**TITLE:**

Automatic Identification of Dendritic Branches and Their Orientation

**SUPPLEMENTAL MATERIAL:**

Several algorithms were employed to detect individual dendritic branches in the fluorescence images and conduct multiple analyses on the results.

**Segmentation**

Segmentation converts a digital image into several segments (image objects) to identify these objects and make them more understandable and available. Here, segmentation was utilized to identify dendritic branches and isolate them from the background. The image segmentation process was split into two stages: the first stage involved detection of dendritic branches as lines, and the second stage involved merging related lines based on distance and direction criteria set by the user.

Line detection:

To achieve optimal representation of the dendritic branches, two main functions from the Python OpenCV[[1]](#endnote-1) library were used:

1. HoughLinesP()[[2]](#endnote-2) - Probabilistic Hough Line Transform, OpenCV function.  
   Hough Transform is one of the basic methods for recognizing geometric shapes from images. It is widely used and has many improved algorithms. It is mainly used to separate geometric shapes (such as lines, circles) with some identical features from the image. The most basic Hough transform is to detect straight lines from black and white images (i.e., line segment). This function is called the probabilistic version of the Hough transform because it only analyzes a subset of points and estimates the probability that these points belong to a straight line. This is the optimized version of the standard Hough transform and therefore, it is more suitable for the SOA. The direct input of the Hough line transformation can only be an edge binary image. Therefore, an edge detection pre-processing is required before applying the transform.
2. Canny edge detection[[3]](#endnote-3), OpenCV function.   
   Canny Edge Detection is used to detect the edges in an image. It analyzes all edges and their connections, keeping the true edges and removing the non-obvious ones. It accepts a gray scale image as input, turns it to a binary image, and uses a 4 step algorithm:

* Use of Gaussian filter to reduce image noise. In OpenCV, image smoothing (also known as blurring) can be done using a variety of methods, one of which is the Gaussian filter function[[4]](#endnote-4). GaussianBlur (src, (5, 5), 0) convolves the image with a 2D Gaussian kernel, thus removing high spatial frequencies from the image.
* Computation of gradient. Formally, an image gradient is defined as a directional change in image intensity. At each pixel of the input (grayscale) image, a gradient measures the change in pixel intensity in each direction. By estimating the direction or orientation along with the magnitude (i.e., strength of change in direction), it can detect regions of an image that look like edges.
* Use of non-maximum suppression (NMS) on the edges. After determining the gradient's magnitude and direction, the entire image must be scanned and any pixels that are not boundaries must be eliminated. For each pixel, check whether there is a local maximum in the vicinity of the gradient direction of the pixel.
* Use of dual thresholds on detected edges to remove false positives. This stage mainly determines which are the real boundaries and which are not. Therefore, we need 2 thresholds: Threshold1 and Threshold2. Those higher than Threshold2 are boundaries while those lower than Threshold1 are not boundaries and are reset to zero. Values between the two are confirmed as boundaries based on connectivity. If it is connected to the true boundary, then it is a boundary; otherwise, it is not a boundary and is set to zero.

In SOA, this function will have only one variable: Threshold1, which we call Threshold. For the Threshold1 parameter, we must select different values to match the unique properties of each image. It is worth noting that the lines become more visible as the chosen number decreases, and the higher the number, the less background noise there is. Threshold2 is calculated automatically based on Threshold1. Threshold2 should be three times higher than Threshold1 according to Canny's recommendation. As long as Threshold1 is less than 85, Threshold2 will receive a value three times that of Threshold1. Threshold2 will have a fixed value of 255 if Threshold1 is greater and/or equal to 85.

Line merging:

The above two functions produce an image that may include multiple lines and duplicates located within a single dendritic branch. The lines can be non-overlapping or overlapping, fully or partially, and therefore must be linked or merged. To achieve that, a line merge and deduplication function[[5]](#endnote-5) was used. Because the process includes criteria for merging based on segment proximity and directional difference, the parameters for minimum distance and minimum angle should be updated for each image. If two consecutive segments are close/collinear enough, obtain the common line and find the most distant point:

* If distance < min distance
* Check the angle between lines: if the angle between them < min angle to merge, merge both segments.
* Determine the new merge line by calculating minimum distance from a point and a line segment (i.e., consecutive vertices in a polyline). A polyline is just a series of line segments; hence, the basis of measurement is to loop through all the consecutive pairs of vertices using the minimum distance found. If the closest point does not fall within the line segment, take the shorter distance to an endpoint.
* Update the new lines (after merging) in the line list with their angle, length, and coordinates.

**Dendritic information**

For each line marked, the information on coordinate, angle, and length was collected.

Coordinates:

Coordinates of each marked line were obtained after completion of segmentation. The two endpoints of the detected line segment are represented by four numbers ((x0,y0)(x1,y1)) for each line. These coordinates are the initial data, allowing for calculation of the length and direction angle of each line.

