Automatic Neurites Tracing and Neurons Counting

Daniele Fattò Offidani

Department of Physics

University of California Santa Barbara

Supervisor

Samantha Daly, Ph.D.

In partial fulfillment of the requirements for the degree of

Bachelor of Science in Physics

June 12, 2021
Name   Signature   Date

Savonette Daly   

Advisor

Committee Chair
Acknowledgements

First, I would like to thank Professor Samantha Daly for giving me the chance of working on such an interesting project, even during a global pandemic, and for her continuous guidance throughout my undergraduate and research career. I would also like to thank Professor Christian Franck at University of Wisconsin for making this entire work possible. Moreover, I would like to acknowledge and thank everyone that supported and cheered me throughout my research experience, starting from the Daly Group. Particularly, I want to thank Dr. Neal Brodnik for his never ending advice and continuous help in brainstorming and troubleshooting my ideas, at times even outside of working hours. I would like to thank Luke Summey from Franck’s group for his continued availability during the past year and for his help in understanding the biological concepts surrounding this research project. In conclusion, I would like to thank my family and all the people I have met at UCSB during my undergraduate education. You made me enjoy every single moment of it.
Dedicated to my mother, father and brothers.
Contents

1 Introduction ........................................... 1
   1.1 Traumatic Brain Injury .......................... 1
   1.2 Neurons, Neurites and Astrocytes ............... 2
   1.3 Soft Matter Mechanics And Strain Mapping .......... 3
       1.3.1 Continuum Soft Matter Mechanics .............. 3
       1.3.2 Local Strain Mapping ....................... 4

2 Methods ............................................ 6
   2.1 3D Dataset Growth and Gathering ................ 6
   2.2 3D Image Thresholding ........................... 7
       2.2.1 K-means and MiniBatches .................... 8
       2.2.2 K-number determination and thresholding .... 8
   2.3 Soma Erosion and Labeling ...................... 11
   2.4 Cell Reconstruction and Post-Processing .......... 12
       2.4.1 Label Expansion ............................ 12
       2.4.2 NeuronStudio, Rayburst Algorithm and Neurite Segmentation ... 14
   2.5 Astrocytes Identification and Final Cell Counting .... 16
       2.5.1 Feature Identification ....................... 16
       2.5.2 Probability Distributions & Group Assignment .... 19

3 Results and Discussion ............................. 22
   3.1 Performance of Auto-Tracing Algorithm ............ 22
   3.2 Evaluation of Probability Mapping and Neuron Counting ... 28

4 Conclusions ........................................ 32
List of Figures

2.1 Example of Metrics used to evaluate KMeans clustering. Up from left to right: Elbow Method with Distortion, Elbow Method with Inertia and Calinsky-Harabasz Score. Down from left to right: Davies-Bouldin Score, Silhouette Score and Computation time. ........................................ 9

2.2 Examples of bad (b) and good (c) K-means based thresholding compared to the original grey-scale image (a). ....................................................... 10

2.3 Slice of binarized image after erosion (a) and after labeling the remaining clumps (b). ................................................................. 12

2.4 Slice of image after the reconstruction process. All the cells present in the dataset are now labeled, and it possible to extract single neurons .. 13

2.5 3D view of an exported neuron in ImageJ (a) and the corresponding traced skeleton (b). ................................................................. 15

2.6 Histograms of all features considered for astrocytes and neurons respectively. The plots also show the fitted Weibull distributions of each feature both for astrocytes and neurons (A copy of these plots is in Appendix B at higher scale). The area under the curve and of the histograms is normalized. 20

3.1 All the automatically traced neurons shown on top of the original dataset. 22

3.2 Output of locally mapped axial strain from MatLab code. ................. 24

3.3 XZ slice of output neuron. Rayburst Sampling traces multiple big nodes on the z-axis above and below the soma, thus elongating the overall output. 25

3.4 Examples of inaccuracies due to thresholding. From left to right: jagged edges (a), small holes in the cells (b) and spherical cellular debris (c). 26

3.5 Output of locally mapped axial strain from MatLab code following the post-processing steps to fix morphological inaccuracies. .................. 27
3.6 Slice of Probability Map. Lighter colors correspond to a less accurate guess, while for very dark colors the cell is almost certainly either a neuron or an astrocyte. ............................................................... 29

3.7 Distributions of probabilities that a given cell is an astrocyte or a neuron. The distributions are very peaked towards high probability values. The dashed line indicates 75% probability of attribution. ............................... 31

4.1 Maximum z-axis projection of two segmented cells that are connected by their branches. ................................................................. 33

A.1 Representation of simple shear. Red indicates original configuration while blue indicates the deformed configuration. ................................. 38

B.1 Histograms and corresponding fits for surface areas of astrocytes and neurons. 41
B.2 Histograms and corresponding fits for bounding box areas of astrocytes and neurons................................................................. 42
B.3 Histograms and corresponding fits for the first moment of inertia eigenvalues of astrocytes and neurons. ............................... 42
B.4 Histograms and corresponding fits for the second moment of inertia eigenvalues of astrocytes and neurons. ............................... 43
B.5 Histograms and corresponding fits for the third moment of inertia eigenvalues of astrocytes and neurons. ............................... 43
B.6 3D view of segmented datasets with traced spines (preprocessing version). 44
Chapter 1

Introduction

If I have seen further it is by standing on the shoulders of Giants.

Isaac Newton, 1675

1.1 Traumatic Brain Injury

Traumatic Brain Injury (TBI) is a serious condition that affects thousands of people across the world every year. In 2019, there have been more than 61,000 deaths related to TBI in the United States alone [1]. More that 200,000 cases of TBI require hospitalization every year, and in more than 70% of cases the patients do not improve within 5 years of the injury taking place. Since TBI is so widespread, and the medications and therapies currently available have limited success [2], [3], it is of the utmost importance to understand the mechanisms of how and why TBIs occur. To this end, researchers have dedicated their time and resources in investigating TBI from multiple angles. One of these approaches consists of investigating the damaging and rupture of axonal and dendritical branches in single neurons under extreme strains and strain rates.

Many experiments have been performed to investigate the damage thresholds in neurons both in one and two-dimensional geometries with good success. Moreover, thanks to the continuous improvement of computational tools and software, it is now possible to carry on stress-strain simulations on single neurons in three dimensions. However, there is little research being performed on the local mapping of strains on the individual branches of neurons [4], in part due to the variety of challenges presented by locally mapping strains on living structures at the micrometer scale in three dimensions. A very high
voxel spatial resolution is required to produce any kind of reliable simulation. Moreover, the neurons’ capability of making connections with each other can often render the isolation process for modeling purposes tricky if at all possible. Therefore, the representation of isolated 3D neuron models is currently the bottleneck of this type of research.

Recent work by Professor Christian Franck’s group at University of Wisconsin-Madison has employed local mapping of strains in the investigations of damage thresholds in neurons[5]. The current, state-of-the-art process for creating reliable models of neurons requires 3D imaging of cellular cultures followed by a subsequent structure-tracing step. This segmentation process has been carried out manually in NeuronStudio, a program designed for this purpose at the Icahn School of Medicine at Mount Sinai by the Computational Neurobiology and Imaging Center [6]. For a single dataset, it could take up to many hours to obtain a sufficiently accurate three dimensional representation of multiple neurons so that the strains can be mapped. For this reason, the goal of this project is to design a semi-automatic program in Python that uses features of NeuronStudio to reliably and rapidly output three-dimensional neuron models from images gathered via multi-photon confocal microscopy. Moreover, the program should clean the originally gathered datasets of all the cellular debris and unwanted non-neuronal cells and indicate the number, position, and type of all the cells present. Finally, it should lay the foundation for future work in terms of improvement of existing tasks and the automatic separation of neurons that are either connected or whose branches are interlaced.

1.2 Neurons, Neurites and Astrocytes

Before delving in the core of this work, it is important to define some terms that might be unfamiliar to general audiences. The work that is being done on Traumatic Brain Injuries currently focuses on the damage occurring to the neurons and their branches, which can lead to necrosis and cellular death. The branches that stem from the body of the neurons, termed the soma, are called neurites. Neurites is an umbrella term that includes the dendrites, the axons, and in general any kind of neuronal process or projection that starts from the neuronal somas. It is often used when referring to young neurons, when it is not possible to distinguish the axons from the dendrites due to the early stage of cellular growth.
Among the cells that grow in the collagen cultures, there are some that are not neurons. Once the neuronal somas are isolated from the brain cortex of rats, other cells can make their way through to the culture. These cells are called astrocytes, or astrocytic glial cells. Astrocytes are different from neurons both in their appearance and in their function. For instance, they provide nutrients for the neural tissues, they maintain extracellular ion balance, and they provide support during the repairing processes after traumatic damage to the spinal cord[7]. Moreover, astrocytes do not have axons, but they have branches that are morphologically (and functionally) similar to the neuronal dendrites. For the purposes of our project, we want to identify these unwanted cells and eliminate them from the 3D dataset.

1.3 Soft Matter Mechanics And Strain Mapping

1.3.1 Continuum Soft Matter Mechanics

It is not the purpose of this work to define the mathematics used to map strains onto the neurites. Nonetheless, it is worth noting how these strains are calculated and how the output obtained at the end of the segmentation is used. In the domain of simple shear deformation, where the displacement of the deformation plane and the distance between planes are respectively $\delta$ and $L$, the angle formed between the deformed and undeformed plane is:

$$\gamma = \tan \frac{\delta}{L}$$

(1.1)

In order to rigorously define a deformation, two different deformation tensors are used. The Greene-Lagrange strain tensor ($E_{GL}$) describes the deformation with respect to the axes of the undeformed configuration, while the Euler-Almansi strain tensor ($e_{EA}$) defines it with respect to the axes in the deformed configuration. The two stress tensors can be expressed in matrix form as follows:
1.3 Soft Matter Mechanics And Strain Mapping

\[
E_{GL} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & k \\
0 & k & k^2
\end{bmatrix}, \quad e_{EA} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & k \\
0 & k & -k^2
\end{bmatrix}
\]

with \( k = \tan \gamma \). Since we are operating on a visco-elastic material, it is appropriate to work out the strain rate tensors as well. Since the only time-dependent component that affects the strain tensors is \( \delta(t) \), the stress rates are obtained by directly differentiating the strain matrices. We have \( k(\delta) \), and hence \( k(t) \). Then \( \frac{d}{dt}k = \dot{k} \) and \( \frac{d}{dt}k^2 = 2k\dot{k} \).

Plugging this in the strain tensors yields:

\[
\frac{d}{dt} E_{GL} = \dot{E}_{GL} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \dot{k} \\
0 & \dot{k} & 2k\dot{k}
\end{bmatrix}, \quad \frac{d}{dt} e_{EA} = \dot{e}_{EA} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \dot{k} \\
0 & \dot{k} & -2k\dot{k}
\end{bmatrix}
\]

Note that we are working with the variable \( k \) instead of \( \delta \) for simplicity. We can define a value of \( k \) and \( \dot{k} \) for the tensors based on a far-field magnitude approximation, rather than working with the \( \delta \) and \( v \), with \( v \) being the displacement rate (speed of deformation). For reference, the explicit tensor matrices, both with and without small angle approximation, are shown in Appendix A.

1.3.2 Local Strain Mapping

Using the basic tools explained in the previous section, it is possible to map strains onto the neuronal branches. First, it is necessary to build a skeletonized morphological representation of the neuron in three-dimensional space. Up until now, this has been done in NeuronStudio with a combination of automatic and manual neuron tracing, and is currently the bottleneck of the entire mapping process. Once the skeletonized image is obtained, it will look as a sequence of nodes connected by line segments. The line

\footnote{A full derivation of the tensors is worked out in Appendix A}
segments and nodes are subsequently rearranged so that they are all of the same length, without modifying the overall morphological structure of the neuron. This is a necessary step needed for the mapping process.

With the nodes equally spaced between each other, we can define a local orthonormal vector basis between each pair of nodes. The basis will be formed by a unit vector oriented in the same direction as the line segment connecting the nodes. The other two basis vectors will be both mutually perpendicular and perpendicular to the line segment. These vector basis is called a Frenet basis, and it is used to construct the rotation matrix to locally map the strain and strain rate tensors onto the branch segment of interest using the continuum mechanics mentioned in the previous section.
Chapter 2

Methods

2.1 3D Dataset Growth and Gathering

Before imaging the datasets needed for the modeling, it is necessary to grow a culture of neuronal cells in-vitro in a collagen matrix. The living cells are directly obtained from the brain of neonatal rats. This operation is carried out by making them unconscious through hypothermia and extracting pieces of their brain cortexes [5]. The brain cortexes obtained in this way are then isolated and submerged in a solution of Papain at 37° for 30 minutes, so that the neuronal cells get detached from the supporting structures. The cells remaining after this step are collected in a pellet and are centrifuged and filtered twice until only individual somas are left. The pellet is subsequently diluted in cortical complete media in the proportions needed for the desired cellular concentration [8]. Next, the solution is mixed with a Collagen-I hydrogel solution. After the polymerization process, the neuronal cells are uniformly distributed across the collagen matrix, and they can grow to the desired size. In the collagen matrix configuration, the cellular neurites are free to grow in all directions because of the collagen fibers that support them [8].

