# Online Neuron Reconstruction

by

Jonathan Stoller

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016



Author . . . . . .
Department of Electrical Engineering and Computer Science
May 16, 2016

Certified by . . . . . .
Nir Shavit
Professor
Thesis Supervisor

Accepted by . . . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

# Online Neuron Reconstruction

by

Jonathan Stoller

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

## Abstract

In this thesis, I designed an algorithm that traces neurons through the images of various datasets extracted from mouse brains. The algorithm is implemented in Python and relies on the output of a recently developed Fully Convolutional Neural Network (implemented in C) that runs on the underlying images.
The images themselves are generated using state of the art technologies that provide high resolution and extremely accurate representations of the original brain matter. The algorithm is part of the combined efforts of MIT's Computational Connectomics Group.

Thesis Supervisor: Nir Shavit
Title: Professor

# Acknowledgments

First, I would like to thank my family and friends for their support during my time at MIT.

I would like to thank Professor Nir Shavit for all his help and guidance throughout the year that I spent working on this thesis. His insights and advice were extremely helpful.

Additionally, I would like to thank Alex Matveev, Hayk Saribekyan, Victor Jakubiuk and the various members of MIT's Computational Connectomics Group and Harvard's Hanspeters Group for their suggestions and comments. It was a pleasure working with such outgoing and creative people.

Finally, I would like to give a special thanks to Yaron Meirovitch. Every significant part of my thesis was influenced by him. He was incredibly patient, insightful, available and generous. The fact that I enjoyed working on my thesis as much as I did is to a large extent thanks to him.

# Contents

# List of Figures

# Chapter 1

# Introduction

Recent improvements in imaging technologies have provided neuroscientists with a deluge of data. Terabytes, even petabytes of brain imagery can be generated within a number of hours, or at most a few days. Processing this data, however, remains a challenge.
One particular problem of interest is that of tracing a single neuron. The data associated with a single neuron is a tiny fraction of the total data available. Developing a method that could trace a single neuron would be advantageous in many ways - ideally providing neuroscientists with a tool that could generate results in real time, due to the smaller quantity of data involved.

## 1.1 Online Neuron Reconstruction

The goal of this thesis was to develop an algorithm that would provide an automatic, computerized way of tracing a neuron through brain images, providing neuroscientists with real time results to certain queries.
Online Neuron Reconstruction (ONR) works in real time by accessing the preprocessed, raw brain imagery, processing only those images that are necessary for the algorithm's operation (henceforth, the terms ONR and algorithm are used interchangeably). The primary tool used by ONR is a Fully Convolutional Neural Network (FCNN) that accepts raw images as inputs and outputs an intermediate data type that is used by the algorithm.
Working in this way, and utilizing the fact that the imagery is aligned and of excellent qual-

ity and high resolution, the algorithm is able to apply techniques that grant it the speed it needs to provide real-time, "online" results.

## 1.2 Connectomics - Overview

Mapping the neuroanatomical structure of the brain has been a long term challenge in neuroscience. In the last few years, this objective has seen new light through Connectomics, a field that deals with studying and creating models of the neural connections within the brain. There are many challenges related to Connectomics. The largest problem concerns the sheer volume of data that has to be processed and analyzed. The first milestone in Connectomics was the mapping of the neural structure of the C. elegans worm. Despite the small amount of neurons involved[1], the entire process, which was done by hand, lasted ten years. Recently, scientists and engineers have begun to map small sections of the mouse brain. Overall, the brains of mice are comprised of $\sim 70,000,000$ neurons. However, at present, only very small subsets of the brain are being analyzed (our project focuses on one cubic millimeter).
Despite the deceptively small volume these subsets occupy, the size of their corresponding image datasets can be huge, spanning from terabytes to even petabytes of memory.

### 1.2.1 Brain Imagery

The images used by the MIT Computational Connectomics Group are generated by a method known as Electron Microscopy (EM). Essentially, EM employs special microscopes that fire beams of electrons on the original brain tissue. By analyzing the stream of electrons returned from the brain tissue, the microscope is able to generate images of extremely high quality. All the images used by the algorithm and by the MIT Computation Connectomics Group (MCCG) have been provided and aligned by Harvard's Hanspeter Lab [1].

[1] 302

## 1.3 Motivation

One of the many tasks that require optimization in the field of neuroscience is the tracing of individual neurons throughout a dataset of images. In other words, given a particular neuron, how can the various components in its structure (e.g., the soma, pre/post synaptic connections, etc.) be identified?

At present, there is no program or algorithm that can be used to answer this question in a fast enough manner. The only way to track a neuron is for an experienced researcher to spend hours, painstakingly sifting through hundreds of images until finding the various components of the neuron in question.

For example, the research of Morgan et al. [2] revolved around the study of postsynaptic thalamocortical neurons located in an image dataset comprised of 100 trillion voxels. The work done by the researchers could surely have been aided by an approach capable of quickly tracing individual neurons, rather than having to wait for the analysis of an entire dataset that contained huge amounts of irrelevant data.

## 1.4 Usage

ONR can be used directly from the command line, by providing a number of arguments. Asides from the arguments, the only input required is the raw brain imagery. Due to the enormous size of the data, the three dimensional data set is divided into a large number of vertically aligned stacks of two dimensional images. The algorithm navigates through this data set, applying the FCNN only where necessary.

Given an initial three dimensional sequence of coordinates (corresponding to a voxel within a neuron), the algorithm traces the neuron throughout the three dimensional space.

In accordance with the underlying two dimensional data representation, the algorithm outputs the approximate location of the neuron's "centers" within the various two dimensional images it traverses, as well as the set of voxels belonging to it within each image.

As an example, the algorithm can be called in the following way:

```
python tracer.py 8 7 4 0 290 932 2 50 4 0 1
```

`tracer.py` is the name of the root script that executes the algorithm.

The order of the arguments, as shown above, is as follows:
(1) *3D coordinates of the set of images containing the starting voxel*

As mentioned previously, the imagery is distributed into vertically aligned subsets. Each subset, or block, has a set of 3D coordinates that uniquely identify it.

(2) *3D coordinates of the starting voxel within the specified block*

(3) *The vertical direction of exploration*

Most likely, the neuron expands vertically in both directions, i.e., in both positive and negative vertical (Z) directions. This argument allows the user to specify whether the algorithm should trace the expansion of the neuron in both directions, or in just one of the two directions.

(4) *Maximum depth of vertical traversal*

This argument provides the user with a way of instructing the algorithm to terminate after exploring a certain vertical depth (e.g., a value of fifty would cause the algorithm to terminate after tracing the neuron through a depth of fifty images in the Z direction).

