Automated Seed Point Selection in Confocal Image Stacks Of Neuron Cells

Gregory P. Bilodeau

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

Csaba L. Egyhazy, Chair
Ing-Ray Chen
Gregory Kulczycki

January 2, 2013
Falls Church, Virginia

Keywords: image analysis, DAIDEM, neuron tracing

Automated Seed Point Selection in Confocal Image Stacks of Neuron Cells
Gregory P. Bilodeau

ABSTRACT

Research into neurological disease and function requires an understanding of how neuron cells connect and interact – a function of their shape, or morphology. To determine the morphology of a neuron, researchers first obtain a series of images of the neuron through a microscope taken at different focal depths called an image stack. This stack is then used to manually trace the neuron's 3-dimensional structure, usually using a computer program to manipulate the stack images and track the points created by the researcher.

Although the technology necessary to capture these image stacks has advanced in terms of speed and cost, this process of reconstructing the morphology of neuron cells from image stacks remains a manual, time-consuming and subjective process sometimes taking weeks to months to complete. Fully automated systems capable of constructing these 3-dimensional models with little to no input from the researcher would greatly speed up the research process. Previous attempts to solve this problem have utilized many different techniques, and a recent competition called the DAIDEM Challenge sought to compare different approaches in a competitive manner in order to determine the most effective approaches and spur further innovation in the field.

Many algorithms require the user to supply seed points, specific coordinates in 3-dimensional space that represent a known area of the morphology from which the algorithm can begin a search. Proper selection of seed points is a critical task, as many algorithms utilize the local image information around the seeds to direct the tracing process. Knowledge of the global image space is useful in fine tuning the filters and kernels used to produce a successful tracing.

In this paper, I propose a fully-automated method of seed point selection that produces both a comprehensive overview of the global image environment and a set of high-probability seed points while performing faster than conventional means. I demonstrate that seed points can be successfully selected from a single image representation of the image stack known as a maximum intensity projection (MIP). I then propose that constructing the MIP from a sub-sample of the total collection of slice images produces equivalent seeding results while providing an increase in execution speed.

The resulting seed set produced from this approach is then compared to a gold-standard set of reconstruction points to show their validity. The seeds generated by this approach are suitable for use in any number of tracing algorithms that require initial seed placement, and the global image information catalogued may be used by successive tracing algorithms to better adapt to image irregularities and select appropriate tracing filters and kernels.

Contents

1 Introduction 1
1.1 Definitions and Terminology 1
1.2 Goals and the Current Status 3
1.3 Problem Statement 6
2 Background Research 9
2.1 Global Search Methods 10
2.2 Local Search Methods 14
3 Materials & Implementation 20
3.1 Materials 20
3.2 Implementation 21
3.2.1 Algorithm Objectives 21
3.2.2 Seed Metric Definitions 22
3.2.3 Algorithm Description 23
3.2.3.1 Collapse Search Space 23
3.2.3.2 Thresholding and Segmentation 29
3.2.3.3 Intensity Examination 31
3.2.3.4 Seed Point Selection 32
3.2.3.5 Establish Connectivity and Seed Pruning 37
3.2.3.6 Group Connections 42
3.2.3.7 Preparing and Exporting Data 44
4 Results 46
5 Discussion 80
6 Conclusion 86
Works Cited 89
Appendix A – Notes on Data Sets 91
Appendix B – Source Code 92

List of Figures

Figure 1 - three sequential slices from an image stack with exaggerated z-axis displacement 2
Figure 2 - the OP_1 stack's gold-standard SWC file 3
Figure 3 - a 3-dimensional rendering of the OP_1 image stack from the gold-standard SWC file 3
Figure 4 – a screenshot of the Neuromantic program showing a manual reconstruction overlaid on a slice from an image stack 4
Figure 5 - animation from point A to B over 24 frames 24
Figure 6 - animation from point A to B over 12 frames 24
Figure 7 - animation requiring more than two points to describe over 24 frames 24
Figure 8 - three points are enough to describe the basic path of motion 24
Figure 9 - two frames does not contain enough information to describe the path accurately 25
Figure 10– images 28 and 29 from the OP_1 image stack 25
Figure 11– subtractive difference between slices 19-20, 20-21, and 19-21 26
Figure 12– completely collapsed ray projections of the OP_1 and OP_2 image stacks 26
Figure 13– ray projections of the OP_1 and OP_2 image stacks using half the slices 27
Figure 14– difference map of the OP_1 image stack between the 1 and 2 sample rates (inverted to show fine detail) 27
Figure 15– the thresholding algorithm in process, showing the lower threshold (LT), mean (ME), mode (MD) and upper threshold (UT) values 29
Figure 16– the original ray projection (left) and thresholded version 29
Figure 17– detailed view of original ray projection 30
Figure 18– detailed view of thresholded ray projection 30
Figure 19– a centerline described using maximal disks 33
Figure 20– the seed selection kernel at size 1 and 2. The blue square reflects the current center of the potential Coord. The yellow squares reflect the pixels being tested. Detection of a blank pixel triggers the movement of the center pixel before expanding the kernel 34
Figure 21– the results of the seeding algorithm 35
Figure 22– Coord signal averaging and peak determination over indices; red arrow indicates change in slope sign, peak found to be index 20 36
Figure 23– the seeds colored by depth; red indicates higher on the z-axis, blue lower 37
Figure 24– the result of the seed connection algorithm 40
Figure 25– the refined connections 42
Figure 26– fragment assignment and singletons; red borders represent higher top-extrema of fragment members, blue lower 43
Figure 27 - example output file 45
Figure 28– left column: the seeds matching the gold standard (winners set); right column: seeds rejected (losers set), at tolerance levels of 1 (a), 2 (b), 3(c) and 4(d) pixels for OP_1 at sample rate 2 47
Figure 29 - OP_1 MIP 49
Figure 30 - OP_2 MIP 49
Figure 31 - OP_3 MIP 49

Figure 32 - OP_4 MIP........ 49
Figure 33 - OP_5 MIP........ 50
Figure 34 - OP_6 MIP........ 50
Figure 35 - OP_7 MIP........ 50
Figure 36 - OP_8 MIP........ 50
Figure 37 - OP_9 MIP........ 51
Figure 38 - set of final seeds for OP_1, sample rate 1........ 52
Figure 39 - set of final seeds for OP_1, sample rate 2........ 52
Figure 40 - winner set, sample rate 1, 1 pixel threshold........ 52
Figure 41- winner set, sample rate 2, 1 pixel threshold........ 52
Figure 42- winner set, sample rate 1, 4 pixel threshold........ 52
Figure 43- winner set, sample rate 2, 4 pixel threshold........ 52
Figure 44 - set of final seeds for OP_2, sample rate 1........ 54
Figure 45 - set of final seeds for OP_2, sample rate 2........ 54
Figure 46 - winner set, sample rate 1, 1 pixel threshold........ 54
Figure 47- winner set, sample rate 2, 1 pixel threshold........ 54
Figure 48- winner set, sample rate 1, 4 pixel threshold........ 54
Figure 49- winner set, sample rate 2, 4 pixel threshold........ 54
Figure 50 - set of final seeds for OP_3, sample rate 1........ 56
Figure 51 - set of final seeds for OP_3, sample rate 2........ 56
Figure 52 - winner set, sample rate 1, 1 pixel threshold........ 56
Figure 53- winner set, sample rate 2, 1 pixel threshold........ 56
Figure 54- winner set, sample rate 1, 4 pixel threshold........ 56
Figure 55- winner set, sample rate 2, 4 pixel threshold........ 56
Figure 56 - set of final seeds for OP_4, sample rate 1........ 58
Figure 57 - set of final seeds for OP_4, sample rate 2........ 58
Figure 58 - winner set, sample rate 1, 1 pixel threshold........ 58
Figure 59- winner set, sample rate 2, 1 pixel threshold........ 58
Figure 60- winner set, sample rate 1, 4 pixel threshold........ 58
Figure 61- winner set, sample rate 2, 4 pixel threshold........ 58
Figure 62 - set of final seeds for OP_5, sample rate 1........ 60
Figure 63 - set of final seeds for OP_5, sample rate 2........ 60
Figure 64 - winner set, sample rate 1, 1 pixel threshold........ 60
Figure 65- winner set, sample rate 2, 1 pixel threshold........ 60
Figure 66- winner set, sample rate 1, 4 pixel threshold........ 60
Figure 67- winner set, sample rate 2, 4 pixel threshold........ 60
Figure 68 - set of final seeds for OP_6, sample rate 1........ 62
Figure 69 - set of final seeds for OP_6, sample rate 2........ 62
Figure 70 - winner set, sample rate 1, 1 pixel threshold........ 62
Figure 71- winner set, sample rate 2, 1 pixel threshold........ 62
Figure 72- winner set, sample rate 1, 4 pixel threshold........ 62
Figure 73- winner set, sample rate 2, 4 pixel threshold........ 62
Figure 74 - set of final seeds for OP_7, sample rate 1........ 64
Figure 75 - set of final seeds for OP_7, sample rate 2........ 64
Figure 76 - winner set, sample rate 1, 1 pixel threshold........ 64
Figure 77- winner set, sample rate 2, 1 pixel threshold........ 64

Figure 78- winner set, sample rate 1, 4 pixel threshold 64
Figure 79- winner set, sample rate 2, 4 pixel threshold 64
Figure 80 - set of final seeds for OP_8, sample rate 1 66
Figure 81 - set of final seeds for OP_8, sample rate 2 66
Figure 82 - winner set, sample rate 1, 1 pixel threshold 66
Figure 83- winner set, sample rate 2, 1 pixel threshold 66
Figure 84- winner set, sample rate 1, 4 pixel threshold 66
Figure 85- winner set, sample rate 2, 4 pixel threshold 66
Figure 86 - set of final seeds for OP_9, sample rate 1 68
Figure 87 - set of final seeds for OP_9, sample rate 2 68
Figure 88 - winner set, sample rate 1, 1 pixel threshold 68
Figure 89- winner set, sample rate 2, 1 pixel threshold 68
Figure 90- winner set, sample rate 1, 4 pixel threshold 68
Figure 91- winner set, sample rate 2, 4 pixel threshold 68
Figure 92 - image mean for the stacks 73
Figure 93 - standard deviation for the stacks 73
Figure 94 - signal to noise ratio for the stacks 73
Figure 95 - threshold levels for the stacks 73
Figure 96 - number of pixels in the ray projection that were found to have a intensity value greater than 0 74
Figure 97 - number of seeds found during prelininary seed discovery and resulting from additions during slice assignment 74
Figure 98 - average seed radius 75
Figure 99 - average seed intensity 75
Figure 100 - number of areas identified as being ambiguos and in need of extra refinement 75
Figure 101 - hit percentage for OP_1 77
Figure 102 - hit percentage for OP_2 77
Figure 103 - hit percentage for OP_3 77
Figure 104 - hit percentage for OP_4 77
Figure 105 - hit percentage for OP_5 77
Figure 106 - hit percentage for OP_6 77
Figure 107- hit percentage for OP_7 78
Figure 108 - hit percentage for OP_8 78
Figure 109 - hit percentage for OP_9 78
Figure 110 - average difference in seed radius from matching SWC points 78
Figure 111 - execution times for the sample rates 1 and 2 for all stacks 79
Figure 112 - visual validation of final seed set, over laid ray projection 80
Figure 113 - example of the image ambiguity areas record in the output.txt file 82
Figure 114 - seed number by sampling rate 83
Figure 115 - detailed view of the complex region of OP_4 84
Figure 116 – average hit percentages of the sample rates 85

List of Tables

Table 1 - break down of prior algorithms by local or global methods 19

Table 2 - list of the nine drosophila olfactory axonal projection image stacks and their slice counts 20

Table 3 - image statistics results 70

Table 4 - seeding results 70

Table 5 - image ambiguity alert results 71

Table 6 - SWC validation hits/misses for sample rate 1 71

Table 7 - SWC validation hits/misses for sample rate 2 72

Table 8 - hit percentages for each stack, sampling rate and error margin 76

Table 9 - average execution time over 3 runs per sampling rate 79

Table 10 - areas of ambiguity identified by sample rate 81

1 Introduction

The use of computer algorithms to assist researchers in the investigation of biological systems has become commonplace. Many areas of study, such as genomics, would be impossible without specialized programs to help sort, store and interpret the vast amounts of data generated within this domain. When the data to be processed represents easily verified and quantifiable units – such as base pairs within a gene or heart beats per minute – the application of computer processing techniques is fairly straightforward. However there are other types of data that are not so easily interpreted, even by human researchers with our potent pattern recognition systems.

Many investigations of systems within the human body must rely on data gained through imaging methods. X-rays, MRIs and echocardiograms can provide researchers with a view of internal organs, bones and tumors. Similarly, smaller structures such as blood, bacteria or nerve cells may be viewed using microscopes connected to computers to capture the images. Algorithms used to interpret these images must be able to overcome several complications deriving from the imaging process that introduce possible errors into the data. The smaller the structure being imaged, the larger the impact image errors can produce.

This paper attempts to advance the pool of algorithms designed to create 3-dimensional reconstructions from image stacks of nerve cells. The approach outlined here is meant to serve as a stepping stone for other algorithms by providing an efficient method of selecting starting seed points, estimating the gross topology of the cell and providing data about the images that can be used to minimize the effect imaging errors can have on the final output.

1.1 Definitions and Terminology

For those who may not be familiar with the subject of biology, I will first define some terms that will be used in this paper.

A nerve cell, or neuron, is a specialized cell within the body that functions as a message propagation system utilizing electrical and chemical signals. They are found within the brain and nervous system. Neurons form connections with other neuron, called synapses, and pass messages between their connections using chemicals called neurotransmitters. A neuron consists of three main parts: a single soma, an axon and multiple dendrites.

The soma is the main body of the neuron. It contains the cell nucleus and most of the organelles. It is generally the largest portion of the cell in terms of volume. From the soma branch the axons – long fiber-like projections that carry signals away from the soma during propagation. In the opposite direction across the soma from the axons are the dendrites, which function as input receptors for signals across a synapse. A neuron may have any number of dendrites, but only a single soma and axon – although the axon commonly branches many times over its length.

A principle concern of researchers is the three-dimensional shape of the neuron, or the morphology. Neuron cells are often described in terms of arbors for their tree-like structures

across the 3 dimensions. In order to convey this three-dimensional information via the two-dimensional image plane of the microscopes, researchers create image stacks of the cells. Each stack contains multiple slices, images of the cell at a specific sequential focal depth. Focal depth can be described as the distance from the viewer of a plane orthogonal to the image plane of the microscope.

Figure 1 - three sequential slices from an image stack with exaggerated z-axis displacement

Images of neurons are captured using various forms of microscopes. The data sets used in this project were captured using a technique called confocal microscopy. In contrast to wide-field microscopy, which simply illuminates an entire subject with light and produces an image comprised of intensities throughout the structure including an out-of-focus background, confocal microscopy makes use of point illumination and a spatial pinhole to eliminate the out-of-focus signals. While this results in a much clearer image of the subject at the specified focal depth, it also means that the signals being received are lower in intensity as they are partially obscured by the pin hole. The images obtained in the data sets used here have a good focal cohesion.

The goal of reconstruction algorithms is to convert this image stack into a tracing or reconstruction. A reconstruction is a representation of the neuron's morphology that is concise and complete. Reconstructions consist on a tracing of the centerlines of the neuron and an estimation of its volume and shape at specific points in 3 dimensional space.

Reconstructions of neurons are saved as a SWC formatted file, a non-proprietary format designating nodes and interconnections. Each line of the SWC file denotes a neuron classification, node identity, node location in the X/Y/Z space, node radius and is a form of directed graph where each node indicates its parent by referencing a previous node at the end of its line.

```
OP_1.swc - Notepad
File  Edit  Format  View  Help
# Neurolucida to SWC conversion from L-Measure.
1 2 30.979 429.04 0.000 0.303 -1
2 2 30.567 428.01 0.336 2.2816 1
3 2 30.182 427 0.681 3.9617 2
4 2 29.797 426.07 1.052 4.4086 3
5 2 29.521 425.09 1.419 4.5102 4
6 2 29.249 424.09 1.784 4.5753 5
7 2 29.033 423.06 2.142 4.2784 6
8 2 28.876 421.95 2.482 4.3753 7
9 2 28.73 420.84 2.819 4.3905 8
10 2 28.615 419.7 3.152 4.3738 9
11 2 28.57 418.58 3.491 4.042 10
```

Figure 2 - the OP_1 stack's gold-standard SWC file

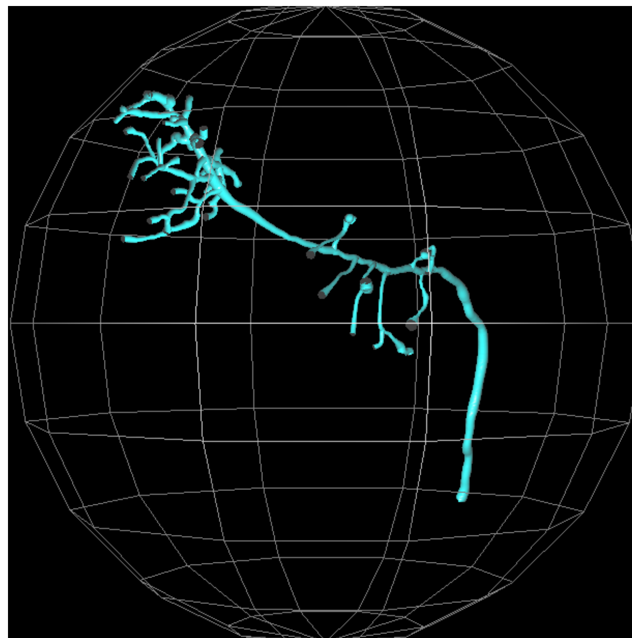


Figure 3 - a 3-dimensional rendering of the OP_1 image stack from the gold-standard SWC file

1.2 Goals and the Current Status

The function of a neuron cell derives from its shape. Neurons propagate signals between themselves by way of chemical signals released at synapses, small channels between adjacent neuron cells. How these signals flow, or do not, can be the difference between a healthy and diseased individual. Studying the morphology of neuron cells can reveal how different cells may connect – or not – at various points in the nervous system. From a greater understanding of how these cells fit together researchers may be able to identify causes for diseases and generate treatments and cures.

However neurons are simultaneously microscopic, morphologically complex and can have a huge number of interconnections. Being able to map or reconstruct these cells accurately is a daunting task and currently a manual and time-consuming one. Researchers must painstakingly photograph these cells through a microscope, adjusting the focal depth

progressively to obtain a complete record of the 3-dimensional structure. These image sequences or stacks are then manually examined, with the researcher zooming to pixel-level detail to determine the centerlines, diameters and connectivity of branch of the neuron's tree like structure. Each point in the stack that describes a change in the neuron's structure must then be selected, quantified and assigned an order to produce a full reconstruction.

This process may take weeks to months of continuous effort. Creating these image stacks takes far less time, resulting in an ever-expanding backlog of neuron image stacks awaiting analysis.

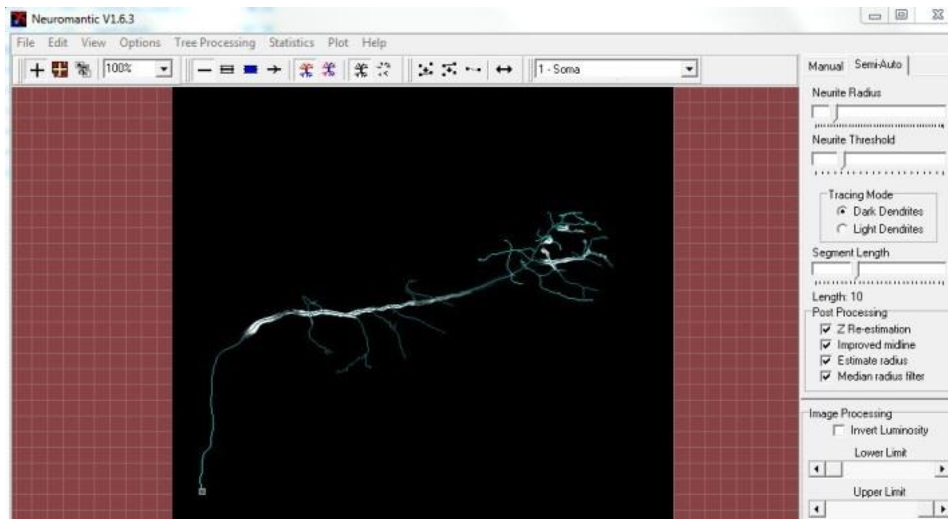


Figure 4 – a screenshot of the Neuromantic program showing a manual reconstruction overlaid on a slice from an image stack

A fully automated method of analyzing these image stacks would greatly accelerate the pace of neuron research and potentially have an impact on the release of new medicines and treatments for disease. However all attempts to create such an automated method to date have fallen short of the requirements of researchers. Many require users to provide numerous seed points for their algorithms to start from, shifting the burden of gross-pattern recognition back to the human. A researcher must still visually inspect the stack, select points that they determine are a part of the neuron structure and pass them along to the program.

Some systems utilize training stacks and reconstructions to calibrate their systems, requiring researchers to do additional collection and evaluation. These systems also tend to be slower in execution, as they require time to build an understanding of the image space and establish models to work from. Ultimately the larger the pool of information that must be read into memory from physical media the slower the system will function.

The ultimate goal of researchers is a program that is fully automated, fast and accurate. Requirements such as seed points, training stacks and other user-supplied inputs present a large obstacle toward processing these stacks at the speed required to stay current with the pace of microscopy capture. However systems that are slow or require specialized equipment are not desirable either, as they represent a barrier to entry to some research groups. Accuracy in

reconstructions is unfortunately difficult to quantify, and will most likely always require some level of user validation [1].

With this in mind, the immediate goal of research in this domain should be on providing solutions that are fully automated, fast and run on common equipment. If algorithms that generate reconstructions can quickly and automatically produce SWC files that then only need to be validated by a researcher, which would represent a much needed improvement over the current manual method of tracing.

A recent competition called the DIADEM (Digital Reconstruction of Axonal and Dendritic Morphology) Challenge sought to stimulate development of semi- and fully-automated reconstruction algorithms and to gauge the state of the art techniques currently being used. Although the goal of reaching a 20 fold increase in the time required to reconstruct a neuron from an image stack was not met, the winning algorithms came very close.

Competitors were supplied with a collection of image stacks, along with their manually created ‘gold-standard’ reconstructions, and were given a year to develop an algorithm capable of reproducing the reconstructions within a set tolerance. They were then presented with an additional set of image stacks that they had not seen before, and the top performers were invited to the final competition. The specific finalist algorithms and their features and results are discussed in the background research section below.

1.3 Problem Statement

Computer image analysis is a well-studied field. Techniques such as edge and blob detection have a large collection of literature associated with them, and are valuable in the domain of computer vision and face recognition. Various approaches also exist that utilize generalized models of objects to estimate and match their shapes to objects imaged. All of these techniques are concerned with extracting specific information from an image by utilizing local image information, but may also use generalized information of the entire data set to provide heuristics to guide the search process.

The ability to examine an image and extract relevant information requires the ability to distinguish between what are and are not connected bodies. This is not a trivial problem when one considers that real objects exist in a 3-dimensional space and may be partially obscured behind other objects, cut off in the frame of the image or severely blurred. When such situations occur, it becomes difficult to determine which elements comprise the obscured object (if such a situation could even be detected) and how best to reconstruct the object in its original shape. When images are blurred substantially, it may even become difficult to determine where one object ends and another begins with any certainty.

Imaging itself is an imprecise task. In any given image there may be a certain amount of noise present among the signal that represents the true scene. These defects can occur from defects in the lenses or other elements of the imaging system, light diffusion or refraction from various elements within the scene or even random particles that might enter the field of view during capture. In any image there exists a ratio of true signal to noise that can quantify just how "noisy" an image is.

Images of neurons suffer from all of these problems. In order to capture images of the neurons through the microscope, researchers first apply a dye to the cells in order to make them opaque, a technique called staining. Different areas of the cell take up this dye at different concentrations, and therefore variations in the darkness of the neuron occur even on the same image plane. When we consider that a discontinuity in a neuron cell on any given image plane must also be considered a false gap caused by the staining process, the difficulty of accurately finding the neuron in the stack becomes apparent.

As the neurons are 3-dimensional structures resembling trees, on any given image within the stack there can be discontinuities between areas of the cell as they pass between the different layers of the stack. This is analogous to a thread being sewn through a piece of fabric: at each point where the thread crosses the fabric we can see only that portion that exists in the hole made as it pushes through. Neighboring threads may or may not be from the same branch or even the same structure in the sample.

Due to the microscopic size of the subject and limited ability to manipulate the scene of image capture, image stacks may contain structures other than the neuron that were present in the sample when the images were taken. This may include other cellular debris, forms of contaminant introduced in the sampling or dying stages or even other neurons that could not be removed from the desired cell.

As focal adjustments are made to capture different levels of the stack, the edges of the neuron are progressively blurred. Although the use of confocal microscopy helps to eliminate the blurring of edges from parts of the neuron not in the immediate focal plane, there is still a loss of resolution around the edges of the neuron structure described by the point spread function. This blurring occurs at the boundaries of the neuron cell where the image moves from the foreground cell structure to the background pixels. This loss of resolution makes the determination of the true cell diameter difficult, and therefore complicates the calculation of the cellular centerline through 3 dimensional space.

The partial volume effect also contributes to the loss of resolution within these stack images. When the structure being imaged is smaller than twice the full width at half maximum resolution, the intensity recorded for the structure will be under-sampled [2]. This means that there will be areas within a neuron cell that simply cannot be imaged at the same intensity as other areas simply because they fall below this threshold. Smaller structures such as spurs – nodules that protrude from the cell wall and may form connective synapses with other cells – may not be captured at all, despite their importance to the overall communicative structure of the neuron.

In order to properly reconstruct the neuron morphology from source images containing such imperfections, a system must be able to: correctly eliminate noise and identify pixels that are part of the neuron; properly group connected pixels into segments; correctly identify the centerlines and diameters of these segments; and connect adjacent segments that appear across one or more levels of the image stack.

This represents the first part of the problem. Once the raw image processing is done, the system must be able to take the resulting segmented representation of the neuron and correctly define a series of points and parent-children relationships such that the entire neuron is reduced to a simple graph structure. This requires searching through the collection of generated points, connecting the correct points into parent-child relationships and correctly identifying points where the neuron bifurcates into different branches. This is not a trivial task.

The reconstruction of a neuron is basically the construction of a sub tree of an edge-weighted graph through 3 dimensional space utilizing a collection of individual points derived from the image stacks. The construction of such a sub tree may seem possible utilizing various heuristics to guide the process, such as spatial proximity and relative size of the points in question. However, due to the nature of the neuron structure and the fact that the true path may exist as a joining of two or more trees of such points, the problem might also be considered to be a variant of the k-cardinality tree problem, which is known to be NP-hard. As each sub tree of the reconstruction would be a 2 cardinality tree (since a neuron cell can only branch into two sub-branches at any given point, each node can have at most two children) there are approaches that may be applicable (see Blum [3]) but the problem remains challenging.

The techniques currently in use for automated reconstruction of neurons can be divided into two distinct groups – global and local. Global techniques require the program to examine each pixel (or voxel – a 3 dimensional volume object representing a pixel) of each image in the

stack and perform several complex, slow calculations on a local image area to determine image intensity, gradient direction and object orientation. Once the entire image stack has been processed, the result is a matrix of coded pixels or voxels with particular value assignments indicating the likelihood of being a member of the foreground neuron structure (or the background) and a directional measurement indicating the most likely path of similar voxels. This approach is costly in terms of computational power, and utilizes every pixel of every image in the image stack.

Local tracing utilizes a set of seed points provided by the researcher, and then examines adjacent image information to determine in which direction to continue the process. Once a similar region has been found, the process continues to iterate until a stopping criteria is met. As neurons are tree-like in structure, these methods may employ either a depth-first or breadth first approach to branching points, and reversing the trace once an end point is reached can account for branches occurring on both sides of the cell soma. Local techniques only examine and calculate values based on a small subset of the available image information.

Global techniques give the best results in tracing, as they are able to account for global image information, and may ensure complete neuron connectivity by examining the coded pixels/voxels to find areas of discontinuity and adjust for them. However they are computationally expensive and scale poorly with larger dimension image stacks. Local tracing is much quicker and scales better to larger image stacks, but does not guarantee connectivity and requires significant user interaction.

A method of automating seed point selection while also considering global image information would help provide a quick, inexpensive method of combining the advantages of both approaches. A method that also is able to help guarantee connectivity would provide a best of both worlds solution. In this paper, I present a system for generating a set of high-quality seed points from image stacks that can then be used for a variety of other local tracing methods, with the additional benefit of providing neuron width estimates, morphology models and global image-stack thresholding information that can be used by algorithms to refine their results.

2 Background Research

The problem of neuron reconstruction has been studied for many years, and has been greatly influenced by similar attempts at automating the reconstruction of other biological systems that have similar morphologies, such as the study of vascular systems and MRI images. Many of the techniques used in neuron reconstruction have been adapted from the methods used in these other areas, but the fine structure of the cells and the available level of detail in their imaging presents additional complications.

At its most basic, the problem of neuron reconstruction becomes one of separating elements of the foreground from a background across 3 dimensions. The additional hurdles of high signal to noise ratios and the partial volume effect require additional techniques to produce accurate data. As such, and work on this topic would be based heavily on the concepts of edge/blob detection and various image analysis theories.

However the combination of these techniques has generated an interesting array of approaches to the problem. A good historical account of the process beginning in the 17th through modern day computer-assisted methods may be found in Senft [4] and a review of the modern state-of-the-art computer algorithm approaches in Meijering [5].

It should be noted before discussing individual efforts in this domain to note that a common element to almost all of the approaches discussed is the use of a smoothing convolution to remove image noise. In the images within the stack, image noise may be present on the form of random pixels that are of brighter intensity than the black of the background pixels. When a convolution kernel is applied to the area it produces a blurring effect, and the noisy pixels can be eliminated by averaging their intensity value over the local pixel neighborhood.

This technique is used prior to any other type of signal processing from the images, and has the dual effect of reducing image noise and further blurring the intensity signal from the neuron. Smaller structures that may have been successfully imaged may be lost during the process, and the gradient between the neuron body and the background areas are extended over a greater pixel length.

Most approaches utilize the convolution product with a Gaussian kernel of a set size. Some kernels may be adaptive or run at several different sizes in order to evaluate the noise of the image at different scales. However all of the approaches mean that the algorithms discussed below are essentially evaluating a synthetic, modified version of the original image information present in the slices of the stack.

There are two main categories of reconstruction algorithm: those that exploit local image information and those that take a global approach and analyze every pixel within the image stack. Within these two categories there have been a number of different approaches offered. I outline each approach and summarize the related prior work in the two sections that follow.

2.1 Global Search Methods

Those methods classified here as global search methods all exhibit the same characteristic: they examine all available image data within a stack and attempt to segment the foreground neuron structure from the background by evaluating each pixel or voxel in terms of its place in a gradient of the image intensity. This evaluation is then converted into a coding scheme that is used to label each pixel/voxel and derive the neuron structure by following peaks in the coding values through the data space.

Once the segmentation is complete, this class or algortihms offers a variety of methods to perform the additional steps of centerline extraction, width estimation and the building of the tree reconstruction. The main advantage of the approach is its ability to provide a fulled-connected view of the neuron structure and, by extension, eliminate extraneous structures that may have been in the sample during imaging.

One of the most straightforward techniques was proposed by He et al [6]. He attempted to produce skeletonizations of neuron cell 3 dimensional data sets using a thinning algorithm that produced a centerline tracing of the cell. These source data sets were binary representations of the segment neuron, where the intensity value for neuron voxels was set to 1 and the background voxels to 0. He established the three main criterion for thinning neurons in 3 dimensions: connectivity preservation, thinness and geometry preservation.

The first criterion means that the input and output images should share connected components, and that no holes that are apparent in the input data should be closed nor new holes introduced in the output image. This criterion ensures that whatever data was present in the input image, it is reflected in the out. However, as discussed in the problem statement, often the input data can be faulty where regions of the neuron are less intense due to dye uptake or the partial volume effect.

The second criterion requires that output images of the neurons represent the neuron at any point in 3 dimensional space by as small a unit as possible. In his approach, this would mean that the skeleton produced would have to be 1 voxel thich, except where additional voxels were required to retain the other two properties.

The third property is inherienty subjective. The basic definition is that the skeleton produced should "appear similar to" the input data set, and appear at the center of the object. While somewhat trival in nature, the criteria captures the essence of the reconstruction goal and means that valid skeletonizations should not pass through the detected boundaries of the neuron object.

To accomplish this, He proposed an iterative method of coding the pixels according to a count of how many of their neighboring voxels are in the neuron set and deleting the ones that fall into the outer most layer border – that is,share a common face on their voxel cube with a voxel of the background. This process is repeated until only a single pixel remains, defining the skeleton.

Sato et al [7] proposed a mthod of enhancing curvilinear structures in medical images using a 3 dimensional line enhancement filter based on a combination of the eigenvalues of the 3-D Hessian matrix. The approach was designed to specifically test for line structures in images while removing the effects of other shapes and reducing the effects of non-uniform and noisy artifacts.

Noting that the combination of the Gaussian convoultion with the second derivative of intensities has been used in line enhancement filtering, Sato extends this approach to 3 dimensions using the Hessain matrix. This matrix describes the second-order values of local intensity variations around a point in a 3 dimensional image. The positive eigenvector derived from this matrix determines the direction of the line detected. Because the system tests for linear structures, noisy blob like elements are separated. Once separated from the linear set, non-linear shapes can be eliminated through connectivity assesment.

An important contribution of Sato's work was the concept of multiscale analysis. Working from the detection of lines in 2 dimesnions, the importance of varying the size of the Gaussian convolution kernel in order to respond to lines of various sizes promted him to apply the multiscale approach to the detection of 3 dimensional tubular objects. However one difficulty with this approach is that the response function results of intensity and its derivative are decreasing functions of scale and need to be normalized for comparison.