When the cells are ready for imaging, the collagen culture is stained with the fluorescent compound (CO)CH$_2$OH, commercially known as Neuron Orange or NeuO. This compound absorbs light of wavelength $\lambda_{ab-max} = 470$ nm and emits a wavelength of $\lambda_{em-max} = 555$ nm [9]. NeuO can be absorbed from living neuronal cells without damaging them, and thus allows for the collection of living neurons dataset by means of multi-photon confocal microscopy. Confocal microscopes and the fluorescence of NeuO staining enable the collection of multiple two dimensional images of a given living dataset.
at fixed depths. This in turn allows for the construction of a three-dimensional image, by stacking subsequent 2D FOVs. The output is then formatted in a .tiff hyper-stack file and is ready for the subsequent segmentation steps. The stack will have different planar and depth micron-to-pixel ratios, depending on the spacing between the images collected, which will have an effect on the final outputs. All the datasets used in this project were gathered and generously shared by Professor Christian Franck’s research group at University of Wisconsin.

2.2 3D Image Thresholding

The confocal laser microscopy outputs a three-dimensional figure formed by a stack of multiple two-dimensional layered fields of view. Inevitably, as each FOV contains information about the underlying ones, the image output will be gray-scaled. Performing morphological analysis on 3D, grey-scale datasets presents a challenge in itself, and it is generally easier to work with binary versions of the grey-scale images. This requires the use of a thresholding algorithm that is both accurate in the representation of all the cellular features in the dataset and does not add any artificial voxels in the binarized version of the image. The thresholding process is further complicated by the nature of the 3D dataset, and by the variation in average image intensity present among subsequent 2D planes of the 3D stack.

Thresholding algorithms commonly used in image processing and computer vision, such as Otsu and Huang methods [10], [11], perform well on the two dimensional slices of the 3D image dataset, but are not consistent at different depth levels because of the aforementioned differences in the backgrounds’ intensities between different slices [12], [13]. For this reason, a different thresholding method involving machine learning was implemented.

Raw k-means clustering is computationally taxing for large datasets. Considering a minimum 1024x1024 resolution of a single FOV and a depth size ranging from 100 to 200 pixels, the amount of data points to cluster in an average dataset is in the order of $10^7$. The computation time for a single iteration of k-means clustering computer can take up to one hour for convergence. We implemented a k-means mini-batches approach to reduce the computation time without sacrificing precision. The number of batches as
well as reassignment ratios were heuristically determined by comparing the minibatches outputs with raw k-means ones.

2.2.1 K-means and MiniBatches

K-means clustering is a widely used clustering method. Its function is the one of partitioning a given dataset in a $k$ numbers of clusters [14]. To this end, each data point in the dataset is represented as a vector in a n-dimensional space, where $n$ is the number of the data-set’s characteristic features. A k-means algorithm works on the premise of minimizing the variance of the datapoints present within the same cluster across the entire set of clusters:

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

where $C$ is the set of clusters partitioned, $k$ is the number of clusters, $x$ is a vector present in a partition and $\mu_i$ are the means of each cluster, the commonly so-called centroids [15].

The computation time for normal k-means algorithms scales as $O(kns)$ with $k$ being the number of clusters, $n$ being the total number of datapoints to cluster and $s$ being the number of features per data vector [16]. For this reason, it is taxing for larger datasets and alternative techniques are required. The mini-batches optimization helps in maintaining a low computation time for large dataset. Essentially, when optimising a k-means algorithm with mini batches, instead of sampling the entire dataset at every iteration, only a random subset of all of the data vectors is sampled. At every iteration, the subset randomly changes, and updates the centroids of the previous iteration using the new batch. The process continues until convergence is reached. Although mini batches save computation time by sacrificing accuracy, the practical difference in the output clusters was negligible in the thresholding process, and the real time needed for the thresholding was scaled down from one hour to 10-20s$^1$.

2.2.2 K-number determination and thresholding

There is no standard way to uniquely determine the number of clusters for K-means clustering. Many metrics have been developed to judge the quality of clustering. Nonetheless, not all of these metrics work at all times, and there is an art to choosing the correct num-

$^1$All the specifics of the computer used for this project are in Appendix C
ber of clusters to suit ones needs. This was also the case in our approach to thresholding. In our case, there were two variables under our control:

- k, the actual number of clusters
- which clusters to label as background, and which to label as cell matter

The commonly used metrics were not advantageous. Giving an in depth look at Fig 2.1 we can see that the two elbow metrics employed did not help as much. What we would be looking for in that kind of plot is for a point where the slope of the plot stabilizes (an elbow) [17], [18]. However, both in the case of distortion and inertia values, there is no such point clearly visible. Distortion is the average of the square distances of each point in a cluster to the corresponding cluster centroid. On the other hand, inertia is the sum of the square distances from each data point to the corresponding centroid. Both are used as a measure to determine how compact the clusters are, and an elbow point in the distortion or inertia plots would indicate that there is no need to further increase the k-number.

The Calinski-Harabasz score measures the ratio of the similarity of each datapoint to the others in its cluster, and to its difference from the ones in the other clusters. Ideally, a

![Figure 2.1](image_url)

**Figure 2.1** Example of Metrics used to evaluate KMeans clustering. Up from left to right: Elbow Method with Distortion, Elbow Method with Inertia and Calinsky-Harabasz Score. Down from left to right: Davies-Bouldin Score, Silhouette Score and Computation time.
good value of k would be one where the CH score is significantly higher than the ones of the other k-values, indicating denser and more well-defined clusters [19]. Similarly, the Silhouette score measures the difference between the intra-cluster similarity and extra-cluster difference, and normalizes it. Hence, a good Silhouette score for clustering would be close to 1, while a bad one would be close to -1 [20]. Finally, the Davies-Bouldin index is a score based not on the samples, but on the overall clusters. It is computed by taking the pairwise ratio of the sum of the inner spreads of two clusters divided by their respective distance from each other. A low value of Davies-Bouldin index would indicate clusters that are well separated and dense [21].

Even these three metrics are not really helpful in judging what the best cluster is, as visible from Fig 2.1. While Davies-Bouldin would suggest that the K-Numbers 3 and 7 are better than the others, the Silhouette scores do not agree with this and indicate that 2 is the right number of clusters. However, by selecting 2 as our k-value, we obtain an output that is similar to what a Otsu thresholding would yield, with much of the details of the neuronal branches and the 3D interconnection information lost in the process.

Figure 2.2 Examples of bad (b) and good (c) K-means based thresholding compared to the original grey-scale image (a).
2.3 Soma Erosion and Labeling

Instead, we choose to run through various values of k and clusters to remove and stop when the desired accuracy of details is reached, which produced better results. This process is the only manual part needed to jump start the automatic segmentation, and it is a pre-processing step that is needed for a better morphological reconstruction of the image. Generally, we found that a k-value of 6 provided sufficient detail across all images, with the only variable being the number of clusters to label as background. If the wrong number of clusters is chosen, the error is apparent and can be immediately fixed, as shown in Fig 2.2. Moreover, it was found that when the right k-values are set, they hold for all datasets gathered under the same conditions, with no further refinement needed.

2.3 Soma Erosion and Labeling

Once the binary mask for the dataset is obtained, the process of cell counting and labeling begins. It is impossible to immediately identify the number of single cells present within a dataset from the raw image. The vast amount of cellular debris and the fact that neurons interact with each other via their dendrites prevents any sort of counting based on isolated structures in the input array. The most evident geometrical feature of the neurons is their somas, or the body of the cell. Thus, an intuitive approach of counting cells would be to isolate the somas present in the dataset and count them. We achieve this through a morphological erosion algorithm.

Morphological erosion works by superimposing on each voxel of the original image a pre-determined structuring element. Then, if the entire structuring element centered at a specific voxel is contained in the image, that voxel will not be eroded. Conversely, if the structuring element is not entirely contained, the voxel being parsed will be eroded. Assuming that an average cell soma has a spherical shape and considering the difference in depth resolution with respect to planar resolution, the most suitable structuring element for the erosion process is an ellipsoid. Setting the size of the ellipsoid requires prior knowledge of the average radius size of the somas present in the dataset. With that information, it is possible to determine the length of the horizontal semi-axes for the ellipsoidal structuring element and compute the vertical semiaxis’ length. Defining $a, b$ and $c$ to be respectively the semiaxes oriented along the x, y and z axes and $Res_{x,y}$ and
2.4 Cell Reconstruction and Post-Processing

Figure 2.3 Slice of binarized image after erosion (a) and after labeling the remaining clumps (b).

Res\textsubscript{z} the planar and depth micron-to-pixel ratios:

\[ c = \frac{\text{Res}_{x,y}}{\text{Res}_z} \times a \]

Since some neurons might have smaller somas than average, it is more appropriate to use a reduced version of the ellipsoid for erosion purposes. For this reason, all the semi-axes of the ellipsoid are shortened by 25%. Once the structuring element is determined, the erosion process is performed. The original binary mask is reduced to a set of small clumps of voxels centered at the somas present in the image, while the axons, dendrites, and all the cellular debris are eroded. At this point, it is possible to start the counting. Every separated clump of voxels is considered as a singular entity and is labeled with an integer value, as shown in Fig 2.3. These labeled clumps are the starting points for the reconstruction process.

2.4 Cell Reconstruction and Post-Processing

2.4.1 Label Expansion

The erosion and labeling process reduce the original binary mask to a group of colored chunks. We then need to reconstruct the original image by expanding each clump until
it obtains its original shape. To this end, we implemented a reconstruction loop divided in two parts. The first part consists in the label expansion. The program selects one of the labeled clumps and determines the position of its borders. Then, it expands the clump in every direction by a voxel, while keeping the same label value. The second part consists in an erosion step that ensures that no artificial voxels have been created. This is accomplished with an element-by-element matrix multiplication between the binary mask obtained at the beginning and the expanded labeled image. Every voxel created at the same coordinate position of background voxels in the original image gets eroded, leaving only accurate representations of the neurons. These two steps are carried on in a loop until the reconstructed image of the $n^{th}$ iteration of the loop is identical to the one of the $n^{th}-2$ iteration, which means that no new voxels will be reconstructed in future iterations. This is the most time-consuming step in the entire algorithm, and requires from 40-150 minutes of computation time. This slowness is to be attributed to a specific function in the scipy.ndimage package called distance_transform_edt, which essentially performs on the image an exact Euclidean distance transform. Basically, for every label present in the labeled image, the function computes the distance from each voxel in the label to the background, and assign that distance value to the voxel. This is needed to determine where to perform the label expansion [22]. Considering that the function parses every labeled voxel in a dataset of hundreds of millions, and it can take up to 300 iterations of

![Figure 2.4](image.png)
2.4 Cell Reconstruction and Post-Processing

the reconstruction loop to complete the process, the extensive computation time makes sense. Work is currently ongoing to reduce the computation time by subdividing the dataset at every iteration, thus shortening the total computation time, and by looking at alternatives for this operation.

At the end of the reconstruction process, we obtain the original dataset without any cellular debris and with each neuron labeled with a different value, as shown in Fig 2.4. At this point, the export process begins. Depending on whether they are touching other neurons or not, each cell is classified as connected or isolated. Isolated cells are directly exported as tiff stacks in two different ways: first, they are exported in a box that contains only the cell, and second, in a stack that has the same size of the original dataset, such that the information about the position of the neuron relative to the others is conserved. This will be needed later for composing the final skeletonized image.

2.4.2 NeuronStudio, Rayburst Algorithm and Neurite Segmentation

Once the singled neurons have been exported they can be uploaded to NeuronStudio, a program designed by the Computational Neurobiology and Imaging Center at the Mount Sinai School of Medicine that is geared towards the semiautomatic segmentation of neuronal structures. NeuronStudio has been used as a manual tracing software for neuronal structures in Professor Franck’s group up until now. However, one of its strong suits is the built-in automatic segmentation function. While the auto segmentation does not perform accurately on full datasets, it instead works well if single neurons are imported and the right parameters are set. The auto segmentation function is based on the Rayburst Sampling.

Rayburst Sampling consists of propagating an N number of equally spaced rays from a sampling core, either in 2D or 3D, and to obtain information as these rays interact with an image, thought as a mosaic of intensity values [23]. Generally, the intensity values at any point in space are not determined, as only specific points in the 3D grid that composes the image contain the intensity information. By means of piece-wise trilinear interpolation[6], however, it is possible to estimate the intensity everywhere in space based on the finite number of surrounding intensity data points (the vertices of the lattice making up the grid). As the rays propagate through space they will cross multiple intensity values, and
2.4 Cell Reconstruction and Post-Processing

![Figure 2.5](image)

**Figure 2.5** 3D view of an exported neuron in ImageJ (a) and the corresponding traced skeleton (b).

from the gradient of intensity it is possible to determine the boundaries (faces of exit) of 3D structures within the image. It is important to note that Rayburst Sampling is only effective when applied to convex 3D structures, as it is possible to univocally determine the faces of exit in this case [23].

Among the important information that can be computed with Rayburst Sampling, the most relevant to this work is the length of the diameter of tubular structures in space. The way that NeuronStudio skeletonizes the dendrites and axons is based on the interpretation of these branches as a succession of cylinders with varying radius at each node. Rayburst sampling can accurately estimate the diameter of a dendritic section by computing the median lower band diameter (MLBD) of the circular cross section perpendicular to the axis of the cylindrical structure. MLBD is computed by sorting the lengths of all the rays being sampled (each ray vector is added with the one going in the opposite direction, so there are N/2 lengths to sort) and choosing the distance indexed at position N/8 to be the median of the lower band (inferior half of the indexed distances) [6]. This allows us to obtain the tubular diameter of the branching structures with good confidence and in a way that is resistant to local irregularities in the image, thus aiding NeuronStudio in the search for cylindrical structures.