(5) *Sparsity of gathered points (sub sampling factor)*

The output of the algorithm includes the voxels belonging to the neuron. In order to make the output smaller, it is possible to output a sub sampling of these voxels.

For example, a sub sampling value of 2 would cause the algorithm to output every $2^{nd}$ voxel along the X and Y axes, i.e., $\frac{1}{4}$ ($= \frac{1}{2}^2$) of the total voxels belonging to the neuron.

(6) *Data type*

The envisioned use case of the algorithm is to serve as a tool for neuroscientists to quickly analyze new datasets that have not yet been processed. That being said, if the data has already been processed, it is possible to utilize the already existing output of the FCNN, thus speeding up the running time of the algorithm.

(7) *Forking*

The algorithm has a notion of uncertainty that allows it to terminate when it is relatively unsure as to how to continue tracing a neuron through the dataset. Typically, when the algorithm reaches such a point, it can still make an educated guess as to where the neuron continues. The *Forking* parameter instructs the algorithm to produce additional "forks", or "branches" that continue the tracing along such paths.

It is important to note that the tracing of the neuron along such forks is more prone to error. The algorithm's output includes a confidence score that indicates, for each image traversed (on every fork), the likelihood that the computed results are correct.

## 1.5 Contributions

- ONR provides neuroscientists with a way of reconstructing the structure of a neuron in real time, on raw brain imagery.
- ONR can also be used as a way of improving the results of a dense segmentation. Dense segmentations are the primary tool used to analyze an entire dataset of brain

imagery. The output of a dense segmentations maps every voxel in the three dimensional space to a unique neuron.

Currently, while extremely powerful, dense segmentations are prone to making split errors (i.e., erroneously splitting a single neuron into two distrinct neurons). Running ONR on the output of a dense segmentation can allow neuroscientists to identify such errors and fix them by merging the split neurons into a single neuron.

# Chapter 2

# The Brain Imagery

The brain imagery used by ONR was prepared by the Hanspeter Group from Harvard. The process involved a number of distinct stages[3] that will now be outlined.

## 2.1 Creating the dataset

The ONR datasets were extracted from brain tissues whose sizes were approximately 0.1 $mm^3$. The brain tissue was sliced at an extremely fine granularity (29 nm) using a diamond knife.

After slicing the tissue, the resulting sections were scanned using an electron microscope that employed backscattered electron detection [3], providing a resolution per section of 3x3 nm, or in total 3x3x29 (XYZ axes, respectively). Recently, the procedure has incorporated a new microscope capable of speeding up the process by employing a number of scanning beams working in parallel[4]. For computational reasons this resolution was down-sampled to 6x6x29. That being said, when finer details are required, our systems use the original resolution instead.

## 2.2 Properties of the dataset

One crucial property of the dataset is that the images are aligned. Alignment was achieved via the use of affine image transformations [3]. The fact that the data was both aligned and of such high resolution allowed ONR to make use of various techniques that previously could not have been applied.

Generating such a high resolution comes with a price - the resulting images take up a very large amount of storage. For example, at a resolution of 3x3x29 nm the images generated from a 0.13 $mm^3$ volume require a few terabytes of storage.

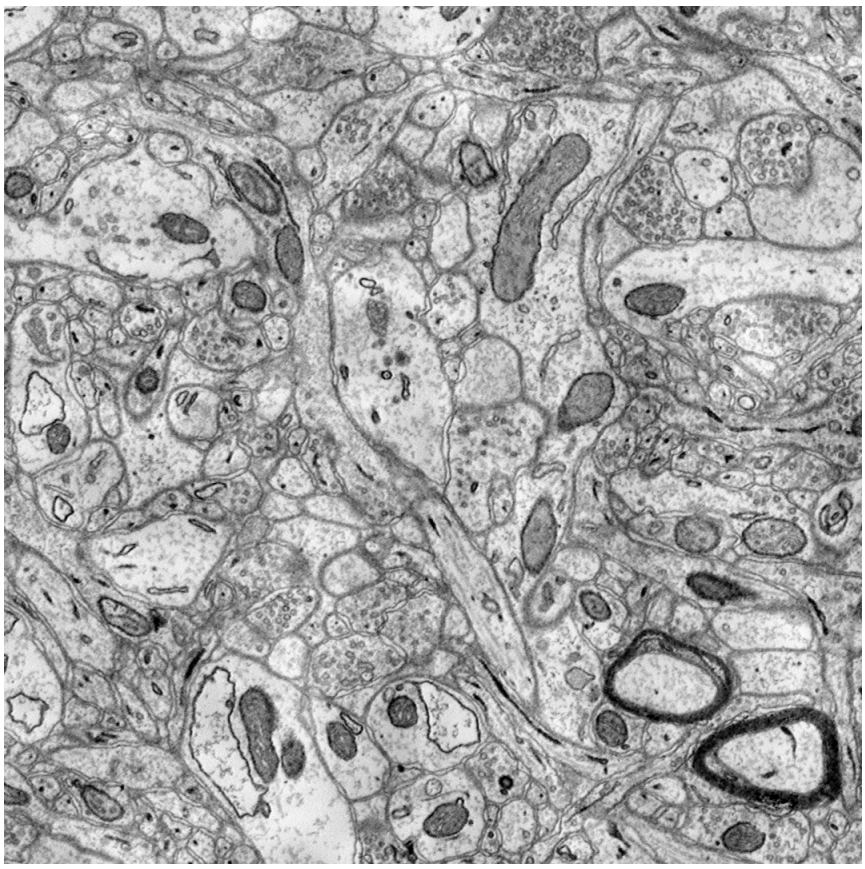

**Figure 2-1:** A sample image generated by the Hanspeter Group

## 2.3 FCNN

As stated above, ONR does not make direct use of the images. Instead, it invokes a FCNN on the raw imagery that outputs a probability map in which values of 1 correspond to voxels that are considered to be membranes, and values of 0 correspond to the interior of a cell. Figure 2-2 is the FCNN probability map outputted for the image displayed in Figure 2-1.