This focus on multiscale analysis was built upon by Krissian et al [8]. Continuing on the concept of tubular detection and using the Hessian matrix for their discirimination, Krissian goes on to propose a new centerline detection scheme built on a gradient-based response function. A simple model of a cylndrical vessel is used with a circular Gaussian cross section for the detection of the centerlines, which result in a estimation of the centerline when multiple scales of the Gaussian kernel are applied.

The approch uses three steps: the computation of the multi-scale response at a descrete set of sclaes; extracting local intensity maxima in the response to find centerlines; and reconstrcuting the vessels using the centerline and size information. The response function is locally maximal at the center of the tubuklar object, and the scale of the response is a function of the size (radius) of the tube.

The use of multi scale heuristics was also employed by Alyward and Bullitt [9] in their application, which utilized dynamic scale intensity ridge transversal to trace tubular structures. This approach is insensitive to initial parameter settings, tracks at sub-voxel accuracy, passes branching points and is insensitive to substantial image noise. Based on the "cores" approach of Pizer [10], the system simultaneously solves for centerlines and width during transversals, however this approach differs in that uncouples the centerline and width estimations from the measurement windows to maintain noise insensitivity.

The Hessian matrix is also used in this approach, with the critical parameter being the scale at which the intensity, gradient and and matrix is calculated at any given point in the data space.Three other parameters – tolerance for equal-to-zero values in N-1 the eigenvectors of the

matrix, the value of the ratio of eigenvalues in the direction normal to the ridge, and the seed point to start the search – must be input before the system will run.

In order to compensate for the anisotropic nature of the image stack, Alyward and Bullitt use cubic spline interpolation to super-sample the Z axis and produce isotropic voxel data in their results. Without this step, they note that there is a five-fold increase in processing time.

To recover from local discontinuities arising from branching, low image intensity and image noise, they used four heurictics: the smoothing of tangent directions, reducing step size during high-curvature areas, detecting vector swapping at singularities and purturbing the image data to remove such irregularities.

In order to select seed points for their algortihm, Aylward and Bullitt used histogram analysis. Selecting a lower threshold of the 5th percentile and an upper threshold of the 95th percentile, they used the intermediary intensity values to serve as their source voxels for connected components. The 99.5^{th} percentile voxels were selected to serve as seed points for their tracings.

Four of the finalist algorithms in the DAIDEM competition utilized global tracing techniques. Chothani and Stepanyants [11] utilized pre-processed image stacks. First they determine all available structures within a stack, then select the structure of interest. Structures are segmented according to their number of end points along a pre-set path legth, then merged together from lowest to highest end-point count according to a cost function evaluating the distance between end points, cosine of the angles formed in the merge, the avergae normalized intensities and the curvature of the structure formed. This cost function requires the setting of thirteen paramters, only some of which can be determined using a learning algorithm.

Of note in their approach was the use of a specialized binary version of the image to drive their voxel coding technique. This means that the establishment of a correct thresholding level is of vital importance. The resulting synthetic interpretation of the image stack would be useless if the thresholding levels were too high or low. They also used multiscale Laplacian of Gaussian center surround filters in place of the slower Hessian matrix calculation or steerable filters.

Turetken et al [12] used a probablistic approach. In their alogorithm, they first calculate and encode the tubularity of each voxel using the Hessian matrix and steerable filters. Then selecting only the voxels with the highest probability values, they eliminate the surrounding voxels until the remaining anchor points are equally spaced as possible.

The next step is to create tree graphs from the anchor points that represents the reconstruction. They connect these anchor points using the maximum probability paths via the ant optimization scheme, determining the optimal k value of the k-minimum spanning tree problem by assigning probabilistic costs to the trees constructed and selecting the least costly. They use Dijkstra’s algorithm to find the points between two anchor points, using the voxel probabilities within the image graph.

Of the global techniques, my approach is most closely related to that of Narayanaswamy et al [13] and Peng et al [14]. In [13], they algorithm produces a set of syntehtic images from the image stack that it uses to perform complex calculations. These synthetic images are based on: the curvlet transform; perceptual grouping by scalar voting; adaptive focus detection; and depth estimation. Curvelets are determined to be the best choice of wavelet transforms due to the fact that neurons are tubular in structure, they eliminate image artifacts and provide structure orientation at a per-voxel scale.

Perceptual grouping is accomplished through the use of tensor voting and supplimented with scalar voting at branch points, eliminating fringe artifacts. The adaptive focus image is comprised of a ray projection through the image stack, with each image having been focus-adjusted to produce a best-focus version.

The depth map is tracks the slice distribution of the pixels across the Z index of the stack, and is then used in the SWC file generation.

In [14], an initial over reconstruction of the neuron is produced and gradually scaled back by itertive pruning to achieve a reconstruction while assuring connectivity. First the average intensity value of the image is used as a global foreground threshold level, then a graph is created connecting all voxels to their immediate spatial neighbors. The resulting graph is undirected and represents an over reconstruction of the neuron morphology. From this over reconstruction, the connections are pruned until a true tracing is left.

Dijkstra's algorithm is employed to prune nodes from the graph, using the calculation of the geodesic metric calculation as edge weights between the voxels. Once the graph has been pruned using this approach, three additional pruning approaches are used. These approaches are applied to both leaf nodes and interconnected nodes.

The first approach is called dark leaf pruning. This type of pruning sets an additional threshold for the voxels and removes those that fall below it. In the paper Peng establishes 30 as his dark leaf threshold and removes the nodes equal or less than this value (hence "dark") from his graph. The second approach is called covered leaf pruning. This type of pruning defines a sphere of adjustable radius at a leaf node and its connector and enlarges it until 0.1% of internal voxels are below the average intensity level. Then the number of bright voxels in the intersection of the two spheres is examined to determine if there is significant coverage between the two nodes. If so, the leaf is pruned.

Inter-node pruning is responsible for removing redundant nodes connecting leaf nodes to branching nodes, or branching nodes to a root node. For each remaining leaf, check if the area defined by the node covers the area defined by the parent. If a overlap is detected, the parent is deleted and the next connected node established as the new leaf parent, repeating until a branching point or the root is reached. The process is then repeated starting with each branching point, recursively executing across all detected branching points until the ultimate root of the tree created is reached.

2.2 Local Search Methods

Those methods classified here as local search methods all exhibit the same local exploitation of image information without requiring the processing of each pixel or voxel in the stack. They tend to be quicker in execution and lighter in memory requirements, but cannot assure connectivity of the reconstruction due to discontinuities in the image and lack of global image knowledge.

The early approaches in local neuron tracing was based largely on the works of Lorenson and Cline [15], Kass et al [16], Kimmel and Kiryati [17], Pizer et al [10] and Zhou and Toga [18]. The marching cubes algorithm of [15] combined the concept of flood fill with surface contours. A divide and conquer approach, the algorithm creates a polygonal representation of surface contours by examining the intersection of a plane with a logical cube comprised of eight pixels across two adjacent slices. Once the intersection has been determined, the algorithm "marches" to the next cube. In total they model 14 different possible ways a plane can pass through this cube. From an index of already found surface triangles, new surfaces are discovered. The resulting combinations of these models produce a surface reconstruction of the 3 dimensional structure.

The active contour models of the snakes in [16] were capable of interpreting 2-dimensional features by minimizing an energy function. A snake in this context is an energy-minimizing controlled continuity spline influenced by local image data and constrained by user-set parameters. The resulting output is capable of following image contours with high accuracy after being calibrated to a specific boundary.

The skeletonization methods of [17] established maximal disks as a method of center point extraction. Within an imaged object a series of disks may be defined that do not subsume each other and extend over an area of the structure such that an increase in their radius does not add any additional pixels from the structure and extends beyond the borders of the structure into the background. Such maximal disks describe the volume of the shape imaged and their connected center points the skeletonization of the object. The set of maximal disks represents a complete and minimal representation of the shape in question, as both the skeleton and approximation of the surface area are shown by way of the area of the disks.

The generalized maximum ridge of medialness defined as "cores" in [10] introduced the concepts of adaptive kernel shapes and size to local search methods. Pizer noted that linear medialness is defined as merely the intensity gradient using a kernel of a set size – and that an adaptive kernel alters its size and shape in response to the local image information for best results. In the discussion of the approach several kernels are discussed individually, including the best choices for use on tubular structures such as vessels and similar structures.

The voxel coding and clusters approach of [18] added a more global perspective. Using distance and voxel coding techniques, the authors devise a system that assures smoothness and connectivity without being sensitive to object boundary complexity as templates are. Two main methods are used to achieve a skeletonization of the voxel cluster and establishment of the reconstruction points – boundary peeling and distance coding.

The boundary peeling scheme is based on the coding of voxels by their neighbor count. The authors establish three types of neighboring voxel: vertex, sharing only one vertex of the voxel logical cube with a neighbor; edge, sharing two vertices (i.e., a single edge) with a neighbor; and face, sharing four vertices or an entire face of the logical cube. Voxels with connections at all four vertices are considered inside voxels – otherwise they are placed in the boundary set for peeling.

Distance coding uses two different approaches. The first is called boundary-seeded and generates a minimum distance field according to object boundaries. This property assures connectivity in the resulting reconstruction. The second is called single-point-seeded and generates a distance field based on a single seed point. This property, also called the cluster code, provides the initial seed points for voxel coding and defines the clusters in the data.

The voxel coding approach begins from the seeds identified in the distance coding operation and consists of two parts. First is propagation, where the system assigns codes to the voxels indicating distance from the seed points. Then extraction is performed, a reverse search based on the voxel code field.

Al-Kofahi et al [19] noted the poor scaling of both skeletonization and branch point analysis and edge detection and chaining techniques, due to the multiple processing required for each pixel. Local tracing approaches – also known as vectorization – were the preferred method in terms of performance. They suggested the use of generalized 3D models and directionally-sensitive kernels.

Noting that dendrites and axons are well represented by generalized cylinders over a short distance, the authors use a set of 4,096 5xK templates perpendicular to the structure's direction, varying the size of the templates from 8 to 50, with each template representing a possible orientation of a cylinder in 3 dimensional space over 1,024 different directions. At each point found at a set step value from an initial seed point, different templates were used to determine the direction the structure had turned to. Not all templates were used at each step, as limitations on the neuron's tortuosity prevent rapid changes in direction.

This approach also gathered statistics on the image stack. The system catalogued the median of the 3 dimensional image, median of the foreground (as the median of the disks centered at seed points), median of the background and each standard deviation. They also recorded the average, maximum and minimum dimensions of the neuron near seed points.

Rodreguez et al [20] built on the marching cubes algorithm. They begin with addressing a problem of alignment of workspace – adjusting the origin of the images within the stack so that features from different slices align in the proper way. The method of tracing is based on flood-fill approaches, utilizing seed points to start a two-pass system and using a threshold level to mask further refinement.

Al-Kofahi et al [21] later adapted Canny edge detection for neurite tracing. Using an adaptive 2 dimensional kernel, they manage to prune the search space to limit memory

requirements and test only for the direction of edges found within the stack. Correlational kernels traditionally used to perform traces are partitioned into multiple sub-kernels, and the median of their response functions used as the guiding criterion improved the tracing accuracy of the system substantially.

Schmitt et al [22] achieved segmentation and skeletonization by applying cylindrical models and cross sections to user defined branch points. This approach differs slightly from the other local vectorization techniques in that it is designed around a different classification of seed point. Most algorithms attempt to begin their tracing routines from a simple seed point – that is, a region that defines an area within a generalized cylinder and provides vectors from which the next step in detection may be taken. This approach uses the areas joining these simple cylinders, typically an area of high image ambiguity. However, because the areas of ambiguity are identified before tracing begins, the program is able to utilize the positions of these points and join them with generalized cylinders to overcome the local uncertainties.

Srinivasan et al [23] utilized probabilistic region merging to achieve segmentation, building regions from user supplied seed points. Utilizing a collapsed version of the image stack called a Maximum Intensity Projection (or MIP) they use a low-complexity 2 dimensional tracking approach except when cross over events are detected. The algorithm then switches to a cross-sectional tracking approach until the cross over event is passed.

The resulting MIP image is then segmented using a combination of the seeded-watershed algorithm and the mean-shift algorithm. A guided region growing approach is also used to complete the segmentation when the watershed algorithm fails.

Xie et al [24] used a triangle thesholding method to segment their images. The local scale of the neurite structure is estimated using the distance transform, and the generated seed points are pruned using a neighbor-voting scheme to eliminate the seeds of low intensity. This ensures that the resulting seed set only encompasses the seeds of the highest intensity value.

The shortest path between the seeds is determines using Johnson's alogrithm, as the seed set has already been pruned and is therefore sparse. The path between the seeds is further refined using a cost function that encorporates a smoothness factor, making the resulting trace have as few rapid changes in tortuosity as possible.

A gap indicator was used to ensure that seeds that were close to each other yet on different branches were not connected. The seeds located near to a bifurcation point were merged, as a method of dealing with the local image ambiguities typical of these areas of the neuron.

My approach is also related to the work done by Flasque et al [25] in that they too used all images in the stack as part of their thresholding. However they also use the settings determined from a reference slide to recompute the intensity values of all remaining slides. The initial background threshold is determined as the first local minima in the reference slice histogram. Two threholds are then found for the remaining voxels – 15-20% was used for the lower threshold, and 80-85% for the upper threshold. The resulting thresholds were then utilized

in a recomputation of the other slice images using linear interpolation so that the levels of all the slice images aligned according to the threshold levels.

The tracing algorithm utilizes a search box centered around a seed point with a directional vector. The box is progressively reduced in volume till a structural marker is hit. Voxel clustering is used to group voxels into regions of interest using the intensity of voxels and the neighborhood configuration. A neighbor voting scheme using both voxels and regions and a threshold level continues the grouping, and regions of less than a certain size are deleted.

Three of the final algorthms in the DAIDEM challenge may be considered local tracing. Zhao et al [24] used cylinderical model templates and refined their initial local trace with shortest paths. After describing a similar distinction between the advantages and costs of global and local search methods, the authors propose a system utilizing the local tracing system and refined with a shortest path connection scheme based in global image techniques.

The approach is based around the fitting of adaptive cylindrical models to the data utilizing seed points, radius estimates, directional vectors and scale values. They used a local Hessian matrix of minimal size – 3 x 3 x 3 – to find seed points, and used the triangle method over histogram to find thresholds. The discovered seeds were then connected based on their spatial distance, being limited to 20 pixels away – twice the height of the cylinder templates used.

The authors also discussed the issue of connection types between cylinder model methods, where two cylinders may connect either in an end-to-end method where end points of each cylinder are judged to be connected, or in an end-to-body method where the end point of one cylinder is attached to a point between the two end points defining the other cylinder. The first case defines a path along a neuron branch with no bifurcations. The second decribes a bifurcation event not as the division of a path but as the intersection of two already defined paths.

In the detection and resolution of cross over events, the authors determine that the best method of descriminating a true cross over from a proximal approach of two branches is by the angle formed by the two neurite end vectors. That is to say, the limits on neuron tortuosity may be relied on to provide resolution of a cross over where the angles formed by the intersecting branches must be within a set tolerance to be labeled as correctly indicating the cross over event. They accomplish cross over detection by finding the best set of pairings based on minimizing the total of join angles using the Hungarian algorithm, fusing the resulting pairs. This is performed before the use of the minimum spanning tree algorithm.

Wang et al [26] proposed using a preprossing step to produce a binary fore- and back-ground image which was used in seed selection, followed by application of open-curve snakes adapted from Kass [12]. The preprocessing step consists of four parts. First the curvlet and scalar voting algorithm of Kass is used to de-noise the image and provide continutiy enhancement, and then the gradient vector flow (GVF), a regularized form of the image gradient, is computed. The GVF is used as the primary deforming force in the snakes tracing scheme, and points directly to the centerlines of the neuron, preserves edges and avoids axon diffusion when the fibers are close. The GVF also supersedes the need of selecting a scale for the Gaussian smoothing step.

The second step is the application of Frangi's vesselness measure to the Jacobian matrix computed with the GFV to make a vesselness image. Third, this image is segmented using a graph-cut segmentation algorithm to produce a binary image roughly separating the foreground from the background. Finally, the combination of the vesselness and binary segmentation images is used to performe seeding.

The binary image of the segmented stack is used to avoid snake leakage during the tracing algortihm and bind the snakes within the foreground neuron voxels. The tracing process consumes the list of seeds found from the seeding step, and terminates when all seeds have been removed from the list. As before, branching within the tracing is determined where two snakes collide.

Finally Bas et al [27] used curves designed to explicitly mimic those of the underlying biological system. The authors argue that in order to construct a system capable of reconstructing the neuron arbor in a reliable manner, an algorithm must make use of mathematical models that inherit the underlying geometry of biological structures. They propose using principle curves as a model for the skeletonization of neurite branching structures, using local statistics to define centerlines and voxel connectivity.

From a seed point, their approach iteratively traces the principle curve of a structure in space using a recursive tracing method based on identified potential branches. For each branch so traced, a check is made to see if there are any nearby potential branches to which this new one might be joined. For each found, the tracing process through space is called and the process continued until the potential branches set if exhausted.

Author	Local	Global
Al-Kofahi et al	x	
Al-Kofahi et al	x	
Alyward & Bullitt		x
Bas et al	x	
Chothani & Stepanyants		x
Flasque et al	x	
He et al		x
Kass et al	x	
Kimmell & Kiryati	x	
Lorensen & Cline	x	
Narayanaswamy et al		x
Peng et al		x
Pizer et al	x	
Rodreguez et al	x	
Sato et al		x
Schmitt et al	x	
Srinivasan et al	x	
Turetken et al		x
Wang et al	x	
Xie et al	x	
Zhao et al	x	

Table 1 - break down of prior algorithms by local or global methods

3 Materials & Implementation

3.1 Materials

In building my system and validating its results, I am relying on the image stacks and manual 'gold-standard' reconstructions distributed as part of the DIADEM challenge. Specifically I am using the nine stacks of olfactory projection fibers used in the challenge – the first three are the stacks used by the competitors to build and test their system, the second three are the qualifier stacks the teams used to qualify to compete in the final round of the competition where the last three stacks were used to determine the winner.

These stacks are all 512 x 512 pixels in dimension with between 61 and 102 slices in each stack. The space between each slice in these stacks is equivalent to 3.03 pixels compared to the width and height of each image. The image slices are saved as grayscale TIF images and all images are 8 bits per pixel. The stack names are prefixed with the letters OP – for olfactory projection. I refer to the stacks in the remainder of this paper by their abbreviated stack names.

Stack Name	Slice Count
OP_1	60
OP_2	88
OP_3	62
OP_4	67
OP_5	76
OP_6	101
OP_7	71
OP_8	85
OP_9	92

Table 2 - list of the nine drosophila olfactory axonal projection image stacks and their slice counts

As part of my system I use an additional file within the image stack folder called ino.txt. The purpose of this file is to provide my system with information about the stack required to properly calculate distances between the slices, specifically the 3.03 separation value mentioned above. This information is generated at the same time the images are acquired, so no additional time or effort on the part of the researcher is required. However should this file not be found, the system prompts the user for this value.

The info.txt file in the stack folders also contains the 3-dimensional coordinates of the cell soma as determined by the researchers. Although I do not utilize or require this information in my system, the data is included to represent a complete set of the available information I had to work from. These values are determined from the 'gold-standard' reconstructions generated by hand tracing the image stack, so would not be available to a system such a mine before an automated reconstruction was attempted. Therefore I have not used these values to guide my system.

Each stack also has an accompanying ‘gold standard’ reconstruction, produced by neural researchers using the manual technique described earlier. These reconstructions will be the basis of comparison for the output of my system, although as I am only producing a limited number of seed points, it should not be a complete reflection of the manual reconstruction. However, the goal is that my seeds do in fact coincide with points of the manual reconstruction within a set tolerance, and that I do not generate an excessive number of seeds that exceed this tolerance.

3.2 Implementation

The system described here was coded in Java using the Eclipse IDE. I utilize the JAI library to accommodate the use of TIF files in the image stacks. The experiments were carried out on a Windows 7 computer with 3.2 GB RAM.

3.2.1 Algorithm Objectives

Given the nature of the image stacks used in these algorithms, which can range widely in width, height and depth and therefore memory requirements, and the nature of the subject being evaluated, it is advantageous to eliminate as much of the search space as possible when conducting a reconstruction. Many algorithms attempt to do this through the use of seed points – specially designated locations of high probability of belonging to the neuron structure from which a search can be conducted with high confidence of finding adjacent structures. This paper attempts to outline a method of obtaining a set of high-quality seed points from an image stack, using a basic set of assumptions about the stack and pictured neuron(s) to make the process efficient and accurate. I utilize the basic principles of histogram analysis, collision detection along with a sampling approach to determine variable values used in the algorithm.

The approach offered here is designed to accomplish two primary tasks: the first is to produce a set of seed points and accompanying data that can be utilized in other tracing applications. As can be seen from the description of previous efforts in neuron reconstruction, many programs offer robust, highly sensitive and descriptive methods of performing neurite reconstruction but fall short of being fully automated in that they still require the input of a series of seed points from the user before they can begin. Still others require particular parameter settings to achieve the best result, and require a user to adjust them to the particular attributes of each image stack. I attempt to provide a pre-processing step to provide those needed data in an automated fashion.

The second is to propose a method of bridging the benefits of both the global and local search methods. Global searches offer assured connectivity and a better ability to adapt to local discontinuities and irrelevant structures. Local methods offer speed and smaller memory requirements. I attempt to acquire the information needed to assure connectivity and discrimination of false discontinuities and irrelevant structures without requiring vast amounts of time and memory space.

To begin the description of my approach, I should first detail what might be assumed about the nature of the neurons themselves, the images within the stack of these neurons and the seed points I am trying to produce. The first two categories can be validated by inspection of

neuron cells and the image stacks used in this paper. I shall outline the reasoning for the third following its listing.

3.2.2 Seed Metric Definitions

Neurons are completely connected in structure. They have a single root – a point from which their branching factor in either direction is monotonically increasing. Their branches have a limited tortuosity in three dimensions and they cannot pass through themselves. They have a varying diameter along their length, where branches of the neuron taper to an end and do not exhibit end points with large diameters – with the possible exception of the main cell body, or soma. It should be noted that spurs, round projections from the main neurite body, do exist. However the detection of spurs is beyond the scope of this paper.

Image stacks encompass the length of the entire neuron. Each slice contains at least some pixels that comprise the neuron, with the possible exception of the top and bottom-most slices. There exists a separation between the foreground and background that can be determined by examining the pixel intensity. The bulk of the neuron structure tends to be located in the center of the image, in terms of both two and three dimensions. The slices within a stack all have fairly uniform distributions of image noise and non-neurite structures.

Seed points should be located near the centerline of the neuron. They must give an approximation of the neuron diameter around the center point. They should be as evenly spaced as possible, and can be scored from most desirable to least by: proximity to the cell soma, or body (which is the root); designating the true end of a branch; designating a branching point; denoting a path from a leaf or branching point; and lastly, denoting a path from an interior seed to another interior seed.

The first two requirements of seed points follow from their definition. The requirement that they be as even spaced as possible allows us to make generalizations about the overall structure of the completed neuron tree from the seed list, and would provide the end-user tracing system with a method of rapidly pruning the search space and determining the best required models, kernels or directional filters to use in the tracing.

Seeds proximal to the cell soma are desirable because they represent a highly-probable direction for the trace to begin from an easily verifiable landmark. Somas tend to be the largest, brightest elements within the image stack and as the root of the cell present the best starting point for a trace of the tree structure. Seed points that mark a true end of a branch (seeds that are located on the actual terminus of a branch and not on a section of false discontinuity) are desirable because they present a true stopping criterion for the tracing algorithm. Supplied with nothing but the location of the soma and all end points, a full trace of a neuron is all but guaranteed.

Seeds located at branching points are valuable in that they can alert to areas of ambiguity in the local image information for the tracing system. Many algorithms have difficulty at branching points because their method of calculating the local image gradient to determine directional vectors suddenly becomes inconclusive or may even be interpreted as a terminal point. Seeds demarking the direction away from end point of leaf seed, as well as those showing

the direction(s) away from a branching point are valuable in that they provide a heuristic method for tracing algorithms to overcome the aforementioned difficulty. Seeds merely denoting the passage between a single parent and single child are the least valuable, as they tell tracing systems little about the overall image and structure of the neuron and would most likely be included in a trace beginning from any of the other seed points.

3.2.3 Algorithm Description

The general outline of my algorithm is as follows:

1. Collapse Search Space: determine the number of slices in the image stack and select every other image for analysis
2. Threshold: determine the best thresholding levels to segment the neuron from the background
3. Examine Intensity Values: discover the locations of the remaining pixels with intensities above the threshold
4. Seed: create seed points from resulting set of pixels and establish z-index assignments
5. Connect: establish seed connectivity, prune extraneous structures and consolidate seeds
6. Group: assign resulting connected seeds into groups that show overall structure
7. Output: generate a list of the seeds with high-confidence levels of being in the neuron, image stack attributes and a comparison of submitted points with the gold-standard reconstructions.

The various steps are discussed in their corresponding sections below. I give descriptions of my motivations and justifications and discuss how and why I determined how to approach each step.

3.2.3.1 Collapse Search Space

When examining an image stack, it is quickly apparent that the information contained on any given slice is of small help in understanding the whole structure of the neuron. A single slice may have a wide range of apparent structures on it, but without the greater context established by its proceeding and trailing neighbors it is impossible to say how they are connected or even if they are part of the neuron at all.

When an image stack is viewed with an application that allows the user to scroll through the slices sequentially, the information from one slice prepares the viewer to contextualize the intensities seen in the following slice, which in turn helps explain its next neighbor, and so on through all of the images in either direction. The sequence is the important aspect of the slice relationship; neighboring slices say much about each other, but slices separated by too great of a distance or presented out of order cannot provide the proper context for each other.

This method of visual examination of the image stack as a sequential presentation of images showing objects translated, scaled and rotated in the X/Y plane is analogous to watching an animated cartoon. In a cartoon, characters appear to move from point to point due to small changes in their position between accompanying frames. The speed the character moves and the

level of detail of his movements is dependent on the total number of frames in the sequence, and how far he moves over the course of the animation.

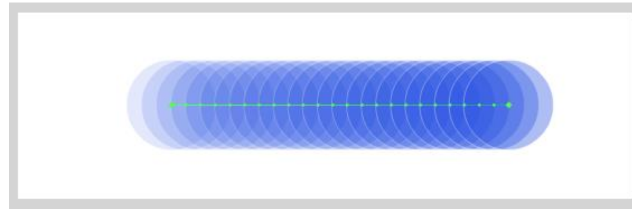


Figure 5 - animation from point A to B over 24 frames

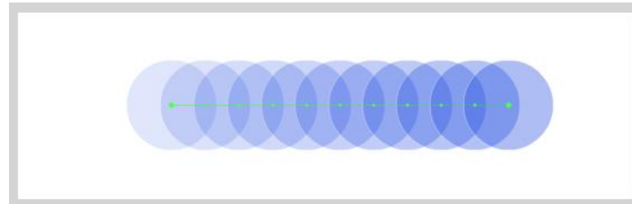


Figure 6 - animation from point A to B over 12 frames

Extending this analogy further, if we are interested in describing the path of the character through space over the course of the animation all we need to do is examine his location in the first and last frame. A line connecting these points can be said to describe the intervening steps taken between the first and last image. However if the path taken by the character was not direct (say the animation was of a character jumping over a small hole) than this description of his path is insufficient. It does not take into account the variation of the path from the beginning to the end, since it does not examine the intervening frames.

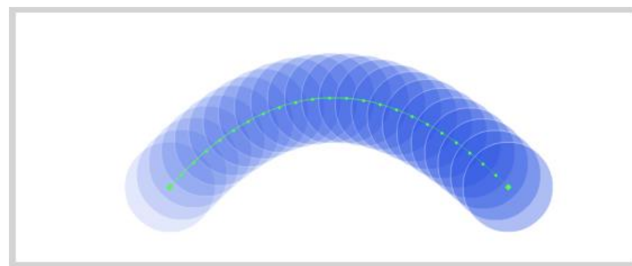


Figure 7 - animation requiring more than two points to describe over 24 frames

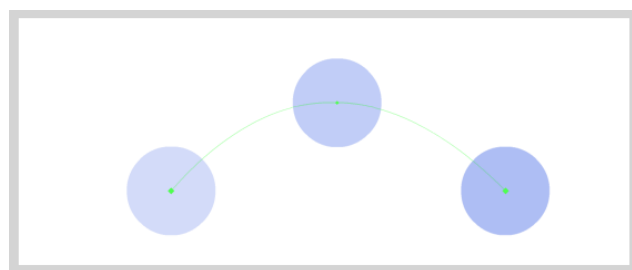


Figure 8 - three points are enough to describe the basic path of motion

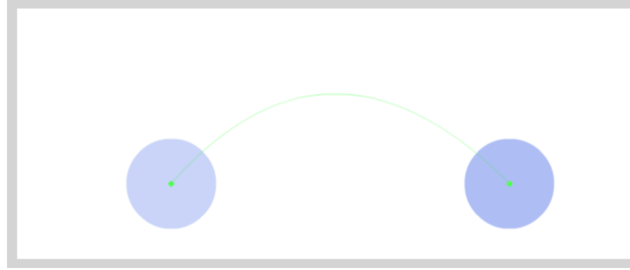


Figure 9 - two frames does not contain enough information to describe the path accurately

How few frames than are required to properly describe the path of animation? That of course depends on the speed and level of detail of the animation, described above. If the character is known to move a very small distance, than fewer frames would be required to fully describe his path with a high degree of confidence, as there are fewer possible variations of position per frame (assuming the animation is of a realistic character operating in a real environment). If there are few frames in the animation and the character travels a great distance, than more frames would be needed as each would represent a larger percentage of the total distance traveled.

Applying this method of thinking to the domain of neuron image stacks, we find our animations to be fairly uniform. The variations between neighboring images within the stack are usually very slight, generally with at least some portion pixels with a common intensity. See figure 10 for an example. In the set of image stacks used in this project, the slices are all separated by a pixel value of 3.03.

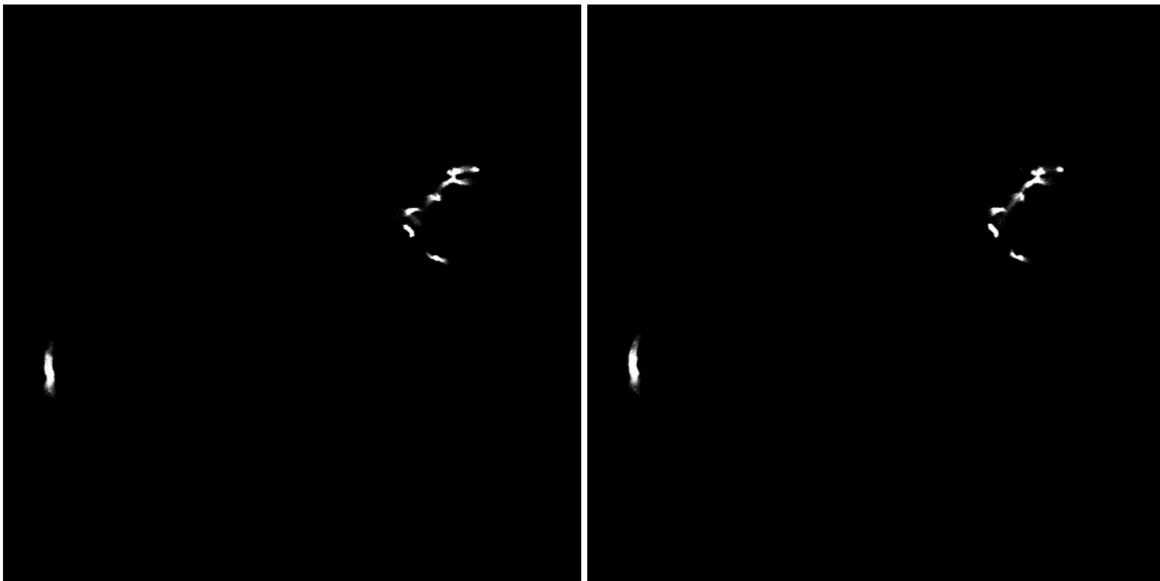


Figure 10– images 28 and 29 from the OP_1 image stack

An important quality of the images in the stack is that they are anisotropic between the three directions. We cannot convert the pixels of the stack directly into voxels, as the distance between each slice is not equal to the width or height of a pixel. Some algorithms attempt to normalize the images before beginning their tracing methods, but as [9] noted, this process can require up to five times as long to accomplish and is a computationally costly process.

I attempt to leverage this quality to quickly prune the search space while not compromising the required qualities of our rough reconstruction. The image stacks captured by confocal microscopy are of micron scale, and as a connected structure must exhibit limits on the ability to change position, direction or size in rapid fashion. This property allows us to state the following hypothesis:

In cases where the differences between neighboring slices is dramatic, it can be shown that the differences between those slices and their next neighbors in either direction are smaller. Therefore it can be said that in the worst case, and difference between a slice and its neighbor is not significantly worse than the difference between that same slice and its secondary neighbor. See figure 11 for a visual proof.