Once all the neuronal ramifications have been identified, NeuronStudio creates a skeletonized version of the neurons themselves, as shown in Fig 2.5 and outputs them as an
2.5 Astrocytes Identification and Final Cell Counting

2.5.1 Feature Identification

Another goal of this automatic algorithm is the composition of an image that has been cleaned of all the cellular debris as well as everything that is not part of neurons. This includes also the astrocytes that are formed during the cell growth process. The NeuO stain used for the microscopy purposes is designed to only affect living neuronal cells. Nonetheless, it is the case that also astrocytical somas absorb this substance. While this compromises the direct counting of the cells in the dataset, we devised a method of removing the astrocytes after the labeling process was carried out.

The astrocytes visible in the 3D images are distinguishable from the neurons for two reasons:

- Astrocytes have a notably inferior number of branches with respect to neurons
- NeuO seems to only stain the body of astrocytes, and is minimally absorbed by their branches

Because of this, only the somas of the astrocytes appear in the images together with a couple of attached and shortened dendrites. Thus, although these cells are not eliminated during the erosion process because their somas have roughly the same size of neuronal ones, it is possible to threshold them out based on their distinctive features.

We gathered all the morphological metadata that could be extracted from the output images. In order to make a generalized threshold across multiple dataset with different resolutions, all the morphological measurements done on the samples were normalized in standard measuring units. In particular, the lengths, surface areas, and volumes of the cells were gathered in $\mu m$, $\mu m^2$ and $\mu m^3$ respectively instead of pixel and voxel units. Since in most of the cases there is a difference in resolution in the z axis with respect to x and y in the original dataset, a spline trilinear interpolation was used to reformat
the input arrays so that the micron-to-pixel ratio was constant across all dimensions. 5 distinctive morphological feature vectors have been identified:

- The surface area of the cell, measured in \(\mu m^2\) and computed using a marching cubes algorithm
- The 3 eigenvalues of the moment of inertia tensor
- The surface area of the box containing the entirety of the cell.

The reason to include the surface area as a metric is as follows: the branching structures stemming from the somas of the neurons add a considerable amount of surface area to the cell. Since astrocytes have many less branches than the neurons, it follows that their surface area will generally tend to be inferior.

A similar argument can be made for the eigenvalues of the moment of inertia tensor. The magnitude of the eigenvalues will be directly proportional to the amount of mass (or voxels in our case) and to the distance of the mass from the center. More branches will then inflate the magnitude of the eigenvalues, as they carry more morphological mass at distances far from the center of the neurons (i.e. they are more elongated in the directions of the principal axes). The surface area of the bounding box is an indirect measure of how much the cell spans in 3D space, which takes into account how all the branches stemming from the somas propagate both in the x,y and z direction. The bounding box areas is easily computed by counting the number of external pixels composing the box and then converting them in \(\mu m^2\). On the other hand, the calculation of the cellular surface area and moment of inertia values is not as intuitive.

The surface area was computed firstly by creating a surface mesh of the cell. The polygonal mesh itself was obtained employing a marching cubes algorithm. Marching Cubes works by representing the image as a set of cubes and then fitting a triangular mesh withing them based on their positions and intensity values. This process transforms the original image in a polygon formed by a multitude of connected triangles of identical shape, thus faithfully approximating the surface of the 3D object. The value of the surface area is then computed by counting the number of triangles composing the surface mesh and multiplying them by the surface value of a single triangle (in pixel units). The computed area is then converted to \(\mu m^2\), knowing the micron-to-pixel ratio of the input image. In order to compute the eigenvalues of the moment of inertia tensor, we first
2.5 Astrocytes Identification and Final Cell Counting

need to construct the covariance matrix made up by second order central moments of the image:

\[
\text{cov}[\mathbf{I}(x, y, z)] = \begin{bmatrix}
\mu'_{200} & \mu'_{110} & \mu'_{101} \\
\mu'_{110} & \mu'_{020} & \mu'_{011} \\
\mu'_{101} & \mu'_{011} & \mu'_{002}
\end{bmatrix}, \quad \mu'_{ijk} = \sum_{x, y, z} f(x, y, z)(x^i)(y^j)(z^k)
\]

where \( f(x, y, z) \) is the intensity value of the image \( \mathbf{I} \) at point \( (x, y, z) \), and we have centered the origin of the axes at the centroid of the image. We can see how this matrix relates to the moment of inertia matrix [24].

\[
\mathbf{J} = \begin{bmatrix}
J_{11} & J_{12} & J_{13} \\
J_{21} & J_{22} & J_{23} \\
J_{31} & J_{32} & J_{33}
\end{bmatrix}, \quad J_{ij} = \sum_{k=1}^{N} m_k (x^2 + y^2 + z^2) \delta_{ij} - \sum_{k=1}^{N} m_k (x^i)(y^j)(z^k)
\]

where \( \delta_{ij} \) is the Kronecker delta. This is the physical definition of the inertia tensor. If we substitute the physical point mass with the intensity function we can now compare the diagonal and off diagonal elements of the two matrices:

\[
\mu'_{200} = \sum_{x, y, z} f(x, y, z) x^2 \quad \& \quad J_{11} = \sum_{x, y, z} m(x, y, z) (y^2 + z^2)
\]

\[
\mu'_{110} = \sum_{x, y, z} f(x, y, z) xy \quad \& \quad J_{12} = -\sum_{x, y, z} m(x, y, z) xy
\]

From these two equations, it is evident that the matrices are highly similar and must be related. The Inertia matrix is equivalent to:

\[
\mathbf{J} = \text{tr}(\mathbf{C}) \mathbf{I} - \mathbf{C}
\]

where \( \mathbf{I} \) is the identity matrix and \( \text{tr}(\mathbf{C}) \) is the trace of \( \mathbf{C} \). Knowing the eigenvalues \( \lambda \) and eigenvectors \( \mathbf{v} \) of \( \mathbf{C} \), and that \( \text{tr}(\mathbf{C}) \mathbf{v} = \text{tr}(\mathbf{C}) \mathbf{I} \mathbf{v} \), we can reach the conclusion that:
Hence, if we compute the covariance matrix and its eigenvalues we can also compute the eigenvalues of the Inertia matrix: $\lambda_{in} = tr(C) - \lambda_{cov}$. The moment of inertia eigenvalues for the images have units of [intensity pixel$^2$], which can be converted to metric units as [intensity µm$^2$] by multiplying them by the micron-to-pixel ratio. Note that the dimensions of intensity in this case do not have a real physical interpretation. Rather, they are adimensional and therefore the result could be interpreted in units of length squared.

2.5.2 Probability Distributions & Group Assignment

After running the segmentation code on multiple datasets, it was possible to visually group two sets of cells representative of exemplar neurons and astrocytes. Both sets contained more than 40 cells (42 neurons and 81 astrocytes). These two sets were used to model two statistical probability density functions for each one of the morphological features described in the previous section. Instead of modelling the distributions from scratch, it was decided to examine the features of the distributions and try to fit them to well-established pdfs. In all cases, the distributions of the cellular features appear to follow a curve slightly skewed towards higher feature values, suggesting the use of a Gamma distribution. Instead of using the normally defined Gamma distribution, however, it was appropriate to fit the data to an exponentiated Weibull distribution. Exponentiated Weibull functions were chosen for two reasons: first, they represent a generalized version of the Gamma distribution, thus providing more freedom in fitting the available data; and second, although morphological features of neurons and astrocytes have been rarely examined statistically, it was shown in prior work that their cross sectional areas can be assumed to be a Weibull’s random variable [25], a special case of the exponentiated Weibull distribution.

Weibull probability density function follows the formula [26], [27]:

$$f(x, k, \lambda, \alpha, \theta) = \frac{\alpha k}{\lambda} \left(\frac{x - \theta}{\lambda}\right)^{k-1} \left[1 - \exp\left(-\frac{x - \theta}{\lambda}\right)^k\right]^{\alpha-1} \exp\left(\left(\frac{x - \theta}{\lambda}\right)^k\right)$$
Figure 2.6 Histograms of all features considered for astrocytes and neurons respectively. The plots also show the fitted Weibull distributions of each feature both for astrocytes and neurons (A copy of these plots is in Appendix B at higher scale). The area under the curve and of the histograms is normalized.

where \(k\) and \(\alpha\) are the first and second shape parameters respectively (the exponents characterizing the behavior of the function), \(\lambda\) is the scale parameter that governs the spread of the distribution, and \(\theta\) is the location parameter, that determines the start of the distribution.

Since all of the measured features did not have negative values, the location parameter was set at the origin, thereby reducing the exponentiated Weibull distribution to its 3-parameter version \(f(x, k, \lambda, \alpha)\), for faster and more accurate fitting. All the distributions were fitted by means of nonlinear regression, and the resulting fits modeled faithfully the discrete distributions, as shown in Fig 2.6.

The probability density functions obtained from the gathered cell sets were used to construct an algorithm that can determine the probability of a random cell of either being an astrocyte or a neuron. The calculation is based on the version of Bayes theorem for a
discrete random variable (the group that the cell is assigned to) and a continuous random variable (the morphological feature of the cell):

\[ P(A \mid x) = \frac{P(A) f(x \mid A)}{P(x)} = \frac{P(A) f(x \mid A)}{P(A) f(x \mid A) + P(N) f(x \mid N)} \]

\[ P(N \mid x) = 1 - P(A \mid x) = \frac{P(N) f(x \mid N)}{P(A) f(x \mid A) + P(N) f(x \mid N)} \]

Here \( P(A) \) and \( P(N) \) are the probabilities that a cell is an astrocyte or a neuron without any information about it, and they constitute our priors. The priors were estimated by taking the ratio of astrocytes and neurons used for the curve fitting to the overall number of cells. \( f(x \mid A) \) and \( f(x \mid N) \) are respectively the values of the probability density functions at \( x \) given that the cell is an astrocyte or a neuron. Finally, \( P(A \mid x) \) and \( P(N \mid x) \) are the probabilities that the cell belongs to the astrocyte group or a neuron group, given the feature value of \( x \).

These probabilities were computed for all the features of each cell, and then averaged to obtain the cumulative probability that the cell is either an astrocyte or a neuron. The threshold for grouping was set at 50% probability; however this might not be the optimum threshold (see Results and Discussion). Following the thresholding process and the elimination of all the neurons present at the boundaries, a final count of the good neurons in a dataset was obtained, concluding the counting process.
Chapter 3

Results and Discussion

3.1 Performance of Auto-Tracing Algorithm

After all steps of the algorithm are carried out, including a cycle of Rayburst Sampling and skeletonization within NeuronStudio, a typical output will look similar to what is shown.

Figure 3.1 All the automatically traced neurons shown on top of the original dataset.
in Fig 3.1. In that figure, a total of eight cells have been reconstructed and traced in 3D, in a total time span of less that 50 minutes. This alone is a tremendous improvement with respect to the time of manual tracing, as the process of visually identifying good neurons to trace and manually segmenting them from the original grey-scale images, even if aided by Rayburst sampling, can take up to 2-3 hours. Moreover, the amount of neurons that are present under ideal conditions for manual segmentation is similar to the ones obtained through the automatic segmentation pipeline.

From this standpoint, it is possible to manually obtain many more neurons to automatically trace from the reconstructed labeled dataset output. The code automatically detects which neurons are in contact with each other and can export them pairwise. Although this process is not currently automatized, one could potentially split the neurons if it was possible to visually detect where one ends and the other one begins. One possible way to address the splitting problem would be the development of a neural network capable of identifying neurons based on their morphological structure. The neural network would then be trained to separate two neurons based on their characteristic features.

Looking at the output of the strain mapping process, shown in Fig 3.2, it is possible to evaluate the automatic skeletonization process. Notably, almost all of the segmented cells appear to have a soma that is elongated in the z direction. This characteristic is attributed to the reduced micron-to-pixel ratio of the z axis of the confocal laser microscopy. This translates into an elongated morphological reconstruction of the neurons, as shown in the figure. NeuronStudio incorporates settings designed to account for much of this image distortion/smearing if the the correct pixel-to-micron ratios are input. Nonetheless, it appears that these functions are not enough to bypass this issue altogether. When running Rayburst Sampling on these distorted images, the neuronal soma’s elongated shape will be traced as the soma itself and an extra branch connected to it. Depending on the presence or absence of other branches originating above or below the soma, the additional point might be defined by the automatic segmentation process as either a splitting point or an additional neurite point (see Fig 3.3).

In addition to vertical distortions, there exist other smaller morphological inaccuracies in the output images (Fig 3.4). Some of these inaccuracies, such as jagged edges or small holes present in the branches, are caused by the thresholding process. Whereas the K-means approach employed is versatile and can be accurate, as explained in the
Figure 3.2 Output of locally mapped axial strain from MatLab code.
Methods section, it will still impact the binarized image, as do all thresholding algorithms. Particularly, the edges of the neurons are where there is the greatest intensity gradient, where pixels of medium-high intensity are surrounded by others of lower intensities, which will be thresholded away. This will manifest in jagged edges appearing in the final morphological reconstructions. Similarly, but less often, there may be some isolated darker-than-normal pixels present within parts of the neurons themselves, which will cause the holes that can be seen in Fig 3.4. The third type of defects look like spherical structures floating within the cell cultures. These and other smaller particles are cellular debris residual from the centrifuging process performed to separate the cells from the cerebral cortex.

These large structures are wiped away during the erosion process outlined above, and in the rare case that they surpass that step, they are clustered together with the astrocytes and are thresholded with them. Sometimes, neurons will happen to be located closely to these cellular debris, such that their branches will grow attached or going through them. In this case, they will be detected as parts belonging to the morphology of the neuron. They therefore may cause issues when Rayburst Sampling is used, as they may be recognized as very short branches that are usually oriented vertically. While this can introduce error in the analysis of the locally mapped strain, it is usually negligible compared with the overall number of all line segments used to compute the averaged strain.