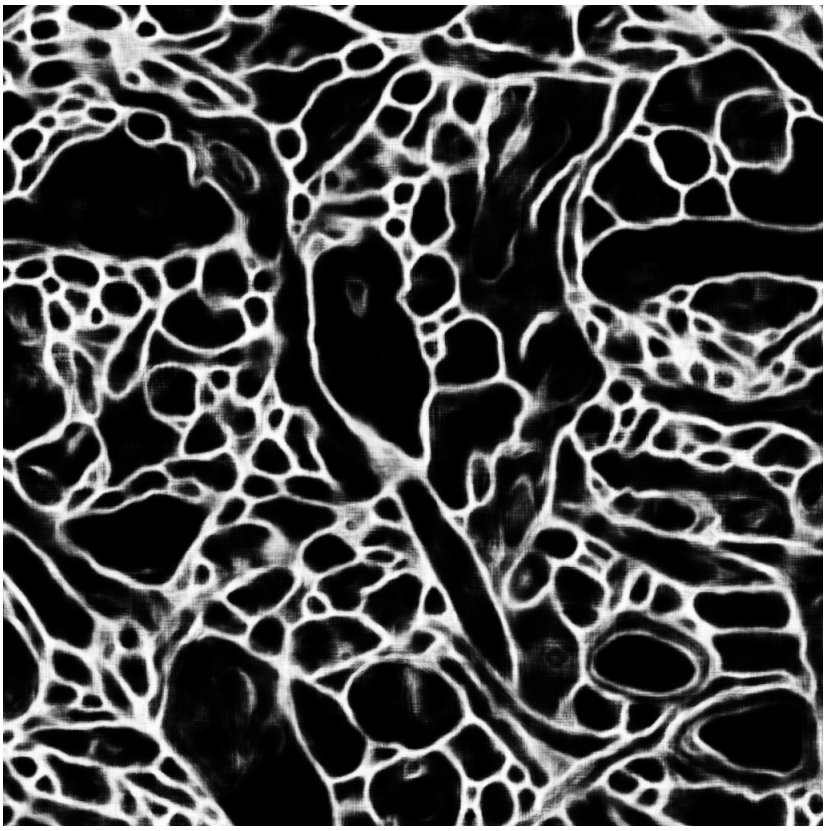

**Figure 2-2:** An example of a FCNN probability map

Note that while most of the image is white or black (corresponding to values of 1 or 0, respectively), various shades of gray also appear. The reason for this is that the values in the probability map directly correspond to the probability of the underlying voxel being a membrane, and the FCNN is often uncertain as to the correct designation. For the most part, ONR sets a threshold, conceptually binarizing the values into 1s and 0s.

### 2.3.1 Training the FCNN

The accuracy of the FCNN is a result of the extensive efforts that went into its construction and training. The training data of the neural network was a dataset that was manually annotated, providing a groundtruth that was used not only by the FCNN, but also by the ONR algorithm.

# Chapter 3

# Related Work

The need for an algorithm capable of reconstructing neurons in real time from brain imagery has existed for many years. The first attempts at reconstructing neuronal structures using computers date back to the mid-1960s.
Many solutions, with varying degrees of success, have been developed. For various reasons that will be discussed in this section, these solutions were not an appropriate choice for the imagery used by our group.

## 3.1 Kalman filters

In 1960, R. E. Kalman released a seminal paper that introduced what came to be known as Kalman filters [5]. Over the course of the last fifty years, Kalman filters have performed very well at predicting future states of noisy data. At first glance, it appears that Kalman filters would be an ideal tool for ONR. Indeed, Kalman filters have been used successfully in the past in similar endeavors. One such example involved tracing neurons in the optic tract of an embryonic zebra fish:

### 3.1.1 Axon tracking

Jurrus et al [6] describe the method for tracking a neuron's axon throughout the imagery of the optic tract: After the initial processing of the imagery, a Kalman filter tracking methodology was developed that allowed the researchers to follow an axon through several slices of images. As stated in the paper, the location of the axon in each image slice is predicted using the Kalman filter, through the use of contours and optic flow that serve as velocity and positional estimates.

Nonetheless, while the Kalman filter approach was very good at tracking axons within the optic tract imagery of zebra fish, it would not provide an ideal solution for our mouse brain imagery, for a number of reasons:

Firstly, the Kalman filter approach requires two seconds of searching per image slice. The optic tract imagery had a total of 100 images, which brought the total search time to 200 seconds. However, the mouse brain imagery that we are working with has approximately 2,000 images, and this number will increase significantly when we process larger portions of the brain. Furthermore, each of our image slices has a resolution of 3x3x29 nm, while the optic tract imagery had a resolution of 26x26x50 nm. In addition to the increase in resolution, our images are also vertically aligned with each other (unlike the images of the zebra fish). These two improvements suggest that it would be possible to develop a new algorithm to track neurons that could incorporate these advantages into its operation. A rough estimate shows that using the Kalman filter approach to track neurons in our image set could require roughly an hour of work, which is far too slow for our goal of online reconstruction.

## 3.2 Segmentation based approaches

Many approaches for reconstructing neuronal structures revolve around the use of Dense Segmentation (DS). The major difference between DS and techniques such as Kalman filters and ONR is that DS reconstructs *all* neurons in the dataset, rather than a single neuron. There are many variants of DS, but for the most part they follow a similar process. One

form of DS, as employed by the Lichtman Lab at Harvard [7], works as follows:

1. Every image in the dataset is passed through a classifier, such as a Random Forest classifier or a convolutional neural network. For every voxel in each image, the classifier determines the probability of it being a cell membrane (as opposed to the interior of a cell).

   After the classifier finishes computing, each image has a probability map that is the same size as the original image.

2. A watershed is run on the probability maps at different heights. This creates a set of different watershed transformations for each of the original images.

3. Using the resulting set of watershed transformations, an integer programming algorithm attempts to determine the best watershed transformation for each original image, such that the set of all chosen watershed transformations are optimally compatible with each other (compatibility is assessed using parameters such as overlap/similarlity between adjacent images, etc.)

This is a brief description of one DS approach. The important thing to note is that while DS is a very powerful tool that yields accurate results for the entire dataset, it necessarily has to perform *many* computational steps across the *entire* dataset.
The corrolary of this fact is that it is far slower than a technique such as ONR that merely seeks to reconstruct a single neuron, and can afford to ignore the vast majority of data.
So while DS is an excellent tool for many applications, it cannot (currently) be considered a valid tool for online reconstruction.

# Chapter 4

# The Online Neuron Reconstruction Algorithm

In this section I will present the ONR algorithm in detail and describe its implementation.

Some helpful terminology:

- *layer* - refers to the set of images with the same Z coordinate
- *neuron_local* - the area of a neuron within a given layer
- *neuron_cont* - the area of the continuation of *neuron_local* within a subsequent layer

## 4.1 The Algorithm's objective

At a basic level, the algorithm's objective can be described as follows:

*Given a voxel that represents the (approximate) center of a neuron,*
*find the center of the same neuron in the subsequent layer.*
*Repeat...*

Using the terminology introduced above, this could be rephrased as:

*Given the center of neuron_local, find the center of neuron_cont.*

*Repeat...*

This presents us with two questions:

1. How do we determine the center of a neuron within a given layer?

2. How can we find *neuron_cont* given *neuron_local*?

## 4.2 Determining the center of a neuron

ONR uses two primary methods to determine the center of a neuron within a given layer (i.e., the two dimensional XY center): Calculating the center by averaging the voxels belonging to the neuron; Applying the Euclidean Distance Transform.