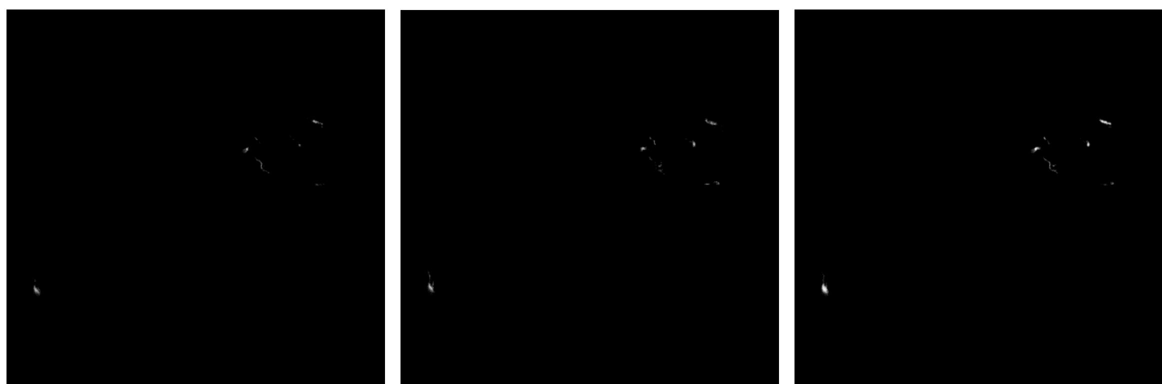


Figure 11– subtractive difference between slices 19-20, 20-21, and 19-21

Using this assumption, we can safely reduce the amount of data needed to perform a rough search of the image stack by half. This is a key step in balancing the computational costs of examining the entire image stack with the requirement that we include the entire topology of the neuron. Since I am not attempting to perform a complete trace of the neuron with all of its detail, the loss of the information from the other half of the stack does not prevent the system from completing successfully.

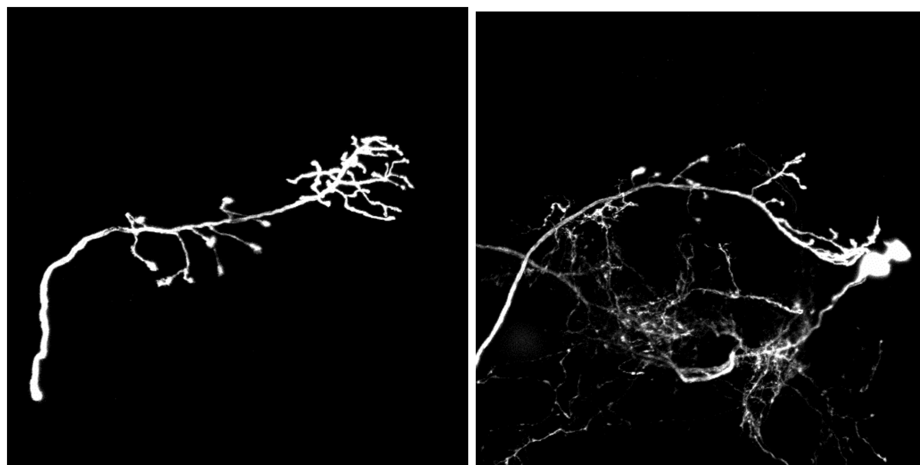


Figure 12– completely collapsed ray projections of the OP_1 and OP_2 image stacks

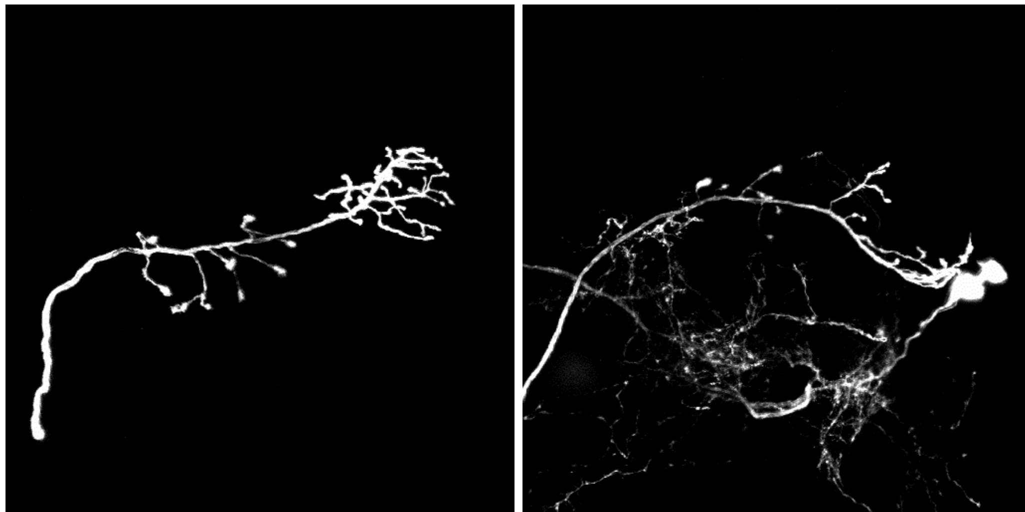


Figure 13– ray projections of the OP_1 and OP_2 image stacks using half the slices

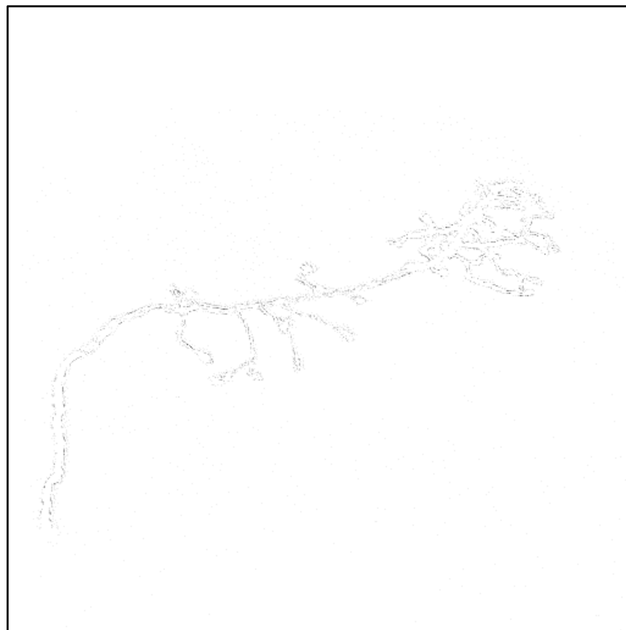


Figure 14– difference map of the OP_1 image stack between the 1 and 2 sample rates (inverted to show fine detail)

Utilizing half the stack also provides my system with a good representation of the entire stack. Due to the assumptions outlined above about the uniform distribution of image artifacts, extraneous structures and imaging errors throughout the stack images, it can be said that the collection of every other slice within the stack constitutes a reliable sample from which qualities about the entire stack may be drawn. This is useful in the following steps of my algorithm.

Within my program, the slice selection is conducted by the StackInspector class. Once provided with a file selected by the user, the StackInspector scans the current directory to determine how many images are in the stack. It also determines in an info.txt file is present in the

directory. If the file is found, the value for the z-axis offset is read into the system. If the file is not found, the user is prompted to enter the value before the program can continue.

Once the total number of images has been determined, the StackInspector creates a new Slice class instance for every other image slice. These Slice objects contain the actual pixel data from the image slice in the stack, along with their z-index position.

The StackInspector then creates the ray projection, or maximum intensity projection (MIP) of the sampled images. This image is essentially an additive combination of all of the sampled slices and is equal in high and width to any of the slice images. Each pixel of the ray projection is the maximum intensity value from all of the sampled slice images, as if each image were stacked on top of each other and the brightest intensity pixels were allowed to "shine through" darker ones above them. See figures 12 and 13 for examples.

The ray projection is a synthetic image created from the data set that is then used for further calculations. The ray projection has many useful properties that I exploit, including providing a global image overview that is useful in determining noise levels, discovery of extraneous structures in the stack and establishing the bounds of the neuron pixels. These properties represent the benefits of the global search methods and can be of value to faster local tracing algorithms that would not be able to otherwise obtain this information.

Using the ray projection, the SliceInspector also collects some important statistical information about the image stack to report later with the resulting seed set. The information collected includes:

- Slice image dimensions
- Number of images in the stack
- Image mean
- Highest/lowest intensity values
- Standard deviation
- Signal to noise ratio
- Thresholding level T

The signal to noise ratio is defined as simply the image mean divided by the standard deviation.

The thresholding level T is the intensity value below which we eliminate a pixel from consideration for being included as part of the neuron structure. Thresholding and its application to my data sets are discussed below, but I will outline the method used to determine this value in this section.

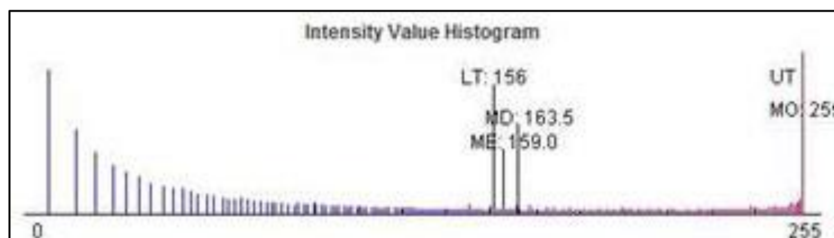


Figure 15– the thresholding algorithm in process, showing the lower threshold (LT), mean (ME), mode (MD) and upper threshold (UT) values

The method used to determine the thresholding level for the image is called iterative bisection, or means shifting. An initial threshold is set as the midpoint of the range of data represented by the image histogram. For each sub-histogram on either side of this line, the mean is then calculated. The mean of these means is then taken as the new thresholding level, and the process iterates until there is no further change in the thresholding level.

The process of calculating these thresholds was done utilizing JAI's Histogram class and the getIterativeThreshold method for the grey band of the ray projection image. In previous versions of my code I had done the calculations using my own methods and the results were identical (see figure 15 above); the JAI method was used for the sake of keeping the code more compact.

3.2.3.2 Thresholding and Segmentation

The next step of the process is to apply the new thresholding level to the ray projection of the sampled images within the stack. The original ray projection image is taken and redrawn, where all pixels with intensity values below the threshold level are simply omitted. The pixels that remain are all of intensity value T or higher.

Figure 16– the original ray projection (left) and thresholded version

The goal of thresholding the ray projection is to remove image defects and noise, and retain only portions of the image pixels that can be said with a high degree of confidence to be part of the neuron. As discussed earlier, images within the stack contain noise in the form of brighter-than-black pixels in areas where the neuron is not present due to the imaging process, and the neuron itself exhibits a gradient change in brightness from its central core through its edge as a result of the point spread function. Removing the pixels below this threshold has the combined effect of nearly eliminating image noise from the ray projection but also contracting the neuron itself slightly.

The use of thresholding in image segmentation is common practice, and has both advantages and disadvantages. The main disadvantage to thresholding as a segmentation technique is that aggressive threshold settings can introduce false discontinuities – create gaps in a connected body due to variations in gradient that happen to fall below the set level – while a permissive level does not eliminate enough of the image noise or bleed over from proximal branches to allow a clear understanding of the neuron shape.

Figure 17– detailed view of original ray projection

Figure 18– detailed view of thresholded ray projection

Figures 17 and 18 show a detailed view of my two ray projection images. The areas encircled in red show segments that have discontinuities introduced to branches by the thresholding process. While a lower, more permissive thresholding value may have prevented this discontinuity, I have found the iterative bisection method is still the superior choice for estimating the thresholding level without a priori knowledge of an image stack or requiring user intervention.

The relative speed and accuracy of the technique make it an attractive choice, and the results are fairly consistent across the data sets. My system is designed to deal with such discontinuities from thresholding by making use of both the synthetic ray projection image and the original image slice images. See the seeding section below for details on how artificial discontinuities are dealt with.

Most algorithms designed to reconstruct neurons from image stacks apply a Gaussian convolution to the images before processing them to deal with noise. This approach is computationally costly, and the resulting images are then synthetic and lose most of the fine detail captured by the imaging process. Further processing done to such altered images would be incapable of utilizing the gradient details between proximal branches in producing their reconstructions.

By using a combination of the synthetic ray projection and simply thresholding, I am quickly and efficiently eliminating the noise present in the stack. I am also reducing the search space for my seeding class to only a pool of pixels of very high probability of being within the

neuron structure. However because I apply the threshold to only the synthetic ray projection, all of the sampled slice image data used to construct it remain unaltered and available for finer examination in later steps.

3.2.3.3 Intensity Examination

Once the stack has been reduced as above and thresholded, the next step is to examine the intensity values of the pixels in the modified ray projection. For this system I am assuming the images processed show the neurons in a lighter color value on a dark to black background. The system could handle input of the opposite contrast if the intensity values (explained below) are simply inverted.

The images within the stacks used in this paper are in grayscale TIF format. I use the grey channel information of a pixel to determine its brightness intensity value. The values range from 0 (least intense) to 255 (brightest intensity). A pixel representing a neuron will have a higher intensity value than one representing the background. Pixels between the neuron and background will show a change in intensity over distance. Noise can be present in the image as lighter intensity pixels among background pixels or darker pixels among lighter.

To define the classes of pixels formally, I state the following definitions:

$P_i = \{x_i, y_i, i_i\}$ where $0 \leq i \leq 255$
$N = \{P_1, P_2, \dots P_n\}$ where $P_i(i) > T$
$B = \{P_1, P_2, \dots P_n\}$ where $P_i(i) < T$

Where P_i is the ith pixel in an image slice in a stack having properties x and y setting its position on the X and Y axis, and i indicating its intensity value, N is the collection of pixels compromising the neuron within the image, B is the collection of pixels defining the background non-neuron elements, and T is the threshold that divides members of N from members of B.

The seeding is performed by my SeedFinder class. It takes the thresholded ray projection image as an input and returns a list of data structures called Coords. A Coord instance contains the seed's center X and Y coordinate in the image plane, the z-index of its position in the image stack, the Z coordinate in pixels, the seed's radius and an average intensity value for the pixels it encircles. The goal of the program is to produce a set of these Coord objects that represent the best choices for a local tracing algorithm to begin its search.

The initial process of seed creation is to scan the input ray projection image and find pixels with an intensity greater than 0. Remember that the input ray projection has been thresholded and is therefore comprised only of pixels belonging to set N. All the pixels of set B have had their intensity set to 0 and can therefore be ignore, as they cannot under any circumstances be considered as members of set N.

To determine the intensity value of a pixel, I perform a bit-shift operation on it using the following:

Intensity value = pixel RGB data & 0x00000FF

When a pixel of set N is found, a new Coord object is created. The center X and Y values are set to the pixels x and y properties, and the average intensity for the Coord is set to the pixel's intensity value. The Z value for the Coord is set to 0 – later steps will alter this value to properly place the Coord on the correct slice index. Once the scan of the ray projection is complete, the resulting collection of Coord objects represents the entire N set of pixels.

It is quickly apparent that using this set of Coord objects as seed points is useless – not only do they contain no depth information, but they represent an over-representation of the neuron cell. Not every pixel of the thresholded ray projection is important in describing the shape of the cell, and simply finding the brightest pixels within the stack does not give us any information about the overall structure of the neuron or the relative thickness of the cell as it travels through 3 dimensional space. Further processing is required to produce a seed set of useful starting points for tracing programs.

3.2.3.4 Seed Point Selection

The collection of Coord objects obtained from the intensity examination represents my pool of highest probability neuron pixels, from which my seed selection will occur. The goal is to find a set of seeds that represent areas of high probability of belonging to the neuron structure, accurately estimate the width of the cell structure at that point while being as equally spaced along the neuron tree as possible.

There are multiple techniques that are employed to perform centerline extraction and provide for estimations of object diameter. [17] [10] [18] However, the methods all utilize a binary image to perform their functions. The ray projection image used in my algorithm has multiple intensity values, and such information is important in determining the best seed point selection.

My approach is similar to the maximal disk method discussed in [17]. The maximal disk method provides centerline and area estimation of an image object through the use of a series of disks laid over the object. The formal definition is:

There exists in the shape a disk, for which there is no other disk in the shape that contains it

In any given shape, we can describe a series of disks such that each disk is maximal in terms of the area of the object it contains, and no disk is subsumed by any other. If we were to then take and connect the center points of these disks, the line created would be the true centerline of the object. We would also be able to determine the width of an object at given skeleton point by the size of the disk centered at that point.

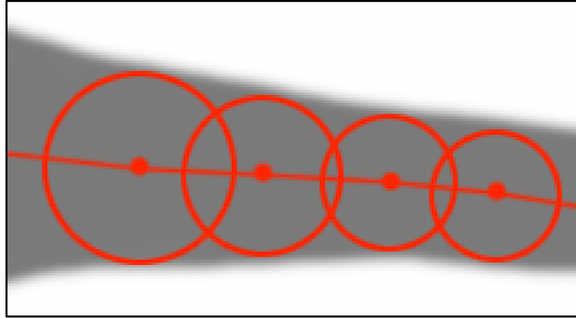


Figure 19– a centerline described using maximal disks

This approach gives a very detailed skeletonization of an image object – yet again ignores any underlying data the pixels in the object may have to offer in the consideration of our seed points. Being accurate in terms of the centerline of the neuron is only as important as producing seeds of high average intensity, as those constitute the points where the likelihood of hitting the neuron is the highest.

I adapt the maximal disk approach by utilizing a kernel based search to find the maximal disks around the highest intensity pixels. By isolating the disks to the high intensity pixels, I am ensuring that the resulting seed points are very likely to be part of my neuron. By ensuring that no seed found can be subsumed by another I ensure that I generate a trail of center points nearly as detailed as the maximal disk method.

The pixels with the highest intensity value are the most likely candidates for inclusion in the neuron structure. However, as discussed previously, it is possible for the image to have noise or other artifacts present that might remain after thresholding has occurred. Some of these remaining noisy pixels could potentially be of very high intensity.

The key difference between high-intensity pixels that should be considered as indicating likely seed points and those that should be eliminated is the average intensity found within the pixels neighborhood. Noisy pixels represent abrupt changes in intensity compared to the pixel neighborhood, whereas pixels within the neuron generally fall into a gradient of intensity. By iteratively exploring the pixel neighborhood of the highest intensity Coord objects, we can quickly tell which are centered at noise artifacts and which are, in fact, parts of the neuron structure.

The collection of Coord objects is first sorted by intensity value. I then iterate through the list conducting a recursive kernel search on each Coord beginning with the most intense. A kernel of size 1 is applied to the Coord's x and y coordinates on the ray projection. Each of the neighboring pixels is then tested for an intensity value of 0 – as the thresholding process has eliminated any pixel below the level T, any pixel remaining that is surrounded by pixels of intensity 0 must be noise. If no blank pixels are encountered, the kernel is expanded by one. Another check for blank pixels is conducted at the kernel's center top, bottom, left and right pixels as well as at the corner pixels.

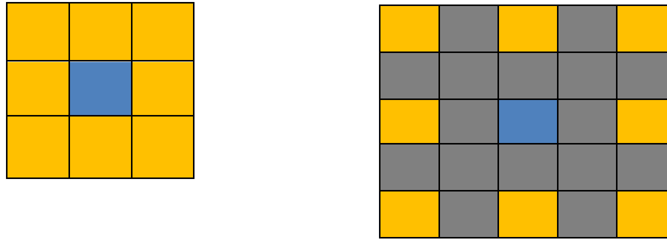


Figure 20– the seed selection kernel at size 1 and 2. The blue square reflects the current center of the potential Coord. The yellow squares reflect the pixels being tested. Detection of a blank pixel triggers the movement of the center pixel before expanding the kernel

At this point encountering a blank pixel at any testing site will either cause the kernel to re-center and test again or terminate the search. A blank pixel found along one or two adjacent sides of the kernel will increment the kernel's center coordinates along the X and Y axis where another test will be conducted – if no blank pixels are found, then the process continues and the kernel expands.

If blank pixels are found on opposite sides of the kernel it terminates the search. This property is to encourage the resulting seeds to be circular in nature. Limiting my seed shape to circles and spheres makes distance, connection and angular calculations much simpler, and is a better representation of a seed point than a cylinder would be, although some algorithms do use generalized cylinders to describe the tube-like shape of a neuron.

Once the search has completed, I am left with a kernel covering a subset of group of N pixels, with a center point as close as possible to the original high-intensity pixel. If the kernel is large enough, I create a new Coord object using the kernel's X and Y coordinate information, and set the new Coord's radius equal to the kernel size minus one – to offset the last kernel expansion before the stopping criterion was met – and its average intensity equal to that of all pixels found within the area of the circle calculated using these values.

I purposefully reject new Coords with a kernel size of 2 or smaller to eliminate surviving noise elements and smaller details of the neuron that are beyond the scope of my approach. Although the end of neuron branches tend to be smaller in size than other parts of the cell and end point seeds were previously identified as being of significant value to understanding the overall morphology, they also tend to be dimmer in intensity and therefore require special consideration to differentiate them from extraneous elements. As I am attempting to only produce seeds with high probability of being within the neuron tree, dimmer end points, although valuable, are not reliable indicators. In order to maintain the quality of maximal disks that none can be subsumed by a different disk, I also run a comparison on each new Coord to ensure that it is not entailed by an already found seed. Any such seed is then rejected.

Figure 21– the results of the seeding algorithm

The resulting set of seeds, which are now considered candidates, can be seen in figure 21. The seeds appear to be maximal and encompass the entire area defined in the thresholded ray projection, although there is a loss of detail on areas where discontinuities were introduced.

Although this set of seeds appears to describe well the shape of the neuron, it still does not contain any depth information. All of the seeds currently have a z-index and pixel value of 0. In order to make use of these seeds, we need to establish where in the image stack they can be placed.

As discussed previously, one of the problems with collapsing the image stack into a single image is that crossovers – points at which branches of the neuron cross over and under each other – become impossible to resolve. We can detect when they occur as they introduce cycles into an ordered graph of the candidates, but we cannot disambiguate a higher branch from a lower. We are also unable to discern when a branch may be traveling mainly along the z-axis for any period of time, as all of the pixels describing that movement would be lost.

Recall that I saved all of the slice images within Slice objects during the collapsing of the search space. Each of these images can tell us exactly on which slice each of these seeds is the most intense.

A new class called the SliceSorter accomplishes this task. Taking as input the SliceInspector instance and candidate list, the SliceSorter is responsible for scanning the images from the stack, examining the average intensity described by each candidate at each level and determining what the appropriate z-index classification should be.

For each slice image used in the ray projection, the SliceSorter first applies the global threshold to the image. It then iterates through the candidate list and calculates the average intensity for the pixels described by the candidate’s x, y and radius properties. If this average is found to be greater than 0, the Coord has the slice index and intensity passed to it for later

calculation. Once the SliceSorter has finished examining each stack image, it tells each of the candidate Coords to calculate their own best z-index value.

The candidate Coord now looks at an internal array of both z-indices and intensity values that was built by the SliceSorter. Although the array is mostly empty, intensity peaks within this array signal that this Coord should be replicated and added to the candidate collection with a second z-index value. This process establishes Coord object above and below points of crossover, resolving the problem discussed earlier.

However due to the earlier discussed property of these image stacks where the actual difference between successive and even once-removed slices is fairly small, it is possible for a candidate Coord that is not part of a cross over event to have multiple intensity responses recorded on many neighboring slices. Furthermore, it is possible for these responses to contain several local maxima which could result in an over population of Coord objects to describe this particular area of the neuron.

In order to compensate for this effect I utilize a signal averaging scheme to find the best fit for each Coord. As the Coord moves through the array, for each index number it calculates the average intensity of the previous, current and next index number. This eliminates the problem of multiple local maxima for Coords not part of cross over events.

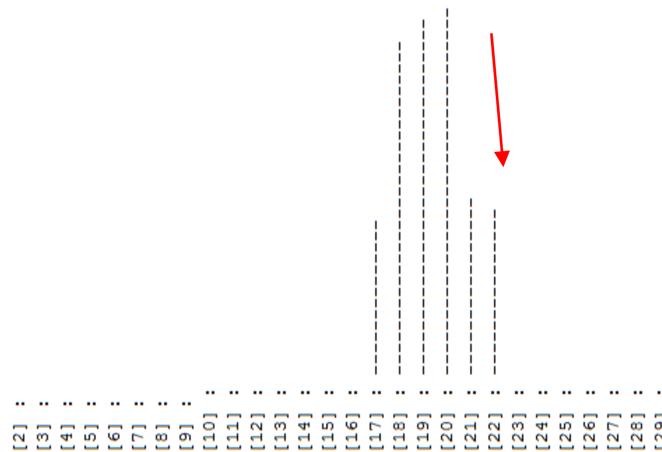


Figure 22– Coord signal averaging and peak determination over indices; red arrow indicates change in slope sign, peak found to be index 20

A change in the sign of the tangent of the function of the intensity values over the indices indicates a true local maxima and the correct setting for the z-index. For Coord objects that do not occur above or below crossovers this setting is ideal. However, in cases where a crossover has occurred, there will be two such local maxima. In this case, the Coord needs to do another check to see if the second peak is significant.

Because the candidate Coord objects were derived from a flattened ray projection of the images in the stack, they do not necessarily occur at x/y coordinates in any slice at the average intensity they record from the ray projection. Often their intensity value is lower, as the Coord's

center is offset from the data present on any given slice. Occasionally the Coord will have a radius that may straddle a portion of the neuron from a higher section of the stack and a lower portion, creating a second signal peak in the average although it does not describe the center point or radius of the distant area.

In order to ensure that new Coord objects are not created in these instances, a threshold is set for the intensity found at a second peak before another Coord object will be generated. In order for this new slice to have meaning, it must represent at least 80% of the intensity value registered at the other index to be considered valid. This threshold is intentionally high as the addition of new Coord objects is to be discouraged except in cases where confidence is high that the new object does indeed represent a section of the neuron.

If the second peak is found to meet or exceed the threshold, a new Coord object is created, cloning the x and y properties of the original Coord. The second z-index value is used to set the new Coord on a different slice and the new Coord is then added to our original set of candidate points.

Now that the candidate Coords have been set with depth information, we can begin to establish connections and relationships between them that describe the overall shape of the neuron being imaged. See figure 23 for the results of our depth estimations.

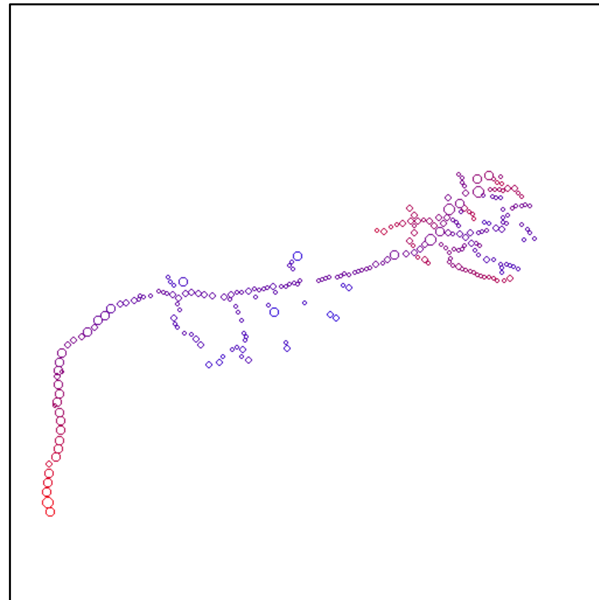


Figure 23– the seeds colored by depth; red indicates higher on the z-axis, blue lower

3.2.3.5 Establish Connectivity and Seed Pruning

The candidate Coord collection, at this point, represents a fair approximation of points and diameters that can describe the shape of the neuron. However in order to maximize the value of the seeds generated for use in tracing algorithms, further refinement can help us score individual Coords and return the optimal set.

I earlier described various classes of seeds and their relative value toward the task of neuron reconstruction – soma, end nodes, branching points, connectors to end points and branching points, and connector to other connectors in order of value. We can now try to apply these classifications to the candidate set and determine how many of each type we find. In order to do so, we must first establish connections between the Coords. My approach is similar to that used in [11] where a score derived from the relative spatial, intensity and angular values between different Coords is used to determine their connection.

The Connector class is responsible for finding possible connections between the Coords in the candidate collection. The Connector does this by first sorting the candidate list by the x, y and z values of the Coords. For each sorting, it looks at the 50 Coords before and after it in the list and runs a comparison of the positions in the X/Y/Z coordinate space. The value of 50 Coords was set due to the fact that Coords existing in 3 dimensional space may be proximal along any single axis while being separated by large distances across either or both of the other two axis. In order to compensate for this false sorting within an axis value, a large window of examination is needed to find the true potential set of spatially proximal seeds. The value of 50 was found to represent a large enough proportion of the total seed count to achieve the desired results.

The function that compares the positions of the Coords runs a simple check of the Euclidean distance between them and compares this value with the sum of their radii. Specifically I use the following calculations for the distance between the points and the sum of the radii:

$$\text{Distance D} = \sqrt{(P_{1X} - P_{2X})^2 + (P_{1Y} - P_{2Y})^2 + (P_{1Z} - P_{2Z})^2}$$
$$\text{Sum of radii R} = P_{1R} + P_{2R}$$

Where P_1 and P_2 are the two points being compared. So Coords are considered to be connected when $D <= S$.

However there are occasions in the seeding process where a seed of significant size may be proximal to a seed of much smaller size. This can occur when a high intensity pixel happens to occur at a corner of a neuronal arc, or when a spur – a small nodule jutting off from the branch that does not connect to another node yet has a high average intensity value – appear along a branch.

Recall that I am assuming that these neuron cells exhibit gradual changes in diameter over distance, and that I am not attempting to reproduce the neuron cell in all of its detail at this time, merely trying to select the points that most likely represent the neuron structure. In order to better represent the connections among the larger, higher value Coords I limit connections between Coords to cases where the difference between their radii is less than 3. This means that rapid changes in radius between close Coords will disqualify them from connecting to each other.

When two Coord objects are found to be within connection distance, they are added to each other's connectedSeeds list. These lists are used to determine local Coord neighborhoods in

later steps, but are also used at this point for Coord classification by connection count. Connection count gives us an understanding of the local density of Coord objects around our current Coord, and tells us if this Coord represents an area with low population – meaning it is most likely a connection Coord linking two other Coords – or with high population – meaning it is either occurring at a branching point or indicates an area of high complexity within the neuron structure.

The Connector class examines the number of connections for each candidate in the list and assigns them to groups depending on their connection count. The classification categories for Coords and their meaning are:

- Zero connections – Singleton
  Singletons are Coords that have not connected to any other Coords. There are several reasons why a Coord may not be connected, the most important of which is that it may be a localized area of image noise that survived the thresholding and seeding process, or an extraneous structure within the imaged medium. However other singletons may exist due to my connection radius limit discussed above, or from being derived from areas of low dye absorption and therefore false discontinuities.

- One connection – End Point
  Coords having a single connection are regarded as end points. These points represent good starting points for tracing as they describe a vector from which a program may begin. However end point classification suffers from occasional instances where two Coords from an extraneous or noisy area may be linked together (each becoming an end point rather than two singletons) and from local discontinuities.

- Two connections – Middle Point
  Middle points are the least desirable class of seed point Coord, but represent the class that is most easily validated. There are occasions where small Coords proximal to other Coords may form connections with them both and fall into this class without representing a true path along the neuron structure, creating a jutting connection like a spur, but the radius control on connections limits these occasions.

- Three connections – Bifurcation Point
  When a neuron cell splits (or bifurcates) into two sub branches, we would define this as a single Coord object being connected to the last middle point Coord of the parent branch and the first two middle point Coords of the child branches. Coords may also fall into this class when present at areas of high tortuosity, where the neuron structure bends or wraps around itself sharply.

- Four or more connections – Blob Point
  Blob points occur where the complexity of the neuron structure is greatest; they represent areas of high ambiguity and small structural distinctions within the neuron arbor. Although they are not necessarily a useful point from which tracing may begin, they are useful flags for algorithms that employ particular kernels, models or other detection

schemes that are capable of adapting size or orientation that multiple interpretations of this specific sub area may be beneficial.

Figure 24– the result of the seed connection algorithm

Now that I have established the basic connections among my candidate set, I want to refine the connections to remove some of the problem mentioned above in the seed classification explanations. I utilize a class called Refiner to make improvements to the different classification sets. I do not attempt to refine points in the singleton or end point class, as singleton points that are valid in the description of the neuron must exist beside other structures that fall into one of the other classes and end points that exist as merely the connection between two Coords do not necessarily belong to extraneous image information. I rely on later groupings into what I call Fragments to deal with these considerations.

The first group examined is the middle point class. The goal is to eliminate as many of the spur points discussed previously as possible. In order to do so, I first determine in the Coords connected to this middle point are both bifurcation points. I then examine the angle formed between the three points and determine if the connection is too acute to constitute a possible curve in the neuron tree.

If the connected Coords are both bifurcation points – that is, have connections to three other Coord objects – the most likely scenario is that the current Coord is a spur and its two connected Coord objects are actually middle points connected to each other (and an additional Coord in either direction). By deleting the Coord erroneously labeled as a middle point, we correct the spur in the grouping.

If the connected Coord objects do not belong to the bifurcation class, then the angle formed between the three Coord objects must be examined. To determine the angle between the three points in the 3-dimensional space, I calculate the dot product and vector lengths of the coordinate values of the three Coords along the X, Y and Z axis. I then define the cosign of the angle as being the quotient of the dot product divided by the vector length.

$$\cos A = \frac{(X_1 * X_2)+(Y_1 * Y_2)+(Z_1 * Z_2)}{\sqrt{(X_1^2 + Y_1^2 + Z_1^2) * (X_2^2 + Y_2^2 + Z_2^2)}}$$

Where P_1 and P_2 are the child points of the Coord currently under consideration.

Once angle A has been determined, I set a lower limit on the value it can hold before I consider this point a disguised spur. For the purpose of removing as many false spurs as possible, and because of the assumptions discussed previously about the general limit or tortuosity of the neuron branches, I set this value to 90 degrees. And middle point exhibiting an angular connection between its predecessor and successor below this limit is occupying an area of unusual complexity within the neuron and should be so classified.

When this occurs, I collapse the connected Coord objects into a single point. A function in the Refiner class takes the two Coords as input and generates a new Coord that is an average of their coordinates in the X, Y and Z space and an average of their radius and intensity measurements. The Coord currently being considered is now an end point connecting to only this new Coord. This new Coord in added into the list of candidate Coords which will be re-submitted for connection searching after refinement.

The approach for refining the bifurcation class of Coords is somewhat similar, except that I have to compare three angular measurements. The three angles to consider are, labeling the node currently being considered as B and its connected Coords as A,C and D: angle ABC, angle ABD, and angle CBD. If any of these angles is found to be lower than the threshold angle established above, the two connected Coords are averaged and the node we labeled as B becomes a middle point.