The issues outlined above can be partially or completely corrected with post-processing performed before image exportation. The z-axis elongation occurring throughout the dataset is addressed by re-scaling the z-axis such that all three axes have the same micron-to-pixel ratio. To do this, the image is zoomed and fit into an array with the same dimensions in x and y, but with a reduced length of the z dimension. The image is
3.1 Performance of Auto-Tracing Algorithm

Figure 3.4 Examples of inaccuracies due to thresholding. From left to right: jagged edges (a), small holes in the cells (b) and spherical cellular debris (c).

compressed along the z-axis using a spline interpolation, which allows us to compute the intensity values of all the voxels in the reduced configuration starting from the known intensity values of the original image. Once the interpolation is carried out, the image is then newly thresholded to obtain a more realistic model of the neuron. The surface areas of the cells used to derive the distributions in the previous section were also computed using this approach. This was a necessary step to convert the values of the areas from pixels to $\mu m^2$, so that the distributions could be generalized and used for datasets with different image resolutions.

Small output errors, such as the jagged edges and small holes, can also be mitigated to a certain extent using a combination of Gaussian Smoothing and Median Filtering followed by re-binarization. Gaussian Smoothing (or blur) consists in computing an intensity transformation of every voxel in an image. To do so, every voxel in the image is taken as the center of a three-dimensional Gaussian distribution that follows the equation:

$$G(x, y, z) = \frac{1}{(2\pi\sigma^2)^{\frac{3}{2}}} \exp \left[ -\frac{x^2 + y^2 + z^2}{2\sigma^2} \right]$$
Figure 3.5 Output of locally mapped axial strain from MatLab code following the post-processing steps to fix morphological inaccuracies.
This can be interpreted as surrounding each voxel with a series of concentric spherical shells of diminishing intensity, distributed in a Gaussian fashion from the central point. All the spherical intensity distributions are then used to compute a weighted average of the intensity of all voxels, thus creating the "blur" effect. By choosing a small value of sigma it is possible to fill the small holes present (surrounded by bright voxels) without connecting neighboring branches and affecting the edges too much. On the other hand, the jaggedness of the cells’ boundaries is corrected using a median filter with a kernel of voxel size 3x3x3, which will set the intensity value of each voxel to be the median of the ones surrounding it. Since the boundary elements will be surrounded in similar proportions from background and foreground voxels, the median filtering will cause a light blurring that will make the edges of the branches look more continuous.

Application of all of the above measures will result in cleaner and more realistic output traced neurons, as shown in Fig 3.5 Nonetheless, some of these operations might cause inaccuracies themselves. For instance, compression along the z-axis of the 8th cell has caused the shortening of one of the branches and the cancellation of a smaller one. When the interpolation and rebinarization where carries out, one or two voxels of the lost branches were lost in the background. Since the branches were thin to begin with, the only point of connection between them and the neuron was eroded, which resulted in the mutilated output. On the other hand, the compression process also suppressed some small vertical branches that were artificial and present in the original output because of the stretched neuronal reconstructions.

3.2 Evaluation of Probability Mapping and Neuron Counting

The analysis devoted to differentiating between astrocytes and neurons produced a high success rate. The probability density functions mapping the distributions of the five feature vectors were used in combination with Bayes theorem to estimate the probability of a cell belonging to either the astrocytes or neuron groups. More than 300 cells were processed using this method, and the success rate of clustering was $\sim 92.54\%$ when compared with a visual examination of the 3D cell models. Cells were labeled astrocytes if the probability of them being astrocytes averaged over all feature vectors was more
3.2 Evaluation of Probability Mapping and Neuron Counting

than 50%. Conversely, they were labeled neurons if they had a weighted 50% probability of being neurons. This was set arbitrarily to obtain preliminary results, and it must be noted that more careful approaches in thresholding should be considered, as the results of the probability maps are only indicative of the likelihood of a cell of belonging to either group. In order to judge differentiation, each cell was mapped using a diverging color-map centered at 50% probability, so that it was possible to simultaneously see both the morphological structure and their probability of belonging to either group. A slice of such a map is shown in Fig 3.6.

A 93% accuracy in a dataset of 25-60 isolated cells corresponds to an absolute error of clustering varying from 1 to 5 cells. From visual inspection of the probability maps, it was found that most of the cells wrongly grouped corresponded to ones that had close to 50% probabilities of belonging to either group. One kind of cells that failed the probability test could either be neurons that were still in early stages of development (they had multiple branches, but were noticeably smaller than the rest of the cells) that were grouped as astrocytes. Moreover, huge masses of cellular debris floating in the collagen matrix can be clustered as neurons due to their large surface area, although they have more modest moment of inertia eigenvalues than the neurons themselves. The probability maps allow

![Figure 3.6 Slice of Probability Map. Lighter colors correspond to a less accurate guess, while for very dark colors the cell is almost certainly either a neuron or an astrocyte.](image-url)
for manual removal of these more challenging cells to group. Nonetheless, the inclusion of other, currently unknown, feature vectors could result in a more accurate clustering procedure. A possible feature to include might be the number of appendages stemming from the cells, although there is currently no direct way to include this in the function.

Furthermore, the thresholding process might be improved in the future. Instead of grouping each cell using a “more than 50%” threshold, it might be valuable to set an immediate threshold at 75% likelihood of belonging to either group instead. The distributions of the probabilities for each cell group, in fact, are highly peaked towards the higher probability values rather than lower ones. As shown in Fig 3.7, where the histograms of the probabilities are plotted, by setting a threshold at 75% probability we would still group correctly more than 80% of the astrocytes and 76% of the neurons, while the remaining cells would be assigned to a “grey” group and either manually or automatically sorted with different techniques. The 75% threshold indicated, however, might be furthered lower by carefully analyzing what is the lowest value where the automatic sorting actually fails.

Finally, another way to further improve the current sorting process might be to exclude the averaging step. Although this step is necessary to ensure that all parameters are considered, sometimes averaging might be misleading if a particular feature carries with it a probability of assignation to either group that significantly differs from the others. For instance, for the case of the huge clumps of cellular debris, the surface area of the cell had a much bigger impact on the overall probability of attribution that all the other features. Given that all the probability density functions employed were Weibull’s functions, it should be possible to construct a multivariate probability density so that the averaging step is skipped altogether. With a multivariate version of the pdf that spans over all the features of interest, we would require only one iteration of Bayes theorem to obtain the final probability. Alternatively, the averaging step can be modified by assigning a weight to each feature, although this would be arbitrary in nature, and therefore not as accurate.
Figure 3.7 Distributions of probabilities that a given cell is an astrocyte or a neuron. The distributions are very peaked towards high probability values. The dashed line indicates 75% probability of attribution.
Chapter 4
Conclusions

In this study, we developed an algorithm capable of reconstructing accurate three-dimensional representations of neurons from datasets gathered via confocal laser microscopy. The proposed goals of this project were:

- Shortening the overall time needed to manually trace neurites
- Developing a way to automatically distinguish astrocytes from neurons
- Produce an accurate estimate of the number of neurons present in a given dataset
- Build the bases for extracting 3D models of interacting neurons

The algorithm is successful in shortening the time needed for tracing neurites, and it does so in a completely automatic way, when the required preprocessing steps are completed. It was found that the total time needed to process a dataset and obtain the morphological skeletons varies from 50 to 150 minutes. On the other hand, the time invested to produce a similar output by manually tracing the neurons from the original is about 3 hours. This corresponds to roughly a 3-fold improvement in time consumption, and it also does not require manual intervention. Work is currently ongoing to further reduce the computation time, as it is mainly dependent on the morphological reconstruction of the cells. Moreover, a method for distinguishing neurons from astrocytes was also devised. By using the morphological data gathered via first part of the algorithm, it was possible to compute a conditional probability that a cell is a neuron or an astrocyte based on its features. The assigned probabilities are then used to build a probability map of the 3D image, where all the cells displayed are colored depending on their probabilities of being attributed to either group. Furthermore, this information is used to eliminate the
non-neuronal cells from the initial datasets. This, together with the erosion of cellular
debris and the elimination of boundary elements leads to a precise count of the neurons
present in a given dataset. Future work should focus around the improvement of the
statistical distributions used for the probability estimation and on the search for addi-
tional features that might be useful in distinguishing neurons from astrocytes. Better
averaging methods for the probabilities across all features should also be investigated,
and the construction of a multivariate distribution seems to be a viable way to research.
In conclusion, the algorithm that has been developed should ultimately serve as a basis or
a skeleton for further research projects. In particular, using the high-fidelity morphologi-
cal reconstructions of the cells, it would be possible to better investigate their shape and
size, and possibly statistically model these features. Moreover, it should also be possible
to build a neural network capable of deconvolving neurons whose branches either touch
or intersect, like the ones shown in Fig 4.1, since this process still requires manual work,
and it is not always possible to carry out.

![Figure 4.1](image)

**Figure 4.1** Maximum z-axis projection of two segmented cells that are connected by
their branches.
References


Appendix A

Additional Derivations and Formulas

A.1 Green-Lagrange and Euler-Almansi tensors

The Green-Lagrange and Euler-Almansi tensors for a given configuration can be obtained by first computing the right and left Cauchy-Green deformation tensors, which are respectively defined:

\[ C = F^T F, \quad B = FF^T \]

where \( F \) is the deformation gradient tensor which contains the information of the derivative of a vector in the deformed configuration with respect to the components of the vector in the original configuration [28]. Or, mathematically:

\[ F = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} \]

where \( \mathbf{x} \) and \( \mathbf{X} \) are the the same vector in the deformed and undeformed configurations respectively.

Under the assumption of homogeneous simple shear deformation (deformation in only one direction), we can look at the components of the deformation gradient tensor:

\[
F = \begin{bmatrix}
\frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \frac{\partial x_1}{\partial X_3} \\
\frac{\partial x_2}{\partial X_1} & \frac{\partial x_2}{\partial X_2} & \frac{\partial x_2}{\partial X_3} \\
\frac{\partial x_3}{\partial X_1} & \frac{\partial x_3}{\partial X_2} & \frac{\partial x_3}{\partial X_3}
\end{bmatrix}
\]

Looking at the representation in Fig A.1, we can see that all the components of the vector in the original configuration remain the same except for the component in the \( e_1 \) direction. There we have \( x_1 = X_1 + X_2 \tan(\gamma) = X_1 + X_2 \tan(\tan(\delta L)) \). So the deformation
A.1 Green-Lagrange and Euler-Almansi tensors

**Figure A.1** Representation of simple shear. Red indicates original configuration while blue indicates the deformed configuration.

The gradient tensor will have all the diagonal components of unit value and the 12 component of value $\tan(\gamma)$. All the other components will be 0. Hence,

$$F = \begin{bmatrix} 1 & \tan(\gamma) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From the gradient deformation tensor, we can compute the right and left Cauchy-Green deformation tensors:

$$C = \begin{bmatrix} 1 & \tan(\gamma) & 0 \\ \tan(\gamma) & \tan^2(\gamma) + 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} \tan^2(\gamma) + 1 & \tan(\gamma) & 0 \\ \tan(\gamma) & \tan^2(\gamma) + 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can obtain the Green-Lagrange and Euler-Almansi strain tensors, which are:

$$E_{GL} = \frac{1}{2}(C - I), \quad e_{EA} = \frac{1}{2}(I - B^{-1})$$
With

\[
B^{-1} = \begin{bmatrix}
1 & -\tan(\gamma) & 0 \\
-\tan(\gamma) & \tan^2(\gamma) + 1 & 0 \\
0 & 0 & 1
\end{bmatrix},
\]

we find that:

\[
E_{GL} = \frac{1}{2} \begin{bmatrix}
0 & \tan(\gamma) & 0 \\
\tan(\gamma) & \tan^2(\gamma) & 0 \\
0 & 0 & 1
\end{bmatrix}, \quad e_{EA} = \frac{1}{2} \begin{bmatrix}
0 & \tan(\gamma) & 0 \\
\tan(\gamma) & -\tan^2(\gamma) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

By setting \(k = \tan(\gamma)\), and rotating the axes so that instead of using the x and y axes for the derivation we instead employ y and z, we obtain the familiar matrices from the Introduction section:

\[
E_{GL} = \frac{1}{2} \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & k \\
0 & k & k^2
\end{bmatrix}, \quad e_{EA} = \frac{1}{2} \begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & k \\
0 & k & k^2
\end{bmatrix}
\]
A.2 Strain Tensors and their derivatives in explicit form

Here \( v \) represents the derivative of \( \delta \) with respect to time

\[
E_{GL} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \tan(\tan(\frac{\delta(t)}{L})) \\
0 & \tan(\frac{\delta(t)}{L}) & \tan^2(\frac{\delta(t)}{L})
\end{bmatrix}
\]

\[
e_{EA} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \tan(\frac{\delta(t)}{L}) \\
0 & \tan(\frac{\delta(t)}{L}) & -\tan^2(\frac{\delta(t)}{L})
\end{bmatrix}
\]

\[
\dot{E}_{GL} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \frac{v}{L} \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L}) \\
0 & \frac{v}{L} \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L}) & 2\frac{v}{L} \tan(\tan(\frac{\delta(t)}{L})) \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L})
\end{bmatrix}
\]

\[
\dot{e}_{EA} = \frac{1}{2} \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & \frac{v}{L} \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L}) \\
0 & \frac{v}{L} \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L}) & -2\frac{v}{L} \tan(\tan(\frac{\delta(t)}{L})) \sec^2(\tan(\frac{\delta(t)}{L})) \sec^2(\frac{\delta(t)}{L})
\end{bmatrix}
\]
Appendix B

Additional Images

B.1 Higher Scale High-Res Fitting Plots

Figure B.1 Histograms and corresponding fits for surface areas of astrocytes and neurons.
Figure B.2 Histograms and corresponding fits for bounding box areas of astrocytes and neurons.