### 4.2.1 Treating the center as an average of the voxels

The first method is straightforward: Given a set of voxels belonging to a neuron within a layer, the algorithm calculates the average X and Y values. It then treats the computed average of the two values as the center of the neuron.

This method is convenient and simple, but it suffers from certain disadvantages.
Firstly, it's possible that the average of the voxels lies outside the actual body of the neuron, as illustrated by Figure 4-1.

The second disadvantage is slightly more subtle. Although the images that are used by the algorithm are of very high quality, a certain amount of noise is almost unavoidable (noise can be caused in a number of ways, including post-mortem damage to the brain tissue).

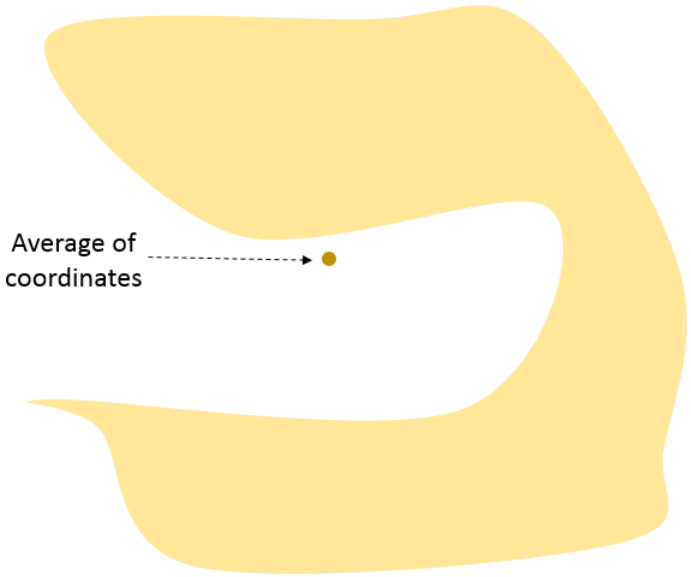


**Figure 4-1:** Occasionally, averaging the coordinates yields a center that is outside the body of the neuron

Such "noise" often exhibits itself in the form of a blotted out area within the image, obscuring or entirely replacing whatever neuron or other brain tissue existed at the original location.
ONR has a hard time dealing with such noise. If the computed center of the neuron falls within a noisy area, the algorithm may fail to continue the search, terminating prematurely.

Consequently, this method for determining the center of a neuron is only used when the algorithm is confident that the center lies within the body of the neuron, and does not overlap with a noisy area.
In most cases, this method is supplemented with the following approach.

### 4.2.2 Euclidean Distance Transform

The Euclidean Distance Transform (EDT) is an algorithm that operates on images. For each voxel in an image, EDT computes its euclidean distance from the nearest zero valued voxel. In the probability maps generated by the FCNN, neuron membranes are given a value of 1, while voxels that are inside the body of a neuron are given a value of 0.

Applying EDT to the probability maps (that are effectively the same as images) allows us to determine the distance of a voxel from neuronal membranes and from noisy areas that are treated by the algorithm as membranes.
Thus, after computing the average of the coordinates of the voxels within the body of the neuron, the algorithm calls EDT on a large patch of voxels centered around the computed average (the size of the patch will vary based on the size of the neron).
The algorithm will use the EDT values to decide which voxel should be designated as the neuron's center. The decision is a function of the voxels' proximity to the computed average, as well as the distance from membranes and noisy areas.

Supplementing the previous method with EDT provides ONR with a robust way of determining a neuron's center within a given layer. That being said, ONR will avoid using EDT when possible, due to its relatively high computational cost. For a more detailed explanation of EDT see Felzenszwalb (2004) [8].

## 4.3 Finding the continuation of a neuron

Having discussed how the algorithm determines the center of a neuron, it is time to turn to the primary question: Given *neuron_local*, how can *neuron_cont* be determined?

Recall that the input to the algorithm includes the approximate center of a neuron within a given layer. Assuming the algorithm can invoke subroutine `find_area` to find the area of the neuron within the current layer (this subroutine will be expanded on later in this section), the algorithm can proceed to find the continuation of the neuron as follows:

1. Using the assumption that the XY location of the center of *neuron_cont* is relatively close to the center of *neuron_local*, invoke `find_area` on the same XY location within the subsequent layer (i.e., only the Z coordinate is changed).

2. Compare the resulting area with the area of *neuron_local*. If the overlap is sufficiently high, declare the discovered area as *neuron_cont*.
   If the overlap is not high enough, continue invoking `find_area` on all unexplored locations in the subsequent layer, within a certain radius of the original location. The magnitude of the radius is primarily a function of the size of the neuron.

3. If the algorithm fails to find *neuron_cont* within the subsequent layer, explore the next two layers as well.

Note that once the algorithm has traced a neuron through a number of layers, it can optimize its search by utilizing additional information, such as the neuron's average XY drift from layer to layer and information regarding the shape of the neuron.
Furthermore, there are certain biological insights that are used by the algorithm to verify that a continuation is valid, such as the fact that a neuron cannot be enclosed within the body of another neuron.

One weakness of the algorithm (that will hopefully be addressed in the near future) is that it does not allow backtracking:
As described above, when failing to find a replacement within a subsequent layer, the algorithm will search for a replacement within the following two layers. For example, if the center of *neuron_local* is at XYZ location `(10,30,5)`, the algorithm will search for the continuation at Z layers `6`, `7` and `8`.
However, on rare occasions the neuron will not immediately advance to the next layer, but rather loop back to previous layers (using the aforementioned example, looping back to layer `4` before continuing to layer `6`), as demonstrated in Figure 4-2). In such cases the algorithm will not discover the continuation of the neuron, and the tracing will be terminated prematurely.
That being said, as stated, such occurrences do not appear to be frequent.