The refinement of blob points is somewhat tricky. As discussed previously, blob points represent areas of extreme uncertainty and complexity. Refining connections within such areas is difficult and would require substantial calculations. However, as my goal is not to definitively resolve such ambiguity but to adjust and signal for it, I utilize the following method of slimming down the blob point class.

As noted earlier, the goal of this system is to produce a set of high-probability seed points for tracing algorithms. As such, the points that present the best chance of actually showing an area of the neuron structure are those that have the largest radius and highest average intensity. Blob points exist in areas where there is a high density of possible neuron pixels, and as such are more likely to belong to either the bifurcation of middle point classes.

A clear determination cannot be made, however, so in the interest of preserving as much information as possible for the destination program about these areas, a connection limit of three Coords is used to remove the blob point class members. The Coords connected to a blob point are first sorted by their radius values. Then, all but the top 3 radius Coords are deleted from the candidate set.

Once this refinement of the candidate list of Coord objects has been completed, the list is passed again to the Connector class. The new list is processed and the resulting connected structures are then used in the next step, assigning them to fragments.

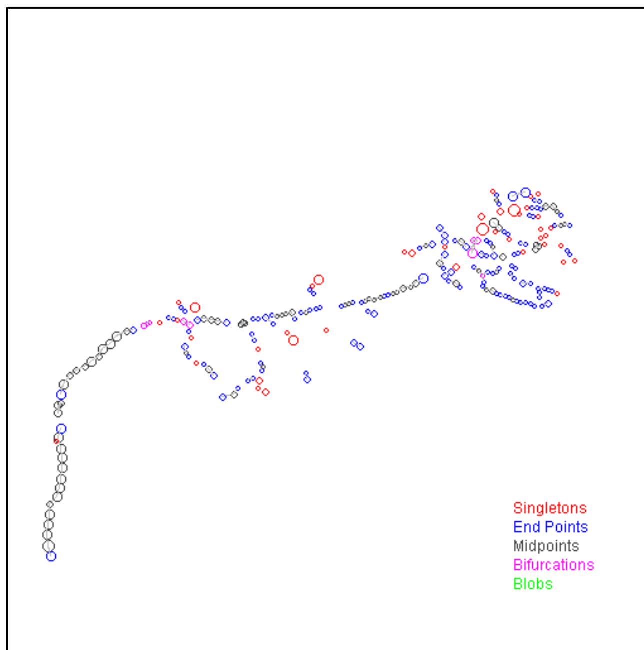


Figure 25– the refined connections

3.2.3.6 Group Connections

The candidate list now represents the best estimation of the stack volume that contains neuron information, and has within it a rough estimation of portions of morphology. However the connections formed in the previous steps and the various Coord classifications do not ensure a fully connected version of the neuron. In order to produce a list of seed points that encompasses as much of the total neuron structure as possible, further refinement must be done.

In order to accomplish this, I use a data structure called a Fragment. A Fragment contains a list of Coords that all have each other as either members of their connectedSeeds list, or have ancestral connections through other Coords. For each member of the candidate list, a new Fragment object is created and the Coord is added to it. Then a depth-first search is conducted of the Coord's connectedSeeds list, assigning each Coord found to the same Fragment. What results is a Fragment describing an area of well-understood topology.

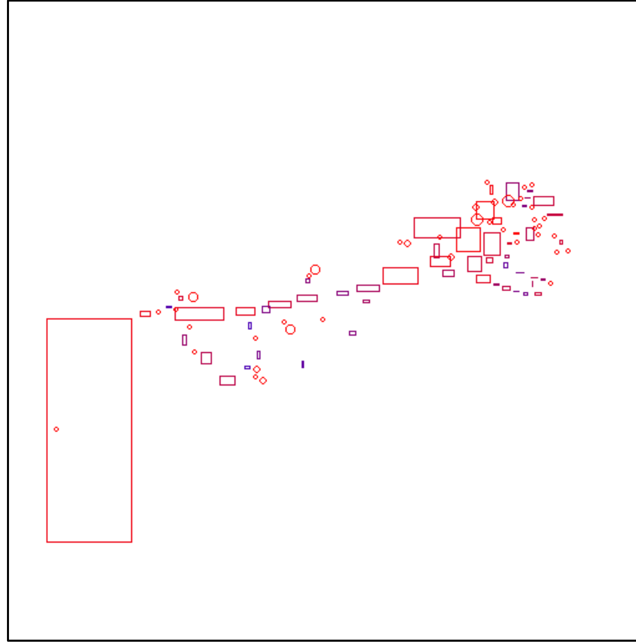


Figure 26– fragment assignment and singletons; red borders represent higher top-extrema of fragment members, blue lower

Fragments also show sections where discontinuities have been introduced; we can also tell from neighboring fragments how these discontinuities may best be overcome. Large fragments proximal to other fragments of significant size can most likely be safely connected at their closest members, whereas fragments that overlap along the X and Y axis may also be joined along the Z.

Now that I have the collection of Fragment objects, I build my final set of potential seed points by asking each to submit its best points. As discussed in the section on the refinement of blob points, the largest and brightest points are the obvious choice for inclusion.

Each Fragment first sorts its members according to their radius value. Then the top members from the Fragment are selected and appended to the list of final seed choices. The number of seeds selected from the Fragment is varied depending on the Fragment's member count.

Fragments with large member numbers will be skewed toward certain sides of the image stack – see figure 26 for an example. If we were to select a set number from each Fragment, then we would end up over sampling of one side of the stack. Instead, I first test the number of members of the Fragment and return a different proportion depending on the member count.

Fragments with more than 10 members will return80% of their top Coords whereas smaller Fragments return 90%. This ensures that Fragments representing small structures will be almost entirely represented in the seed set; whereas the finer detail of larger Fragments will be paired down to give a more evenly spaced representation of that area.

Of the initial candidate list, those that were added to a fragment object had more than one connection. That means that in addition to the points that now compromise the final seed set, I have a list of singletons that need to be considered. However this list may contain Coord objects that were created from initial image noise of extraneous structures in the image stack and need to be screened to allow only the most likely candidate seeds to be included in the final seed set.

In order to ensure that the seeds selected from the singletons set have a high chance of representing areas of the neuron structure, only those that represent a large seed should be considered. Coords derived from image noise or extraneous structures are typically smaller in size and can therefore be partially eliminated from this list by selecting only Coords on a certain size. In order to select such Coords, I select only the top 20% of the singletons list once the list has been sorted by radius size.

3.2.3.7 Preparing and Exporting Data

I have now constructed a list of seed points for the image stack that contains areas where there is a very high probability that neuron structures exist. I have also ensured that these seed points are widely distributed across the neuron and that they reflect a fully-connected version of the structure being imaged. I have also established points where the local image information is complex and areas of where multiscale analysis or other adaptive tracing methods may best be applied.

In order to prepare and export this data, it must first be noted that construction of a SWC file is both impractical and impossible from this data set. A SWC file describes a fully ordered set of reconstruction points, whereas this list contains at best a number of undirected graphs. Further, SWC file generation is the goal of the receiving tracing program and as such should not be used as an input element.

I choose a plain text file to consolidate and store the information collected in this system. The statistics concerning the overall stack properties are listed first, followed by a tab delimitated output of the X, Y, Z and radius values of the Coords comprising the final seed set. This format is flexible enough for all programs to work from without artificially constructing a SWC file.

Similar to the statistics reporting used in [19], my system collects various statistical properties of the image stack and reports them to the subsequent tracing algorithm. Specifically, I record the following values about the image stack in the output file:

- Slice pixel offset value
- Slice image dimensions
- Slice count
- Mean of the collapsed ray projection image
- Extrema intensity values
- Standard deviation
- Signal to noise ratio
- The optimal thresholding level discovered

OP_1_sr_1.txt - Notepad

File Edit Format View Help

```
File selected as first in stack:        C:\Users\gbilodeau\Documents\0 - Olfactor
Slice Pixel Offset:     3.03
Sampling rate:  1
Image dimensions:       512px x 512px
Total number of slices in stack:        60
Total number of slices sampled: 60

Image statistics:
Mean of the ray projection:     6.852874755859375
Extrema intensity values:       high = 254.0; low = 0.0
Standard Deviation:     5.668754637140687
Signal to Noise ratio:  1.2088854068511876
Threshold level:        117 (out of 255)

Begin seeding
Found 6930 pixels with intensity greater than 0
Found these many potential seeds: 257
Average seed radius: 4.120622568093385
Average intensity of seeds: 133.15953307392996

Start slice assignment
Pre-slice assignment candidate count: 257
Post-slice assignment candidate count: 282
```

Figure 27 - example output file

4 Results

In order to make a quantitative, non-subjective analysis of my result set, I use the 'gold-standard' SWC reconstruction files provided by the DIADEM competition. These files are the result of a manual reconstruction of each image stack by an experienced researcher and represent a full collection of points to describe the neuron being imaged. The points listed in the SWC file have X, Y, Z coordinate and radius properties and denote the ordered connection between each point.

There exists a difference, however, between the points described in my result set and the points described in the SWC file. The points in the SWC file are set along the true centerline of the neuron, are entirely connected for the full neuron morphology, adjusted precisely for local image irregularities and take fine structures and spurs into account. My result set is merely a collection of points that, if properly processed and examined by another program, may result in the definition of the SWC point. As such there cannot be an analogue in my result set for each point in the SWC file, and for each point defined within my result set a certain margin for error must be allowed.

In the evaluation of my system, I establish four testing error margins of 1, 2, 3 and 4 pixels. The margin of 1 pixel will show me which of my points were found within a sub-pixel neighborhood of a gold standard point. The upper limit of 4 encompasses the pixel separation value between the slices and as such controls for my sampling of half the image stack.

To run this comparison, I utilize a class called SWCComparator. This class opens an associated SWC file designated by the user and compares each point in the results set to each point in the SWC at the various error margins. It also tracks the average difference in radius between the points.

The SWCComparator class returns a set of Coord objects (if any) that have been found to match points defined within the SWC file below the current tolerance limit. This set is called the winners set and represents points that may be used as seed points by tracing algorithms with a high degree of confidence. The class also returns a list of Coord objects (if any) that fail to be matched to a SWC point. This set is called the loser set. See figure 29 for an example.

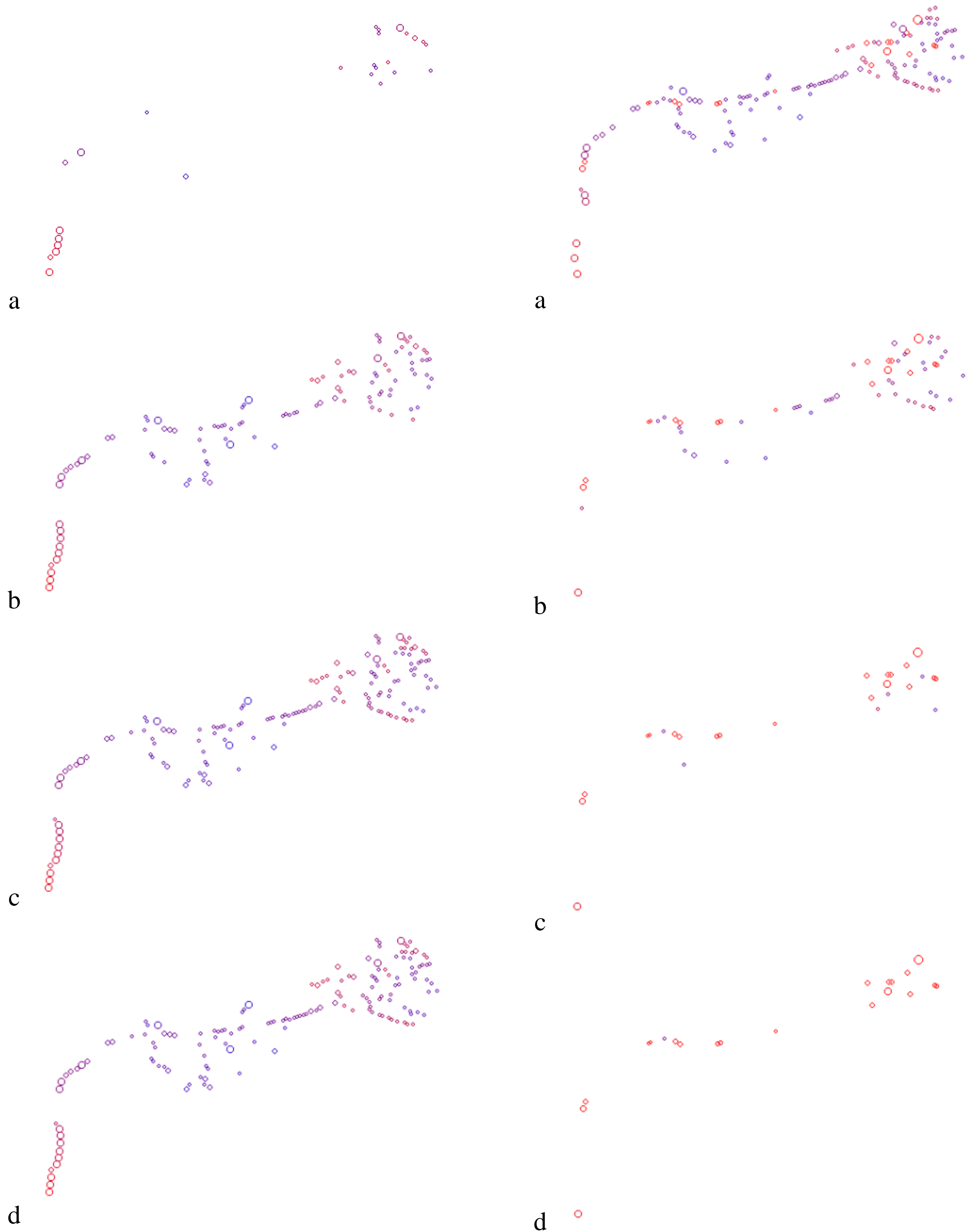


Figure 28– left column: the seeds matching the gold standard (winners set); right column: seeds rejected (losers set), at tolerance levels of 1 (a), 2 (b), 3(c) and 4(d) pixels for OP_1 at sample rate 2

Presented below are the resulting images of each of the nine stacks used in this paper. Each page shows the comparison between the sample rate of 1 and 2 slices within the stack

starting with the seed set submitted for comparison purposes. As the collection of pixels from which seeding is conducted is different for each sampling rate, both source seed sets are shown for comparison. I also show the winner sets for both sampling rates at the 1 and 4 pixel thresholding levels. These encompass the results that are most precise and least precise, respectively.

For the sake of discussing the results, I am also providing the collapsed versions of each original stack below. These MIP ray projections were constructed using the sample rate 1, which means no information is being left out and represents all of the image data present in the stack.

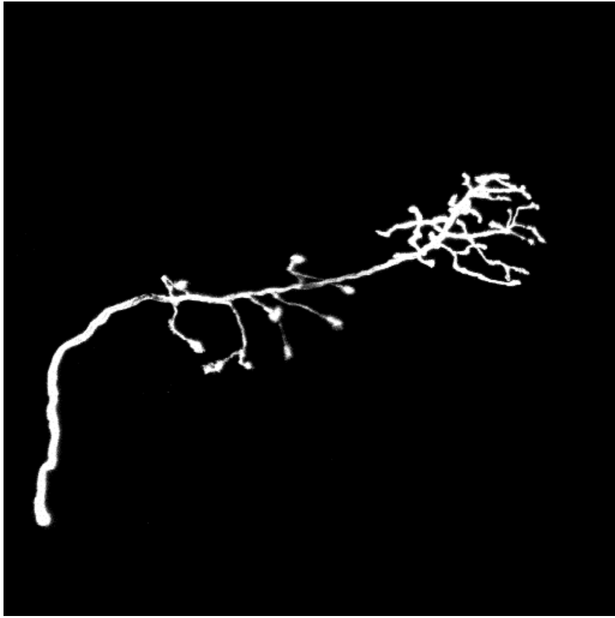


Figure 29 - OP_1 MIP

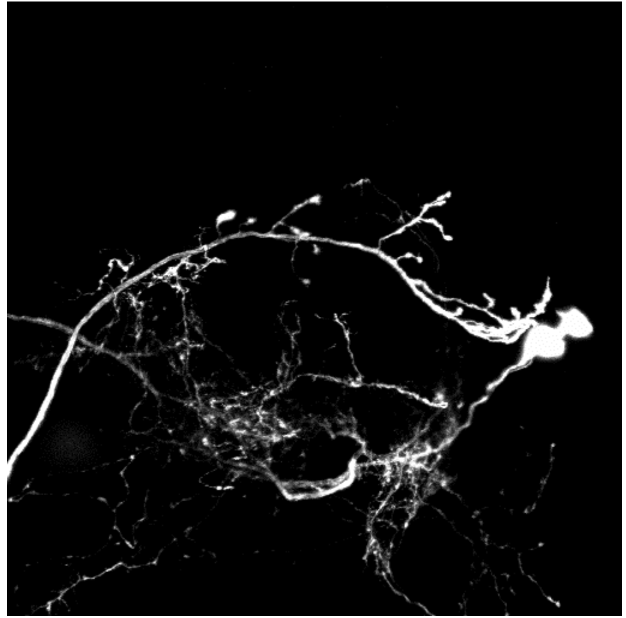


Figure 30 - OP_2 MIP

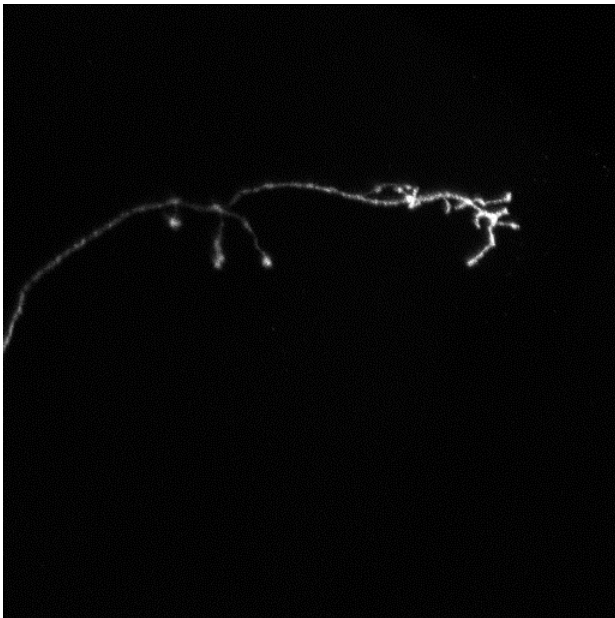


Figure 31 - OP_3 MIP

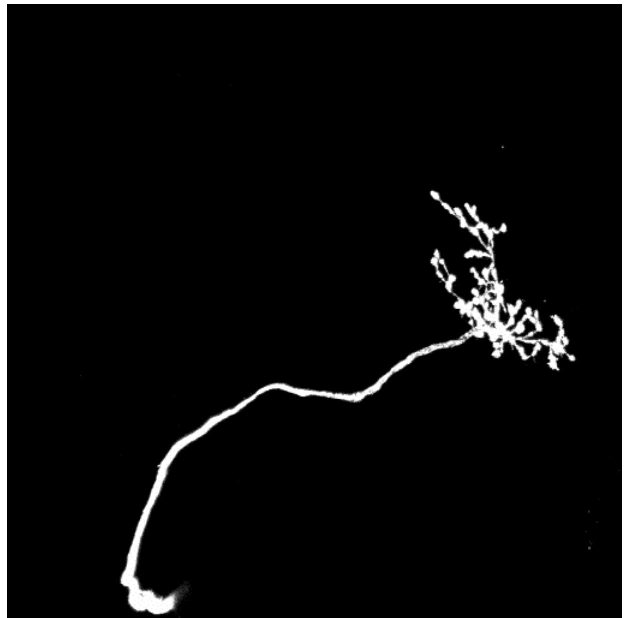


Figure 32 - OP_4 MIP

Figure 33 - OP_5 MIP

Figure 34 - OP_6 MIP

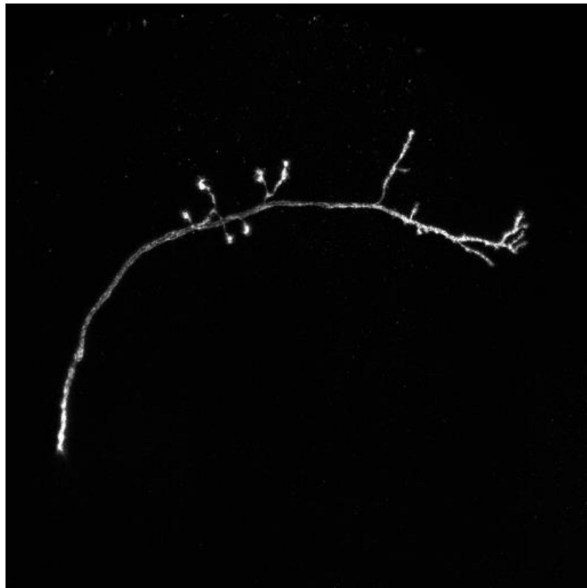


Figure 35 - OP_7 MIP

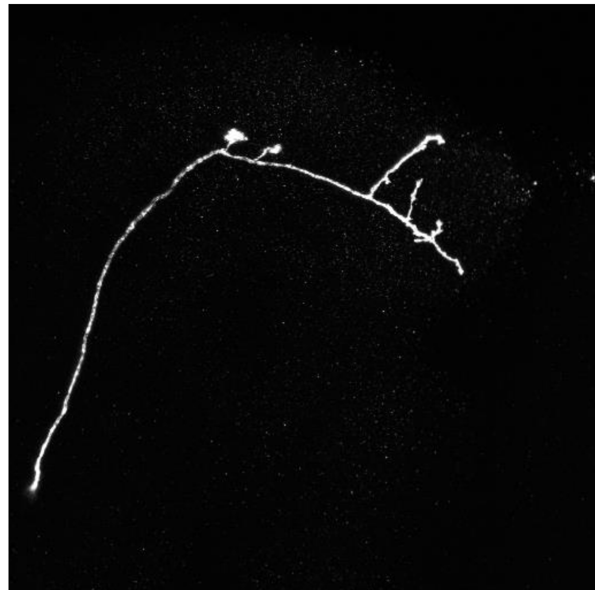


Figure 36 - OP_8 MIP

Figure 37 - OP_9 MIP

OP_1 Results

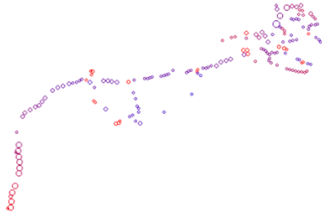


Figure 38 - set of final seeds for OP_1, sample rate 1

Figure 39 - set of final seeds for OP_1, sample rate 2

Figure 40 - winner set, sample rate 1, 1 pixel threshold

Figure 41- winner set, sample rate 2, 1 pixel threshold

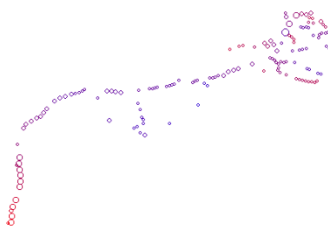


Figure 42- winner set, sample rate 1, 4 pixel threshold

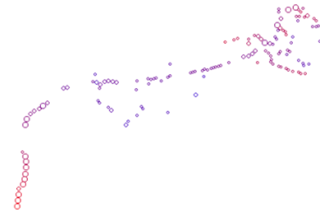


Figure 43- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_1 image stack contains 60 slices, all of which were read for sample rate 1 and 30 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 6.8529, with a standard deviation of 5.6688 and signal to noise ratio of 1.2089. The mean calculated for sample rate 2 was 6.4319, with a standard deviation of 5.6688 and signal to noise ratio of 1.1346.

Seeding Results:

The MIP for sample rate 1 contained 6,930 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 117. Seeding the thresholded version of the MIP yielded 257 potential seeds, with an average radius of 4.1206 pixels and an average intensity of 133.1595. Following the slice assignment process, the seed count was increased to 282 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 168 Coord objects.

The MIP for sample rate 2 contained 6,519 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 117. Seeding the thresholded version of the MIP yielded 249 potential seeds, with an average radius of 4.0763 pixels and an average intensity of 129.8715. Following the slice assignment process, the seed count was increased to 260 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 148 Coord objects.

Image Ambiguity Alerts:

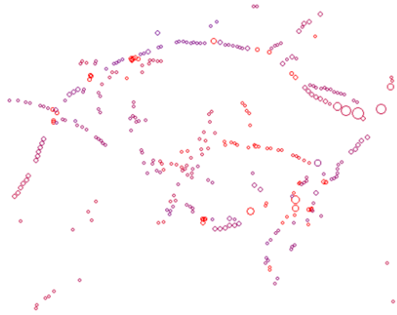
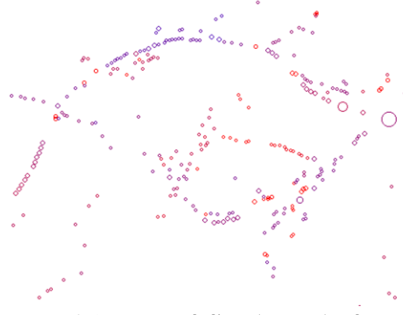
The sample rate of 1 produced a set of 2 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 4 such Coords.

Accuracy to SWC Seeds:

Of the 168 seeds submitted for validation by sample rate 1 against the ‘gold standard’ SWC reconstruction 88 were confirmed valid and 80 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.36 pixels. 135 were validated and 33 rejected at the threshold of 2 pixels, with an average radius difference of 1.57. 140 were validated and 28 rejected at the threshold level of 3 pixels, with an average radius difference of 1.56 pixels. 142 were validated and 26 rejected at the threshold level of 4 pixels, with an average radius difference of 1.58 pixels.

Of the 148 seeds submitted for validation by sample rate 2 against the ‘gold standard’ SWC reconstruction 28 were confirmed valid and 120 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.9 pixels. 98 were validated and 50 rejected at the threshold of 2 pixels, with an average radius difference of 1.7. 127 were validated and 21 rejected at the threshold level of 3 pixels, with an average radius difference of 1.55 pixels. 133 were validated and 15 rejected at the threshold level of 4 pixels, with an average radius difference of 1.55 pixels.

OP_2 Results

 Figure 44 - set of final seeds for OP_2, sample rate 1	 Figure 45 - set of final seeds for OP_2, sample rate 2
Figure 46 - winner set, sample rate 1, 1 pixel threshold	Figure 47- winner set, sample rate 2, 1 pixel threshold
Figure 48- winner set, sample rate 1, 4 pixel threshold	Figure 49- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_2 image stack contains 88 slices, all of which were read for sample rate 1 and 44 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 11.1259, with a standard deviation of 3.0853 and signal to noise ratio of 3.6061. The mean calculated for sample rate 2 was 9.9191, with a standard deviation of 3.0853 and signal to noise ratio of 3.1249.

Seeding Results:

The MIP for sample rate 1 contained 15,601 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 61. Seeding the thresholded version of the MIP yielded 603 potential seeds, with an average radius of 3.7529 pixels and an average intensity of 96.2637. Following the slice assignment process, the seed count was increased to 683 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 330 Coord objects.

The MIP for sample rate 2 contained 13,867 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 61. Seeding the thresholded version of the MIP yielded 552 potential seeds, with an average radius of 3.7101 pixels and an average intensity of 94.1558. Following the slice assignment process, the seed count was increased to 603 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 275 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 9 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 8 such Coords.

Accuracy to SWC Seeds:

Of the 330 seeds submitted for validation by sample rate 1 against the 'gold standard' SWC reconstruction 5 were confirmed valid and 325 rejected at the sub-pixel threshold of 1, with an average radius difference of 3.2 pixels. 27 were validated and 303 rejected at the threshold of 2 pixels, with an average radius difference of 3.15. 57 were validated and 273 rejected at the threshold level of 3 pixels, with an average radius difference of 3.3 pixels. 79 were validated and 251 rejected at the threshold level of 4 pixels, with an average radius difference of 3.35 pixels.

Of the 275 seeds submitted for validation by sample rate 2 against the 'gold standard' SWC reconstruction 0 were confirmed valid and 275 rejected. 10 were validated and 265 rejected at the threshold of 2 pixels, with an average radius difference of 3.64. 36 were validated and 239 rejected at the threshold level of 3 pixels, with an average radius difference of 3.13 pixels. 64 were validated and 211 rejected at the threshold level of 4 pixels, with an average radius difference of 3.08 pixels.

OP_3 Results

 Figure 50 - set of final seeds for OP_3, sample rate 1	 Figure 51 - set of final seeds for OP_3, sample rate 2
 Figure 52 - winner set, sample rate 1, 1 pixel threshold	 Figure 53- winner set, sample rate 2, 1 pixel threshold
 Figure 54- winner set, sample rate 1, 4 pixel threshold	 Figure 55- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_3 image stack contains 62 slices, all of which were read for sample rate 1 and 31 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 6.5759, with a standard deviation of 1.9066 and signal to noise ratio of 3.4489. The mean calculated for sample rate 2 was 5.8019, with a standard deviation of 1.9066 and signal to noise ratio of 3.043.

Seeding Results:

The MIP for sample rate 1 contained 4,034 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 33. Seeding the thresholded version of the MIP yielded 95 potential seeds, with an average radius of 4.6 pixels and an average intensity of 76.6211. Following the slice assignment process, the seed count was increased to 99 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 53 Coord objects.

The MIP for sample rate 2 contained 3.662 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 33. Seeding the thresholded version of the MIP yielded 100 potential seeds, with an average radius of 4.36 pixels and an average intensity of 74.29. Following the slice assignment process, the seed count was increased to 103 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 50 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 0 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 0 such Coords.

Accuracy to SWC Seeds:

Of the 53 seeds submitted for validation by sample rate 1 against the 'gold standard' SWC reconstruction 3 were confirmed valid and 50 rejected at the sub-pixel threshold of 1, with an average radius difference of 2 pixels. 16 were validated and 37 rejected at the threshold of 2 pixels, with an average radius difference of 3. 25 were validated and 28 rejected at the threshold level of 3 pixels, with an average radius difference of 2.76 pixels. 32 were validated and 21 rejected at the threshold level of 4 pixels, with an average radius difference of 2.6 pixels.

Of the 50 seeds submitted for validation by sample rate 2 against the 'gold standard' SWC reconstruction 2 were confirmed valid and 48 rejected at the sub-pixel threshold of 1, with an average radius difference of 4 pixels. 15 were validated and 35 rejected at the threshold of 2 pixels, with an average radius difference of 2.67. 30 were validated and 20 rejected at the threshold level of 3 pixels, with an average radius difference of 3 pixels. 34 were validated and 16 rejected at the threshold level of 4 pixels, with an average radius difference of 2.82 pixels.

OP_4 Results

Figure 56 - set of final seeds for OP_4, sample rate 1

Figure 57 - set of final seeds for OP_4, sample rate 2

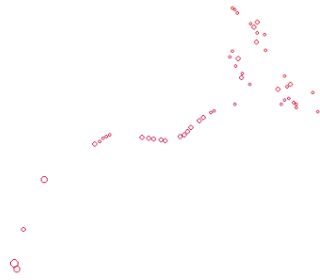


Figure 58 - winner set, sample rate 1, 1 pixel threshold

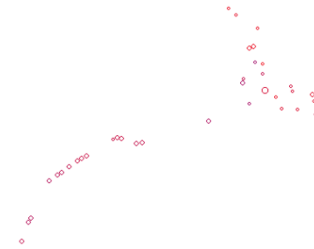


Figure 59- winner set, sample rate 2, 1 pixel threshold

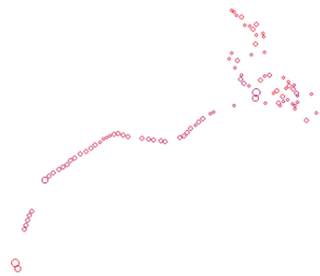


Figure 60- winner set, sample rate 1, 4 pixel threshold

Figure 61- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_4 image stack contains 67 slices, all of which were read for sample rate 1 and 33 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 5.9826, with a standard deviation of 7.7741 and signal to noise ratio of 0.7696. The mean calculated for sample rate 2 was 5.7028, with a standard deviation of 7.7741 and signal to noise ratio of 0.7336.

Seeding Results:

The MIP for sample rate 1 contained 6,264 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 110. Seeding the thresholded version of the MIP yielded 178 potential seeds, with an average radius of 4.5506 pixels and an average intensity of 144.5169. Following the slice assignment process, the seed count was increased to 200 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 116 Coord objects.

The MIP for sample rate 2 contained 5,989 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 110. Seeding the thresholded version of the MIP yielded 195 potential seeds, with an average radius of 4.3231 pixels and an average intensity of 141.9077. Following the slice assignment process, the seed count was increased to 213 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 120 Coord objects.

Image Ambiguity Alerts:

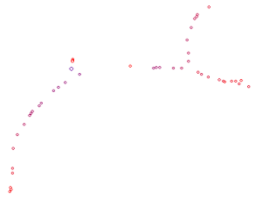
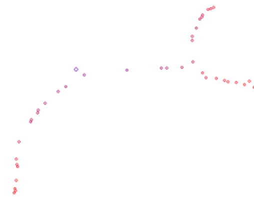
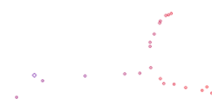
The sample rate of 1 produced a set of 2 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 2 such Coords.

Accuracy to SWC Seeds:

Of the 116 seeds submitted for validation by sample rate 1 against the ‘gold standard’ SWC reconstruction 52 were confirmed valid and 64 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.72 pixels. 73 were validated and 43 rejected at the threshold of 2 pixels, with an average radius difference of 1.87. 87 were validated and 29 rejected at the threshold level of 3 pixels, with an average radius difference of 1.98 pixels. 94 were validated and 22 rejected at the threshold level of 4 pixels, with an average radius difference of 1.97 pixels.

Of the 120 seeds submitted for validation by sample rate 2 against the ‘gold standard’ SWC reconstruction 36 were confirmed valid and 84 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.87 pixels. 88 were validated and 32 rejected at the threshold of 2 pixels, with an average radius difference of 1.73. 102 were validated and 18 rejected at the threshold level of 3 pixels, with an average radius difference of 1.67 pixels. 110 were validated and 10 rejected at the threshold level of 4 pixels, with an average radius difference of 1.64 pixels.