Figure B.3 Histograms and corresponding fits for the first moment of inertia eigenvalues of astrocytes and neurons.
B.1 Higher Scale High-Res Fitting Plots

Figure B.4 Histograms and corresponding fits for the second moment of inertia eigenvalues of astrocytes and neurons.

Figure B.5 Histograms and corresponding fits for the third moment of inertia eigenvalues of astrocytes and neurons.
Figure B.6 3D view of segmented datasets with traced spines (preprocessing version).
Appendix C

Coding Resources

C.1 Specifics of Computer Used for Segmentation

<table>
<thead>
<tr>
<th>CPU Speed</th>
<th>2.3Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>8-core Intel Core i9</td>
</tr>
<tr>
<td>Memory (RAM)</td>
<td>16 GB 2667 MHz DDR4</td>
</tr>
<tr>
<td>Graphics</td>
<td>Intel UHD Graphics 630 1536 MB</td>
</tr>
<tr>
<td>OS Version</td>
<td>macOS Big Sur Version 11.3.1</td>
</tr>
</tbody>
</table>
## C.2 Fitting Parameters

### NEURON FIT PARAMETERS (FIXED LOCATION PARAMETER)

<table>
<thead>
<tr>
<th>Feature</th>
<th>First Shape Parameter ($k$)</th>
<th>First Shape Parameter ($\alpha$)</th>
<th>Scale Parameter ($\lambda$)</th>
<th>Location Parameter ($\theta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Area</td>
<td>0.4310</td>
<td>2536</td>
<td>38.25</td>
<td>0</td>
</tr>
<tr>
<td>Bounding Box Area</td>
<td>0.5280</td>
<td>24.52</td>
<td>19030</td>
<td>0</td>
</tr>
<tr>
<td>First Moment of Inertia Eigenvalue</td>
<td>0.3591</td>
<td>227.0</td>
<td>2.154</td>
<td>0</td>
</tr>
<tr>
<td>Second Moment of Inertia Eigenvalue</td>
<td>0.4947</td>
<td>40.61</td>
<td>13.83</td>
<td>0</td>
</tr>
<tr>
<td>Third Moment of Inertia Eigenvalue</td>
<td>0.3833</td>
<td>160.0</td>
<td>0.9766</td>
<td>0</td>
</tr>
</tbody>
</table>

### NEURON FIT PARAMETERS (NON-FIXED LOCATION PARAMETER)

<table>
<thead>
<tr>
<th>Feature</th>
<th>First Shape Parameter ($k$)</th>
<th>First Shape Parameter ($\alpha$)</th>
<th>Scale Parameter ($\lambda$)</th>
<th>Location Parameter ($\theta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Area</td>
<td>0.3332</td>
<td>36940</td>
<td>4.000</td>
<td>-124.6</td>
</tr>
<tr>
<td>Bounding Box Area</td>
<td>0.7598</td>
<td>9.021</td>
<td>59460</td>
<td>44.77</td>
</tr>
<tr>
<td>First Moment of Inertia Eigenvalue</td>
<td>0.3456</td>
<td>300.3</td>
<td>1.565</td>
<td>-1.787</td>
</tr>
<tr>
<td>Second Moment of Inertia Eigenvalue</td>
<td>0.4946</td>
<td>40.75</td>
<td>13.81</td>
<td>-0.1276</td>
</tr>
<tr>
<td>Third Moment of Inertia Eigenvalue</td>
<td>0.8634</td>
<td>227.7</td>
<td>0.6683</td>
<td>-1.083</td>
</tr>
</tbody>
</table>

### ASTROCYTE FIT PARAMETERS (FIXED LOCATION PARAMETER)

<table>
<thead>
<tr>
<th>Feature</th>
<th>First Shape Parameter ($k$)</th>
<th>First Shape Parameter ($\alpha$)</th>
<th>Scale Parameter ($\lambda$)</th>
<th>Location Parameter ($\theta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Area</td>
<td>0.5433</td>
<td>231.6</td>
<td>46.23</td>
<td>0</td>
</tr>
<tr>
<td>Bounding Box Area</td>
<td>0.2614</td>
<td>279.6</td>
<td>11.55</td>
<td>0</td>
</tr>
<tr>
<td>First Moment of Inertia Eigenvalue</td>
<td>0.3817</td>
<td>1224</td>
<td>0.2031</td>
<td>0</td>
</tr>
<tr>
<td>Second Moment of Inertia Eigenvalue</td>
<td>0.3796</td>
<td>1516</td>
<td>0.1594</td>
<td>0</td>
</tr>
<tr>
<td>Third Moment of Inertia Eigenvalue</td>
<td>0.8153</td>
<td>99.46</td>
<td>3.143</td>
<td>0</td>
</tr>
</tbody>
</table>

### ASTROCYTE FIT PARAMETERS (NON-FIXED LOCATION PARAMETER)

<table>
<thead>
<tr>
<th>Feature</th>
<th>First Shape Parameter ($k$)</th>
<th>First Shape Parameter ($\alpha$)</th>
<th>Scale Parameter ($\lambda$)</th>
<th>Location Parameter ($\theta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Area</td>
<td>0.5436</td>
<td>231.8</td>
<td>46.34</td>
<td>-0.8003</td>
</tr>
<tr>
<td>Bounding Box Area</td>
<td>0.2341</td>
<td>553.6</td>
<td>3.294</td>
<td>-0.9063</td>
</tr>
<tr>
<td>First Moment of Inertia Eigenvalue</td>
<td>0.4263</td>
<td>849.1</td>
<td>0.4256</td>
<td>-2.520</td>
</tr>
<tr>
<td>Second Moment of Inertia Eigenvalue</td>
<td>0.4498</td>
<td>647.8</td>
<td>0.5159</td>
<td>-2.322</td>
</tr>
<tr>
<td>Third Moment of Inertia Eigenvalue</td>
<td>0.8103</td>
<td>119.0</td>
<td>3.058</td>
<td>-0.6404</td>
</tr>
</tbody>
</table>

## C.3 Source Code (Python3)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

Created on Tue Jun 1 16:53:22 2021

@Author: danielefattooffidani
```
# Packages needed in Python:
# - Numpy
# - scikit-image
# - matplotlib
# - scipy
# - scikit-learn
# - tifffile
# _timeit (should be natively installed)

# Import all Needed Packages and Functions
import copy
import numpy as np
from skimage import io
import matplotlib.pyplot as plt
from timeit import default_timer as timer
from sklearn.cluster import MiniBatchKMeans
from tifffile import imsave
from skimage.filters import threshold_otsu
from scipy.ndimage import binary_erosion as ers
from scipy.ndimage.measurements import label
from skimage import draw
from skimage.segmentation import find_boundaries as bnd
from skimage.filters import gaussian as gaus
from skimage.filters import median_filter
from skimage.measure import regionprops as rg
from skimage.segmentation import expand_labels as exp
from skimage.measure import marching_cubes, mesh_surface_area
from scipy.ndimage import zoom
from scipy import stats
from matplotlib.widgets import Slider

# Chunking function, see below. YOU NEED TO RUN THIS SECTION BEFORE THE NEURON RECONSTRUCTION

def chunkIt(seq, num):
    avg = len(seq) / float(num)
    out = []
    last = 0.0
    while last < len(seq):
        out.append(seq[int(last):int(last + avg)])
        last += avg
    return out

# Assigning Probabilities. YOU NEED TO RUN THIS BEFORE CONDITIONAL PROBABILITY MAP

# Loads Precomputed pdfs
pdf_astro_area, pdf_a1_in, pdf_a2_in, pdf_a3_in, pdf_astro_bbox = np.load('pdf_astro_area.npy'), np.load('pdf_a1_in.npy'), np.load('pdf_a2_in.npy'), np.load('pdf_a3_in.npy'), np.load('pdf_astro_bbox.npy')

pdf_neuron_area, pdf_n1_in, pdf_n2_in, pdf_n3_in, pdf_neuron_bbox = np.load('pdf_neuron_area.npy'), np.load('pdf_n1_in.npy'), np.load('pdf_n2_in.npy'), np.load('pdf_n3_in.npy'), np.load('pdf_neuron_bbox.npy')
def cond_prob(a):
    working_copy = zoom(a,(0.61/1.075,1,1))
    working_copy[working_copy>0] = 255

    vts, fs, ns, cs = marching_cubes(a)
    surface = (mesh_surface_area(vts,fs)*0.61**2)

    inertia = rg(a)[0].inertia_tensor_eigvals
    bbox_area = rg(a)[0].bbox_area*0.61**2

    prob_given_surface = (stats.exponweib.pdf(surface, *pdf_astro_area)*0.63)/(stats.exponweib.pdf(surface, *pdf_astro_area)*0.63+stats.exponweib.pdf(surface, *pdf_neuron_area)*0.37)

    prob_given_in1 = (stats.exponweib.pdf(inertia[0], *pdf_a1_in)*0.63)/(stats.exponweib.pdf(inertia[0], *pdf_a1_in)*0.63+stats.exponweib.pdf(inertia[0], *pdf_n1_in)*0.37)

    prob_given_in2 = (stats.exponweib.pdf(inertia[1], *pdf_a2_in)*0.63)/(stats.exponweib.pdf(inertia[1], *pdf_a2_in)*0.63+stats.exponweib.pdf(inertia[1], *pdf_n2_in)*0.37)

    prob_given_in3 = (stats.exponweib.pdf(inertia[2], *pdf_a3_in)*0.63)/(stats.exponweib.pdf(inertia[2], *pdf_a3_in)*0.63+stats.exponweib.pdf(inertia[2], *pdf_n3_in)*0.37)

    prob_given_bbox = (stats.exponweib.pdf(bbox_area, *pdf_astro_bbox)*0.63)/(stats.exponweib.pdf(bbox_area, *pdf_astro_bbox)*0.63+stats.exponweib.pdf(bbox_area, *pdf_neuron_bbox)*0.37)

    weighted_prob = 0.25*(prob_given_surface+prob_given_in1+prob_given_in2)+0.125*(prob_given_in3+prob_given_bbox)

    if weighted_prob >= 0.5:
        print('Cell is an Astrocyte with probability %.3f%%' % (100*float(weighted_prob)))
    else:
        print('Cell is a Neuron with probability %.3f%%' % (100*(1-float(weighted_prob))))

    astro_prob = weighted_prob
    neuron_prob = 1-weighted_prob

    return (astro_prob, neuron_prob)
# Optional Functions

## Displays multiple slices of 3D array in one go

```python
def display(im3d, cmap="gray", norm = '', step=2):
    _, axes = plt.subplots(nrows=6, ncols=10, figsize=(16, 14))
    if norm == 'norm':
        vmin = 0
        vmax = 1
    elif norm == '2norm':
        vmin = -1
        vmax = 1
    else:
        vmin = im3d.min()
        vmax = im3d.max()

    for ax, image in zip(axes.flatten(), im3d[::step]):
        ax.imshow(image, cmap=cmap, vmin=vmin, vmax=vmax)
        ax.set_xticks([])
        ax.set_yticks([])
```

## This section loads the figure as a 3D array

```python
stk = io.imread('good.tif') # import image

# There is a bug with the tiff files. When opened in Python they can appear as 1024x1024 single slices images. To fix this, open the tiff file in Imagej and save it as tiff with the same name. It will now be loaded as a tiff stack. Not sure what causes the issue, but it’s consistent throughout all image reading functions I have tried, even tiff specific ones.

# 3D KMeans Cleaning

```python
start = timer()
data_set = stk.copy() # creates dataset copy to keep original intact

for i in range(data_set.shape[0]):
    data_set[i] = data_set[i]/np.max(data_set[i])*255 # Normalize all slices of Dataset after importation

k = 6 # sets KNumber
z, x, y = data_set.shape # shape of image
cl_image = data_set.reshape(x*y*z,1) # reshape image array representation, can only feed column arrays in k-means

# minibatches kmeans initialization
```python
kmeans_cluster = MiniBatchKMeans(n_clusters=k, batch_size=20000, max_iter=500, n_init=100, reassignment_ratio=0.5)

predict = (kmeans_cluster.fit_predict(cl_image)).reshape(z,x,y)  # computes minibatches kmeans and assigns each voxel to a cluster

unique, counts = np.unique(predict, return_counts=True)
count_list = counts.tolist()
count_list.sort()  # List of clusters in descending order of datapoints contained

#binarization via cluster tresholding
#For old datasets, 1 was fine, for newer datasets 2 are needed
for n in range(1, 3):
    #Sets number of clusters to assign to background
    predict[predict == unique[np.where(counts == count_list[len(count_list)-n])[0][0]]] = 15  # 15 is dummy index

#Tresholding
predict[predict != 15] = 255
predict[predict == 15] = 0

end = timer()
predict = predict.astype('uint8')  # integers datatype as unlabeled 8bit integers, no waste of space

print(end - start)

#plots zeroth and 60th slice for quality check
fig, ax = plt.subplots(2, 2)
ax[0,0].imshow(stk[0], cmap='Greys')
ax[0,0].set_title('Original Zeroth Slice')
ax[0,1].imshow(predict[0], cmap='Greys')
ax[0,1].set_title('Binarized Zeroth Slice')

ax[1,0].imshow(stk[60], cmap='Greys')
ax[1,0].set_title('Original Sixtieth Slice')
ax[1,1].imshow(predict[60], cmap='Greys')
ax[1,1].set_title('Binarized Sixtieth Slice')

#Saves binarized output as binarized file. Option to save tiff stack as well
np.save('Cleaned.npy', predict)
imsave('Cleaned.tif', predict)

#Repeat this function with different n range if result is wrong (it is obvious when it is)