In this section we have described the general strategy of the algorithm: Given the center of a neuron within some layer, ONR will attempt to iteratively discover the centers of the neuron

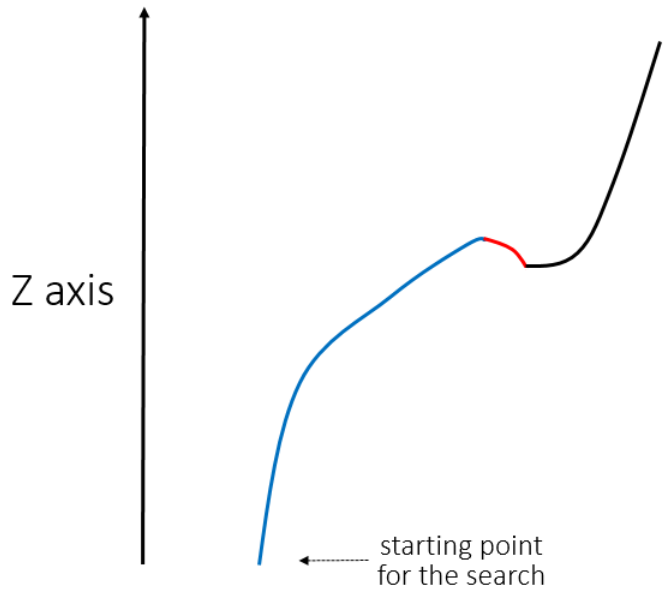


**Figure 4-2:** Failure to backtrack: After tracing the blue portion of the neuron, the algorithm would terminate the search at the beginning of the red portion, as it cannot backtrack

within subsequent layers (the entire algorithm will run once for the positive Z direction, and once again for the negative Z direction) until it either fails to find a continuation or until it reaches the end of the dataset. One critical component, which will now be discussed, is the `find_area` subroutine.

## 4.4 The `find_area` subroutine

Given a voxel as an input, `find_area` will attempt to discover the area of the neuron the voxel is contained within. In other words, `find_area` returns all voxels that belong to the same neuron as the inputted voxel.

In practice, the subroutine consists of a number of methods. The subroutine first attempts to apply the floodfill method (see below). If the overlap between *neuron_local* and the area discovered by the floodfill method is not sufficiently high, the subroutine will apply the rectangle search method. Failing that, it will attempt the next method, etc.

If all available methods fail, the algorithm will search for a valid continuation within a subsequent layer, as explained previously.

### 4.4.1 Floodfill

The starting voxel provided to the algorithm is treated by Floodfill as the first voxel belonging to the neuron. Floodfill then examines each of its neighboring voxels. Any voxel that is not a cell membrane is considered to be part of the area of the neuron as well.
Floodfill then recursively reapplies this approach to all these voxels (the neighboring voxels that were not cell membranes), examining *their* neighboring voxels. Floodfill continues in this manner until it has effectively "filled" the interior of the neuron, i.e., the end result is a set of voxels that belong to the interior of a neuron, all of which are surrounded by voxels that are part of a membrane.

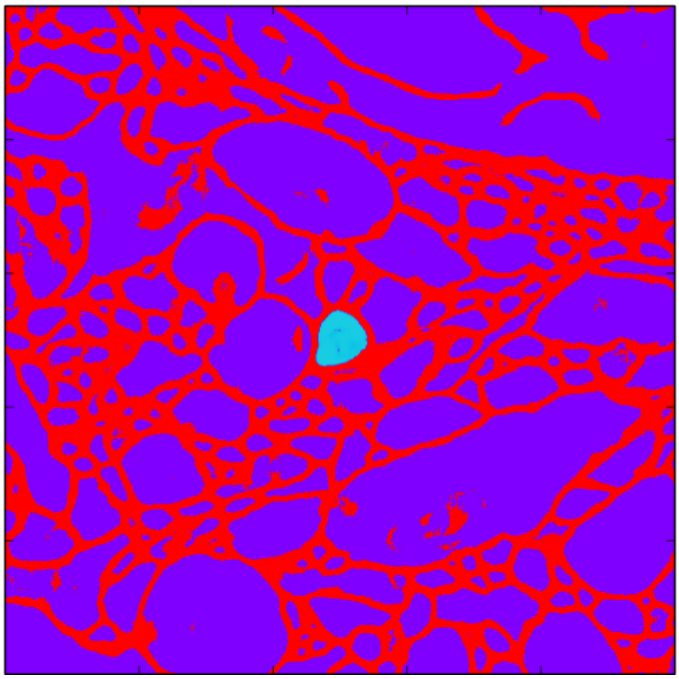

**Figure 4-3:** A neuron whose area was discovered using the Floodfill method

### 4.4.2 Rectangle Search

In most cases Floodfill performs well, in addition to being quite efficient. However, in some cases, it fails miserably. In reality, all neurons are enveloped by a membrane. However, due to noise present in the underlying image, as well as errors made by the FCNN, the proba-

bility maps occasionally label some voxels incorrectly, creating "holes" in the membranes. As a result, Floodfill will sometimes combine the area of multiple neurons! Figure 4-4 demonstrates such an occurrence.

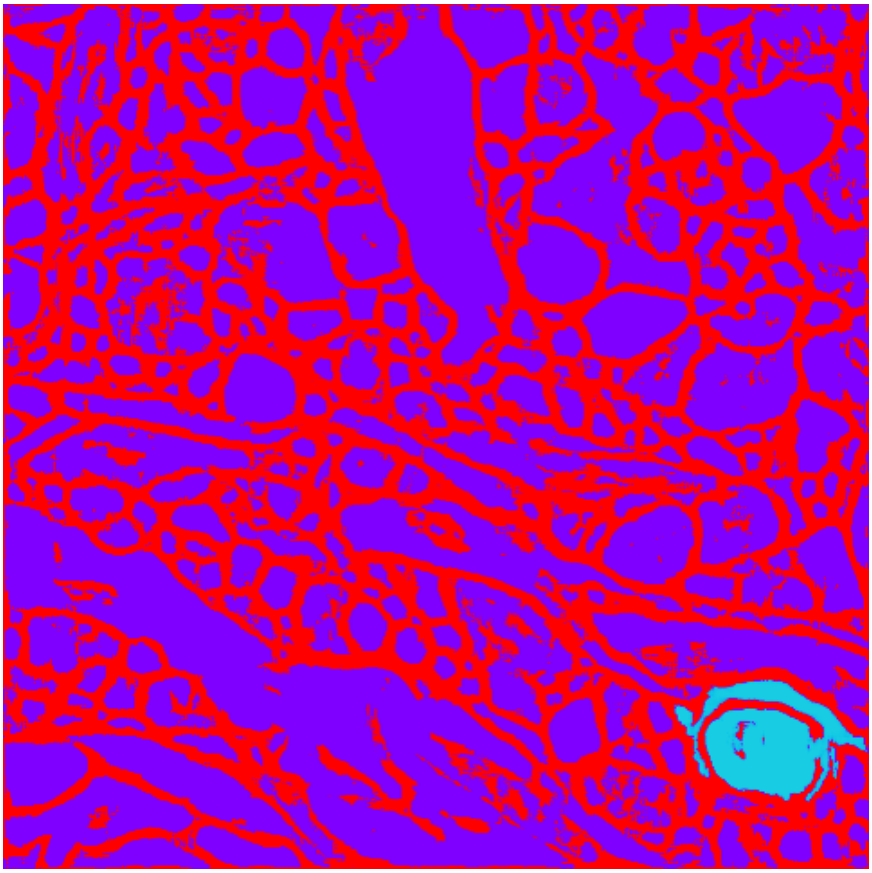

**Figure 4-4:** Floodfill erroneously combines the area of multiple neurons, due to a hole in the right side of the central neuron's membrane

In contrast to Floodfill, the Rectangle Search method is capable of handling such situations almost without failure. Essentially, Rectangle Search uses the starting voxel as a point from which it expands a large number of lines, each at a different angle. The lines are expanded until they encounter a cell membrane.

