by raising \[ s_{\text{min}} \] seeds in these areas might be suppressed enough to disregard these regions in the final segmentation.

- The question which error measure should be used cannot be answered. Both VI and OCE seem to be in line with my subjective judgements of segmentation quality during labeling.
Table 5.5 – Comparing the ground-truth segmentations with “blocky” superpixel segmentations. The $300 \times 300 \times 300$ test dataset was divided into cubes with side lengths $l$; each was assigned a unique label. Best values over one row are highlighted in green.

Figure 5.8 – Hinton diagrams for the VI and OCE error measures between the automatic segmentation (learning with $C = 2$, conservative labeling on TD$_1$) and the ground-truth of either sparse or grown skeletons. The area of each squares visualizes the agreement $1 - \text{VI}$ or $1 - \text{OCE}$, thus bigger squares are better. $t_{\text{max}}$ is the threshold on the probability map for seed creation and $s_{\text{min}}$ the minimum allowed seed size.
Applying the first steps of the segmentation procedure to FIBSEM volume images

For each feature set $FS_i$ and each labeling strategy $LS_j$, four error measures $VI$ over sparse skeletons (upper left in cell), $VI$ over grown skeletons (upper right in cell), OCE over sparse skeletons (lower left in cell), OCE over grown skeletons (lower right in cell) are shown.

<table>
<thead>
<tr>
<th>$FS_0$</th>
<th>$FS_1$</th>
<th>$FS_2$</th>
<th>$FS_3$</th>
<th>$FS_4$</th>
<th>$FS_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = 2$, conservative on TD</td>
<td>$1.7500$</td>
<td>$1.9100$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, conservative on TD</td>
<td>$1.7500$</td>
<td>$1.9100$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, sparse on TD</td>
<td>$1.5000$</td>
<td>$1.7200$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, sparse on TD</td>
<td>$1.5000$</td>
<td>$1.7200$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
</tbody>
</table>

For each labeling set $PS_i$, and each labeling strategy $LS_j$, four error measures $VI$ over sparse skeletons (upper left in cell), $VI$ over grown skeletons (upper right in cell), OCE over sparse skeletons (lower left in cell), OCE over grown skeletons (lower right in cell) are shown.

Table 5.6 - For each feature set $FS_i$ and each labeling strategy $LS_j$, four error measures $VI$ over sparse skeletons (upper left in cell), $VI$ over grown skeletons (upper right in cell), OCE over sparse skeletons (lower left in cell), OCE over grown skeletons (lower right in cell) are shown.

For each error measure the smallest value (best agreement) over a grid search of $t_{max}$ and $s_{min}$ was selected.

For a single error measure, the best value over one row is highlighted in green.