# %% Alternative Cleaning if KMeans does not converge (ONLY USE IF EVERYTHING ELSE FAILS, NOT AS ACCURATE)
data_set = stk.copy()
otsu = np.empty(data_set.shape)
for i in range(data_set.shape[0]):
```
C.3 Source Code (Python3)

```python
c.3 Source Code (Python3)

thresh = threshold_otsu(data_set[i])
otsu[i] = (data_set[i]>thresh)*255 #Thresholding based on intensity histogram

otsu = otsu.astype('uint8')

#Quality Check
fig, ax = plt.subplots(2, 2)
ax[0,0].imshow(stk[0], cmap = 'Greys')
ax[0,0].set_title('Original Zeroth Slice')
ax[0,1].imshow(otsu[0], cmap = 'Greys')
ax[0,1].set_title('Binarized Zeroth Slice')

ax[1,0].imshow(stk[60], cmap = 'Greys')
ax[1,0].set_title('Original Sixtieth Slice')
ax[1,1].imshow(otsu[60], cmap = 'Greys')
ax[1,1].set_title('Binarized Sixtieth Slice')

#Saving options
#np.save('otsu.npy', predict)
#imsave('otsu.tif', otsu)

# % Erosion

#Load image array or tiff stack depending on format
new_stk = np.load('Cleaned_20210302_S22_Neu00_G.npy').astype('uint8')
#new_stk = io.imread('binary.tif').astype('uint8')

average_diameter = float(input("Enter average diameter of soma in pixels:"))

cmap_reversed = plt.cm.get_cmap('gnuplot2_r')

xy_semiaxis = int(round((average_diameter/2)*0.75)) #creates semiaxes
z_semiaxis = int(round(((average_diameter/2)*0.75*(1.075/0.61)))

struc = draw.ellipsoid(z_semiaxis, xy_semiaxis, xy_semiaxis) #takes into account different pixel-to-micron ration for x/y and z axes.

eroded = ers(new_stk, structure=struc, iterations = 1).astype('uint8') #Erosion process. For each voxel it checks the neighbors

#based on the structure. If they are all full it keeps them, else it erodes. Output is clumps of voxels, ideally the centers of the somas of the cells
structure = np.ones((3, 3, 3)) #connectivity structure to identify neighboring voxels horizontal, vertical and diagonal

labels, ncomponents = label(eroded, structure) #labels voxel clumps based on connectivity

print(ncomponents) #prints number of labels (i.e. number of separated clumps). Most important index
```

51
```python
display_labels = (labels.copy().astype('float')) + 255 - np.max(labels)
# adding integer step to better distinguish labels from background
display_labels[display_labels == np.min(display_labels)] = 0
# plots 2 slices to visually check the result
fig, ax = plt.subplots(2, 2)
ax[0,0].imshow(new_stk[20], cmap = 'Greys')
ax[0,0].imshow(eroded[20], cmap = 'jet', interpolation = 'none', alpha = 0.6)
ax[0,0].set_title('Eroded 20th slice on original binary')
ax[0,1].imshow(new_stk[80], cmap = 'Greys')
ax[0,1].imshow(eroded[80], cmap = 'jet', interpolation = 'none', alpha = 0.6)
ax[0,1].set_title('Eroded 80th slice on original binary')
ax[1,0].imshow(display_labels[20], cmap = cmap_reversed)
ax[1,0].set_title('Labeled Eroded 20th Slice')
ax[1,1].imshow(display_labels[80], cmap = cmap_reversed)
ax[1,1].set_title('Labeled Eroded 80th Slice')

# Saving Options (Current Directory)
labels = labels.astype('uint8')
np.save('Labeled_20210302_S22_NeuO0_G.npy', labels)
imsave('Labeled_20210302_S22_NeuO0_G.tif', labels)
np.save('Eroded_20210302_S22_NeuO0_G.npy', eroded.astype('uint8'))
imsave('Eroded_20210302_S22_NeuO0_G.tif', eroded.astype('uint8'))

# % Neuron Reconstruction
# This section of the code reconstructs the neurons starting from the
# clumps obtained in the erosion section. At every iteration
# of the while loop, the labeled clumps are expanded in every direction.
# Then, we multiply the expanded array with the original
# binary array (functions as a mask). This prevents the creation of
# artificial voxels. At the end, the code checks whether the
# output expanded dataset is the same as the one of the previous
# iteration. If it is, it means that the reconstruction has
# reached the plateau and the while loop breaks. The time it takes for
# this process varies based on the size of the dataset
# and the number and topology of the 3D neurons. As of right now, it is
# estimated to range from 35 to 80 minutes for
# 1024x1024x188 images.

# Run the chunking function before starting

# if not imported, import label clumps and or mask:
new_stk = np.load('Cleaned_20210302_S22_NeuO0_G.npy').astype('uint8')
labels = np.load('Labeled_20210302_S22_NeuO0_G.npy').astype('uint8')

# Initialize labeled dataset
```
mask = (new_stk / 255).astype('uint8')
new_labels_0 = labels.copy().astype('uint8')
old_labels = new_labels_0.copy().astype('uint8')
count = 0  # counting iterations
computation_times = []  # computation times per iteration, useful for optimization purposes

while True:  # loop breaks after neurons are totally reconstructed
    begin = timer()  # needed for iteration time
    # after first iteration, updates old_labels to previous dataset
    if count > 0:
        globals()['new_labels_%s' % count] = globals()['new_labels_%s' % (count-1)].copy()
        old_labels = globals()['new_labels_%s' % (count-1)].copy()

    finished_neurons = []  # list of labeled clumps that no longer change, no need to compute them at every iteration
    for regions in rg(globals()['new_labels_%s' % count]):  # this function examines every labeled region
        if (regions.label in finished_neurons):
            continue  # if the label is the one of a region that does not change, skip it
        minz, minx, miny, maxz, maxx, maxy = regions.bbox  # find bounding boxes for labeled clumps
        zlow, xlow, ylow, zhigh, xhigh, yhigh = minz-3, minx-3, miny-3, maxz+2, maxx+2, maxy+2  # increasing box size
        # making sure bounds are not imaginary. If out of bounds, set to dataset shape
        if zlow < 0:
            zlow = 0
        if xlow < 0:
            xlow = 0
        if ylow < 0:
            ylow = 0
        if zhigh > labels.shape[0]-1:
            zhigh = labels.shape[0]-1
        if xhigh > labels.shape[1]-1:
            xhigh = labels.shape[1]-1
        if yhigh > labels.shape[2]-1:
            yhigh = labels.shape[2]-1
        # further chunking might be beneficial for larger datasets. Or maybe at later iterations, there is currently something wrong with the finished neurons characterization, needs some work to understand what
# create 3-axial divisions
zrange = np.arange(zlow, zhigh+1, dtype = 'uint8')
xrange = np.arange(xlow, xhigh+1, dtype = 'uint8')
yrange = np.arange(ylow, yhigh+1, dtype = 'uint8')

# divides each neuron in 8 parts for faster processing
divs_z = chunkIt(zrange, 2)
divs_x = chunkIt(xrange, 2)
divs_y = chunkIt(yrange, 2)

# Label expansion binary mask multiplication loop. If statements needed to handle boundaries of dataset. Ugly but works

# THE FOLLOWING LINES HAVE BEEN DEINDENTED FOR SPACING ISSUES
# REACH OUT TO ME IF YOU WANT AN INDENTED COPY
for i in divs_x:
    for j in divs_y:
        for k in divs_z:
            if np.min(i) == 0:
                if np.max(i) == labels.shape[1]-1:
                    if np.min(j) == 0:
                        if np.max(j) == labels.shape[2]-1:
                            if np.min(k) == 0:
                                if np.max(k) == labels.shape[0]-1:
                                    globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                    = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], distance = 1.74)
                                    globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                    = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], mask[np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)])
                                    else:
                                        if np.max(k) == labels.shape[0]-1:
                                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                            = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], distance = 1.74)
                                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                            = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)])
                                        else:
                                            pass
                                    else:
                                        if np.max(k) == labels.shape[0]-1:
                                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                            = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], distance = 1.74)
                                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
                                            = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)])
                                        else:
                                            pass
                                else:
                                    pass
                        else:
                            pass
                    else:
                        pass
                else:
                    pass
            else:
                pass
        else:
            pass
    else:
        pass
else:
    pass
```python
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)], mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)])
 else:
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)]
 = exp(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)], distance = 1.74)
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)], mask[np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)])

else:
 if np.min(k) == 0:
 if np.max(k) == labels.shape[0]-1:
 globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = exp(globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
 globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], mask[np.min(k):np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1])
 else:
 globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = exp(globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
 globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1], mask[np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)])

else:
 if np.max(k) == labels.shape[0]-1:
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = exp(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1])
 else:
 if np.max(k) == labels.shape[0]-1:
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = exp(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
 globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
 = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1], mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1])
```

```python
mask[np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
else:
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1]
    = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j):np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j):np.max(j)+1]
    = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j):np.max(j)+1],
      mask[np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j):np.max(j)+1])

else:
if np.max(j) == labels.shape[2]-1:
    if np.min(k) == 0:
        if np.max(k) == labels.shape[0]-1:
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j)-1:np.max(j)]
        = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i),np.min(j)-1:np.max(j)]
        = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1],
          mask[np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j)-1:np.max(j)])
    else:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j)-1:np.max(j)]
        = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j)-1:np.max(j)]
        = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1],
          mask[np.min(k):np.max(k)+1,np.min(i):np.max(i),np.min(j)-1:np.max(j)])
else:
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i),np.min(j)-1:np.max(j)]
```

```python
= exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)], mask[np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)])
else:
    if np.min(k) == 0:
        globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], mask[np.min(k):np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1])
else:
    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], mask[np.min(k):np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], mask[np.min(k)-1:np.max(k), np.min(i):np.max(i), np.min(j)-1:np.max(j)+1])
else:
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1], mask[np.min(k)-1:np.max(k)+1, np.min(i):np.max(i), np.min(j)-1:np.max(j)+1])
```

57
C.3 Source Code (Python3)

```python
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j)-1:np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j)-1:np.max(j)+1],
mask[np.min(k)-1:np.max(k)+1,np.min(i):np.max(i),np.min(j)-1:np.max(j)+1])

else:
    if np.min(j) == 0:
        if np.max(j) == labels.shape[2]-1:
            if np.min(k) == 0:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)], distance = 1.74)
                    globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)],
mask[np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)])
                else:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1 np.min(i):np.max(i)+1, np.min(j):np.max(j)], distance = 1.74)
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)],
mask[np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)])
            else:
                globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)], distance = 1.74)
                globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1, np.min(j):np.max(j)],
mask[np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)])
        else:
            globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1, np.min(j):np.max(j)], distance = 1.74)
            globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i):np.max(i)+1, np.min(j):np.max(j)],
mask[np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)])
    else:
        if np.max(k) == labels.shape[0]-1:
            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1, np.min(j):np.max(j)], distance = 1.74)
            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1, np.min(j):np.max(j)],
mask[np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j):np.max(j)])
        else:
            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)], distance = 1.74)
            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)],
mask[np.min(k)-1:np.max(k)+1,np.min(i):np.max(i)+1,np.min(j):np.max(j)])
```
C.3 Source Code (Python3)

```python
mask[np.min(k)-1:np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)

else:
    if np.min(k) == 0:
        globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], mask[np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1])
    else:
        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)

else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)

else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)

else:
    if np.min(k) == 0:
        globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], mask[np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1])
    else:
        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)

else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j):np.max(j)+1], distance = 1.74)
```

if np.max(j) == labels.shape[2]-1:
    if np.min(k) == 0:
        globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = exp(globals()["new_labels_%s" % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)], distance = 1.74)
        globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = np.multiply(globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)],
                      mask[np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)])
    else:
        globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = exp(globals()["new_labels_%s" % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)], distance = 1.74)
        globals()["new_labels_%s" % count][np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = np.multiply(globals()["new_labels_%s" % count][np.min(k):np.max(k)+1, np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)],
                      mask[np.min(k):np.max(k)+1,np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = exp(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)], distance = 1.74)
        globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
        = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)],
                      mask[np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)])
    else:
        if np.min(k) == 0:
            if np.max(k) == labels.shape[0]-1:
                globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
                = exp(globals()["new_labels_%s" % count][np.min(k):np.max(k), np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)], distance = 1.74)
                globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
                = np.multiply(globals()["new_labels_%s" % count][np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)],
                              mask[np.min(k):np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)])
            else:
                globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
                = exp(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k), np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)], distance = 1.74)
                globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)]
                = np.multiply(globals()["new_labels_%s" % count][np.min(k)-1:np.max(k),np.min(i):np.max(i)+1, np.min(j)-1:np.max(j)],
                              mask[np.min(k)-1:np.max(k),np.min(i):np.max(i)+1,np.min(j)-1:np.max(j)])
```python
import numpy as np