OP_5 Results

 Figure 62 - set of final seeds for OP_5, sample rate 1	 Figure 63 - set of final seeds for OP_5, sample rate 2
 Figure 64 - winner set, sample rate 1, 1 pixel threshold	 Figure 65- winner set, sample rate 2, 1 pixel threshold
 Figure 66- winner set, sample rate 1, 4 pixel threshold	 Figure 67- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_5 image stack contains 76 slices, all of which were read for sample rate 1 and 38 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 5.0631, with a standard deviation of 1.8429 and signal to noise ratio of 2.7473. The mean calculated for sample rate 2 was 4.3984, with a standard deviation of 1.8429 and signal to noise ratio of 2.3866.

Seeding Results:

The MIP for sample rate 1 contained 2,032 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 41. Seeding the thresholded version of the MIP yielded 104 potential seeds, with an average radius of 3.4038 pixels and an average intensity of 67.2788. Following the slice assignment process, the seed count was increased to 108 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 46 Coord objects.

The MIP for sample rate 2 contained 1,792 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 41. Seeding the thresholded version of the MIP yielded 98 potential seeds, with an average radius of 3.3878 pixels and an average intensity of 63.3776. Following the slice assignment process, the seed count was increased to 99 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 40 Coord objects.

Image Ambiguity Alerts:

Neither sampling rate alerted to areas of ambiguity on the stack.

Accuracy to SWC Seeds:

Of the 46 seeds submitted for validation by sample rate 1 against the 'gold standard' SWC reconstruction 1 was confirmed valid and 45 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.86 pixels. 7 were validated and 39 rejected at the threshold of 2 pixels, with an average radius difference of 1.97. 24 were validated and 22 rejected at the threshold level of 3 pixels, with an average radius difference of 2.02 pixels. 28 were validated and 18 rejected at the threshold level of 4 pixels, with an average radius difference of 2 pixels.

Of the 40 seeds submitted for validation by sample rate 2 against the 'gold standard' SWC reconstruction 1 was confirmed valid and 39 rejected at the sub-pixel threshold of 1, with an average radius difference of 2.04 pixels. 5 were validated and 35 rejected at the threshold of 2 pixels, with an average radius difference of 2.09. 14 were validated and 26 rejected at the threshold level of 3 pixels, with an average radius difference of 1.99 pixels. 22 were validated and 18 rejected at the threshold level of 4 pixels, with an average radius difference of 1.93 pixels.

OP_6 Results

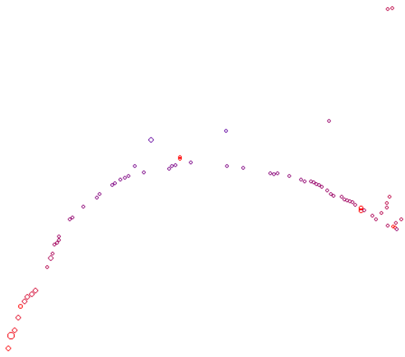


Figure 68 - set of final seeds for OP_6, sample rate 1

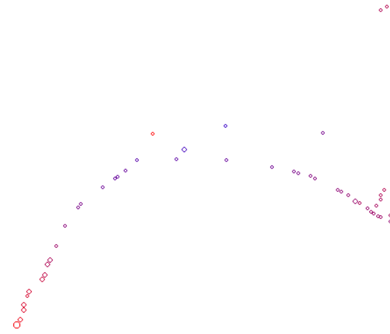


Figure 69 - set of final seeds for OP_6, sample rate 2

Figure 70 - winner set, sample rate 1, 1 pixel threshold

Figure 71- winner set, sample rate 2, 1 pixel threshold

Figure 72- winner set, sample rate 1, 4 pixel threshold

Figure 73- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_6 image stack contains 101 slices, all of which were read for sample rate 1 and 50 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 2.2949, with a standard deviation of 4.2640 and signal to noise ratio of 0.5382. The mean calculated for sample rate 2 was 2.1255, with a standard deviation of 4.2640and signal to noise ratio of 0.4985.

Seeding Results:

The MIP for sample rate 1 contained 2,422 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 104. Seeding the thresholded version of the MIP yielded 131 potential seeds, with an average radius of 3.4885 pixels and an average intensity of 101.5115. Following the slice assignment process, the seed count was increased to 138 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 76 Coord objects.

The MIP for sample rate 2 contained 2,171 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 104. Seeding the thresholded version of the MIP yielded 109 potential seeds, with an average radius of 3.5505 pixels and an average intensity of 97.2752. Following the slice assignment process, the seed count was increased to 110 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 49 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 0 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 0 such Coords.

Accuracy to SWC Seeds:

Of the 76 seeds submitted for validation by sample rate 1 against the ‘gold standard’ SWC reconstruction 5 were confirmed valid and 71 rejected at the sub-pixel threshold of 1, with an average radius difference of 2.41 pixels. 22 were validated and 54 rejected at the threshold of 2 pixels, with an average radius difference of 2.05. 46 were validated and 30 rejected at the threshold level of 3 pixels, with an average radius difference of 2.13 pixels. 55 were validated and 21 rejected at the threshold level of 4 pixels, with an average radius difference of 2.1 pixels.

Of the 49 seeds submitted for validation by sample rate 2 against the ‘gold standard’ SWC reconstruction 0 were confirmed valid and 49 rejected at the sub-pixel threshold of 1. 5 were validated and 44 rejected at the threshold of 2 pixels, with an average radius difference of 2.64. 17 were validated and 32 rejected at the threshold level of 3 pixels, with an average radius difference of 2.18 pixels. 32 were validated and 17 rejected at the threshold level of 4 pixels, with an average radius difference of 2.28 pixels.

OP_7 Results

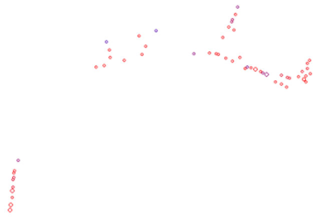


Figure 74 - set of final seeds for OP_7, sample rate 1

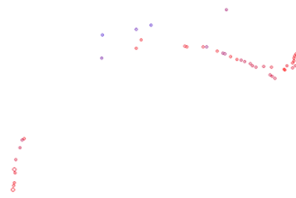


Figure 75 - set of final seeds for OP_7, sample rate 2

Figure 76 - winner set, sample rate 1, 1 pixel threshold

Figure 77- winner set, sample rate 2, 1 pixel threshold

Figure 78- winner set, sample rate 1, 4 pixel threshold

Figure 79- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_7 image stack contains 71 slices, all of which were read for sample rate 1 and 35 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 4.2639, with a standard deviation of 3.8329 and signal to noise ratio of 1.1124. The mean calculated for sample rate 2 was 3.6046, with a standard deviation of 3.8329 and signal to noise ratio of 0.9404.

Seeding Results:

The MIP for sample rate 1 contained 2,062 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 93. Seeding the thresholded version of the MIP yielded 110 potential seeds, with an average radius of 3.3455 pixels and an average intensity of 78.1545. Following the slice assignment process, the seed count was increased to 158 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 60 Coord objects.

The MIP for sample rate 2 contained 1,803 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 93. Seeding the thresholded version of the MIP yielded 103 potential seeds, with an average radius of 3.2136 pixels and an average intensity of 78.699. Following the slice assignment process, the seed count was increased to 113 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 47 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 0 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 0 such Coords.

Accuracy to SWC Seeds:

Of the 60 seeds submitted for validation by sample rate 1 against the ‘gold standard’ SWC reconstruction 1 was confirmed valid and 59 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.29 pixels. 7 were validated and 53 rejected at the threshold of 2 pixels, with an average radius difference of 1.32. 11 were validated and 49 rejected at the threshold level of 3 pixels, with an average radius difference of 1.44 pixels. 12 were validated and 48 rejected at the threshold level of 4 pixels, with an average radius difference of 1.38 pixels.

Of the 47 seeds submitted for validation by sample rate 2 against the ‘gold standard’ SWC reconstruction 2 were confirmed valid and 45 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.48 pixels. 13 were validated and 34 rejected at the threshold of 2 pixels, with an average radius difference of 1.41. 23 were validated and 24 rejected at the threshold level of 3 pixels, with an average radius difference of 1.28 pixels. 27 were validated and 20 rejected at the threshold level of 4 pixels, with an average radius difference of 1.38 pixels.

OP_8 Results

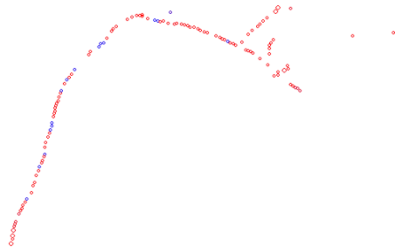


Figure 80 - set of final seeds for OP_8, sample rate 1

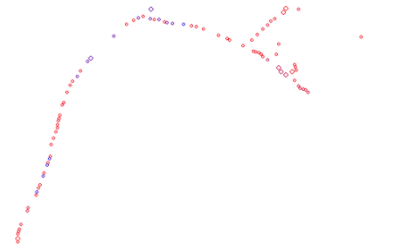


Figure 81 - set of final seeds for OP_8, sample rate 2

Figure 82 - winner set, sample rate 1, 1 pixel threshold

Figure 83- winner set, sample rate 2, 1 pixel threshold

Figure 84- winner set, sample rate 1, 4 pixel threshold

Figure 85- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_8 image stack contains 85 slices, all of which were read for sample rate 1 and 42 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 7.4130, with a standard deviation of 4.0213 and signal to noise ratio of 1.8434. The mean calculated for sample rate 2 was 6.5487, with a standard deviation of 4.0213 and signal to noise ratio of 1.6285.

Seeding Results:

The MIP for sample rate 1 contained 2,945 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 83. Seeding the thresholded version of the MIP yielded 168 potential seeds, with an average radius of 3.3095 pixels and an average intensity of 100.6905. Following the slice assignment process, the seed count was increased to 238 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 130 Coord objects.

The MIP for sample rate 2 contained 2,786 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 83. Seeding the thresholded version of the MIP yielded 159 potential seeds, with an average radius of 3.327 pixels and an average intensity of 95.1824. Following the slice assignment process, the seed count was increased to 216 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 93 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 1 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 0 such Coords.

Accuracy to SWC Seeds:

Of the 130 seeds submitted for validation by sample rate 1 against the 'gold standard' SWC reconstruction 1 was confirmed valid and 129 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.29 pixels. 6 were validated and 124 rejected at the threshold of 2 pixels, with an average radius difference of 1.75. 6 were validated and 124 rejected at the threshold level of 3 pixels, with an average radius difference of 1.75 pixels. 6 were validated and 124 rejected at the threshold level of 4 pixels, with an average radius difference of 1.72 pixels.

Of the 93 seeds submitted for validation by sample rate 2 against the 'gold standard' SWC reconstruction 3 were confirmed valid and 90 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.83 pixels. 11 were validated and 82 rejected at the threshold of 2 pixels, with an average radius difference of 1.88. 18 were validated and 75 rejected at the threshold level of 3 pixels, with an average radius difference of 1.86 pixels. 19 were validated and 74 rejected at the threshold level of 4 pixels, with an average radius difference of 1.93 pixels.

OP_9 Results

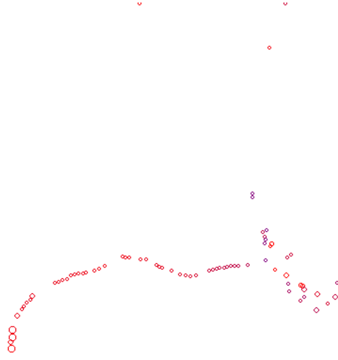


Figure 86 - set of final seeds for OP_9, sample rate 1

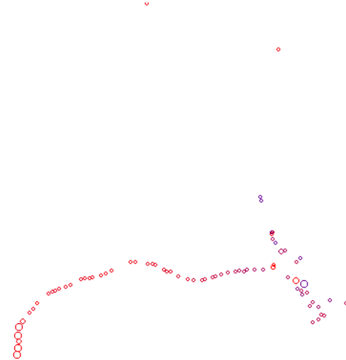


Figure 87 - set of final seeds for OP_9, sample rate 2

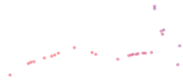


Figure 88 - winner set, sample rate 1, 1 pixel threshold

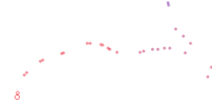


Figure 89- winner set, sample rate 2, 1 pixel threshold

Figure 90- winner set, sample rate 1, 4 pixel threshold

Figure 91- winner set, sample rate 2, 4 pixel threshold

Stack Statistics:

The OP_9 image stack contains 92 slices, all of which were read for sample rate 1 and 46 of which were read for sample rate 2. The image mean calculated for sample rate 1 was 3.1214, with a standard deviation of 4.4558 and signal to noise ratio of 0,7005. The mean calculated for sample rate 2 was 2.8447, with a standard deviation of 4.4558 and signal to noise ratio of 0.6384.

Seeding Results:

The MIP for sample rate 1 contained 3,174 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 104. Seeding the thresholded version of the MIP yielded 144 potential seeds, with an average radius of 3.6528 pixels and an average intensity of 100.7292. Following the slice assignment process, the seed count was increased to 177 seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 79 Coord objects.

The MIP for sample rate 2 contained 2,914 pixels of intensity greater than zero. From this pool of potential pixels, the segmentation threshold was set at a value of 104. Seeding the thresholded version of the MIP yielded 150 potential seeds, with an average radius of 3.4933 pixels and an average intensity of 98.0533. Following the slice assignment process, the seed count was increased to 181seeds. After performing the refinement and fragment assignment/selection of the seeds, the final seed set consisted of 78 Coord objects.

Image Ambiguity Alerts:

The sample rate of 1 produced a set of 0 Coords that represented areas of the neuron structure that had ambiguous or complex morphologies. The sample rate of 2 produced a set of 0 such Coords.

Accuracy to SWC Seeds:

Of the 79 seeds submitted for validation by sample rate 1 against the 'gold standard' SWC reconstruction 27 were confirmed valid and 52 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.38 pixels. 53 were validated and 26 rejected at the threshold of 2 pixels, with an average radius difference of 1.52. 59 were validated and 20 rejected at the threshold level of 3 pixels, with an average radius difference of 1.63 pixels. 65 were validated and 14 rejected at the threshold level of 4 pixels, with an average radius difference of 1.75 pixels.

Of the 78 seeds submitted for validation by sample rate 2 against the 'gold standard' SWC reconstruction 29 were confirmed valid and 49 rejected at the sub-pixel threshold of 1, with an average radius difference of 1.65 pixels. 59 were validated and 19 rejected at the threshold of 2 pixels, with an average radius difference of 1.51. 66 were validated and 12 rejected at the threshold level of 3 pixels, with an average radius difference of 1.75 pixels. 67 were validated and 11 rejected at the threshold level of 4 pixels, with an average radius difference of 1.73 pixels.

The numerical results for the image statistics, seeding results and image ambiguity results of the nine different stacks at each sampling rate are presented in tables 3, 4 and 5. The validation results comparing the final seed set to the 'gold-standard' SWC file are summarized in tables 6 and 7.

Stack	Sample Rate	Mean	High/Low	Std. Dev.	S:N	Threshold
OP_1	1	6.8529	254/0	5.6688	1.2089	117
	2	6.4319	254/0	5.6688	1.1346	117
OP_2	1	11.1259	254/0	3.0853	3.6061	61
	2	9.9191	254/0	3.0853	3.2149	61
OP_3	1	6.5759	254/0	1.9066	3.4489	33
	2	5.8019	254/0	1.9066	3.0430	33
OP_4	1	5.9826	254/0	7.7741	0.7696	110
	2	5.7028	254/0	7.7741	0.7336	110
OP_5	1	5.0631	254/0	1.8429	2.7473	41
	2	4.3984	254/0	1.8429	2.3866	41
OP_6	1	2.2949	254/0	4.2640	0.5382	104
	2	2.1255	254/0	4.2640	0.4985	104
OP_7	1	4.2639	254/0	3.8329	1.1124	93
	2	3.6046	254/0	3.8329	0.9404	93
OP_8	1	7.4130	254/0	4.0213	1.8434	83
	2	6.5487	254/0	4.0213	1.6285	83
OP_9	1	3.1214	254/0	4.4558	0.7005	104
	2	2.8447	254/0	4.4558	0.6384	104

Table 3 - image statistics results

Stack	Sample Rate	Pixels	Pot. Seeds	Avg. Radius	Avg. Intensity	Slice Candidates
OP_1	1	6930	257	4.1206	133.1595	282
	2	6519	249	4.0763	129.8715	260
OP_2	1	15601	603	3.7529	96.2637	683
	2	13867	552	3.7101	94.1558	603
OP_3	1	4034	95	4.6000	76.6211	99
	2	3662	100	4.3600	74.2900	103
OP_4	1	6264	178	4.5506	144.5169	200
	2	5989	195	4.3231	141.9077	213
OP_5	1	2032	104	3.4038	67.2788	108
	2	1792	98	3.3878	63.3776	99
OP_6	1	2422	131	3.4885	101.5115	138
	2	2171	109	3.5505	97.2752	110
OP_7	1	2062	110	3.3455	78.1545	158
	2	1803	103	3.2136	78.6990	113
OP_8	1	2945	168	3.3095	100.6905	238
	2	2786	159	3.3270	95.1824	216
OP_9	1	3174	144	3.6528	100.7292	177
	2	2914	150	3.4933	98.0533	181

Table 4 - seeding results

Stack	Sample Rate	Areas of Ambiguity
OP_1	1	2
	2	4
OP_2	1	9
	2	8
OP_3	1	0
	2	0
OP_4	1	2
	2	2
OP_5	1	0
	2	0
OP_6	1	0
	2	0
OP_7	1	0
	2	0
OP_8	1	1
	2	0
OP_9	1	0
	2	1

Table 5 - image ambiguity alert results

Stack	1 px Validation Hits / Misses	2 px Validation Hits / Misses	3 px Validation Hits / Misses	4 px Validation Hits / Misses
OP_1	88/80	135/33	140/28	142/26
OP_2	5/325	27/303	57/273	79/251
OP_3	3/50	16/37	25/28	32/21
OP_4	52/64	73/43	87/29	94/22
OP_5	1/45	7/39	24/22	28/18
OP_6	5/71	22/54	46/30	55/21
OP_7	1/59	7/53	11/49	12/48
OP_8	1/129	6/124	6/124	6/124
OP_9	27/52	53/26	59/20	65/14

Table 6 - SWC validation hits/misses for sample rate 1

Stack	1 px Validation Hits / Misses	2 px Validation Hits / Misses	3 px Validation Hits / Misses	4 px Validation Hits / Misses
OP_1	28/120	98/50	127/21	133/15
OP_2	0/275	10/265	36/239	64/211
OP_3	2/48	15/35	30/20	34/16
OP_4	36/84	88/32	102/18	110/10
OP_5	1/39	5/35	14/26	22/18
OP_6	0/49	5/44	17/32	32/17
OP_7	2/45	13/34	23/24	27/20
OP_8	3/90	11/82	18/75	19/74
OP_9	29/49	59/19	66/12	67/11

Table 7 - SWC validation hits/misses for sample rate 2

Figures 92 through 100 show charts of the image mean, standard deviation, signal to noise ratio, threshold level, non-zero pixel count, seeding results, average seed radius and intensity, image ambiguity alerts.

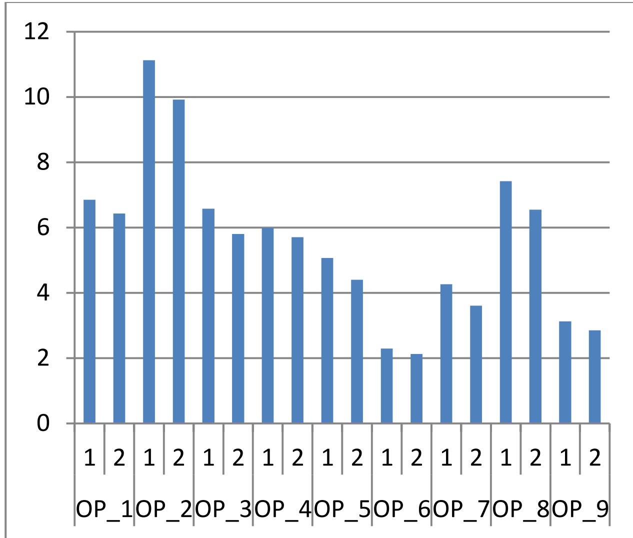


Figure 92 - image mean for the stacks

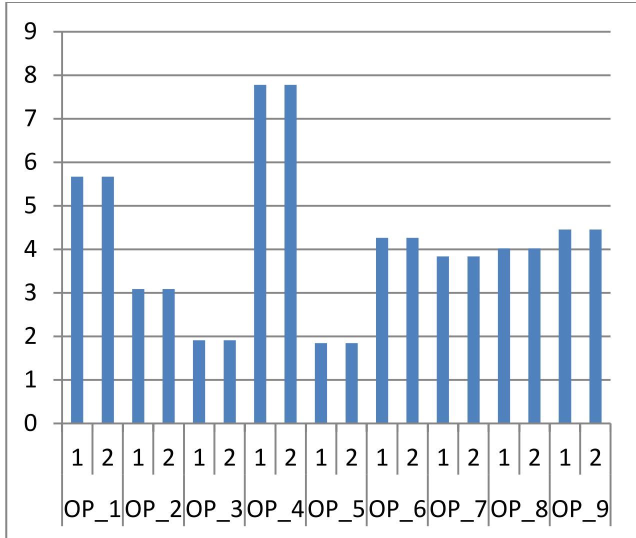


Figure 93 - standard deviation for the stacks

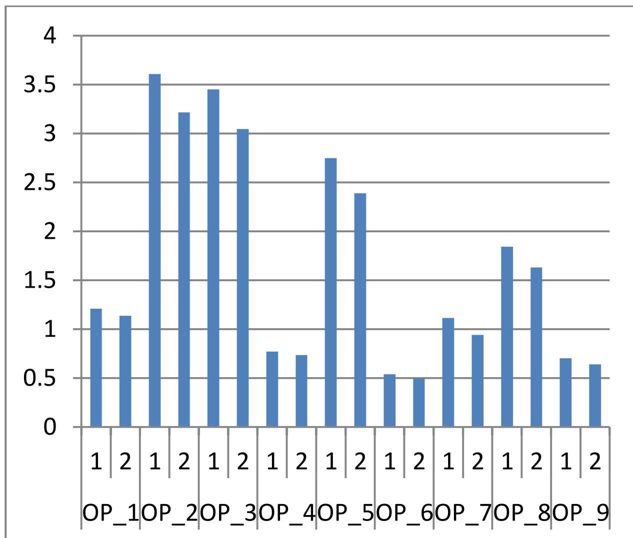


Figure 94 - signal to noise ratio for the stacks

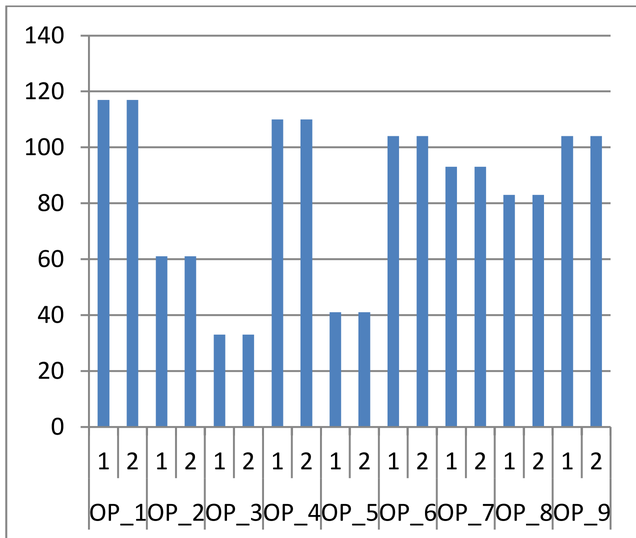


Figure 95 - threshold levels for the stacks

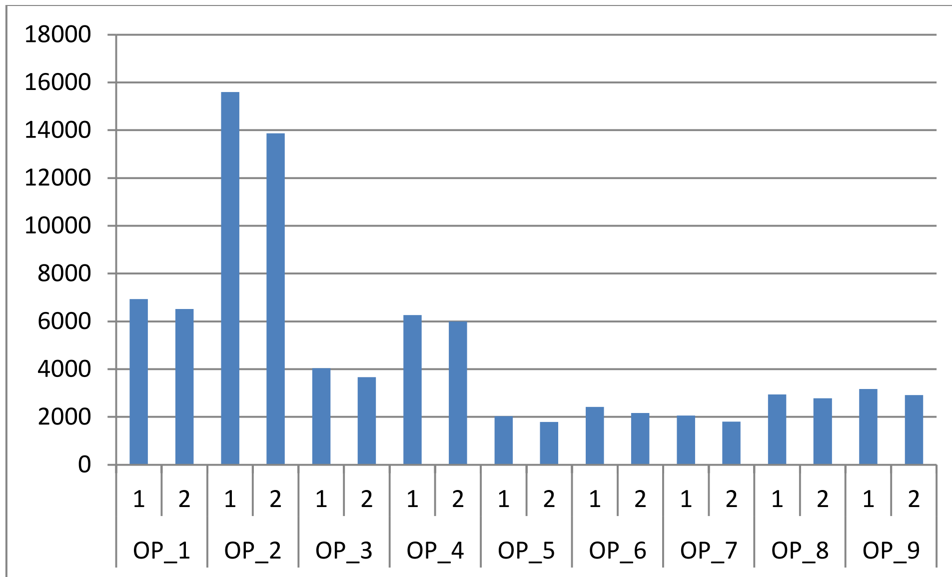


Figure 96 - number of pixels in the ray projection that were found to have a intensity value greater than 0

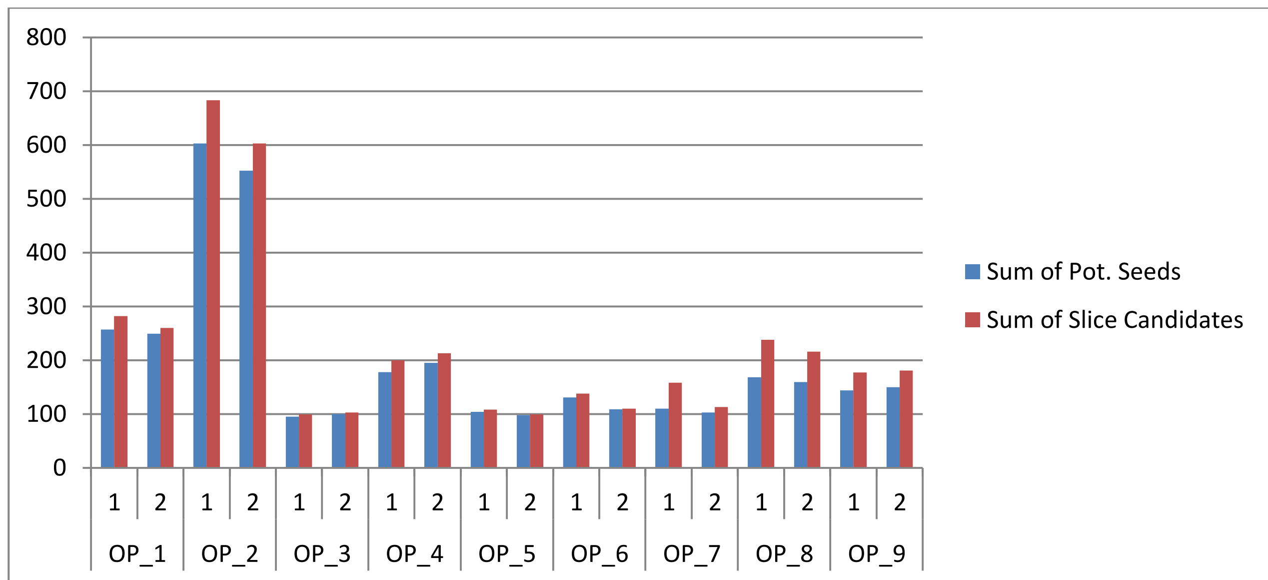


Figure 97 - number of seeds found during prelininary seed discovery and resulting from additions during slice assignment

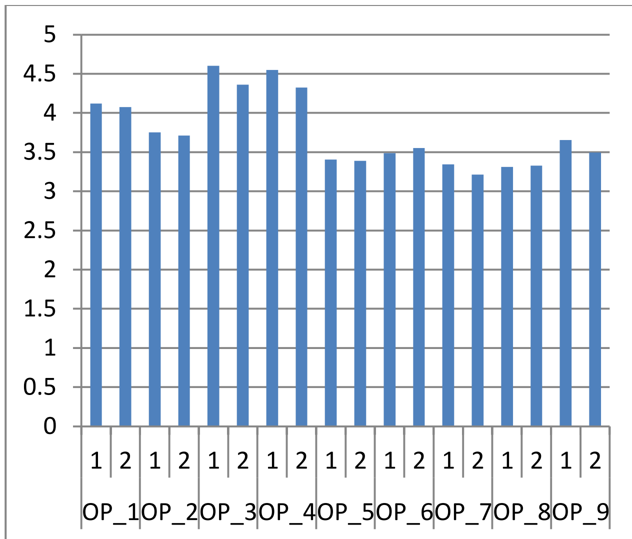


Figure 98 - average seed radius

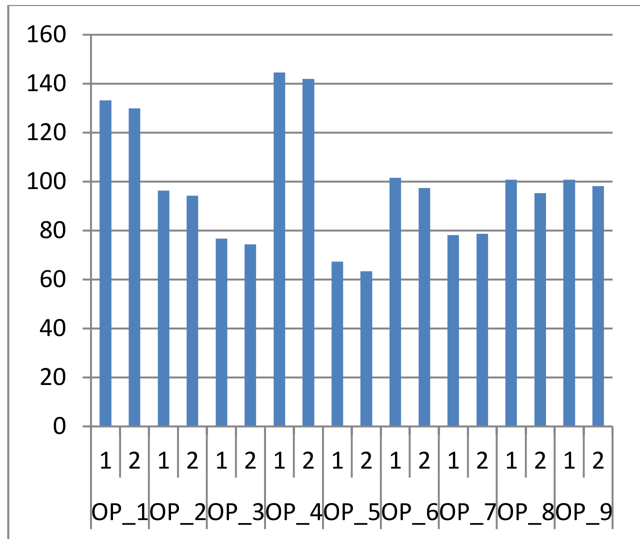


Figure 99 - average seed intensity

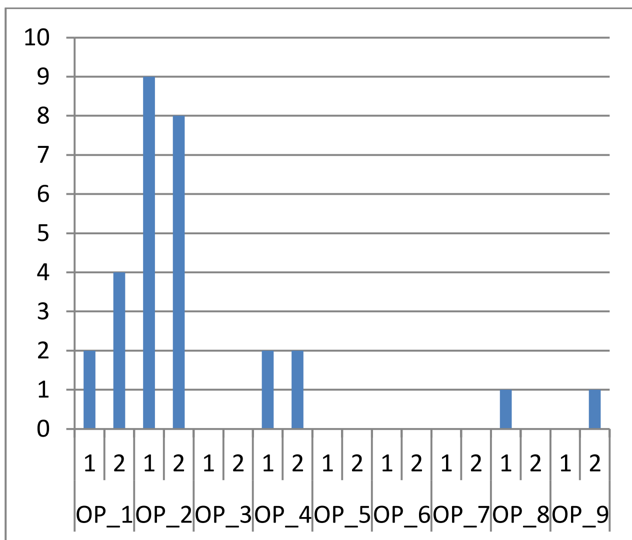


Figure 100 - number of areas identified as being ambiguos and in need of extra refinement

Table 8 shows the hit rate as a percent of the total number of seeds submitted for validation for each threshold and sample rate per image stack. Figures 102 through 109 chart the hit rate percentages of each sampling rate in a comparative fashion. Figure 110 shows the average difference in radius from the matching SWC points for the winner set of seeds.