After all lines have been expanded, the maximum and minimum X and Y coordinates of each line are analyzed. The median value for each of the four values is used to determine the dimensions of a rectangle within which a localized floodfill is applied.

Confining the floodfill to the rectangle (whose dimensions are determined by the expanded lines) makes it extremely unlikely that the floodfill will include voxels not belonging to the

actual neuron.

The justification for this claim is as follows: In order for one of the sides of the rectangle to protrude past the interior of the neuron, a *majority* of the expanded lines - each of which were expanded in a different direction - would have to have encountered a hole in the membrane of the neuron. Given the overall accuracy of the images and the FCNN, this is highly unlikely.

Despite its extremely low error rate, rectangle search is still only used when the original, standalone Floodfill fails. The reason for this is twofold:

Firstly, Rectangle Search is almost twice as slow as Floodfill. Secondly, while it is unlikely that the area discovered by Rectangle Search will contain voxels not belonging to the target neuron, it will almost definitely not include all the voxels that *do* belong to the neuron, due to the rectangular constraint on the area, as shown in Figure 4-5.

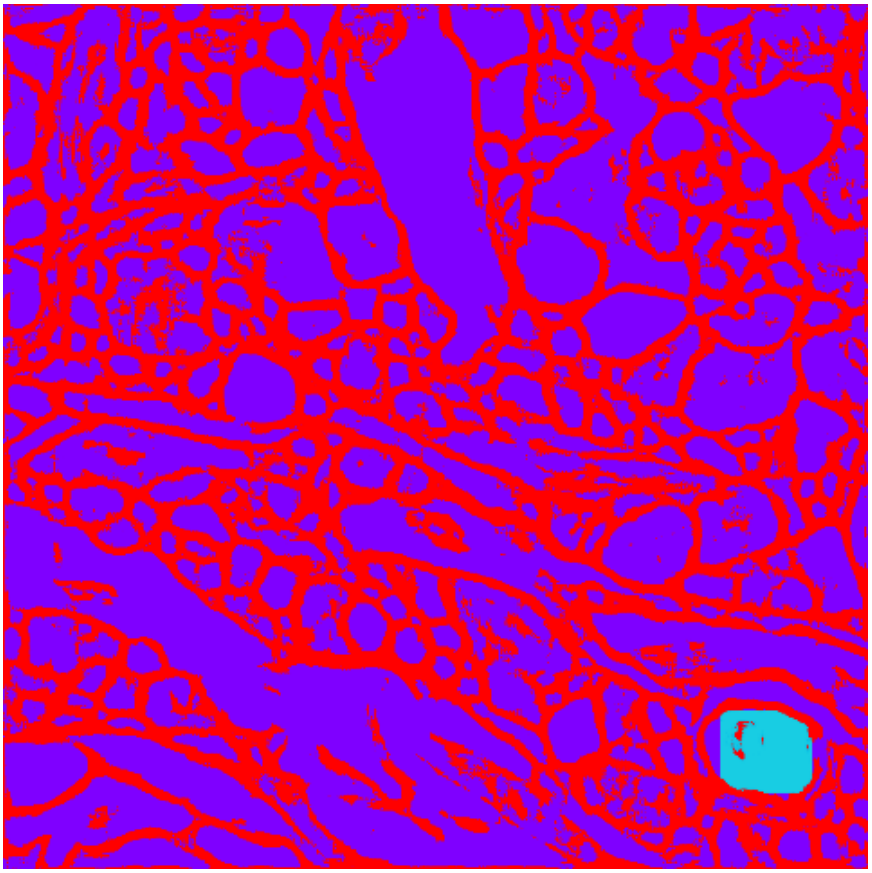

**Figure 4-5:** Rectangle Search. Note the hole in the right side of the membrane

### 4.4.3 Exhaustive Floodfill / Rectangle Search

The starting voxel for both Floodfill and Rectangle Search is based on the center of *neuron_local*. If the two methods fail to find a valid continuation using this starting voxel, it will exhaustively attempt to launch both methods using all voxels within a certain radius of the center of *neuron_local* as starting points. Although there may be many dozens of voxels within the radius, the algorithm will only attempt to launch the methods for voxels that have not yet been discovered by Floodfill or Rectangle Search. So in practice, only 2-3 new searches are launched.

### 4.4.4 Probabilistic Elimination

There are two likely causes for why the exhaustive application of Floodfill and Rectangle Search could fail:

1. The area of the image in which the neuron is found is particularly noisy.

2. The neuron is extremely small and the XY overlap between layers is relatively minor.

In such cases, the method that has proven to be most effective is Probabilistic Elimination: The Probabilistic Elimination method traces out the area of all neurons within a large radius surrounding the center of *neuron_local*, in the succeeding layer. For each neuron, the method calculates the probability that it is a continuation of *neuron_local*.
This calculation relies on the same considerations as those used by previous methods, i.e., XY overlap, similarity in shape, distance, etc. On the basis of these calculations, the method declares a neuron as *neuron_cont* if the calculated probability exceeds a certain threshold **and** the probabilities of all other neurons are significantly less than a second, lower threshold.
In other words, Probabilistic Elimination attempts to find one neuron that is a far better candidate for being the continuation to *neuron_local* than any other neuron, despite the fact that the actual XY overlap is not significant (due to noise in the image or the small size of the neuron).

## 4.5 Data representation

As mentioned previously, the sizes of the datasets used by the algorithm are enormous. It is not feasible to store an entire dataset in RAM, nor is it practical to store it as a single matrix on disk.

Instead, the images of the dataset are stored in dozens or even hundreds of folders, based on a methodology that corresponds to the layout of the images in three dimensional space. The algorithm uses an intermediary class to abstract this fact - the methods and subroutines used by ONR reference all voxels using an absolute coordinate system, and the intermediary class does the necessary conversion in order to fetch the appropriate data from RAM/disk.