def compute_distance(labels, distance):
    new_labels = np.zeros_like(labels)
    for i in range(labels.shape[1]):
        for j in range(labels.shape[2]):
            if np.max(i) == labels.shape[1] - 1:
                new_labels[:, i, j] = np.exp(np.multiply(new_labels[:, i, j], mask[:, i, j])
    return new_labels
```

This code snippet computes a new label set based on the existing label set and a given distance parameter. It iterates over each element in the label set, applying the exponential function to the corresponding element of the new label set. If the maximum index of the current row is equal to the last index of the label set, it applies the exponential function to the entire row of the new label set, weighted by the mask. This process is repeated for each index, effectively updating the new label set based on the current label set and the specified distance.
if np.max(j) == labels.shape[2]-1:
if np.min(k) == 0:
globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)]
= exp(globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)], distance = 1.74)

globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)]
= np.multiply(globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)], mask[np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)])

else:
globals()[f'new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i)-1:np.max(i), np.min(j):np.max(j)]
= exp(globals()[f'new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i)-1:np.max(i), np.min(j):np.max(j)], distance = 1.74)

else:
if np.max(k) == labels.shape[0]-1:
globals()[f'new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)]
= exp(globals()[f'new_labels_%s' % count][np.min(k)-1:np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)], distance = 1.74)

else:
if np.min(k) == 0:
if np.max(k) == labels.shape[0]-1:
globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)+1]

else:
if np.min(k) == 0:
if np.max(k) == labels.shape[0]-1:
  globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)+1]

else:
if np.min(k) == 0:
if np.max(k) == labels.shape[0]-1:
  globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)+1]

else:
if np.min(k) == 0:
if np.max(k) == labels.shape[0]-1:
  globals()[f'new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)+1]
```python
= exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j):np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], mask[np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1])
else:
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j):np.max(j)+1])
    else:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], distance = 1.74)
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1] = np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1], mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j):np.max(j)+1])
else:
    if np.max(j) == labels.shape[2]-1:
        if np.min(k) == 0:
            if np.max(k) == labels.shape[0]-1:
                globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
```

63
```python
= exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)],
distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)],
mask[np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)])
else:
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)],
distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1, np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)],
mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)])
else:
if np.max(k) == labels.shape[0]-1:
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1]
= exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)],
distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)],
mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)])
else:
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)],
distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)]
= np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1, np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)],
mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)])
else:
if np.min(k) == 0:
if np.max(k) == labels.shape[0]-1:
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1]
```

64
```python
= exp(globals()['new_labels_%s' % count][np.min(k):np.max(k), np.min(i)-1:np.max(i), np.min(j)-1:np.max(j)+1],

distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],
mask[np.min(k):np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1])
else:
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],

distance = 1.74)
globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],
mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1])
else:
if np.max(k) == labels.shape[0]-1:
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],

distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],
mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1])
else:
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],

distance = 1.74)
globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1] =
np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1],
mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i),np.min(j)-1:np.max(j)+1])
else:
if np.min(j) == 0:
```
if np.max(j) == labels.shape[2]-1:
    if np.max(k) == 0:
        globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
        exp(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
        distance = 1.74)
    globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    mask[np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)]
else:
    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    exp(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    distance = 1.74)
    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)]
else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
        exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
        distance = 1.74)
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)]
else:
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    exp(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    distance = 1.74)
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)] =
    np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)],
    mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j):np.max(j)]
else:
if np.min(k) == 0:
    if np.max(k) == labels.shape[0]-1:
        globals()[f'new_labels_{count}'] = 
            exp(globals()[f'new_labels_{count}'][:max(k), np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
                distance = 1.74)
        globals()[f'new_labels_{count}'] = np.multiply(globals()[f'new_labels_{count}'][:max(k), np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            mask[:max(k), np.min(i)-1:max(i)+1, np.min(j):max(j)+1])
    else:
        globals()[f'new_labels_{count}'] = exp(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            distance = 1.74)
        globals()[f'new_labels_{count}'] = np.multiply(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            mask[:max(k)-1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()[f'new_labels_{count}'] = 
            exp(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
                distance = 1.74)
        globals()[f'new_labels_{count}'] = np.multiply(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            mask[:max(k)-1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1])
    else:
        globals()[f'new_labels_{count}'] = exp(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            distance = 1.74)
        globals()[f'new_labels_{count}'] = np.multiply(globals()[f'new_labels_{count}'][:max(k)+1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1],
            mask[:max(k)-1, np.min(i)-1:max(i)+1, np.min(j):max(j)+1])
else:
    if np.max(j) == labels.shape[2]-1:
        if np.min(k) == 0:
            if np.max(k) == labels.shape[0]-1:
                globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                mask[np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                else:
                    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    mask[np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                else:
                    if np.max(k) == labels.shape[0]-1:
                        globals()['new_labels_%s' % count][np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        mask[np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    else:
                        if np.max(k) == labels.shape[0]-1:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        else:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
        else:
            if np.max(k) == labels.shape[0]-1:
                globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                distance = 1.74
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    distance = 1.74
                else:
                    if np.max(k) == labels.shape[0]-1:
                        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        distance = 1.74
                    else:
                        if np.max(k) == labels.shape[0]-1:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            distance = 1.74
                        else:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            distance = 1.74
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    mask[np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                else:
                    if np.max(k) == labels.shape[0]-1:
                        globals()['new_labels_%s' % count][np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        mask[np.min(k):np.max(k)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    else:
                        if np.max(k) == labels.shape[0]-1:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        else:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
        else:
            if np.max(k) == labels.shape[0]-1:
                globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                distance = 1.74
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    distance = 1.74
                else:
                    if np.max(k) == labels.shape[0]-1:
                        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        distance = 1.74
                    else:
                        if np.max(k) == labels.shape[0]-1:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        else:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
        else:
            if np.max(k) == labels.shape[0]-1:
                globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                distance = 1.74
            else:
                if np.max(k) == labels.shape[0]-1:
                    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                    distance = 1.74
                else:
                    if np.max(k) == labels.shape[0]-1:
                        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        distance = 1.74
                    else:
                        if np.max(k) == labels.shape[0]-1:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                        else:
                            globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
                            mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)]
mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)+1]
else:
    if np.min(k) == 0:
        if np.max(k) == labels.shape[0]-1:
            globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
            distance = 1.74
            globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1] -= np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1], mask[np.min(k):np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
        distance = 1.74
        globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1] -= np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1], mask[np.min(k)-1:np.max(k),np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1])
else:
    if np.max(k) == labels.shape[0]-1:
        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
        distance = 1.74
        globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1] -= np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1], mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1])
else:
    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
    distance = 1.74
    globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1] -= np.multiply(globals()['new_labels_%s' % count][np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1], mask[np.min(k):np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1])
else:
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
    distance = 1.74
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1] -= np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1], mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1])
else:
    globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1, np.min(j)-1:np.max(j)+1]
```python
= np.multiply(globals()['new_labels_%s' % count][np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)+1],
mask[np.min(k)-1:np.max(k)+1,np.min(i)-1:np.max(i)+1,np.min(j)-1:np.max(j)+1])

if count >1:
    if (globals()['new_labels_%s' % count][zlow:zhigh, xlow:xhigh, ylow:yhigh] ==
        globals()['new_labels_%s' % (count-2)][zlow:zhigh, xlow:xhigh, ylow:yhigh]).all():
        finished_neurons.append(regions.label)  # Checks if there is no change
        # with respect to 2 previous iterations. If no change, the neuron will no
        # longer be updated.

stop = timer()
comp = stop-begin
computation_times.append(comp)  # appends computation time for current iteration

if count > 2:
    del globals()['new_labels_%s' % (count-3)]  # delete dataset from 2 previous iteration, no need to keep it

if (globals()['new_labels_%s' % (count-2)] == globals()['new_labels_%s' % count]).all():
    break  # breaks loop when dataset no longer changes (reconstruction is complete)

count +=1  # update count
# optional printing
print(count)

# plot one slice to visualize output
plt.figure('Reconstructed Neurons')
plt.imshow(globals()['new_labels_%s' % count][60, cmap = plt.cm.get_cmap('gnuplot2_r'))

# saving output
reconstructed = globals()['new_labels_%s' % count].copy().astype('uint8')
np.save('Reconstructed.npy', reconstructed)
imsave('Reconstructed.tif', reconstructed)

# %% Find Borders and detect connected labels
```
reconstructed = np.load('Reconstructed.npy').astype('uint8')

borders = bnd(reconstructed, connectivity = 3, mode = 'inner').astype('uint8')  # detects boundary elements of labels as binary mask
borders = np.multiply(borders, reconstructed)  # labels the mask with through the original

neighbor_array = np.array([borders, np.roll(borders,+1,axis=0), np.roll(borders,-1,axis=0), np.roll(borders,+1,axis=1), np.roll(borders,-1,axis=1), np.roll(borders,+1, axis=2), np.roll(borders,-1, axis=2)])
# if you want neighbors to include wraparounds, use above; if not, prune
# reshape to a 2d array entries x neighbors
neighbor_list = np.reshape(neighbor_array,[7,-1]).T
# get uniques into a dictionary
neighbor_dict = {}
for num in range(1, len(np.unique(borders)+1)):
    neighbor_dict[num] = np.unique(neighbor_list[np.where(neighbor_list[:,0]==num)])

# This part needs much refining. Ideally I would put the labels in each specific list. Right now only singles are good

single = []
double = []
triple = []
quadrupole = []
quintupole = []
sextupole = []
ectupole = []
enneapole = []
decapole = []

for i in range(1, len(neighbor_dict)+1):
    if len(neighbor_dict[i]) == 2:
        single.append(i)
    elif len(neighbor_dict[i]) == 3:
        double.append(neighbor_dict[i])
        # double.append(i)
    elif len(neighbor_dict[i]) == 4:
        triple.append(neighbor_dict[i])
        # triple.append(i)
    elif len(neighbor_dict[i]) == 5:
        quadrupole.append(neighbor_dict[i])
        # quadrupole.append(i)
    elif len(neighbor_dict[i]) == 6:
        quintupole.append(neighbor_dict[i])
        # quintupole.append(i)
    elif len(neighbor_dict[i]) == 7:
        sextupole.append(neighbor_dict[i])
        # sextupole.append(i)
    elif len(neighbor_dict[i]) == 8:
        ectupole.append(neighbor_dict[i])
        # ectupole.append(i)


```python
#ectupole.append(i)
elif len(neighbor_dict[i]) == 9:
    octupole.append(neighbor_dict[i])
#octupole.append(i)

elif len(neighbor_dict[i]) == 10:
    enneapole.append(neighbor_dict[i])
#enneapole.append(i)

elif len(neighbor_dict[i]) == 11:
    decapole.append(neighbor_dict[i])
#decapole.append(i)

double = np.unique(double, axis=0)

labeled_3D = reconstructed.copy()
boundaries = np.concatenate((np.unique(labeled_3D[0,:,:]), np.unique(labeled_3D[:,0,:]), np.unique(labeled_3D[:,:,0]), np.unique(labeled_3D[int(labeled_3D.shape[0]-1),:,:]), np.unique(labeled_3D[:,int(labeled_3D.shape[1]-1),:]), np.unique(labeled_3D[:,:,int(labeled_3D.shape[2]-1)])))

for i in np.unique(boundaries):
    if i in single:
        single.remove(i)

# Singling out Neurons.
from tifffile import imsave
from timeit import default_timer as timer
from skimage.measure import regionprops as rg

for i in single:
    full_data = reconstructed.copy()
    limz = []
    limx = []
    limy = []
    for regions in rg(reconstructed):
        if regions.label == i:
            minz, minx, miny, maxz, maxx, maxy = regions.bbox
            limz.extend((minz, maxz))
            limx.extend((minx, maxx))
            limy.extend((miny, maxy))

    test = reconstructed[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)].copy()

    test[test == i] = 255
    test[test<255] = 0
    test = test.astype('uint8')
    full_data[full_data != i] = 0
    full_data[full_data == i] = 255
```
# Saves full dataset with single neuron and a smaller image with the neuron alone
imsave('Neuron_in_3D%d.tif'%(i,), full_data)
imsave('Neuron_%d.tif'%(i,), test)

# Neurons Postprocessing
improved_full_data = np.empty(reconstructed.shape)  # Creates empty array
count = 1
neurons_list = []  # List of neurons and their respective characteristics

for i in single:  # List of neurons and their respective characteristics
    full_data = np.empty(reconstructed.shape).astype('uint8')
    limz = []
    limx = []
    limy = []
    for regions in rg(reconstructed):
        if regions.label == i:
            minz, minx, miny, maxz, maxx, maxy = regions.bbox
            limz.extend((minz, maxz))
            limx.extend((minx, maxx))
            limy.extend((miny, maxy))  # Find bounds of enclosing box
    disc = reconstructed[np.min(limz):np.max(limz), np.min(limx):np.max(limx), np.min(limy):np.max(limy)].copy()
    disc[disc == i] = 255
    disc[disc < 255] = 0  # Binarize Neuron
    improved_neuron = zoom(disc, (0.61/1.075, 1, 1))
    improved_neuron = median_filter(improved_neuron, size = (2,2,2))
    improved_neuron = gaus(improved_neuron, sigma = [0.15, 0.1, 0.1])  # Gaussian Smoothing
    improved_neuron[improved_neuron > 0] = 255
    improved_neuron = improved_neuron.astype('uint8')  # Appends neurons and characteristics to list
    neurons_list.append([limz, limx, limy, improved_neuron])

    # creates full dataset with only one neuron to append (comment out if not needed)
    full_data[np.min(limz):np.max(limz), np.min(limx):np.max(limx), np.min(limy):np.max(limy)][np.where(improved_neuron == 255)] = 255
    full_data = full_data.astype('uint8')

    # save single neuron
    imsave('Singled Neurons/good_single_%d.tif'%(count,), improved_neuron)
    # save single neuron in Full Field Format
    imsave('Singled Neurons/FFOV_good_single_%d.tif'%(count,), full_data)
    count+=1
for i in range(len(neurons_list)):
C.3 Source Code (Python3)

```python
improved_full_data[np.min(neurons_list[i][0]):np.max(neurons_list[i][0]),np.min(neurons_list[i][1]):np.max(neurons_list[i][1]),np.min(neurons_list[i][2]):np.max(neurons_list[i][2])][np.where(neurons_list[i][3] == 255)] = 255