Stack	Threshold	Sample Rate 1	Sample Rate 2
OP_1	1	52%	18%
	2	80%	64%
	3	83%	83%
	4	85%	85%
OP_2	1	3%	0%
	2	8%	4%
	3	17%	13%
	4	24%	23%
OP_3	1	6%	4%
	2	30%	30%
	3	47%	60%
	4	60%	68%
OP_4	1	45%	30%
	2	62%	73%
	3	75%	85%
	4	81%	91%
OP_5	1	2%	3%
	2	15%	13%
	3	52%	35%
	4	61%	55%
OP_6	1	7%	0%
	2	29%	10%
	3	61%	35%
	4	72%	65%
OP_7	1	2%	4%
	2	12%	28%
	3	18%	49%
	4	20%	57%
OP_8	1	1%	3%
	2	5%	12%
	3	5%	19%
	4	5%	20%
OP_9	1	34%	37%
	2	67%	76%
	3	75%	85%
	4	82%	86%

Table 8 - hit percentages for each stack, sampling rate and error margin

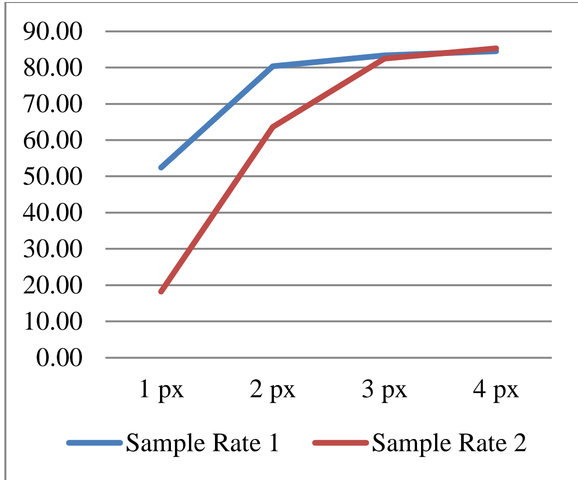


Figure 101 - hit percentage for OP_1

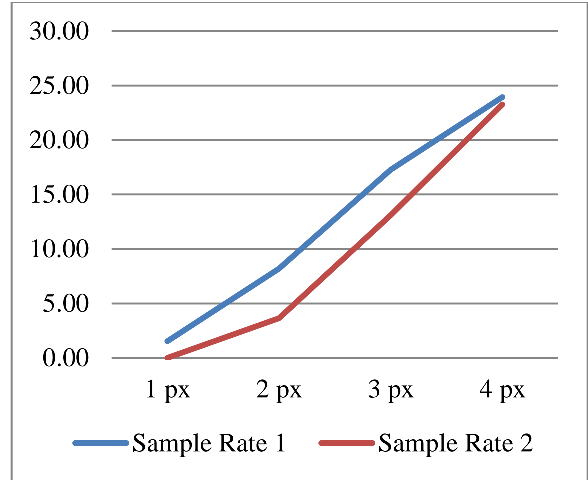


Figure 102 - hit percentage for OP_2

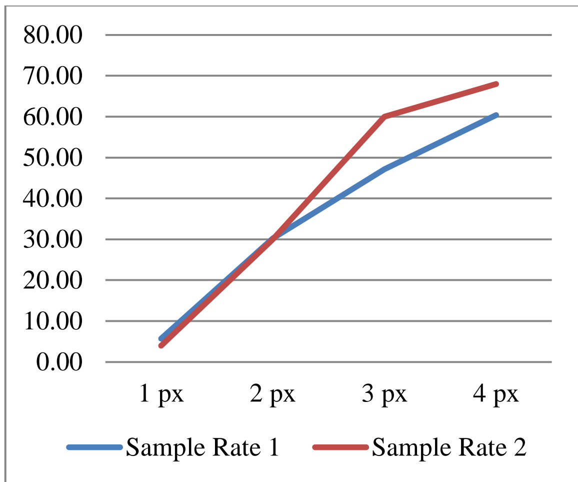


Figure 103 - hit percentage for OP_3

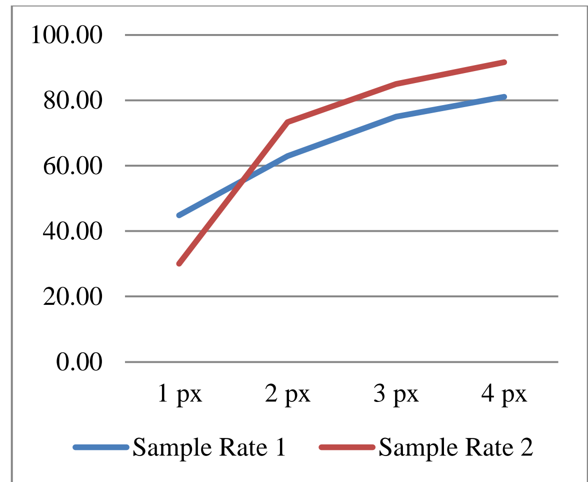


Figure 104 - hit percentage for OP_4

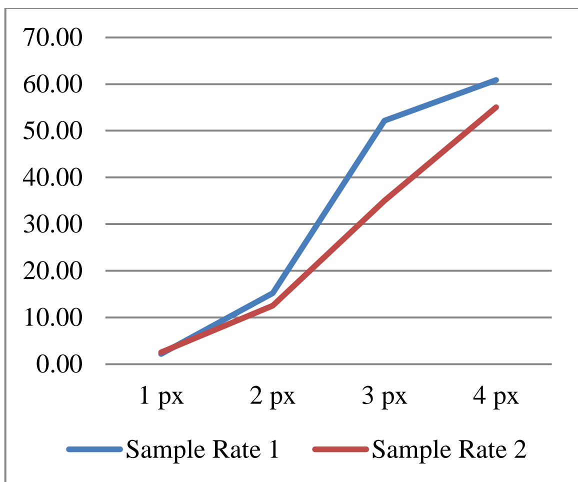


Figure 105 - hit percentage for OP_5

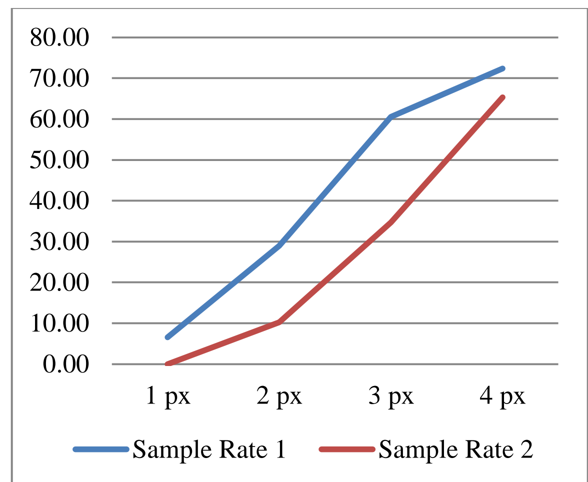


Figure 106 - hit percentage for OP_6

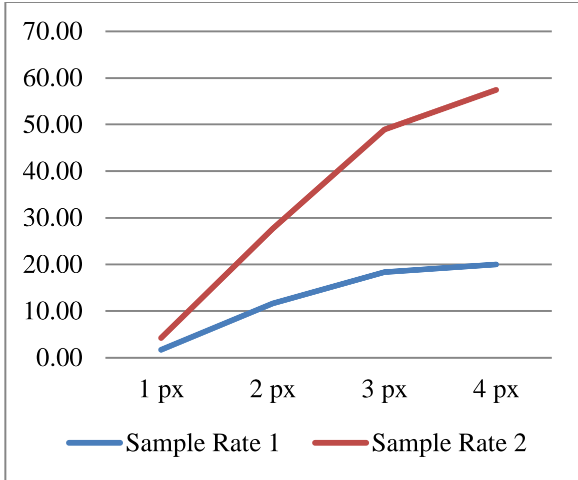


Figure 107- hit percentage for OP_7

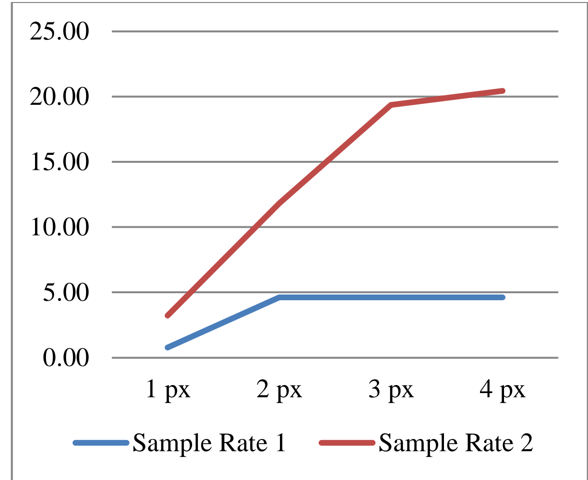


Figure 108 - hit percentage for OP_8

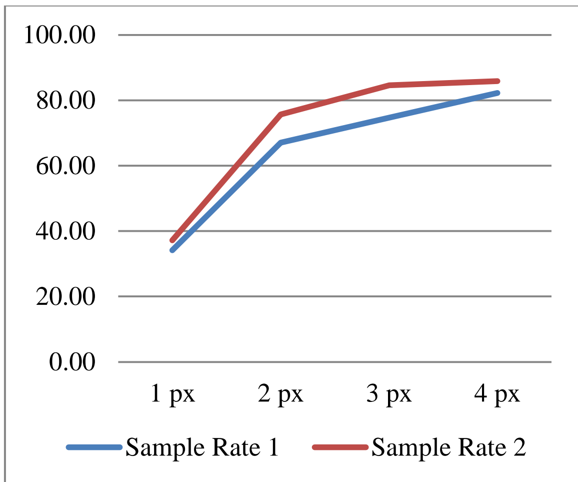


Figure 109 - hit percentage for OP_9

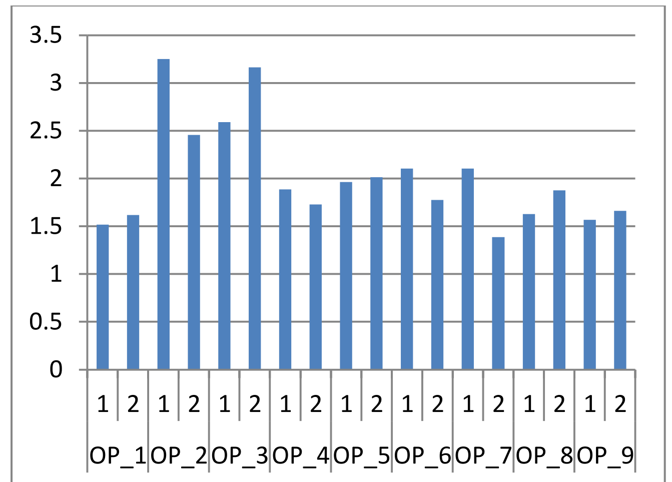


Figure 110 - average difference in seed radius from matching SWC points

Table 9 shows the running time for the program for each sampling rate in milliseconds. This time is calculated from the time the StackInspector begins to examine the images selected to the construction of the finalSeeds list of Coord points. It does not include the time needed to run the validation portion of the program.

The times shown in table 9 are the average times recorded for three consecutive runs of the same image stack with the same sampling rate. This was done to compensate for the use of system cache for the image data and to show the relative execution time of the disk reading and processing of the different stack image numbers.

Image Stack	Run Time Rate1 (ms)	Run Time Rate 2 (ms)
OP_1	2355	1442
OP_2	9283	5650
OP_3	1745	946
OP_4	2164	1644
OP_5	1753	959
OP_6	2337	1642
OP_7	9382	1436
OP_8	9220	5320
OP_9	9587	1869

Table 9 - average execution time over 3 runs per sampling rate

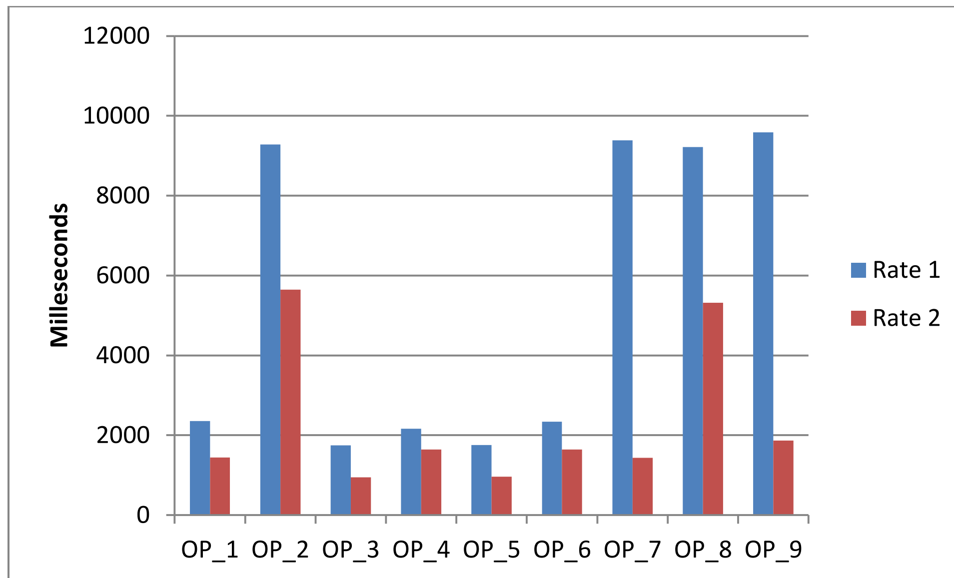


Figure 111 - execution times for the sample rates 1 and 2 for all stacks

5 Discussion

The process of validating my results may seem difficult; I could overlay an image of the collapsed image stack with my generated seed points and make a visual proof that all of my points do, in fact, coincide with neuron structures. However such a visual comparison of the result set is not useful when we remember that progression along the Z axis must also be estimated.

Figure 112 - visual validation of final seed set, over laid ray projection

Providing a program with X and Y coordinates of a seed point is not sufficient. We must also give a Z index value for the information to be of any use. While my system does provide such information as part of the result set, even comparing the points in 3-dimensional space next to a reconstructed neuron would be making use of imprecise, subjective visual inspection. In order to make sure that my seed point selection does provide a good estimation of where tracing algorithms may begin, I need to evaluate my points against a known set of points in 3-dimensional space.

Using the ‘gold-standard’ reconstructions provided by the DAIDEM competition with the data sets used in this paper, I can show exactly how close my discovered points come to points representing the real neuron in all three dimensions. The comparison of the results from the different sampling rates and error thresholds reveals if the sampling of half the stack images is in fact an efficient technique.

<u>Effects of sampling rates on the maximum intensity projections:</u>

Examining the results of the different sampling rates on the maximum intensity projections, we find little difference in the MIP images generated. This is because the loss of

information in the MIP images for the different sampling rates is negligible, with an average difference of only 440 pixels fewer in the sample rate 2 MIP across all nine image stacks. This represents an average loss of only 0.0021% of the information contained in the stack.

As seen in figures 12 and 13, the actual difference in bright pixel number and intensity is very small. Figure 14 shows the data that is lost in the change of sampling rates. Of important note is that the lost image information all seems to come from the boundary area of the neuron with the background, an area that is generally ambiguous at best due to the point spread effect. The information that remains is almost exclusively the high-probability regions of greatest bright-pixel concentrations.

The similarity between the MIPs generated by both sampling rates is further confirmed by an examination of the image statistics. The means for the different sampling rates do not vary highly, with the average variance being just 0.29. The standard deviation and threshold level calculated through the means-of-means approach are exactly the same between the two sample rates. The signal to noise ratio varies slightly, but only because it is calculated as the image mean divided by the standard deviation and varies as the image mean does.

The execution times for the sampling rates show a distinct advantage for the sample rate 2 approach. For the image selection and MIP generation section of the system, the sample rate 2 setting performed 43% faster than sample rate 1 on average.

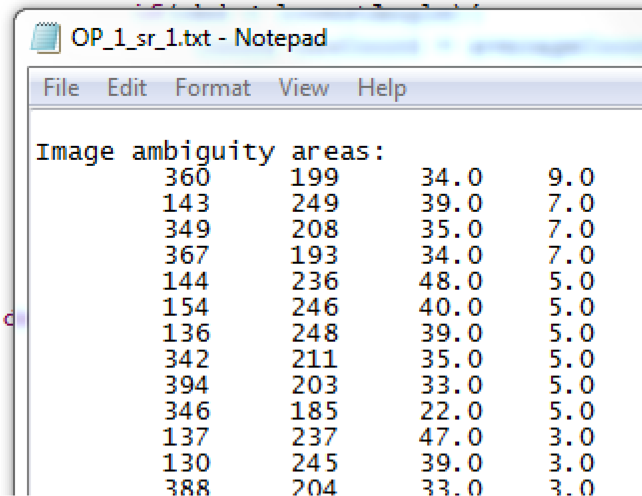
Effects of sampling rate on number of areas of ambiguity:

During the seed point creation, slice assignment and refinement, various areas of the neuron are discovered to have information that cannot be interpreted completely. These areas occur during proximal cross over events, areas of small discontinuity and where optical blurring of small features make the distinction between different areas of the neuron difficult. When such an area is discovered, the system logs the particular Coord object's attributes in the output file so that the subsequent tracing algorithm may approach this area with added analysis tool.

Stack	Sample Rate 1	Sample Rate 2
OP_1	2	4
OP_2	9	8
OP_3	0	0
OP_4	2	2
OP_5	0	0
OP_6	0	0
OP_7	0	0
OP_8	1	0
OP_9	0	1

Table 10 - areas of ambiguity identified by sample rate

Areas of ambiguity in this system are defined as Coord objects that, after the refinement phase, are still being shown to have more than 3 potential connecting Coords in their neighborhood.



OP_1_sr_1.txt - Notepad

File Edit Format View Help

Image ambiguity areas:

360	199	34.0	9.0
143	249	39.0	7.0
349	208	35.0	7.0
367	193	34.0	7.0
144	236	48.0	5.0
154	246	40.0	5.0
136	248	39.0	5.0
342	211	35.0	5.0
394	203	33.0	5.0
346	185	22.0	5.0
137	237	47.0	3.0
130	245	39.0	3.0
388	204	33.0	3.0

Figure 113 - example of the image ambiguity areas record in the output.txt file

When the points identified as being ambiguous are examined, we quickly find that most of the points described are far above the average seed size for the stack. This would mean that the associated Coord point from which this ambiguous area is being defined is very large compared to its average neighboring seed. The connection code, which establishes the number of connected seeds from which the ambiguity alert is determined, would very likely find this point qualifies as a connection for all seeds within the area of average size.

<u>Effects of sampling rates on seed generation in MIPs:</u>

The first observation one might make when examining the images for the final seed sets is that there exists little difference between the seeds created using the different sampling rates. Both seem to span the length of the neuron in equal measure and have roughly the same seed count (see figures 38 and 39, 50 and 51 for examples).

This seems to follow from the observation that the percentage of information lost during the sampling process represents a very small amount of the total stack information, and that it generally describes pixels occurring in less-intense border areas separating the neuron from the background. Such areas are very poor choices for seed points, so we would expect that their elimination should not have a large impact on the eventual seed pool generated.

However small this difference might be in terms of average information lost, it does seem to have some effect on the output seed listing. There are a number of reasons why the final seed set may vary despite there being such a small difference in the initial pixel pool before seeding begins.

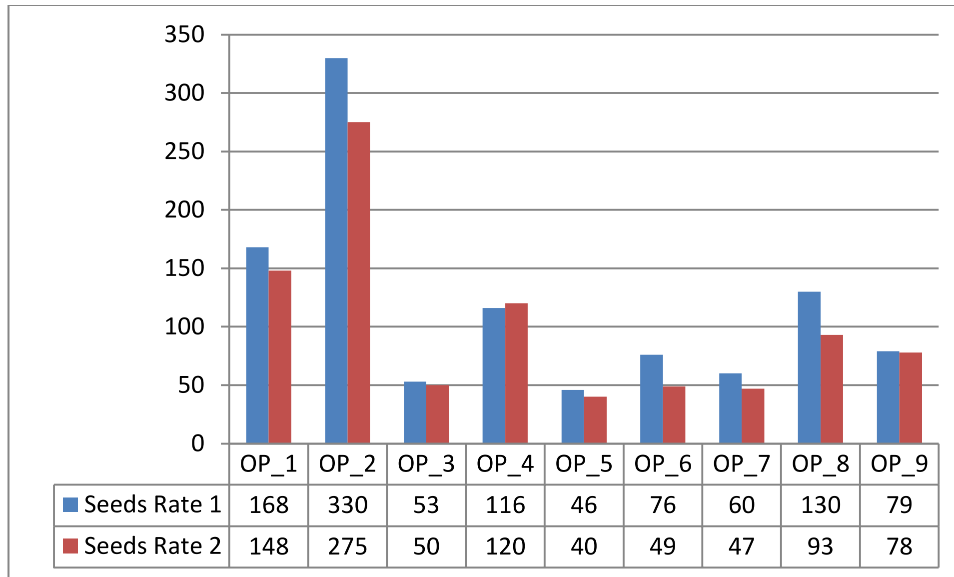


Figure 114 - seed number by sampling rate

Even though the raw number of pixels lost may be low, the seeding algorithm is keyed to the intensity values of the pixels present not the number. If the pixels lost during the sampling are of very high intensity, then the set of initial seed points for the algorithm may change dramatically.

For this situation to occur, the neuron must be moving very rapidly across the X/Y plane across 3 or more slices, such that the difference in local image intensity for the point of origin on the initial slice is reduced substantially by the next slice and almost zero by the third. This is analogous to having a local discontinuity introduced into the MIP by the sample rate. However comparisons of the MIPs for the image stacks do not suggest that such discontinuities are created with the sample rate 2 approach.

Depending on the tortuosity of the neuron, there may be a number of areas where angles on the curves extend over multiple slices and are more likely to spread the collective pixel volume over slices that are skipped during the 2 sampling rate. This would lead to additional pixels being added to the curves during the 1 sample rate. Since the seeding algorithm uses a detection kernel responding to empty pixels at the corners and top, bottom, left and right most pixels, variations in the size of the available pixels around curves may greatly change the resulting seed distribution. A larger seed growing from such a region may consume pixels that could be used to start new seeds had they not been consumed.

As might be expected from using half the available data and starting from a smaller pool of bright pixels, in all but one case the sample rate of 2 produced fewer seed points. Across all of the image stacks, the sample rate of 2 produced 85% of the total number of seeds produced using sample rate 1.

In the case of OP_4, the initial seeding algorithm created 17 more Coords in the sample rate 2 run, and the average radius of all seeds in the set was slightly smaller – 4.55 pixels for sample rate 1 and 4.32 for rate 2. Following slice assignment and refinement, the difference

shrank to 13. Examination of the final seed sets for the sample rates (figures 56 and 57) show that the main driving factor for this difference was the complex branching regions in the top-right of the MIP images.

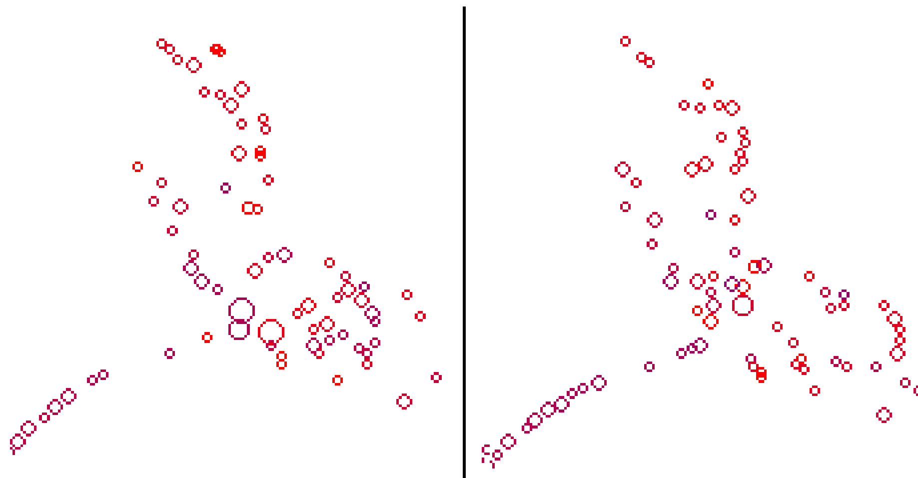


Figure 115 - detailed view of the complex region of OP_4

The large collection of smaller Coord objects grouped in this region suggests that the sample rate 2 approach, even while not introducing false discontinuities into the MIP, does occasionally create a tessellation of image intensity across an area during periods of rapid branching and slice movement of neuron structures. This spacing of groups of intensity clusters would account for the additional, smaller radius Coords found in sample rate 2, as again the seeding algorithm is keyed to pixel intensity. However the slice assignment and refinement steps can reduce this effect somewhat.

Effects of sampling rate on the accuracy of seeds derived from MIPs:

The images showing the winner set for the sampling rates at the 1 and 4 pixel thresholds also appear to be very similar. Even in cases where the winner set for threshold 1 does markedly differ, they seem to converge by the time threshold 4 is reached. Figures 102 through 110 show the difference between the sample rates in terms of hit percentages.

For three of the image stacks (OP_7, OP_8 and OP_9) the sample rate 2 approach produced a higher percentage of seeds that were accurate to the gold standard than the sample rate 1 approach across all four error margin settings. For stacks OP_5 and OP_6, the inverse is true and the sample rate 1 approach had higher percentages across all of the error margins. OP_2 had a lower percentage for sample rate 2 across 3 of the 4 margins, but had almost equal performance at the 4 pixel level.

For stacks OP_1, OP_3 and OP_4 there is a change in the best performing sampling rate at a particular error margin. Sampling rate 2 becomes more accurate than sample rate 1 when we allow our seeds to be up to 2 pixels away from optimum in OP_3 and OP_4. This occurs in OP_1 at the 3 pixel error margin.

It should be stated again that the hit percentages reported here are against a full manual reconstruction of the neuron and are a ‘gold-standard’ set of points that fully describe the cells’ structure. My produced seeds are meant to be used as markers for a subsequent program to use as a basis for producing such a tracing in an automated fashion. Given that my average seed radius does not go far beyond 4 pixels for any stack, it can be said that even at the 4 pixel error margin my seeds still on average contain the true centerline of the neuron.

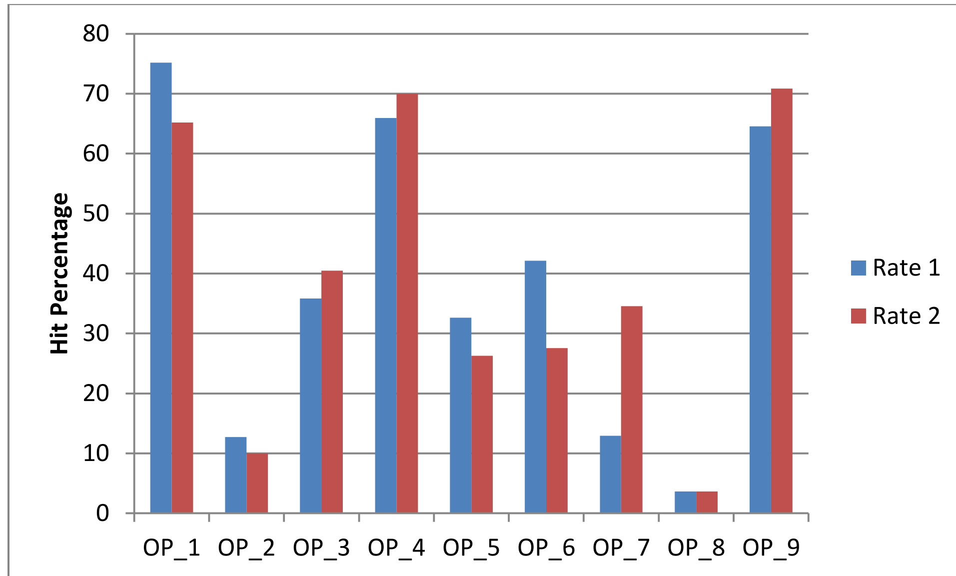


Figure 116 – average hit percentages of the sample rates

As shown in figure 115, the accuracy for seeds derived from MIP images constructed using a sampling rate of 2 are not significantly less accurate to gold standard manually selected points than when all of the stack images are used. On average at the 4 pixel error margin, the sample rate 1 approach had an accuracy of 53% whereas the sample rate 2 approach achieved an accuracy rate of 59%.

6 Conclusion

In this paper I have suggested that the discovery of seed points needed for use in automated local neuron tracing systems – currently mostly supplied through manual interaction with the image stack and program – may be automatically performed from the maximum intensity projection of an image stack with a high degree of accuracy. I have also proposed that the MIP used in the seed generation may be constructed with half of the total images in the stack without a significant loss of accuracy among the seed points generated, while saving processing time by requiring far less disk reads and operations/memory space.

To demonstrate the validity of the first hypothesis, I constructed a system to select seed points from a MIP image. Using this system, I constructed a series of seeds from a set of nine image stacks used in a recent competition on neuron morphology reconstruction. I then validated the seed points generated by my system against the 'gold standard' reconstruction files distributed with the image stacks. The results of this comparison show that seeds generated from MIP images coincide with manually selected proper node placement over 53% of the time with an error margin of 4 pixels.

To see if the number of images used in the MIP affected the resulting seed set accuracy rating, I then repeated the experiment utilizing half of the available slices in the stack. The resulting seed set was slightly more accurate to the 'gold standard' reconstruction points on average, scoring at 59% matching seeds at the 4 pixel margin. When the specific error margin was relaxed to being 4 pixels from the true centerline, the resulting accuracy of the sample 2 approach was just as good across all image stacks as the MIP generated from all of the slices in the stack.

The sample rate 2 approach also proved to be 43% faster in constructing the MIP ray projection images. This time savings is significant when considering that neuron image stacks vary greatly in slice count and the ability to scale well with changes in the number of images in a stack is a critical requirement of rapid tracing algorithms.

<u>How this work may contribute to solutions to the problem:</u>

The seed points generated from this approach are potentially of great use for the class of tracing algorithms defined as local search methods. As described above, these algorithms require the input of a series of user-defined starting seed points from which their tracing algorithm identifies local image characteristics, using these characteristics to define directional and size estimates for the neuron structure.

The current requirement of researcher interaction with the image stack to determine these seed points represents a major bottleneck in the speed of local tracing approaches. This automated approach, which is capable of producing seeds of almost identical position, can greatly accelerate the process by removing the need for user interaction at any stage of the pre-processing chain.

Beyond representing merely exact coordinates for tracing initiation, the seeds generated by this system have been shown to reflect a global representation of the neuron cell. This information can be of value to tracing algorithms in establishing bounding restrictions in the search space, dealing with local discontinuities introduced during cell staining and generating heuristic values to guide the search performed from a local pixel/voxel neighborhood.

The list of areas of ambiguity in the image – along with the various statistics collected during execution – represent a useful set of markers, parameters and global image information that a local search technique may utilize in performing a truly automated tracing of a neuron. Knowing which areas of the neuron are morphologically simple and which are complex could allow a tracing algorithm to alternate between low-complexity, extremely fast approaches and more intricate, time consuming approaches with maximum efficiency.

How this approach may be improved in future works:

There are a number of additional techniques and approaches that may be applied to this system that would increase its speed and/or accuracy but were beyond the scope of my main hypothesis of seed generation from MIP images and stack sampling rates. Some of the additional techniques that may be applied are noise reduction systems, improved seeding approaches and additional connectivity refinement and classification.

Although it was shown that the use of the sample rate 2 approach did in fact reduce the overall image noise present in the MIP image used for seeding, there was not a complete elimination of all image noise. Highly noisy stacks, such as OP_8, could still potentially create a large number of false-positive seed points. Further reduction of image noise before seeding would be a desirable goal.

Many tracing algorithms employ a type of blurring to the slice images before processing them in order to remove noise artifacts. If such an approach were to be used in my system, the resulting reduction in noise would help reduce the number of false positive seeds and increase the accuracy of the generated points. The use of the MIP becomes advantageous again in this context, as the convolution process would only be required of a single image in place of the repetitive and costly convolution of all of the stack images, providing a more efficient method of achieving image noise reduction.

The seed selection algorithm presented here is very basic and designed for simplicity and an over-estimation of the neuron in terms of seed number. There exist other approaches to skeletonization, radius estimation and edge detection that may prove to be more efficient or accurate at producing the seed points. Once the MIP has been generated, the resulting image may be processed by one or more of these techniques, which are efficient for 2-dimensional images, and the results either compared or combined to achieve a more accurate set of seeds.

Finally the seed connection and classification system may be expanded to provide more relevant information to the subsequent tracing algorithm. Currently the system identifies only those areas where the connections between the seeds are said to be ambiguous. Providing the tracing algorithm with a comprehensive list of terminal seeds, suspected branching points and

basic estimations of the 3-dimensional connectivity angles of each point in addition to this information may allow the tracing system to be far more efficient.

The reporting of fragment statistics might also be of great help in the interpretation of the seed set and its application to the image stack. Knowing the member count, neuron volume and 3-dimensional movement path of the members through the fragment could provide a local estimation of the neuron topology in a concise format that local tracing algorithms might exploit. Building on the fragment objects to provide inter-fragment connectivity might extend this topology estimation to a global perspective.

This collective set of data could drive the adaptive techniques described above, allowing tracing algorithms to switch between approaches on the fly to most efficiently achieve their reconstructions.

Works Cited

[1] H. Peng, F. Long, T. Zhao and E. Myers, "Proof-Editing is the Bottleneck of 3D Neuron Reconstrcution: The Problem and Solutions," *Neuroinformatics,* 2011.

[2] E. Hoffman, S. Huang and M. Phelps, "Quantitation in positron emission computed tomography: 1. Effect of object size.," *Journal of Computer Assisted Tomography,* vol. 3, no. 3, pp. 299-308, 1979.

[3] C. Blum and M. Blesa, "New metaheuristic approaches for the edge-weighted k-cardinality tree problem," *Computers & Operations Research,* no. 32, pp. 1355-1377, 2005.

[4] S. Senft, "A Brief History of Neuronal Reconstruction," *Neuroinform,* vol. 9, pp. 119-128, 2011.

[5] E. Meijering, "Neuron Traicing in Perspective," *Cytometry,* vol. 77A, pp. 693-704, 2010.

[6] X. He, E. Kischell, M. Rioult and T. J. Holmes, "Three-Dimensional Thinning Algorithm that Peels the Outermost Layer with Application to Neuron Tracing," *Journal of Computer-Assisted Microscopy,* vol. 10, no. 3, pp. 123-135, 1998.

[7] Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, S. Yoshida, T. Koller, G. Gerig and R. Kikinis, "Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images," *Medical Image Analysis,* vol. 2, no. 2, pp. 143-168, 1998.

[8] K. Krissian, G. Malandain and N. Ayache, "Model-Based Detection of Tubular Structures in 3D Images," *Computer Vision and Image Understanding,* vol. 80, pp. 130-171, 2000.

[9] S. R. Aylward and E. Bullitt, "Intialization, Noise, Singularities, and Scale in Height Ridge Transversal for Tubular Object Centerline Extraction," *IEEE Transactions on Medical Imaging,* vol. 21, no. 2, pp. 61-75, 2002.

[10] S. M. Pizer, D. Eberly and D. S. Fritsch, "Zoom-Invarient Vision of Figural Shape: The Mathematics of Cores," *Computer Vision and Image Understanding,* vol. 69, no. 1, pp. 55-71, 1998.

[11] P. Chothani, V. Mehta and A. Stepanyants, "Automated Tracing of Neurites from Light Microscopy Stacks of Images," *Neuroinform,* vol. 9, pp. 263-278, 2011.

[12] E. Turetken, G. Gonzalez, C. Blum and P. Fua, "Automated Reconstruction of Dentritic and Axonal Trees by Global Optimization with Geometric Priors," *Neuroinform,* vol. 9, pp. 279-302, 2011.

[13] A. Narayanaswamy, Y. Wang and B. Roysam, "3-D Image Pre-processing Algorithms for Improved Automated Tracing of Neuronal Arbors," *Neuroinform,* vol. 9, pp. 219-231, 2001.