<table>
<thead>
<tr>
<th>$PS_0$</th>
<th>$PS_1$</th>
<th>$PS_2$</th>
<th>$PS_3$</th>
<th>$PS_4$</th>
<th>$PS_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = 2$, conservative on TD</td>
<td>$1.7500$</td>
<td>$1.9100$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, conservative on TD</td>
<td>$1.7500$</td>
<td>$1.9100$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, sparse on TD</td>
<td>$1.5000$</td>
<td>$1.7200$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>$C = 2$, sparse on TD</td>
<td>$1.5000$</td>
<td>$1.7200$</td>
<td>$0.3810$</td>
<td>$1.1900$</td>
<td>$0.3850$</td>
</tr>
<tr>
<td>labels</td>
<td>feature set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| \( C = 2 \), aggressive on TD_1 | \( \begin{array}{cccccccc}
0, 15^3 & 0, 15^3 & 10, 11^3 & 10, 19^3 & 20, 11^3 & 20, 13^3 & 5, 11^3 & 10, 15^3 \\
0, 1^3 & 0, 15^3 & 0, 1^3 & 20, 15^3 & 20, 1^3 & 20, 11^3 & 0, 1^3 & 5, 15^3 \\
\end{array} \) |
| \( C = 2 \), conservative on TD_1 | \( \begin{array}{cccccccc}
0, 15^3 & 0, 15^3 & 10, 15^3 & 10, 15^3 & 5, 11^3 & 10, 15^3 & 0, 11^3 & 0, 15^3 \\
0, 1^3 & 0, 15^3 & 10, 1^3 & 20, 19^3 & 5, 1^3 & 5, 15^3 & 0, 1^3 & 0, 19^3 \\
\end{array} \) |
| \( C = 2 \), conservative on TD_2 | \( \begin{array}{cccccccc}
0, 15^3 & 0, 15^3 & 10, 15^3 & 10, 15^3 & 5, 11^3 & 10, 15^3 & 0, 11^3 & 0, 15^3 \\
0, 1^3 & 0, 15^3 & 10, 1^3 & 5, 15^3 & 20, 11^3 & 10, 15^3 & 0, 1^3 & 0, 11^3 \\
\end{array} \) |
| \( C = 2 \), sparse on TD_3 | \( \begin{array}{cccccccc}
0, 19^3 & 0, 19^3 & 10, 19^3 & 10, 19^3 & 10, 1^3 & 10, 19^3 & 0, 15^3 & 0, 15^3 \\
0, 1^3 & 0, 19^3 & 10, 1^3 & 10, 19^3 & 5, 1^3 & 20, 19^3 & 0, 1^3 & 0, 13^3 \\
\end{array} \) |
| \( C = 2 \), sparse on TD_4 | \( \begin{array}{cccccccc}
0, 15^3 & 0, 19^3 & 5, 19^3 & 5, 19^3 & 10, 19^3 & 10, 19^3 & 0, 15^3 & 0, 19^3 \\
0, 1^3 & 0, 19^3 & 0, 1^3 & 5, 19^3 & 0, 1^3 & 10, 19^3 & 0, 1^3 & 0, 19^3 \\
\end{array} \) |
| \( C = 4 \), conservative on TD_1 | \( \begin{array}{cccccccc}
0, 11^3 & 0, 11^3 & 20, 15^3 & 20, 15^3 & 20, 1^3 & 20, 15^3 & 20, 1^3 & 20, 15^3 \\
0, 1^3 & 0, 11^3 & 20, 1^3 & 20, 19^3 & 20, 1^3 & 5, 19^3 & 0, 1^3 & 5, 19^3 \\
\end{array} \) |
| \( C = 4 \), conservative on TD_2 | \( \begin{array}{cccccccc}
10, 11^3 & 0, 15^3 & 20, 15^3 & 20, 15^3 & 5, 19^3 & 5, 19^3 & 0, 19^3 & 0, 19^3 \\
10, 1^3 & 20, 19^3 & 20, 1^3 & 20, 19^3 & 5, 1^3 & 5, 19^3 & 0, 1^3 & 5, 19^3 \\
\end{array} \) |

**Table 5.7** - For each feature set \( FS_i \) and each labeling strategy \( LS_j \), four error measures were computed. The parameters \( t_{max}, s_{min} \) that lead to the smallest error measure are listed in the cell. Clockwise from upper left to lower right: best parameters for VI(sparse), VI(grown), OCE(grown), OCE(sparse).
6 Conclusion

The presented work is concerned with the automatic analysis of both surface-stained SBFSEM and traditionally stained high-resolution FIBSEM volume images of neural tissue. The main focus has been on segmentation, but first steps towards synapse detection have been made as well. As visualization is very important for debugging, exploration and qualitative judgment of all computations on these huge volume images, a performant interactive viewer application has been written.

Segmentation of surface-stained SBFSEM data. The segmentation procedure from Andres et al. (2008b, 2010d) is a promising method for automated analysis of surface-stained SBFSEM data. However, the previous MATLAB implementation could only handle 1/64th of the complete e1088 dataset. To be able to work on the whole dataset, the segmentation procedure had to be improved. This was achieved by making use of MPI parallelization, a fast C++ implementation and efficient data structures.