## 4.6 General implementation details

As stated above, the algorithm itself has been implemented in Python, while making extensive use of a FCNN that is implemented in C. The algorithm imports numerous third party modules, including NumPy and SciPy. Asides from third party modules, the code consists of three primary components:

1. A central script, that manages the tracing of the neuron from layer to layer and decides when to terminate the search

2. An intermediary class that abstracts the access to the data

3. The various `find_area` subroutines

All three components are implemented in Python.

## 4.7 Optimizations

In the interest of clarity, I have glossed over various optimizations made to the algorithm, including two that are now worth mentioning:

- **Skipping layers:** Rather than trace the neuron layer by layer, the algorithm presents the option of skipping adjacent layers, thus speeding up the runtime. This effectively halves both the runtime and the amount of data outputted by the algorithm.

  As the layers that are skipped are preceded and followed by layers in which the neuron *is* traced, the user can still, fairly trivially, deduce the outline of the neuron as a whole. However, if an extremely precise reconstruction is required, this option may not be suitable.

- **Sub-sampling:** Similar to the option presented above, sub-sampling allows the user to prioritize speed over the comprehensiveness of the output. When sub-sampling is enabled, only a representative subset of the area of the neuron within each layer is outputted. This subset is sufficient to allow researchers to deduce the shape/location of the neuron, as well as which other neurons it is adjacent to.

# Chapter 5

# Performance and Analysis

## 5.1 Speed

As its name suggests, the Online Neuron Reconstruction algorithm must execute very quickly in order to provide users with real time results. This consideration is in fact one of the primary concerns of the entire MIT Computational Connectomics Group, guiding the design of all systems developed within the group, including those that do not provide real time results.

Fortunately, the FCNN developed by the group is extremely efficient, utilizing massive parallelization and extensive optimizations that allow it to meet the speed constraints of the group's various systems. The FCNN is the most complex and advanced tool used by ONR, and as such, the work that has gone into guaranteeing its high performance has been essential.

Asides from the FCNN, the components of the algorithm are fairly straightforward. The performance-critical portions of the algorithm have been carefully written in order to maximize the speed of their execution.

Performance related metrics:

- It takes roughly 60 seconds to trace a medium sized neuron (consisting of ~4,000

voxels, translating into an area of roughly 0.144 $mm^3$) through 1,000 layers (29 μm).

- Average number of Python bytecode operations per layer (for a medium sized neuron): ~800,000.

- Percentage of time spent loading data: ~ 12%. This includes the overhead required for an associated cache that aids with data access.

In order to fully understand the benefit of ONR it is helpful to contrast it with the traditional reconstruction of the entire dataset. Currently, the pipeline developed by the MIT Computational Connectomics Group can provide an automatic reconstruction/annotation of a ~90GB dataset in 10 hours. While this is incredibly fast, considering the size of the data and the complexity of the task, a user interested in analyzing a (much) smaller subset of neurons could use ONR to acquire the same information for the few neurons of interest in a fraction of the time.

## 5.2 Accuracy

Before discussing the accuracy of the algorithm, it is important to state that accuracy is inversely related with the depth of the tracing. In other words, the greater the number of layers that are traced by the algorithm for a given neuron, the greater the chance of an error occuring. The algorithm has an internal "tradeoff" parameter that can be configured to determine which of the two to prioritize - the depth of the tracing or the accuracy, i.e., whether the algorithm should attempt to trace the neuron to the greatest extent possible, with an increased risk of making mistakes; or whether the algorithm should act conservatively, minimizing the risk of error at the price of potentially terminating the tracing relatively early.

In order to measure its accuracy, the algorithm was evaluated using the groundtruth of 500 neurons located in the S1 region of a mouse brain. For every layer in which the neuron

was traced by the algorithm, the reported center was compared with the groundtruth to ascertain that it indeed corresponded to the neuron in question. By repeating this evaluation over every traced layer for each of the 500 neurons, the following metrics were reached:

- A medium setting of the tradeoff parameter gives an accuracy of 98%, with a reconstruction depth of approximately 30.5%. That is, on average, the algorithm correctly identified the centers of each of the 500 neurons within $\sim 30\%$ of the layers traversed by each neuron. A small number of additional reported centers were incorrect, belonging to other neurons.
- Setting the tradeoff parameter to prioritize accuracy over depth yields an accuracy of more than 99%, though the reconstruction depth decreases considerably, to only slightly over 20%.

## 5.3 Causes for errors

As alluded to previously in the paper, errors are caused primarily by three culprits: Noise present in images, mistakes in the probability maps generated by the FCNN, and (primarily) neurons with relatively small XY areas.
In fact, once the area of a neuron exceeds a certain size, **no** errors have yet been encountered. Similary, practically all of the early terminations of the algorithm occur when the neuron is relatively small.

Accordingly, the primary area for improvement in ONR is the tracing of small neurons.