[14] H. Peng, F. Long and G. Myers, "Automatic 3D neuron tracing using all-paths pruning," *Bioinformatics,* vol. 27, pp. 239-247, 2011.

[15] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics,* vol. 21, no. 4, pp. 163-169, 1987.

[16] M. Kass, A. Witkin and D. Terzopoulos, "Snakes: Active Contour Models," *International Journal of Computer Vision,* pp. 321-331, 1988.

[17] R. Kimmel and N. Kiryati, "Skeletonization via Distance Maps and Level Sets," *Computer Vision and Image Understanding,* vol. 62, no. 3, pp. 382-391, 1995.

[18] Y. Zhou and A. W. Toga, "Efficient Skeletonization of Volumetric Objects," *IEEE Trandactions on Visualization and Computer Graphics,* vol. 5, no. 3, pp. 196-209, 1999.

[19] K. A. Al-Kofahi, A. Can, S. Lasek, D. H. Szarowski, N. Dowell-Mesfin, W. Shain, J. N. Turner and B. Roysam, "Median-Based Robust Algorithms for Tracing Neurons From Noisy Confocal Microscope Images," *IEEE Transactions on Information Technology in Biomedicine,* vol. 7, no. 4, pp. 302-317, 2003.

[20] A. Rodriguez, D. Ehlenberger, K. Kelliher, M. Einstein, S. C. Henderson, J. H. Morrison, P. R. Hof and S. L. Wearne, "Automated reconstruction of three-dimensional neuronal morphology from laser scanning microscopy images," *Methods,* vol. 30, pp. 94-105, 2003.

[21] K. A. Al-Kofahi, S. Lasek, D. H. Szarowski, C. J. Pace, G. Nagy, J. N. Turner and B. Roysam, "Rapid Automated Three-Dimensional Tracing of Neurons From Confocal Image Stacks," *IEEE Transactions on Information Technology in Biomedicine,* vol. 6, no. 2, pp. 171-187, 2002.

[22] S. Schmitt, J. F. Evers, C. Duch, M. Scholz and K. Obermayer, "New methods for the computer-assisted 3-D reconstruction of neurons from confocal image stacks," *NeuroImage,* vol. 23, pp. 1283-1298, 2004.

[23] R. Srinivasan, X. Zhou, E. Miller, J. Lu, J. Litchman and S. T. C. Wong, "Automated Axon Tracking of 3D Confocal Laser Scanning Microscopy Images Using Guided Probabilistic Region Merging," *Neuroinform,* vol. 9, pp. 189-203, 2007.

[24] T. Zhao, J. Xie, F. Amat, N. Clack, P. Ahammad, H. Peng, F. Long and E. Myers, "Automated Reconstruction of Neuronal Morphology Based on Local Geometrical and Global Structural Models," *Neuroinform,* vol. 9, pp. 247-261, 2011.

[25] N. Flasque, M. Desvignes, J.-M. Constans and M. Revenu, "Acquisition, segmentation and tracking of the cerebral vascular tree on 3D magnetic resonance angiography images," *Medical Image Analysis,* vol. 5, pp. 173-183, 2001.

[26] Y. Wang, A. Narayanaswamy, C.-L. Tsai and B. Roysam, "A Broadly Applicable 3-D Neuron Tracing Method Based on Open-Curve Snakes," *Neuroinform,* vol. 9, pp. 193-217, 2011.

[27] E. Bas and D. Erdogmus, "Principle Curves as Skeletons of Tubular Objects," *Neuroinform,* vol. 9, pp. 181-191, 2011.

[28] K. Svoboda, "The Past, Present and Future of Single Neuron Reconstruction," *Neuroinform,* vol. 9, pp. 97-98, 2011.

[29] W. He, T. A. Hamilton, A. R. Cohen, T. J. Holmes, C. Pace, D. H. Szarowski, J. N. Turner and B. Roysam, "Automated Three-Dimensional Tracing of Neurons in Confocal and Brightfield Images," *Microscopy and Microanalysis,* vol. 9, pp. 296-310, 2003.

Appendix A – Notes on Data Sets

Data set owner: GSXE Jefferis, Department of Zoology, University of Cambridge; L Luo, Department of Biology, Stanford University

Species: Drosophila

Strain: y w hs-FLP UAS-mCD8-GFP/+ or Y; FRTG13 tubP-GAL80/FRTG13 GAL4-GH146 UAS-mCD8-GFP

Nervous System Region: Olfactory Bulb

Fiber type: Axons

Labeling Method: GFP

Image Acquisition Method: 2-channel confocal microscopy

Tracing Method: Neurolucida (Williston, VT); Amira (Chelmsford, MA) extension module hxskeletonize (Evers JF et al., 2005, J Neurophysiol. Vol 93(4))

Objective Lens: 40x oil (NA=1.3) with 1.5x zoom

See Jefferis et al., 2007, Cell Vol 128(6) for general information related to the data

Appendix B – Source Code

MainLauncher.java

```
import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Collections;
import java.util.LinkedList;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import javax.swing.JFileChooser;

public class MainLauncher {
	private final static int RATE = 1;
	private static String stackSampleIdentifier;
	private static LinkedList<Coord> candidates = new LinkedList<Coord>();
	private static LinkedList<Coord> singletons = new LinkedList<Coord>();
	private static LinkedList<Coord> endPointSeeds = new LinkedList<Coord>();
	private static LinkedList<Coord> middlePoints = new LinkedList<Coord>();
	private static LinkedList<Coord> bifurPoints = new LinkedList<Coord>();
	private static LinkedList<Coord> blobPoints = new LinkedList<Coord>();
	private static LinkedList<Fragment> fragments = new LinkedList<Fragment>();
	private static int imgWidth;
	private static int imgHeight;
	private static int imageCount;
	private static double[] stackStats;
	private static int[] levelValues;
	private static int sampleRate;
	private static double zSpacing;
	private static PlanarImage rayProjectionPI;
	private static PlanarImage thresholdedRayProjection;

	public static void main(String[] args) {

		JFileChooser fc = new JFileChooser();
		fc.setDialogTitle("Please select the first image in the stack");
		fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
		int returnVal = fc.showOpenDialog(null);
		File imageFile = null;
		if (returnVal == JFileChooser.APPROVE_OPTION) {
			imageFile = fc.getSelectedFile();
		}else{
			System.out.println("No Image selected.");
			System.exit(1);
		}

		stackSampleIdentifier = imageFile.getParentFile().getName() + "_sr_" + RATE;
```

```
		PrintStream out = null;
		try {
			out = new PrintStream("output\\" + stackSampleIdentifier + ".txt");
		} catch (FileNotFoundException e) {
			System.out.println("Can't write to output file");
			System.exit(1);
		}

		StackInspector si = new StackInspector(RATE, out);
		si.examineStack(imageFile);

		imgWidth = si.getImgWidth();
		imgHeight = si.getImgHeight();
		imageCount = si.getImgCount();

		rayProjectionPI = si.getRayProjection();
		stackStats = si.getStackStats();

		sampleRate = si.getSampleRate();
		zSpacing = si.getZSpacing();

		ImageMaker im = new ImageMaker(imgHeight, imgWidth, stackSampleIdentifier);

		thresholdedRayProjection = im.createThresholdedImage(rayProjectionPI, stackStats[5]);

ColorModel cm = PlanarImage.createColorModel( thresholdedRayProjection.getSampleModel() );
BufferedImage rayProjection = thresholdedRayProjection.getAsBufferedImage(null, cm);

levelValues = si.getLevelValues();

		SeedFinder sf = new SeedFinder( imgHeight, imgWidth, sampleRate, zSpacing, out );

		sf.findSeeds(rayProjection, imageCount);

		candidates = sf.getSeeds();

		SliceSorter ss = new SliceSorter(si, im, candidates );

out.println("");
out.println("Start slice assignment");
out.println("Pre-slice assignment candidate count: " + candidates.size());

ss.findSlices();

out.println("Post-slice assignment candidate count: " + candidates.size());

		Connector con = new Connector(out);
		con.connectSeeds(candidates);

		singletons = con.getSingletons();
endPointSeeds = con.getEndPoints();
```

```
middlePoints = con.getMidPoints();
bifurPoints = con.getBifurPoints();
blobPoints = con.getBlobPoints();

if(blobPoints.size() > 0){
    out.println("");
    out.println("Image ambiguity areas:");
}

Refiner refiner = new Refiner(candidates, zSpacing, con, out);
refiner.refineSeeds();

singletons = con.getSingletons();
endPointSeeds = con.getEndPoints();
middlePoints = con.getMidPoints();
bifurPoints = con.getBifurPoints();
blobPoints = con.getBlobPoints();

LinkedList<Coord> refinedCandidates = refiner.getNewCandidates();

candidates = refinedCandidates;

for(Coord point : candidates){
    point.setExamined(false);
}

for(Coord point : candidates){
    if(!point.hasBeenExamined()){
            // create a fragment and add this point to it
            Fragment frag = new Fragment();
            fragments.add(frag);
            // now add all coords attached to this coord
            buildFrag(point, frag);
    }
}

for(Fragment frag : fragments){
    frag.setFragmentStats();
    double[] fragStats = new double[9];
    fragStats = frag.getFragStats();
    if(fragStats[0] > 1){
            double startX;
            double startY;
            double startZ;
            double endX;
            double endY;
            double endZ;
            startX = fragStats[1];
            startY = fragStats[2];
            startZ = fragStats[3];
            endX = fragStats[4];
```

```
                endY = fragStats[5];
                endZ = fragStats[6];
                double width =  endX - startX;
                double height = endY - startY;
                double depth = endZ - startZ;

                if(depth > 0){
                        // frag spans multiple slices
                        double volume = width * height * depth;
                }else{
                        // frag all on one slice - show area
                        double area = width * height;
                }
        }
    }

    LinkedList<Coord> finalSeedSet = new LinkedList<Coord>();

    for(Fragment frag : fragments){
        LinkedList<Coord> myBest = frag.getBestSeeds();
        finalSeedSet.addAll(myBest);
    }

    Collections.sort(singletons, new CoordSortRad());
    int selectNumber = (int)(singletons.size() * 0.2);
    for(int i = 0; i < selectNumber; i++){
        finalSeedSet.add(singletons.get(i));
    }

    out.println("");
    out.println("Final seed count:\t" + finalSeedSet.size());
    out.println("Final Seed Set List");
                out.println("X\tY\tZ\tRadius");

                for(Coord seed : finalSeedSet){
                        double pointZ = seed.getOptimalZ();
                        pointZ = pointZ / zSpacing;
                        pointZ = pointZ * 100;
                        pointZ = Math.round(pointZ);
                        pointZ = pointZ / 100;
                        out.println(seed.getX() + "\t" + seed.getY() + "\t" + pointZ + "\t" +
seed.getRadius());
                }

    fc.setDialogTitle("Please select SWC file for seed point validation");
                fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
                int returnVal2 = fc.showOpenDialog(null);

                File swcFile = null;

                if (returnVal2 == JFileChooser.APPROVE_OPTION) {
```

```
				swcFile = fc.getSelectedFile();
			}else{
				out.println("No SWC file selected - ending program.");
				System.exit(1);
			}

			SWCComparator compare = new SWCComparator(out);
			compare.openSWC(swcFile, zSpacing);

	out.println("");
			out.println("Evaluating final seed set");
			LinkedList<Coord> resultSet;
			out.println("");

			out.println("Threshold set to 1 pixel");
			resultSet = compare.compareCoords(finalSeedSet, 1.0);
			out.println("\tNumber of seeds validated:\t" + resultSet.size());
			out.println("\tNumber of seeds rejected:\t" + compare.getLosers().size());
			out.println("\tMatching seed percentage:\t" +
( (double)Math.round((double)resultSet.size() / (double)finalSeedSet.size() * 10000 )) / 100);

	out.println("");
			out.println("Threshold set to 2 pixels");
			resultSet = compare.compareCoords(finalSeedSet, 2.0);
	    out.println("\tNumber of seeds validated:\t" + resultSet.size());
			out.println("\tNumber of seeds rejected:\t" + compare.getLosers().size());
			out.println("\tMatching seed percentage:\t" +
( (double)Math.round((double)resultSet.size() / (double)finalSeedSet.size() * 10000 )) / 100);

	out.println("");
			out.println("Threshold set to 3 pixels");
			resultSet = compare.compareCoords(finalSeedSet, 3.0);
	out.println("\tNumber of seeds validated:\t" + resultSet.size());
			out.println("\tNumber of seeds rejected:\t" + compare.getLosers().size());
			out.println("\tMatching seed percentage:\t" +
( (double)Math.round((double)resultSet.size() / (double)finalSeedSet.size() * 10000 )) / 100);

			out.println("");
			out.println("Threshold set to 4 pixels");
			resultSet = compare.compareCoords(finalSeedSet, 4.0);
	out.println("\tNumber of seeds validated:\t" + resultSet.size());
			out.println("\tNumber of seeds rejected:\t" + compare.getLosers().size());
			out.println("\tMatching seed percentage:\t" +
( (double)Math.round((double)resultSet.size() / (double)finalSeedSet.size() * 10000 )) / 100);

			out.close();
	System.exit(0);

	    }

	    public static void buildFrag(Coord seed, Fragment frag){
```

```
		if(!seed.hasBeenExamined()){
			seed.setExamined(true);
			frag.addMember(seed);
			if(seed.getConnectedSeeds().size() > 0){
				for(Coord connection : seed.getConnectedSeeds()){
					buildFrag(connection, frag);
				}
			}
		}
	}

	/**
	* this gets rid of exception for not using native acceleration
	*/
	static
	{
	System.setProperty("com.sun.media.jai.disableMediaLib", "true");
	}
	public static boolean dupCoord(Coord nodeA, Coord nodeB){
		boolean result = false;
		if(nodeA != nodeB){
			double aX = nodeA.getX();
			double aY = nodeA.getY();
			double aZ = nodeA.getOptimalZ();
			double aRad = nodeA.getRadius();
			double bX = nodeB.getX();
			double bY = nodeB.getY();
			double bZ = nodeB.getOptimalZ();
			double bRad = nodeB.getRadius();

			if(aX == bX && aY == bY && aZ == bZ && aRad == bRad){
				result = true;
			}

		}
		return result;
	}
}
```

StackInspector.java

```
import java.awt.image.RenderedImage;
import java.awt.image.renderable.ParameterBlock;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.PrintStream;
import java.util.LinkedList;
import java.util.Scanner;
import javax.media.jai.Histogram;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import javax.media.jai.RenderedOp;
import javax.swing.JFrame;
import javax.swing.JOptionPane;


public class StackInspector {

	private LinkedList<String> imageList = new LinkedList<String>();
	private LinkedList<Slice> slices2 = new LinkedList<Slice>();
	private LinkedList<PlanarImage> slices = new LinkedList<PlanarImage>();
	private JFrame f;
	private PrintStream out;
	private int imgWidth;
	private int imgHeight;
	private int imageCount;
	private double zSpacing;
	private double originX;
	private double originY;
	private double originZ;
	private int samplingRate;
	private int[] levelValues;
	private double imgMean;
	private double extremaHigh;
	private double extremaLow;
	private double stdDev;
	private double sigToNoise;
	private double threshold;
	private PlanarImage rayProjection;

	public StackInspector(int sampleRate, PrintStream outputFile){
		out = outputFile;
		samplingRate = sampleRate;
	}

	public void examineStack(File imageFile){

		int numOfImages = 1;
```

```
String filePath = imageFile.getAbsolutePath();
String fName = imageFile.getName();

out.println("File selected as first in stack:\t" + filePath);

imageList.add(fName);

filePath = filePath.substring(0, filePath.indexOf(fName));

File info = new File(filePath + "info.txt");
if(info.exists()){

	out.print("Slice Pixel Offset:\t");

	try {
		Scanner scanner = new Scanner(new FileReader(info));
		String infoText = scanner.nextLine();
		String[] readInfo = infoText.split(";");
		String[] sliceDepth = readInfo[0].split("=");
		zSpacing = Double.valueOf(sliceDepth[1]);

		out.println(zSpacing);

		String[] originCoords = readInfo[1].split("=");
		String[] origin = originCoords[1].split(",");
		originX = Double.valueOf(origin[0]);
		originY = Double.valueOf(origin[1]);
		originZ = Double.valueOf(origin[2]);

		scanner.close();
	} catch (FileNotFoundException e1) {
		e1.printStackTrace();
	}

}else{
	String s = "";
	while (s == null || s.length() <= 0){
		s = askForInfo();
		zSpacing = Double.parseDouble(s);
	}
	out.println(s);
}

int dotIndex = fName.lastIndexOf('.');
String extension = fName.substring(dotIndex, fName.length());
fName = fName.substring(0, dotIndex);

boolean zeroSeed = false;
String firstChar = fName.substring(0, 1);
String fNameString = "" + fName;
if(firstChar.equalsIgnoreCase("0")){
```

```
        zeroSeed = true;
}

int nextFile = Integer.parseInt(fName) + 1;
String nextFileString = "" + nextFile;
if(zeroSeed){
        if(nextFile < 10){
                nextFileString = "0" + nextFile;
        }
}else{

}
File next = new File(filePath + nextFileString + extension);

while(next.exists()){
        imageList.add(next.getName());
        numOfImages++;

        nextFile++;
        if(zeroSeed){
                if(nextFile < 10){
                        nextFileString = "0" + nextFile;
                }else{
                        nextFileString = "" + nextFile;
                }
        }else{
                nextFileString = "" + nextFile;
        }
        next = new File(filePath + nextFileString + extension);
}

ParameterBlock pbA = new ParameterBlock();
PlanarImage piXtemp = JAI.create("fileload", filePath + fNameString + extension);
slices.add(piXtemp);

for (int i = samplingRate; i <= numOfImages; i+=samplingRate){
        pbA.removeSources();

        String imageNum = "" + i;

        if(zeroSeed){
                if(i < 10){
                        imageNum = "0" + i;
                }else{
                        imageNum = "" + i;
                }
        }else{
                imageNum = "" + i;
        }

        String  fileName = imageNum + ".tif";
```

```
            PlanarImage piA = JAI.create("fileload", filePath + fileName);
            slices.add(piA);

            Slice tempSlice = new Slice(piA, i);
            slices2.add(tempSlice);

            pbA.addSource(piXtemp);
            pbA.addSource(piA);

            piXtemp = JAI.create("max", pbA);
        }

        rayProjection = piXtemp;

        imgWidth = piXtemp.getWidth();
        imgHeight = piXtemp.getHeight();

        imageCount = slices2.size();

        out.println("Sampling rate:\t" + samplingRate);
        out.println("Image dimensions:\t" + imgHeight + "px X " + imgWidth + "px");
        out.println("Total number of slices in stack:\t" + numOfImages);
        out.println("Total number of slices sampled:\t" + slices2.size());

        out.println("");
        out.println("Image statistics:");
        pbA.removeParameters();
        pbA.removeSources();
        pbA.addSource(piXtemp);
        pbA.add(null);
        pbA.add(1);
        pbA.add(1);

        RenderedImage meanImage = JAI.create("mean", pbA, null);

     double[] mean = (double[])meanImage.getProperty("mean");
     imgMean = mean[0];

    out.println("Mean of the ray projection:\t" + mean[0]);

    pbA.removeParameters();
    pbA.removeSources();
    pbA.addSource(piXtemp);
        RenderedImage ri = JAI.create("extrema", pbA, null);
        double[][] extrema = (double[][]) ri.getProperty("extrema");
        extremaHigh = (int)extrema[1][0];
        extremaLow = (int)extrema[0][0];

        out.println("Extrema intensity values:\thigh = " + extremaHigh + "; low = " +
extremaLow);
```

```
        PlanarImage image = JAI.create("fileload", filePath + imageList.get(0));

        ParameterBlock pb = new ParameterBlock();

int[] bins = { 256 };
double[] low = { 0.0D };
double[] high = { 256.0D };

pb.addSource(image);
pb.add(null);
pb.add(1);
pb.add(1);
pb.add(bins);
pb.add(low);
pb.add(high);

RenderedOp op = JAI.create("histogram", pb, null);
Histogram histogram = (Histogram) op.getProperty("histogram");

double[] std = histogram.getStandardDeviation();
stdDev = std[0];

out.println("Standard Deviation:\t" + stdDev);

sigToNoise = imgMean / std[0];

out.println("Signal to Noise ratio:\t" + sigToNoise);

double[] itThresh = histogram.getIterativeThreshold();
threshold = itThresh[0];

out.println("Threshold level:\t" + (int)threshold + " (out of 255)");

  }

  public PlanarImage getRayProjection(){
        return rayProjection;
  }

  public int getImgWidth(){
        return imgWidth;
  }

  public int getImgHeight(){
        return imgHeight;
  }

  public int getImgCount(){
        return imageCount;
  }
```

```
	public double[] getStackStats(){
		double[] stats = new double[6];
		stats[0] = imgMean;
		stats[1] = extremaHigh;
		stats[2] = extremaLow;
		stats[3] = stdDev;
		stats[4] = sigToNoise;
		stats[5] = threshold;
		return stats;
	}

	public int[] getLevelValues(){
		return levelValues;
	}

	public LinkedList<Slice> getSlices(){
		return slices2;
	}

	public int getSampleRate(){
		return samplingRate;
	}

	public double getZSpacing(){
		return zSpacing;
	}

	private String askForInfo(){
		String answer = "";

		answer = (String)JOptionPane.showInputDialog(f, "Please enter the pixel distance
between each slice","Stack Information",JOptionPane.PLAIN_MESSAGE,null,null,"1.0");
		return answer;
	}
}
```

ImageMaker.java

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.awt.image.renderable.ParameterBlock;
import java.util.Iterator;
import java.util.LinkedList;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;


public class ImageMaker {

	private final Color singletonColor = Color.RED;
	private final Color endPointColor = Color.BLUE;
	private final Color midPointColor = Color.DARK_GRAY;
	private final Color bifurPointColor = Color.MAGENTA;
	private final Color blobColor = Color.GREEN;
	private final Color connectorLines = Color.LIGHT_GRAY;
	private int height;
	private int width;
	private String stackSampleIdentifier;

	public ImageMaker(int imgHeight, int imgWidth, String identifier){
		height = imgHeight;
		width = imgWidth;
		stackSampleIdentifier = identifier;
	}

	public BufferedImage showSeeds(LinkedList<Coord> seeds){

		BufferedImage results = new BufferedImage(width, height,
BufferedImage.TYPE_BYTE_GRAY);

		Graphics2D g = (Graphics2D) results.createGraphics();
		g.setPaint ( new Color ( 255, 255, 255 ) );
		g.fillRect ( 0, 0, results.getWidth(), results.getHeight() );
		Iterator<Coord> it2 = seeds.iterator();

		g.setPaint( new Color (0, 0, 0) );

		while(it2.hasNext()){

			Coord coordinate = (Coord) it2.next();
			int xCoord = (int)coordinate.getX();
			int yCoord = (int)coordinate.getY();
			int radius = (int)coordinate.getRadius();

			g.drawOval(xCoord - (radius/2), yCoord - (radius/2), radius, radius);
```

```
		}
		return results;
	}

	public BufferedImage showSeedDepths(LinkedList<Coord> seeds, int depths){

		BufferedImage results = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);

		Graphics2D g = (Graphics2D) results.createGraphics();
		g.setPaint ( new Color ( 255, 255, 255 ) );
		g.fillRect ( 0, 0, results.getWidth(), results.getHeight() );
		Iterator<Coord> it2 = seeds.iterator();

		g.setPaint( new Color (0, 0, 0) );

		double step = 255 / depths;

		while(it2.hasNext()){

			Coord coordinate = (Coord) it2.next();
			int xCoord = (int)coordinate.getX();
			int yCoord = (int)coordinate.getY();
			int radius = (int)coordinate.getRadius();

			double zIndex = coordinate.getZIndex();

			int colorLevel = (int)(zIndex * step);
			int redLevel = 255 - colorLevel;
			int blueLevel = colorLevel;

			g.setPaint( new Color (redLevel, 0, blueLevel) );
			g.drawOval(xCoord - (radius/2), yCoord - (radius/2), radius, radius);

		}
		return results;
	}

	public BufferedImage showFragments(LinkedList<Fragment> fragList, LinkedList<Coord>
singletons){

		BufferedImage results = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);

		Graphics2D g = (Graphics2D) results.createGraphics();
		g.setPaint ( new Color ( 255, 255, 255 ) );
		g.fillRect ( 0, 0, results.getWidth(), results.getHeight() );
		Iterator<Fragment> it2 = fragList.iterator();

		g.setPaint( new Color (0, 0, 0) );
```

```
		double step = 255 / fragList.size();
		int colorLevel = 0;

		while(it2.hasNext()){

			colorLevel = (int) (colorLevel + step);
			g.setPaint( new Color (255 - colorLevel, 0, colorLevel) );

			Fragment frag = (Fragment) it2.next();

			double[] fragStats = new double[9];
	fragStats = frag.getFragStats();

	if(fragStats[0] > 1){
		int startX;
		int startY;
		int endX;
		int endY;
		startX = (int)fragStats[1];
		startY = (int)fragStats[2];
		endX = (int)fragStats[4];
		endY = (int)fragStats[5];

		int width =  endX - startX;
		int height = endY - startY;

		g.drawRect(startX, startY, width, height);
	}

		}

		for(Coord point : singletons){
			g.setColor(singletonColor);

			int xCoord = (int)point.getX();
			int yCoord = (int)point.getY();
			int radius = (int)point.getRadius();

			g.drawOval(xCoord - (radius/2), yCoord - (radius/2), radius, radius);
		}

		return results;
	}

	public BufferedImage showConnections(LinkedList<Coord> seeds){

		BufferedImage results = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
		Graphics2D g = (Graphics2D) results.createGraphics();
		g.setPaint ( new Color ( 255, 255, 255 ) );
		g.fillRect ( 0, 0, results.getWidth(), results.getHeight() );
```

```
Iterator<Coord> it = seeds.iterator();
while(it.hasNext()){
	g.setPaint(new Color(0,0,0));
	Coord coordinate = (Coord) it.next();
	int xCoord = (int)coordinate.getX();
	int yCoord = (int)coordinate.getY();
	int radius = (int)coordinate.getRadius();

	int numChildren = coordinate.getConnectedSeeds().size();

	switch(numChildren){
	case 0:
		g.setColor(singletonColor);
		break;
	case 1:
		g.setColor(endPointColor);
		break;
	case 2:
		g.setColor(midPointColor);
		break;
	case 3:
		g.setColor(bifurPointColor);
		break;
	default:
		g.setColor(blobColor);
		break;
	}

	g.drawOval(xCoord - (radius/2), yCoord - (radius/2), radius, radius);

	if(numChildren > 0){
		for(Coord child : coordinate.getConnectedSeeds()){
			int x2 = (int)child.getX();
			int y2 = (int)child.getY();
			g.setColor(connectorLines);
			g.drawLine(xCoord, yCoord, x2, y2);
		}

	}

}

g.setColor(singletonColor);
g.drawString("Singletons", 400, 400);
g.setColor(endPointColor);
g.drawString("End Points", 400, 415);
g.setColor(midPointColor);
g.drawString("Midpoints", 400, 430);
g.setColor(bifurPointColor);
g.drawString("Bifurcations", 400, 445);
```

```
                g.setColor(blobColor);
                g.drawString("Blobs", 400, 460);

                return results;
        }

        public PlanarImage createThresholdedImage(PlanarImage ray, double threshold){

                PlanarImage thresholdedRayProjection;

                ParameterBlock pb = new ParameterBlock();
                pb.removeParameters();
      pb.removeSources();
      double[] lowVal = { 0.0D };
      double[] highVal = { threshold };
      double[] map = { 0.0D };
      pb.addSource(ray);
      pb.add(lowVal);
      pb.add(highVal);
      pb.add(map);
      thresholdedRayProjection = JAI.create("threshold", pb);

      return thresholdedRayProjection;
        }

}
```

SeedFinder.java

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.PrintStream;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;

public class SeedFinder {

	private PrintStream out;
	private LinkedList<Coord> candidates;
	private double avgRadius;
	private double avgIntensity;
	private int imgWidth;
	private int imgHeight;
	private int sampleRate;
	private double zSpacing;
	private int radiusLimit = 2;
	private BufferedImage image;

	public SeedFinder(int height, int width, int sampling, double zSpace, PrintStream outputFile){
		candidates = new LinkedList<Coord>();
		imgWidth = width;
		imgHeight = height;
		sampleRate = sampling;
		zSpacing = zSpace;
		out = outputFile;
	}

	public void findSeeds(BufferedImage input, int imageCount){

		out.println("");
		out.println("Begin seeding");

		image = input;
		int height = input.getHeight();
		int width = input.getWidth();
		BufferedImage results = new BufferedImage(width, height,
BufferedImage.TYPE_BYTE_GRAY);

		LinkedList<Coord> coords = new LinkedList<Coord>();

		int[] data = new int[width * height];
		input.getRGB(0, 0, width, height, data, 0, width);

		for(int i = 0; i < width; i++){
			for(int j = 0; j < height; j++){
				int index = (i * width) + j;
```

```
                    int intensityValue = data[index] & 0x00000ff;

                    if(intensityValue > 0){
                        Coord newCoord = new Coord(j, i, 0, intensityValue);
                        newCoord.setSampleRate(sampleRate);
                        newCoord.setZSpacing(zSpacing);
                        coords.add(newCoord);
                        Color paint = new Color(intensityValue, intensityValue,
intensityValue);
                        results.setRGB(j, i, paint.getRGB());
                    }
                }
            }

            out.println("Found " + coords.size() + " pixels with intensity greater than 0");

            double sumAllRadii = 0.0D;
            double sumAllIntensities = 0.0D;

            Collections.sort(coords, new CoordSortBright());
    Collections.reverse(coords);


            Iterator<Coord> it = coords.iterator();
            while(it.hasNext()){
                Coord myPoint = (Coord) it.next();
                if(!myPoint.hasBeenExamined()&& !isCoveredBySeed(myPoint)){
                    myPoint.setExamined(true);

                    int centerX = (int)myPoint.getX();
                    int centerY = (int)myPoint.getY();
                    double zIndex = 0.0D;

                    int kernel_size = 1;

                    boolean stopSearch = false;

                    while(!stopSearch){

                        boolean stopTop = false;
                        boolean stopBottom = false;
                        boolean stopLeft = false;
                        boolean stopRight = false;

                        for(int t = -kernel_size; t < kernel_size; t++){
                            int checkY = centerY - kernel_size;
                            int checkX = centerX + t;
                            if(isBlankPixel(results, checkX, checkY)){
                                stopTop = true;
                            }
```

```
		}

		for(int b = -kernel_size; b < kernel_size; b++){
			int checkY = centerY + kernel_size;
			int checkX = centerX + b;
			if(isBlankPixel(results, checkX, checkY)){
				stopBottom = true;
			}
		}

		for(int l = -kernel_size; l < kernel_size; l++){
			int checkX = centerX - kernel_size;
			int checkY = centerY + l;
			if(isBlankPixel(results, checkX, checkY)){
				stopLeft = true;
			}
		}

		for(int r = -kernel_size; r < kernel_size; r++){
			int checkX = centerX + kernel_size;
			int checkY = centerY + r;
			if(isBlankPixel(results, checkX, checkY)){
				stopRight = true;
			}
		}
		if((stopTop && stopBottom) || (stopLeft && stopRight)){
			stopSearch = true;
			break;
		}

		if(stopTop || stopBottom || stopLeft || stopRight){
			if(stopTop){
				centerY++;
			}
			if(stopBottom){
				centerY--;
			}
			if(stopLeft){
				centerX++;
			}
			if(stopRight){
				centerX--;
			}

		}

		kernel_size++;

	}

	if(centerX < 0){
```

```
				centerX = 0;
			}
			if(centerX > width){
				centerX = width;
			}
			if(centerY < 0){
				centerY = 0;
			}
			if(centerY > height){
				centerY = height;
			}

			Coord newCandidate = new Coord(centerX, centerY, zIndex, 100);
			newCandidate.establishSliceIntensityDistribution(imageCount);
			newCandidate.setZSpacing(zSpacing);
			newCandidate.setSampleRate(sampleRate);
			double newRadius = (2 * kernel_size) - 1;
			newCandidate.setRadius( newRadius );

			if(isCoveredBySeed(newCandidate)){

			}else{

				if(newRadius > radiusLimit){
					candidates.add(newCandidate);
					sumAllRadii += newRadius;

					double avgIntensity =
findSeedAverageIntensity(newCandidate);

					sumAllIntensities += avgIntensity;
					newCandidate.setAvgIntensity(avgIntensity);
				}
			}
			myPoint.setExamined(true);

		}else{
			myPoint.setExamined(true);
		}

	}
		avgRadius = sumAllRadii/candidates.size();
		avgIntensity = sumAllIntensities/candidates.size();

		out.println("Found these many potential seeds: " + candidates.size());
		out.println("Average seed radius: " + avgRadius);
		out.println("Average intensity of seeds: " + avgIntensity);

	}

	public LinkedList<Coord> getSeeds(){
```

```
		return candidates;
	}

	public double getAvgRadius(){
		return avgRadius;
	}

	public double getAvgIntensity(){
		return avgIntensity;
	}

	private boolean isCoveredBySeed(Coord thisCoord){
		boolean result = false;

		for(int i = 0; i < candidates.size(); i++){
			if(candidates.get(i) != null){
				Coord seed = candidates.get(i);
				double seedRad = seed.getRadius();

				double distance = findDistance(thisCoord, seed);
				if(distance <= seedRad){
					result = true;
				}
			}
		}

		return result;
	}

	private boolean isBlankPixel(BufferedImage image, int xPos, int yPos){
		boolean result = false;
		int imgWidth = image.getWidth();
		int imgHeight = image.getHeight();
		if((xPos < 0) || (xPos >= imgWidth) || (yPos < 0) || (yPos >= imgHeight)){
			return false;
		}

		int pixelColor = image.getRGB(xPos, yPos);
		Color pixelColorColor = new Color(pixelColor);
		if(pixelColorColor.getRed() == 0 && pixelColorColor.getBlue() == 0 &&
pixelColorColor.getBlue() == 0){
			result = true;
		}
		return result;
	}

	private double findDistance(Coord one, Coord two){

		double threeDimenDistance = Math.sqrt( Math.pow(Math.abs(one.getX() -
two.getX()),2) + Math.pow(Math.abs(one.getY() - two.getY()),2) + Math.pow(Math.abs(1 - 1),2));
```

```
		return threeDimenDistance;
	}

	private double findSeedAverageIntensity(Coord seed){

		int sumIntensity = 0;
		int numPixels = 0;
		double avgIntensity = 0.0D;

		int kernel_size = (int)seed.getRadius();
		int centerX = seed.getX();
		int centerY = seed.getY();

		for(int u = -kernel_size; u < kernel_size; u++){
			for(int v = -kernel_size; v < kernel_size; v++){
				int pointX = centerX + u;
				int pointY = centerY + v;
				if(pointInImage(pointX, pointY)){
					if(pixelInSeed(pointX, pointY, seed)){
						int tempIntensity = (image.getRGB(pointX, pointY)) &
0x00000ff;
						sumIntensity += tempIntensity;
						numPixels++;
					}
				}
			}
		}

		avgIntensity = sumIntensity/numPixels;

		return avgIntensity;
	}

	private boolean pixelInSeed(int x, int y, Coord seed){
		boolean result = true;
		int seedX = seed.getX();
		int seedY = seed.getY();
		double seedRad = seed.getRadius();

		if(  Math.pow(x - seedX, 2) + Math.pow(y - seedY, 2) < Math.pow(seedRad, 2)){
			result = true;
		}

		return result;
	}

	private boolean pointInImage(int pointX, int pointY){
		boolean result = true;

		if(pointX < 0 || pointY < 0){
			result =  false;
```

```
		}
		if(pointX >= imgWidth || pointY >= imgHeight){
			result =  false;
		}

		return result;
	}
}
```