Angle:

Use of the math.atan2() method from the Python math module, which provides access to certain common math functions and constants: math.atan2((y0 – y1), (x0 – x1)) returns atan(y / x), in radians (between -π and π). The vector in the plane from the origin to point (x, y) shows this angle in relation to the positive x axis. The default value of the angle is in degrees (°) (0–180° range).

Length:

Use of the math.sqrt() method from the Python math module. The length of each line is calculated from the distance between the two endpoints using math.sqrt((x1 - x0) \*\* 2 + (y1 - y0) \*\* 2).

**Analysis**

The following analysis can be performed on the gathered data:

1. Classification of Parallel/Non-Parallel dendritic branches

The map() function, part of a built-in function in the Python interpreter, applies a given function to each item of an iterable sequence (list, tuple) and returns a list of the results. The map() function is used to distinguish parallel and non-parallel groups. As line parallelism is measured by comparing angles, we defined parallelism in this study as having a deviation of ±angle(=180/(number of lines)), as long as this deviation does not exceed 5°. For each dendritic branch, we added ±angle(=180/(number of lines)) to the angle of that branch and we got a range of angles. All other branches identified in the image were examined to see if their angle was within this range. Based on that, the number of groups of pairs, triples, quartets of parallel dendritic branches was calculated.

To test whether this growth pattern is random or directed, SOA compares the measured results to the results of a simulation of random line distribution. To simulate random line distribution, the formula of binomial distribution was used:

When *n* represents the number of attempts (number of segments), *k* the number of repetitions (number of parallel lines from each group), *p* is the probability of success (the probability that a segment will fall within the range of another segment), which was defined as *1/n*. The numerical results are fractions. Therefore, for convenience, all the results are multiplied by *n*, providing the number of parallel lines expected in each group:

1. Measurement of average length of parallel vs. non-parallel dendritic branches

The number of dendritic branches as a function of their lengths is displayed as a bar graph for each group. In addition, the average lengths of the parallel and non-parallel dendritic branches are calculated.

1. Measuring and display of angular distribution

A bar graph and an angular scatter plot were used to display the angular distribution groups and the corresponding occurrence of mutual orientation angles of other dendritic branches. The columns on the X axis are divided by 180/(number of lines). In the bar graph, the number of dendritic branches in each group of angles can be obtained. In addition, the identity (ID) of a branch can be related to a particular angle group.

1. Measurement of distance among parallel dendritic branches

As parallelism was defined as having a ±angle deviation (=180/(number of lines)), the distance between a pair of parallel dendritic branches is not constant. Therefore, there was a need for a function to find the shortest distance between pairs of parallel dendritic branches. Using open-source code[[6]](#endnote-6), Numpy.array pairs (a0,a1,b0,b1) was chosen from NumPy, a fundamental package for scientific computing. This function returns the shortest distance between two line segments.

**System**

SOA runs on Windows 10 using Python 3.9 with the following main packages:

* OpenCV 4.5.2.54 - designed to solve computer vision problems.
* NumPy 1.20.3 - fundamental package for scientific computing.
* Matplotlib 3.4.2 - a Python 2D plotting library.
* PyQt5 5.15.4 - to manage the GUI.
* Xlsxwriter 1.4.3 - a Python module for creating Excel XLSX files.
* Seaborn 0.11.1 - Python's statistical data visualization library.
* Scipy 1.6.3 - Python-based ecosystem of open-source software for mathematics, science, and engineering.
* matplotlib-scalebar 0.7.2 - artist for matplotlib to display a scale bar.

**REFERENCES:**

1. . Home - OpenCV. OpenCV at <https://opencv.org/> (2021). [↑](#endnote-ref-1)
2. . Python+OpenCV Image Processing - Line Detection Introduction: - Programmer Sought. https://www.programmersought.com/article/8317733745/ [↑](#endnote-ref-2)
3. . Canny Edge Detection. OpenCV. Available: https://docs.opencv.org/master/da/d22/tutorial\_py\_canny.html [↑](#endnote-ref-3)
4. . OpenCV Gaussian Blur - cv2.gaussianblur() - Example. TutorialKart. TutorialKart; 2021. Available: https://www.tutorialkart.com/opencv/python/opencv-python-gaussian-image-smoothing/ [↑](#endnote-ref-4)
5. . How to merge lines after HoughLinesP? Stack Overflow. 1966. Available: https://stackoverflow.com/questions/45531074/how-to-merge-lines-after-houghlinesp [↑](#endnote-ref-5)
6. . Shortest distance between two line segments. Stack Overflow. 1959. Available: https://stackoverflow.com/questions/2824478/shortest-distance-between-two-line-segments?rq=1 [↑](#endnote-ref-6)