# creates full field dataset with only singled neurons contained

improved_full_data = improved_full_data.astype('uint8')
# Saves FFOV with all single neurons
imsave('Singled Neurons/FFOV_Singles.tif', improved_full_data)
end = timer()
print(end - start)#Keeps Time

#%% Creates Probability map of neurons or astrocytes

count = 1
astro_probs = []
neuron_probs = []
prob_map = np.empty(reconstructed.shape)
neuron_map = np.empty(reconstructed.shape)
astro_map = np.empty(reconstructed.shape)
astro_labels = []

for i in single:
    full_data = reconstructed.copy()
    limz = []
    limx = []
    limy = []
    for regions in rg(reconstructed):
        if regions.label == i:
            minz, minx, miny, maxz, maxx, maxy = regions.bbox
            limz.extend((minz, maxz))
            limx.extend((minx, maxx))
            limy.extend((miny, maxy))

    test = reconstructed[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)].copy()

    test[test == i] = 255
    test[test<255] = 0
    test = test.astype('uint8')

    astro_prob, neuron_prob = cond_prob(test)
    astro_probs.append(astro_prob)
    neuron_probs.append(neuron_prob)

    if neuron_prob > astro_prob:
        prob_map[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)][np.where(test == 255)] = neuron_prob
    else:
```

74
prob_map[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)][np.where(test == 255)] = -astro_prob

astro_labels.append(i)

neuron_map[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)][np.where(test == 255)] = neuron_prob

astro_map[np.min(limz):np.max(limz),np.min(limx):np.max(limx),np.min(limy):np.max(limy)][np.where(test == 255)] = astro_prob

count +=1

# Save Probability Map
imsave('Probability_Map.tif', prob_map)
np.save('Probability_Map.npy', prob_map)

# Plot Probability Map
# Show Probability Maps
idx0 = 0
l = plt.imshow(prob_map[idx0], cmap='seismic_r', vmin=-1, vmax=1)
cbar = plt.colorbar(l)
cbar.set_ticks([-1, -0.5, 0, 0.5, 1])
cbar.set_ticklabels(['Astrocyte', '50% Probability of Astrocyte', 'Unsure', '50% Probability Neuron', 'Neuron'])

axidx = plt.axes([0.17, 0.01, 0.7, 0.03])
slidx = Slider(axidx, 'index', 0, stk.shape[0]-1, valinit=idx0, valfmt='%d')

def update(val):
    idx = slidx.val
    l.set_data(prob_map[int(idx)])
    fig.canvas.draw_idle()

slidx.on_changed(update)
cmap1 = copy.copy(plt.cm.get_cmap('coolwarm_r'))
cmap1.set_bad(color='black')
cmap2 = copy.copy(plt.cm.get_cmap('coolwarm'))
cmap2.set_bad(color='black')

plt.figure(2)
neuron_map[:, :, :] = np.nan
m = plt.imshow(neuron_map[idx0], vmin=0, vmax=1, cmap=cmap1)
cbar2 = plt.colorbar(m)
cbar2.set_ticks([0, 0.5, 1])
cbar2.set_ticklabels(['Astrocyte', 'Unsure', 'Neuron'])

axi = plt.axes([0.17, 0.01, 0.7, 0.03])
sli = Slider(axi, 'index', 0, stk.shape[0]-1, valinit=idx0, valfmt='%d')

def update(val):
    idx1 = sli.val
    m.set_data(neuron_map[int(idx1)])
    plt.draw_idle()

sli.on_changed(update)

# plt.figure(3)
# plt.imshow(astro_map[60], cmap=cmap2, vmin=0, vmax=1)

#Same thing as other plot. If multiple slices are needed use Display in Useful Functions

#%% Finding Neuron Centers (No Astro/No Bounds)
clean = labels.copy()
cell_tbr = []

boundaries = np.concatenate((np.unique(reconstructed[0,:,:]), np.unique(reconstructed[:,0,:]),
                             np.unique(reconstructed[:,:,0]), np.unique(reconstructed[int(reconstructed.shape[0]-1),:,:]),
                             np.unique(reconstructed[:,int(reconstructed.shape[1]-1),:]), np.unique(reconstructed[:,:,int(reconstructed.shape[2]-1)])))

for i in np.unique(boundaries):
    cell_tbr.append(i)

cell_tbr = np.unique(cell_tbr+astro_labels)

for i in cell_tbr:
    clean[clean == i] = 0

centroids = []
for regions in rg(clean):
    centroids.append(regions.centroid)

centroids = np.asarray(centroids)
for i in range(centroids[:,0].shape[0]):
    centroids[:,0][i] = round(centroids[:,0][i])

#%% Obtain Cellular Morphological Data
#load all astrocytes:
estrocytes = []
neurons = []
unknown = []
for i in range(1,82):
    astro = zoom(io.imread('Astrocytes/Astrocyte_%d.tif'%(i,)).astype('uint8'), (0.61/1.075,1,1))
    astro[astro>0] = 255
    astrocytes.append(astro)
for i in range(1,43):
    neuo = zoom(io.imread('Single Neurons/Neuron_%d.tif'%(i,)).astype('uint8'),(0.61/1.075,1,1))
    neuo[neuo>0] = 255
    neurons.append(neuo)

#Preparing lists for cell features
astro_area = []
astro_inertia = []
astro_bbox_area = []
neuron_area = []
neuron_inertia = []
neuron_bbox_area = []
for i in range(len(astrocytes)):
    vts, fs, ns, cs = marching_cubes(astrocytes[i])
    surface = mesh_surface_area(vts,fs)
    astro_area.append(surface*(0.61**2))
    astro_inertia.append(rg(astrocytes[i][0]).inertia_tensor_eigvals)
    astro_bbox_area.append(rg(astrocytes[i][0]).bbox_area*(0.61**2))
for i in range(len(neurons)):
    vts, fs, ns, cs = marching_cubes(neurons[i])
    surface = mesh_surface_area(vts,fs)
    neuron_area.append(surface*(0.61**2))
    neuron_inertia.append(rg(neurons[i][0]).inertia_tensor_eigvals)
    neuron_bbox_area.append(rg(neurons[i][0]).bbox_area*(0.61**2))

#Separating the moment of inertia eigenvalues
a1_in = []
a2_in = []
a3_in = []
	n1_in = []	n2_in = []	n3_in = []

u1_in = []
u2_in = []
u3_in = []
for i in range(len(astro_inertia)):
    a1_in.append((astro_inertia[i-1][0])*(0.61**2))
    a2_in.append((astro_inertia[i-1][1])*(0.61**2))
    a3_in.append((astro_inertia[i-1][2])*(0.61**2))

for i in range(len(neuron_inertia)):
    n1_in.append((neuron_inertia[i-1][0])*(0.61**2))
    n2_in.append((neuron_inertia[i-1][1])*(0.61**2))
    n3_in.append((neuron_inertia[i-1][2])*(0.61**2))

# Fitting Data to Exponentiated Weibull's Distributions
astro_area_fit = stats.exponweib.fit(astro_area, 1, 1, scale=2, floc=0)
astro_bbox_area_fit = stats.exponweib.fit(astro_bbox_area, 1, 1, scale=2, floc=0)
a1_in_fit = stats.exponweib.fit(a1_in, 1, 1, scale=2, floc=0)
a2_in_fit = stats.exponweib.fit(a2_in, 1, 1, scale=2, floc=0)
a3_in_fit = stats.exponweib.fit(a3_in, 1, 1, scale=2, floc=0)
neuron_area_fit = stats.exponweib.fit(neuron_area, 1, 1, scale=2, floc=0)
neuron_bbox_area_fit = stats.exponweib.fit(neuron_bbox_area, 1, 1, scale=2, floc=0)
n1_in_fit = stats.exponweib.fit(n1_in, 1, 1, scale=2, floc=0)
n2_in_fit = stats.exponweib.fit(n2_in, 1, 1, scale=2, floc=0)
n3_in_fit = stats.exponweib.fit(n3_in, 1, 1, scale=2, floc=0)

# Plotting the histograms and fitted distributions
ax1 = plt.subplot2grid(shape=(2,6), loc=(0,1), colspan=2)
ax2 = plt.subplot2grid((2,6), (0,3), colspan=2)
ax3 = plt.subplot2grid((2,6), (1,0), colspan=2)
ax4 = plt.subplot2grid((2,6), (1,2), colspan=2)
ax5 = plt.subplot2grid((2,6), (1,4), colspan=2)
plt.tight_layout(pad=2.0)

ax1.hist(astro_area, 20, density=True, stacked=True, alpha=0.4, color='lightsalmon', label='Astrocyte')
ax1.hist(neuron_area, 20, density=True, stacked=True, alpha=0.4, color='darkturquoise', label='Neuron', edgecolor='black', hatch='o')
ax1.plot(np.arange(0, 1.5*np.max(astro_area)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(astro_area)), *stats.exponweib.fit(astro_area, 1, 1, scale=2, floc=0)), color='darkred', linestyle='dashed', label='Astrocyte Fit')
ax1.plot(np.arange(0, 1.5*np.max(neuron_area)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(neuron_area)), *stats.exponweib.fit(neuron_area, 1, 1, scale=2, floc=0)), color='midnightblue', linestyle='-.', label='Neuron Fit')

ax1.set_ylabel('Frequency', fontsize=12)
ax1.set_xlabel('Cell Surface Area in $\mu m^2$', fontsize=12)
ax1.legend(loc='upper right', prop={'size':12})
C.3 Source Code (Python3)

```python
ax2.hist(astro_bbox_area, 20, density=True, stacked=True, alpha=0.4,
        color='lightsalmon', label='Astrocyte')
ax2.hist(neuron_bbox_area, 20, density=True, stacked=True, alpha=0.4,
        color='darkturquoise', label='Neuron', edgecolor='black', hatch='o')
ax2.plot(np.arange(0, 1.5*np.max(astro_bbox_area)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(astro_bbox_area)), *stats.exponweib.fit(astro_bbox_area, 1, 1, scale=2, floc=0)), color='darkred', linestyle='dashed', label='Astrocyte Fit')
ax2.plot(np.arange(0, 1.5*np.max(neuron_bbox_area)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(neuron_bbox_area)), *stats.exponweib.fit(neuron_bbox_area, 1, 1, scale=2, floc=0)), color='midnightblue', linestyle='--', label='Neuron Fit')
ax2.set_xlabel('Surface Area of Bounding Box in $\mu m^2$', fontsize=12)
ax2.set_ylabel('Frequency', fontsize=12)
ax2.legend(loc='upper right', prop={'size': 12})
```

```python
ax3.hist(a1_in, 20, density=True, stacked=True, alpha=0.4, color='lightsalmon', label='Astrocyte')
ax3.hist(n1_in, 20, density=True, stacked=True, alpha=0.4, color='darkturquoise', label='Neuron', edgecolor='black', hatch='o')
ax3.plot(np.arange(0, 1.5*np.max(a1_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(a1_in)), *stats.exponweib.fit(a1_in, 1, 1, scale=2, floc=0)), color='darkred', linestyle='dashed', label='Astrocyte Fit')
ax3.plot(np.arange(0, 1.5*np.max(n1_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(n1_in)), *stats.exponweib.fit(n1_in, 1, 1, scale=2, floc=0)), color='midnightblue', linestyle='--', label='Neuron Fit')
ax3.set_xlabel('First Moment of Inertia Eigenvalue in $\mu m^2$', fontsize=12)
ax3.set_ylabel('Frequency', fontsize=12)
ax3.legend(loc='upper right', prop={'size': 12})
```

```python
ax4.hist(a2_in, 20, density=True, stacked=True, alpha=0.4, color='lightsalmon', label='Astrocyte')
ax4.hist(n2_in, 20, density=True, stacked=True, alpha=0.4, color='darkturquoise', label='Neuron', edgecolor='black', hatch='o')
ax4.plot(np.arange(0, 1.5*np.max(a2_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(a2_in)), *stats.exponweib.fit(a2_in, 1, 1, scale=2, floc=0)), color='darkred',)
```
linestyle = 'dashed', label = 'Astrocyte Fit')

ax4.plot(np.arange(0, 1.5*np.max(n2_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(n2_in)), *stats.exponweib.fit(n2_in, 1, 1, scale=2, floc=0)), color = 'midnightblue', linestyle = '-.', label = 'Neuron Fit')

ax4.set_xlabel('Second Moment of Inertia Eigenvalue in $\mu m^2$', fontsize = 12)
ax4.set_ylabel('Frequency', fontsize = 12)
ax4.legend(loc = 'upper right', prop={'size': 12})

ax5.hist(a3_in, 20, density = True, stacked = True, alpha=0.4, color = 'lightsalmon', label = 'Astrocyte')
ax5.hist(n3_in, 20, density = True, stacked = True, alpha=0.4, color = 'darkturquoise', label = 'Neuron', edgecolor='black', hatch='o')
ax5.plot(np.arange(0, 1.5*np.max(a3_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(a3_in)), *stats.exponweib.fit(a3_in, 1, 1, scale=2, floc=0)), color = 'darkred', linestyle = 'dashed', label = 'Astrocyte Fit')
ax5.plot(np.arange(0, 1.5*np.max(n3_in)), stats.exponweib.pdf(np.arange(0, 1.5*np.max(n3_in)), *stats.exponweib.fit(n3_in, 1, 1, scale=2, floc=0)), color = 'midnightblue', linestyle = '-.', label = 'Neuron Fit')

ax5.set_xlabel('Third Moment of Inertia Eigenvalue in $\mu m^3$', fontsize = 12)
ax5.set_ylabel('Frequency', fontsize = 12)
ax5.legend(loc = 'upper right', prop={'size': 12})