SliceSorter.java

```
import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;
import java.util.LinkedList;
import javax.media.jai.PlanarImage;

public class SliceSorter {

	private StackInspector si;
	private ImageMaker im;
	private LinkedList<Slice> slices = new LinkedList<Slice>();
	private LinkedList<Coord> candidates = new LinkedList<Coord>();
	double threshold;

	public SliceSorter(StackInspector siInput, ImageMaker imInput, LinkedList<Coord>
candidatesInput){
		si = siInput;
		im = imInput;
		slices = siInput.getSlices();
		threshold = si.getStackStats()[5];
		candidates = candidatesInput;
	}

	public void findSlices(){

		for(int imgIndex = 0; imgIndex < slices.size(); imgIndex++){

			PlanarImage tempSlice =
im.createThresholdedImage(slices.get(imgIndex).getImage(), threshold);

			ColorModel colorModel =
PlanarImage.createColorModel( tempSlice.getSampleModel() );
	BufferedImage sliceImage = tempSlice.getAsBufferedImage(null, colorModel);

	for(int seedIndex = 0; seedIndex < candidates.size(); seedIndex++){

	Coord tempSeed = candidates.get(seedIndex);
	double tempAvgIntensity = findSeedAverageIntensity(tempSeed, sliceImage);

	if(tempAvgIntensity > 0){
		tempSeed.setZ(imgIndex, tempAvgIntensity);
	}
```

```
            tempSeed.setSliceIntensityDistribution(imgIndex, tempAvgIntensity);
           }

        }

        for(int zChecker = 0; zChecker < candidates.size(); zChecker++){
            Coord optimizeCoord = candidates.get(zChecker);
            Coord testCoord = optimizeCoord.setOptimalZ();

            if(testCoord.getX() == 0 && testCoord.getY() == 0 && testCoord.getIntensity() <= 0.0){

            }else{
                    candidates.add(testCoord);
            }
        }

            }

            private double findSeedAverageIntensity(Coord seed, BufferedImage slice){

                    int sumIntensity = 0;
                    int numPixels = 0;
                    double avgIntensity = 0.0D;

                    int kernel_size = (int)seed.getRadius();
                    int centerX = seed.getX();
                    int centerY = seed.getY();

                    for(int u = -kernel_size; u < kernel_size; u++){
                            for(int v = -kernel_size; v < kernel_size; v++){
                                    int pointX = centerX + u;
                                    int pointY = centerY + v;
                                    if(pointInImage(pointX, pointY)){
                                            if(pixelInSeed(pointX, pointY, seed)){
                                                    int tempIntensity = (slice.getRGB(pointX, pointY)) &
0x00000ff;
                                                    sumIntensity += tempIntensity;
                                                    numPixels++;
                                            }
                                    }
                            }
                    }

                    avgIntensity = sumIntensity/numPixels;

                    return avgIntensity;
            }

            private boolean pixelInSeed(int x, int y, Coord seed){
                    boolean result = true;
```

```
		int seedX = seed.getX();
		int seedY = seed.getY();
		double seedRad = seed.getRadius();

		if(  Math.pow(x - seedX, 2) + Math.pow(y - seedY, 2) < Math.pow(seedRad, 2)){
			result = true;
		}

		return result;
	}

	private boolean pointInImage(int pointX, int pointY){
		boolean result = true;

		if(pointX < 0 || pointY < 0){
			result =  false;
		}
		if(pointX >= si.getImgWidth() || pointY >= si.getImgHeight()){
			result =  false;
		}

		return result;
	}
}
```

Connector.java

```
import java.io.PrintStream;
import java.util.Collections;
import java.util.LinkedList;

public class Connector {

	private PrintStream out;
	private LinkedList<Coord> candidates = new LinkedList<Coord>();
	private LinkedList<Coord> singletons = new LinkedList<Coord>();
	private LinkedList<Coord> endPoints = new LinkedList<Coord>();
	private LinkedList<Coord> middlePoints = new LinkedList<Coord>();
	private LinkedList<Coord> bifurPoints = new LinkedList<Coord>();
	private LinkedList<Coord> blobPoints = new LinkedList<Coord>();

	public Connector(PrintStream outputFile){
		out = outputFile;
	}

	public void connectSeeds(LinkedList<Coord> candidatesInput){
		candidates = candidatesInput;

		for(Coord point : candidates){
			point.resetConnectedSeeds();
		}

		Collections.sort(candidates, new CoordSortX());

		findNeighbors();

		Collections.sort(candidates, new CoordSortY());

		findNeighbors();

		Collections.sort(candidates, new CoordSortZ());

		findNeighbors();

		classifySeeds();
	}

	private void findNeighbors(){
		for (int i = 0; i < candidates.size(); i++){
			Coord point = candidates.get(i);

			int maxAway = 50;

			for(int j = (maxAway * -1); j < maxAway; j++){
				if( (j != 0) && (i + j) >= 0 && (i + j) < candidates.size()){
					Coord point2 = candidates.get(i + j);
```

```
                                                if(point2 != point){
                                                        double distance = findDistance(point2, point);

                                                        double sumRadii = point.getRadius() +
point2.getRadius();

                                                        if(point.getRadius() == 1 && point2.getRadius() == 1){
                                                                sumRadii = 5;
                                                        }

                                                        if(distance <= sumRadii){

                                                                double radDiff = Math.abs(point.getRadius() -
point2.getRadius());
                                                                if(radDiff < 3){


        if(!point.getConnectedSeeds().contains(point2)){

        point.addConnectedSeed(point2);
                                                                        }

        if(!point2.getConnectedSeeds().contains(point)){

        point2.addConnectedSeed(point);
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
        }

        private void classifySeeds(){

                singletons.clear();
                endPoints.clear();
                middlePoints.clear();
                bifurPoints.clear();
                blobPoints.clear();

                for(Coord seed : candidates){
                        int connections = seed.getConnectedSeeds().size();
                        switch(connections){
                        case 0:
                                singletons.add(seed);
                                break;
                        case 1:
                                endPoints.add(seed);
```

```
				break;
			case 2:
				middlePoints.add(seed);
				break;
			case 3:
				bifurPoints.add(seed);
				break;
			default:
				blobPoints.add(seed);
				break;
			}
		}

		out.println("");
		out.println("Connection code results:");
		out.println("\tNumber of singletons: " + singletons.size());
		out.println("\tNumber of end points: " + endPoints.size());
		out.println("\tNumber of middle points: " + middlePoints.size());
		out.println("\tNumber of bifurcations: " + bifurPoints.size());
		out.println("\tNumber of blob points: " + blobPoints.size());

	}

	private static double findDistance(Coord one, Coord two){

		double threeDimenDistance = Math.sqrt( Math.pow(Math.abs(one.getX() -
two.getX()),2) + Math.pow(Math.abs(one.getY() - two.getY()),2) +
Math.pow(Math.abs(one.getOptimalZ() - two.getOptimalZ()),2));

		return threeDimenDistance;
	}

	public LinkedList<Coord> getSingletons(){
		return singletons;
	}

	public LinkedList<Coord> getEndPoints(){
		return endPoints;
	}

	public LinkedList<Coord> getMidPoints(){
		return middlePoints;
	}

	public LinkedList<Coord> getBifurPoints(){
		return bifurPoints;
	}

	public LinkedList<Coord> getBlobPoints(){
		return blobPoints;
	}}
```

Refiner.java

```
import java.io.PrintStream;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;

public class Refiner {

	private PrintStream out;
	private double zSpacing;
	private LinkedList<Coord> candidates = new LinkedList<Coord>();
	private LinkedList<Coord> singletons = new LinkedList<Coord>();
	private LinkedList<Coord> endPointSeeds = new LinkedList<Coord>();
	private LinkedList<Coord> middlePoints = new LinkedList<Coord>();
	private LinkedList<Coord> bifurPoints = new LinkedList<Coord>();
	private LinkedList<Coord> blobPoints = new LinkedList<Coord>();
	private Connector conn;

	public Refiner(LinkedList<Coord> candidatesInput, double zOffset, Connector connectorInput,
PrintStream outputFile){
		candidates = (LinkedList<Coord>) candidatesInput.clone();
		conn = connectorInput;
		out = outputFile;
		zSpacing = zOffset;
	}

	public void refineSeeds(){
      for(Coord reset : candidates){
	reset.setExamined(false);
      }

      Collections.sort(candidates, new CoordSortRad());
      Collections.reverse(candidates);

      for(Coord point : candidates){
	double myRadius = point.getRadius();
	LinkedList<Coord> connected = point.getConnectedSeeds();
	Iterator<Coord> it = connected.iterator();
	double diffPercent = 0.9;
	while(it.hasNext()){
		Coord child = it.next();
		double kidRad = child.getRadius();
		double radDiff = Math.abs(myRadius - kidRad);
		if(myRadius >= kidRad){
			if(radDiff > (myRadius * diffPercent)){
			it.remove();
		}
		}else{
			if(radDiff > (kidRad * diffPercent)){
			it.remove();
```

```
				}
			}
		}
	}

	LinkedList<Coord> addTheseSeeds = new LinkedList<Coord>();
	LinkedList<Coord> removeTheseSeeds = new LinkedList<Coord>();
	Coord nodeA, nodeC, nodeD;

	int counter = 0;

	int lowestAngle = 90;

	for(Coord nodeB : candidates){
				LinkedList<Coord> connectedNodes = nodeB.getConnectedSeeds();
		switch (connectedNodes.size()){
			case 0:
				// singleton - ignore
				break;
			case 1:
				// end point - ignore
				break;
			case 2:
				nodeA = connectedNodes.getFirst();
				nodeC = connectedNodes.getLast();
				if(nodeA.getConnectedSeeds().size() == 3 &&
nodeC.getConnectedSeeds().size() == 3){
					removeTheseSeeds.add(nodeB);
				}else{
					double angleFound = findAngle(nodeA, nodeB, nodeC);
					if(angleFound < lowestAngle){
						counter++;
						if( nodeB.getRadius() == 1 && nodeA.getRadius() > 2 &&
nodeC.getRadius() > 2 ){
							removeTheseSeeds.add(nodeB);
						}else{

							Coord newCoord = averageCoords(nodeA, nodeC);
							addTheseSeeds.add(newCoord);
							removeTheseSeeds.add(nodeA);
							removeTheseSeeds.add(nodeC);
						}
					}
				}
				break;
			case 3:
				nodeA = connectedNodes.get(0);
				nodeC = connectedNodes.get(1);
				nodeD = connectedNodes.get(2);

				if( nodeB.getRadius() == 1 ){
```

```
				removeTheseSeeds.add(nodeB);
			}else{
		double abc = findAngle(nodeA, nodeB, nodeC);
		double abd = findAngle(nodeA, nodeB, nodeD);
		double cbd = findAngle(nodeC, nodeB, nodeD);

		if(abc < lowestAngle){
			Coord newCoord = averageCoords(nodeA, nodeC);
				addTheseSeeds.add(newCoord);
				removeTheseSeeds.add(nodeA);
				removeTheseSeeds.add(nodeC);
		}

		if(abd < lowestAngle){
			Coord newCoord = averageCoords(nodeA, nodeD);
				addTheseSeeds.add(newCoord);
				removeTheseSeeds.add(nodeA);
				removeTheseSeeds.add(nodeD);
		}

		if(cbd < lowestAngle){
			Coord newCoord = averageCoords(nodeC, nodeD);
				addTheseSeeds.add(newCoord);
				removeTheseSeeds.add(nodeC);
				removeTheseSeeds.add(nodeD);
		}
			}

		break;
	default:
		if(nodeB.getRadius() == 1 ){
				removeTheseSeeds.add(nodeB);
		}else{

			double pointZ = nodeB.getOptimalZ();
			pointZ = pointZ / zSpacing;
			pointZ = pointZ * 100;
			pointZ = Math.round(pointZ);
			pointZ = pointZ / 100;

			if(!nodeB.hasBeenExamined()){
				nodeB.setExamined(true);
				out.println("\t" + nodeB.getX() + "\t" + nodeB.getY() + "\t" +
pointZ + "\t" + nodeB.getRadius());
			}

			LinkedList<Coord> children = nodeB.getConnectedSeeds();
			Collections.sort(children, new CoordSortRad());
			for(int i = 3; i < children.size(); i++){
				removeTheseSeeds.add(children.get(i));
			}
```

```
                        }
                        break;
            }
        }

        for(Coord newOne : addTheseSeeds){
            boolean addMe = false;
            for(Coord candidate: candidates){
                    if(MainLauncher.dupCoord(candidate, newOne)){
                        addMe = false;
                        break;
                    }else{
                        addMe = true;
                    }
            }
            if(addMe == true){
                    candidates.add(newOne);
            }
        }
        for(Coord oldOne : removeTheseSeeds){
            candidates.remove(oldOne);
        }

        out.println("");
        out.println("Refining Connections:");
        out.println("Altering candidate list: removing " + removeTheseSeeds.size() + " seeds, adding " +
addTheseSeeds.size() + " seeds");
        out.println("\tNew candidate list size: " + candidates.size());
        conn.connectSeeds(candidates);

                TypeSorter nodeTyper = new TypeSorter();
        nodeTyper.sortTypes(candidates);

        singletons = nodeTyper.getSingletons();
        endPointSeeds = nodeTyper.getEndPoints();
        middlePoints = nodeTyper.getMidPoints();
        bifurPoints = nodeTyper.getBifurcations();
        blobPoints = nodeTyper.getBlobs();


            }

            private double findAngle(Coord a, Coord b, Coord c){
                    double angle = 0.0D;
                    double ax, ay, az, bx, by, bz, cx, cy, cz;
            ax = a.getX();
            ay = a.getY();
            az = a.getOptimalZ();
            bx = b.getX();
            by = b.getY();
            bz = b.getOptimalZ();
```

```
        cx = c.getX();
        cy = c.getY();
        cz = c.getOptimalZ();

        double x1, y1, z1, x2, y2, z2;
        x1 = bx - ax;
        y1 = by - ay;
        z1 = bz - az;
        x2 = bx - cx;
        y2 = by - cy;
        z2 = bz - cz;

        double dotProduct = (x1 * x2) + (y1 * y2) + (z1 * z2);
        double vectorLengths = Math.sqrt( (Math.pow(x1, 2) + Math.pow(y1, 2) + Math.pow(z1, 2) ) *
( Math.pow(x2, 2) + Math.pow(y2, 2) + Math.pow(z2, 2) ));
        double resultingVal = dotProduct / vectorLengths;
                angle = Math.toDegrees(Math.acos(resultingVal));
                return angle;
        }

        private Coord averageCoords(Coord a, Coord b){

                double nodeA_x, nodeA_y, nodeA_z, nodeC_x, nodeC_y, nodeC_z, nodeA_intensity,
nodeC_intensity, nodeA_radius, nodeC_radius;
                nodeA_x = a.getX();
                nodeA_y = a.getY();
                nodeA_z = a.getOptimalZ();
                nodeA_intensity = a.getAvgIntensity();
                nodeA_radius = a.getRadius();
                nodeC_x = b.getX();
                nodeC_y = b.getY();
                nodeC_z = b.getOptimalZ();
                nodeC_intensity = b.getAvgIntensity();
                nodeC_radius = b.getRadius();

                int newX, newY, newIntensity;
                double newZ, newRadius;

                newX = (int)(nodeA_x + nodeC_x)/2;
                newY = (int)(nodeA_y + nodeC_y)/2;
                newZ = (nodeA_z + nodeC_z)/2;

                newIntensity = (int) ((nodeA_intensity + nodeC_intensity) / 2);
                newRadius = (nodeA_radius + nodeC_radius) / 2;

                Coord newNode = new Coord( newX, newY, newZ, newIntensity );
                newNode.setAvgIntensity(newIntensity);
                newNode.setRadius(newRadius);
                newNode.setZSpacing(a.getZSpacing());
                return newNode;
```

```
	}

	public LinkedList<Coord> getNewCandidates(){
		return candidates;
	}
}
```

SWCComparator.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.PrintStream;
import java.util.LinkedList;
import java.util.Scanner;

public class SWCComparator {

	private File mySWC;
	private PrintStream out;

	private double slicePixelSeparation;

	private LinkedList<Coord> winners;
	private LinkedList<Coord> losers;

	public SWCComparator(PrintStream outputFile){
		out = outputFile;
	}

	public void openSWC(File swc, double zSpacing){

		mySWC = swc;
		slicePixelSeparation = zSpacing;

		winners = new LinkedList<Coord>();
		losers = new LinkedList<Coord>();

	}

	public LinkedList<Coord> compareCoords(LinkedList<Coord> candidates, double threshold){

		winners.clear();
		losers.clear();

		losers = (LinkedList<Coord>) candidates.clone();

		double radDiffTotal = 0.0D;

		try {
			Scanner scanner = new Scanner(new FileReader(mySWC));

			while(scanner.hasNextLine()){
				String line = scanner.nextLine();
				String[] fields = line.split(" ");
				if(!fields[0].equalsIgnoreCase("#")){
					double x = Double.parseDouble(fields[2]);
					double y = Double.parseDouble(fields[3]);
```

```
					double z = Double.parseDouble(fields[4]);
					double rad = Double.parseDouble(fields[5]);


					for(Coord point : candidates){
						double pointX = point.getX();
						double pointY = point.getY();
						double pointZ = point.getOptimalZ();
						double pointRad = point.getRadius();

						pointZ = pointZ / slicePixelSeparation;
						pointZ = pointZ * 100;
						pointZ = Math.round(pointZ);
						pointZ = pointZ / 100;

						double distance =
findDistance(pointX,x,pointY,y,pointZ,z);
						if(distance <= threshold){

							if(!winners.contains(point)){
								winners.add(point);

								losers.remove(point);
								radDiffTotal += Math.abs(pointRad -
rad);
							}



						}else{

						}

					}
				}
			}
			double radDiffAvg = radDiffTotal / winners.size();
			out.println("\tAverage radius difference:\t" + radDiffAvg);
			scanner.close();
		} catch (FileNotFoundException e1) {
			System.out.println("No SWC for comparison");
			System.exit(1);
		}

		return winners;
	}

	public LinkedList<Coord> getLosers(){
		return losers;
	}
```

```
	private double findDistance(double x1, double x2, double y1, double y2, double z1, double z2){


			double threeDimenDistance = Math.sqrt( Math.pow(Math.abs(x1 - x2),2) + 
Math.pow(Math.abs(y1 - y2),2) + Math.pow(Math.abs(z1 - z2),2));

			return threeDimenDistance;
	}
}
```

Coord.java

```
import java.util.LinkedList;


public class Coord {

	private int xCoord;
	private int yCoord;
	private double slicePixelSeparation;
	private double normal;
	private LinkedList<Double> zCoord;
	private LinkedList<Double> zCoordIntensity;
	private double[] sliceIntensityDistribution;
	private double[] avgIntensityArray;
	private double[] optimalZ;
	private double bestZIndexValue;
	private double bestZPixelValue;
	private int intensityValue;
	private double radius;
	private boolean examined;
	private double avgIntensity;
	private LinkedList<Coord> connectedSeeds;
	private double angleIn;
	private boolean directZSet;

	public Coord(int x, int y, double z, int intensity){
		xCoord = x;
		yCoord = y;
		intensityValue = intensity;
		bestZPixelValue = z;

		angleIn = 0.0D;

		examined = false;
		connectedSeeds = new LinkedList<Coord>();

		zCoord = new LinkedList<Double>();
		zCoordIntensity = new LinkedList<Double>();
		optimalZ = new double[4];
		optimalZ[0] = 0.0D;
		optimalZ[1] = 0.0D;
		optimalZ[2] = 0.0D;
		optimalZ[3] = 0.0D;

		directZSet = false;
	}

	public int getX(){
		return xCoord;
	}
```

```
public int getY(){
	return yCoord;
}

public void setZ(int sliceIndex, double intensity){
	zCoord.add((double) sliceIndex);
	zCoordIntensity.add(intensity);
}

public LinkedList<Double> getZ(){
	return zCoord;
}

public LinkedList<Double> getZInt(){
	return zCoordIntensity;
}

public double getOptimalZ(){

	bestZPixelValue = bestZIndexValue * slicePixelSeparation;

	bestZPixelValue = bestZPixelValue * normal;
	bestZPixelValue = bestZPixelValue * 100;
	bestZPixelValue = Math.round(bestZPixelValue);
	bestZPixelValue = bestZPixelValue/100;

	return bestZPixelValue;
}

public double getZIndex(){
	return bestZIndexValue;
}

public double getZPixel(){
	return bestZPixelValue;
}

public double getRadius(){
	return radius;
}

public boolean hasBeenExamined(){
	return examined;
}

public void setRadius(double value){
	radius = value;
}

public void setExamined(boolean value){
```

```
		examined = value;
	}

	public double getIntensity(){
		return intensityValue;
	}

	public void setAvgIntensity(double value){
		avgIntensity = value;
	}

	public double getAvgIntensity(){
		return avgIntensity;
	}

	public void setAngleIn(double value){
		angleIn = value;
	}

	public double getAngleIn(){
		return angleIn;
	}

	public void establishSliceIntensityDistribution(int size){
		sliceIntensityDistribution = new double[size];
		avgIntensityArray = new double[size];
	}

	public void setSliceIntensityDistribution(int index, double intensity){
		sliceIntensityDistribution[index] = intensity;
	}

	public LinkedList<Coord> getConnectedSeeds() {
		return connectedSeeds;
	}

	public void addConnectedSeed(Coord seed){
		connectedSeeds.add(seed);
	}

	public void removeConnectedSeed(Coord seed){
		connectedSeeds.remove(seed);
	}

	public void resetConnectedSeeds(){
		connectedSeeds.clear();
	}

	public Coord setOptimalZ(){
		Coord returnedCoord = new Coord(0,0,0.0,0);
```

```
if(directZSet){

}else{

        int prevAvg = 0;
        int currAvg = 0;
        int highest = 0;

        boolean goingUp = false;
        double localMax1 = 0;
        double localMax2 = 0;
        double localMax1Value = 0;
        double localMax2Value = 0;

        String result = "";

        for(int i = 0; i < sliceIntensityDistribution.length; i++){

                int previous, current, next;
                previous = i - 1;
                current = i;
                next = i + 1;

                if(previous > 0 && next < sliceIntensityDistribution.length){
                        int sumValue = 0;
                        sumValue += sliceIntensityDistribution[previous];
                        sumValue += sliceIntensityDistribution[current];
                        sumValue += sliceIntensityDistribution[next];
                        int average = sumValue / 3;

                        avgIntensityArray[i] = average;

                        currAvg = average;
                        if((currAvg - prevAvg) > 0){
                                highest = current;
                                goingUp = true;
                        }else{
                                if(goingUp){
                                        goingUp = false;
                                        if(localMax1 == 0){
                                                localMax1 = previous;
                                                localMax1Value =
avgIntensityArray[previous];

                                                optimalZ[0] = localMax1;
                                                optimalZ[1] = localMax1Value;
                                        }else{
                                                localMax2 = previous;
                                                localMax2Value =
avgIntensityArray[previous];
                                                optimalZ[2] = localMax2;
```

```
                                        optimalZ[3] = localMax2Value;
                                    }
                                }
                            }
                            prevAvg = average;

                            result += "\n\t [" + i + "] : ";
                            for(int j = 0; j < average; j++){
                                result += "-";
                            }
                        }

                    }

                    if(optimalZ[3] > 0){

                        double sliceNum;
                        double higherValue;
                        double lowerValue;
                        if(optimalZ[1] > optimalZ[3]){
                            bestZIndexValue = optimalZ[0];
                            sliceNum = optimalZ[2];
                            higherValue = optimalZ[1];
                            lowerValue = optimalZ[3];
                        }else{
                            bestZIndexValue = optimalZ[2];
                            sliceNum = optimalZ[0];
                            higherValue = optimalZ[3];
                            lowerValue = optimalZ[1];
                        }

                        double difference = higherValue - lowerValue;

                        double percentDifference = difference / higherValue;

                        if(percentDifference <= 0.8){
                            returnedCoord = new Coord(xCoord, yCoord, sliceNum, 255);
                            returnedCoord.setRadius(radius);
                            returnedCoord.setAvgIntensity(higherValue);
                            returnedCoord.setZSpacing(this.slicePixelSeparation);
                            returnedCoord.setSampleRate((int)this.normal);
                            returnedCoord.directSetOptimalZ(sliceNum);

                        }

                    }else{
                        bestZIndexValue = optimalZ[0];
                    }
                }

                return returnedCoord;
```

```
	}

	public void directSetOptimalZ(double value){
		bestZIndexValue = value;
		directZSet = true;
		double tempPixelVal = 0.0D;
		tempPixelVal = value * slicePixelSeparation;
		tempPixelVal = tempPixelVal * normal;
		tempPixelVal = tempPixelVal * 100;
		tempPixelVal = Math.round(tempPixelVal);
		tempPixelVal = tempPixelVal/100;
		bestZPixelValue = tempPixelVal;
	}
	public void directSetZPixel(double value){
		bestZPixelValue = value;
	}
	public String reportZs(){
		String report = "Coord [" + this.optimalZ + " : " + this.avgIntensity + "] = ";
		for(int i = 0; i < zCoord.size(); i++){
			report += "[" +  zCoord.get(i) + " : " + (this.avgIntensity - zCoordIntensity.get(i))
+ " (" + zCoordIntensity.get(i) + ")]\t";
		}
		return report;
	}
	public double[] reportStats(){
		double[] stats  = new double[5];

		stats[0] = this.getX();
		stats[1] = this.getY();
		stats[2] = this.getOptimalZ();

		stats[3] = this.getRadius();
		stats[4] = this.getAvgIntensity();

		return stats;
	}
	public void setSampleRate(int sample){
		normal = sample;
	}
	public void setZSpacing(double spacing){
		slicePixelSeparation = spacing;
	}
	public double getZSpacing(){
		return slicePixelSeparation;
	}
	public boolean getDirectSet(){
		return directZSet;
	}
}
```

Slice.java

```
import java.util.LinkedList;
import javax.media.jai.PlanarImage;

public class Slice {

	private PlanarImage sliceImage;
	private LinkedList<Coord> sliceSeeds;
	private int sliceNum;

	public Slice(PlanarImage slice, int index){
		// constructor
		sliceImage = slice;
		sliceNum = index;

		sliceSeeds = new LinkedList<Coord>();
	}

	public PlanarImage getImage(){
		return sliceImage;
	}

	public void addSliceSeed(Coord seed){
		sliceSeeds.add(seed);
	}

	public int reportSeedCount(){
		return sliceSeeds.size();
	}
}
```

Fragment.java

```
import java.util.Collections;
import java.util.LinkedList;

public class Fragment {

	private LinkedList<Coord> members;
	private int memberCount;
	private double avgIntensity;
	private double avgRadius;
	private double minX = Double.MAX_VALUE;
	private double minY = Double.MAX_VALUE;
	private double minZ = Double.MAX_VALUE;
	private double maxX = 0.0D;
	private double maxY = 0.0D;
	private double maxZ = 0.0D;
	private int singletons;
	private int endPoints;
	private int middlePoints;
	private int bifurcations;
	private int blobs;

	public Fragment(){
		members = new LinkedList<Coord>();
		singletons = 0;
		endPoints = 0;
		middlePoints = 0;
		bifurcations = 0;
		blobs = 0;
	}

	public void addMember(Coord seed){
		members.add(seed);
		int nodeConnections = seed.getConnectedSeeds().size();

		switch(nodeConnections){
		case 0:
			singletons++;
			break;
		case 1:
			endPoints++;
			break;
		case 2:
			middlePoints++;
			break;
		case 3:
			bifurcations++;
			break;
		default:
			blobs++;
```

```
				break;
			}
		}

		public void removeMember(Coord seed){
			members.remove(seed);
		}

		public LinkedList<Coord> getMembers(){
			return members;
		}

		public void setFragmentStats(){
			memberCount = members.size();

			double sumIntensity = 0.0D;
			double sumRadius = 0.0D;

			for(Coord node : members){
				double x, y, z, rad, intensity;
				x = node.getX();
				y = node.getY();
				z = node.getOptimalZ();

				if(x < minX){
					minX = x;
				}
				if(y < minY){
					minY = y;
				}
				if(z < minZ){
					minZ = z;
				}
				if(x > maxX){
					maxX = x;
				}
				if(y > maxY){
					maxY = y;
				}
				if(z > maxZ){
					maxZ = z;
				}

				rad = node.getRadius();
				intensity = node.getAvgIntensity();

				sumIntensity += intensity;
				sumRadius += rad;

			}
```

```
		avgIntensity = sumIntensity / memberCount;
		avgRadius = sumRadius / memberCount;
	}

	public double[] getFragStats(){
		double[] stats = new double[9];

		stats[0] = memberCount;
		stats[1] = minX;
		stats[2] = minY;
		stats[3] = minZ;
		stats[4] = maxX;
		stats[5] = maxY;
		stats[6] = maxZ;
		stats[7] = avgIntensity;
		stats[8] = avgRadius;

		return stats;
	}

	public LinkedList<Coord> getBestSeeds(){
		LinkedList<Coord> bestSeeds = new LinkedList<Coord>();

		Collections.sort(members, new CoordSortRad());

		if(members.size() > 1){
			double percentage = 1.0;

			if(members.size() > 10){
				percentage = 0.8;
			}else{
				percentage = 0.9;
			}

			int numberToReturn = (int) (members.size() * percentage);

			if(numberToReturn == 0){numberToReturn = 1;}

			for(int i = 0; i < numberToReturn; i++){
				bestSeeds.add(members.get(i));
			}
		}else{

		}
		return bestSeeds;
	}
}
```

TypeSorter.java

```
import java.util.LinkedList;
public class TypeSorter {
	private LinkedList<Coord> singletonPoints;
	private LinkedList<Coord> endPoints;
	private LinkedList<Coord> midPoints;
	private LinkedList<Coord> bifurcationPoints;
	private LinkedList<Coord> blobPoints;
	public TypeSorter(){
		//
		singletonPoints = new LinkedList<Coord>();
		endPoints = new LinkedList<Coord>();
		midPoints = new LinkedList<Coord>();
		bifurcationPoints = new LinkedList<Coord>();
		blobPoints = new LinkedList<Coord>();
	}
	public void sortTypes(LinkedList<Coord> input){
		singletonPoints.clear();
		endPoints.clear();
		midPoints.clear();
		bifurcationPoints.clear();
		blobPoints.clear();
		for(Coord coordinate: input){
			int numChildren = coordinate.getConnectedSeeds().size();
			switch(numChildren){
			case 0:
				// no connections
				singletonPoints.add(coordinate);
				break;
			case 1:
				// single connection = end point
				endPoints.add(coordinate);
				break;
			case 2:
				// middle point
				midPoints.add(coordinate);
				break;
			case 3:
				// possible bifurcation
				bifurcationPoints.add(coordinate);
				break;
			default:
				// more than 3 = blob
				blobPoints.add(coordinate);
				break;
			}
		}
	}
	public LinkedList<Coord> getSingletons(){
		return singletonPoints;
```

```
	}
	public LinkedList<Coord> getEndPoints(){
		return endPoints;
	}
	public LinkedList<Coord> getMidPoints(){
		return midPoints;
	}
	public LinkedList<Coord> getBifurcations(){
		return bifurcationPoints;
	}
	public LinkedList<Coord> getBlobs(){
		return blobPoints;
	}
}
```

CoordSortBright.java

```
import java.util.Comparator;
public class CoordSortBright implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getIntensity() - c2.getIntensity());
	}
}
```

CoordSortRad.java

```
import java.util.Comparator;
public class CoordSortRad implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getRadius() - c2.getRadius());
	}
}
```

CoordSortInt.java

```
import java.util.Comparator;
public class CoordSortInt implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getAvgIntensity() - c2.getAvgIntensity());
	}
}
```

CoordSortX.java

```
import java.util.Comparator;
public class CoordSortX implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getX() - c2.getX());
	}
}
```

CordSortY.java

```
import java.util.Comparator;
public class CoordSortY implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getY() - c2.getY());
	}
}
```

CoordSortZ.java

```
import java.util.Comparator;
public class CoordSortZ implements Comparator<Coord>{
	public int compare(Coord c1, Coord c2) {
		return (int)(c1.getOptimalZ() - c2.getOptimalZ());
	}
}
```