With these improvements the segmentation procedure could be applied for the first time to volume images as large as eight gigavoxels. Now the whole procedure can be run on the e1088 dataset in less than two weeks on a high memory multi-core machine. All intermediate results are stored as portable HDF5 files, and a Python interface to run all command line programs is available.

Speeding up the procedure has made systematic parameter tuning on large subsets of the whole e1088 dataset possible. An important result has been that parameters need to be adjusted for large datasets: a more conservative choice of parameters is necessary for segmenting all of e1088 than what parameter optimization for a 100×100×100 densely labeled subset would suggest. The choice of labels has been shown to be very important for the success of the procedure; in particular deriving face and intersection labels from dense ground-truth turned out to be inferior to manual labeling.

The CGP algorithm (Andres et al., 2010b) efficiently constructs the topological grid and the list of j-components. The implementation of the topological grid construction has been parallelized using an MPI implementation. Furthermore the CGP library now offers methods to efficiently construct mesh data structures and to visualize 0-, 1- and 2-components with the help of the VTK toolkit. A new class hierarchy reflects the definition of j-cells and j-components. Topology and Geometry data stored on disk can be easily queried with the new high-level classes TopologyReader and GeometryReader.

Visualization. Finding the root cause of under- or over-segmentation in the final output of the complex multi-step procedure is difficult. Being able to visualize the raw data, all intermediate steps and the final result is essential for understanding the circumstances in which these errors occur. As a solution, the SegmentationViewer program has been written from the ground up building on the VTK and Qt libraries. The user is presented with three slice views which show the raw data together with various optional overlays. Any segment can be selected to be shown in a 3D view, where it is easy to grasp its shape and topologic relation to other segments. Furthermore, the success of
the Graphical Model to remove over-segmentation can be judged visually by switching
between watershed and final segmentation.

**SegmentationViewer** uses redundant but memory-layout optimized copies of the data
and a strategy of caching consecutive slices as well as mesh decimation to cope with the
whole 2048 × 1892 × 2048 voxels \(e1088\) dataset.

**Synapse features.** Synapse detection is as important as segmentation for understanding neural circuits. Therefore I have proposed a set of geometric features, consisting of surface area, surface curvature and surface normal gray value statistics. As no ground-truth labels for synapses could be obtained, only first qualitative results have been reported.

**Segmentation of traditionally stained FIBSEM data.** Finally, I have investigated the applicability of the first steps of the segmentation procedure from Andres et al. (2010d) to differently stained EM volume images of neural tissue. Stained cell organelles complicate the segmentation problem. The ILASTIK program (Sommer et al., 2010) has been used for labeling, feature computation, classifier training and prediction as well as for generating ground-truth.

A quantitative evaluation uses both sparse ground-truth in the form of skeletons and derived dense ground-truth. The computed error measures “Variation of Information” and the “object-level consistency error” indicate that a good labeling strategy is not to label mitochondria or vesicles, but only label cell membrane and cell plasma. Areas where labeling is difficult should be left to the Random Forest classifier. In practice, mitochondria and vesicles are less of a problem because these structures often do not get any seeds and are flooded during the watershed segmentation procedure.

Using color, texture and edge features on many scales within ILASTIK was found to perform best. I have generalized the Radon-Like features from Kumar et al. (2010) to three dimensions. Adding these features to the list of voxel features negatively affected the segmentation performance, though parameters were not tuned systematically.
Bibliography


Björn Andres, Ullrich Köthe, Thorben Kröger, and Fred A. Hamprecht. How to extract the geometry and topology from very large 3D segmentations. 2010b. In preparation.


Guillaume Chazarain. iotop, 2010. URL git://repo.or.cz/iotop.git.


Bibliography