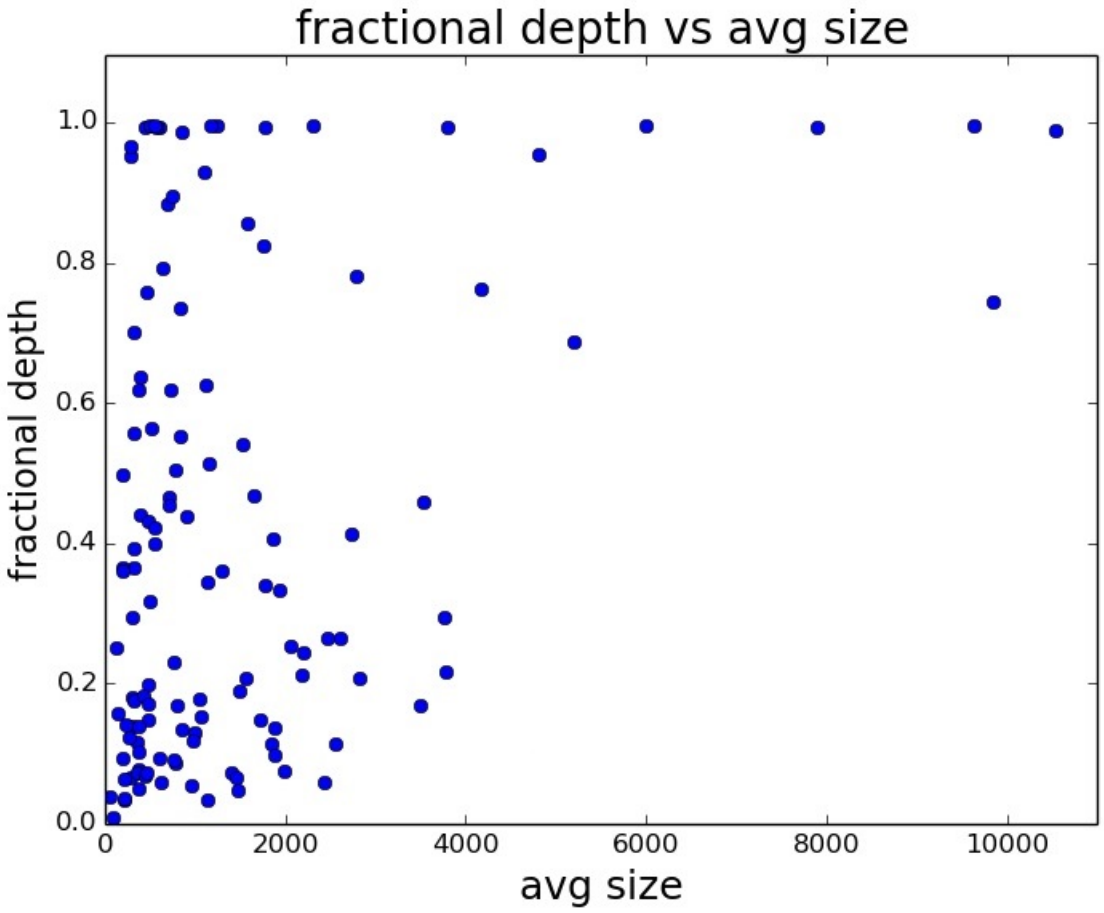


**Figure 5-1:** Fraction of the neuron that was traced successfully versus the average XY area. A value of 4,000 (voxels) corresponds to a medium sized neuron

# Chapter 6

# Future Work

## 6.1 Future applications

ONR's primary use case is to serve as a tool for researchers, providing real time results to queries regarding single neurons. Additionally, it can be used to improve dense segmentations, as outlined earlier in the paper.

One possibility that needs to be explored is using the algorithm to generate seeds for dense segmentations: Currently, dense segmentations initiate a large number of random seeds from which the watershed process (that is part of the segmentation) is begun. Theoretically, the algorithm could be used to improve this selection process, e.g., by verifying that two seeds do not belong to the same neuron.

An additional application that is worth exploring includes applying the algorithm to unaligned datasets. The datasets presently provided to the MIT Computational Connectomics Group by the Lichtman Lab have already been vertically aligned. However, it is possible that with some minor adjustments, ONR will be able to run on unaligned datasets as well - which will shorten the overall time researchers have to wait to run queries on datasets, as the image alignment process could be skipped (note that alignment also increases the size of the dataset considerably, so this optimization would save a considerable amount of storage space as well).

## 6.2 Improving Speed

Currently, the algorithm is implemented in Python (with the exception of the FCNN). Porting the code to C/C++ would undoubtedly speed up the algorthim's runtime.
Another major area for improvement is the way in which the algorithm calls the FCNN. At present, ONR calls FCNN on large blocks (of a predetermined, fixed size) of the underlying image. Most of the information in the resulting probability maps is irrelevant and not used by the algorithm. Making the calls more granular would allow the FCNN to yield results much faster, thus shortening the overall runtime of the algorithm.

## 6.3 Improving Accuracy

One way to improve the accuracy of ONR's results is by executing the algorithm twice, in parallel, using two different types of probability maps: For each layer, one probability map would be generated by the FCNN, while a second map would come from a Random Forest classifier. Overall, the generated probability maps would be extremely similar. However, in some noisy or questionable areas of the images the two maps would likely differ, potentially altering the decision making process of the algorithm. After the two executions of the algorithm would finish running, their results would be compared, and only the layers for which the results were the same would be outputted.
This approach has been tested in practice, and has been shown to improve the accuracy of the reported centers. As the two executions could be run in parallel, the overall runtime would barely increase, except for a short post-processing step after the conclusion of the two executions.

## 6.4 Increasing the average reconstruction depth

The most significant challenge to ONR is the reconstruction of small neurons. Large neurons are handled very well, and in the vast majority of cases, ONR successfully traces them

in their entirety. However, small neurons remain a challenge, often causing ONR to terminate prematurely, rather than being traced throughout the dataset.
Clever modifications and additional techniques will be required in order to successfully handle small neurons and increase the depth of their reconstruction.

## 6.5 Adding a GUI to ONR

In the near future, a GUI will be added to ONR to simplify its use. The envisioned user experience is that the user will open a GUI that presents the original, raw imagery. The user will first set certain optional parameters, such as specifying the vertical direction of exploration, the maximum desired reconstruction depth, etc. After setting these optional parameters, the user will click on any voxel in the image. The coordinates of the voxel will be passed to ONR, which will then perform a reconstruction of the neuron that the voxel belongs to, using the parameters configured by the user.
While the search is being performed, the screen will present the progress of the algorithm, outputting the results of the reconstruction in real time. How exactly the results will be presented could also be determined by the user: The centers of the neuron within each layer could be overlayed on the raw imagery, or alternatively, the area could be displayed. The algorithm provides both outputs.

# Chapter 7

# Conclusion

The Online Neuron Reconstruction algorithm traces neurons throughout image datasets in real time. The algorithm relies on a Fully Convolutional Neural Net and a number of simpler methods and subroutines. While there remains work to be done in order to bring the algorithm to its full potential, ONR is already quite capable of providing researchers with real time results to queries, as well as aiding in improving the results of dense segmentations.

# Bibliography


[1] Amelio Vázquez-Reina, Won-Ki Jeong, Jeff Lichtman, and Hanspeter Pfister. The connectome project: Discovering the wiring of the brain. *XRDS*, 18(1):8–13, September 2011.

[2] Joshua L Morgan et al. The fuzzy logic of network connectivity in mouse visual thalamus: Cell. *Cell*, 165:192–206, 2016.

[3] Narayanan Kasthuri et al. Saturated reconstruction of a volume of neocortex. *Cell*, 162(3), 2015.

[4] A. L. Eberle, S. Mikula, R. Schalek, J. Lichtman, M. L. Knothe Tate, and D. Zeidler. High-resolution, high-throughput imaging with a multibeam scanning electron microscope. *Journal of Microscopy*, 259(2):114–120, 8 2015.

[5] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[6] Elizabeth Jurrus et al. Axon tracking in serial block-face scanning electron microscopy. *Medical image analysis*, 13(1), 2009.

[7] Amelio Vazquez-Reina et al. Segmentation fusion for connectomics. Lichtman Lab. Presented at IEEE International Conference on Computer Vision.

[8] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance Transforms of Sampled Functions. 2004.