Root Necrosis Code Documentation

Aaron Lin

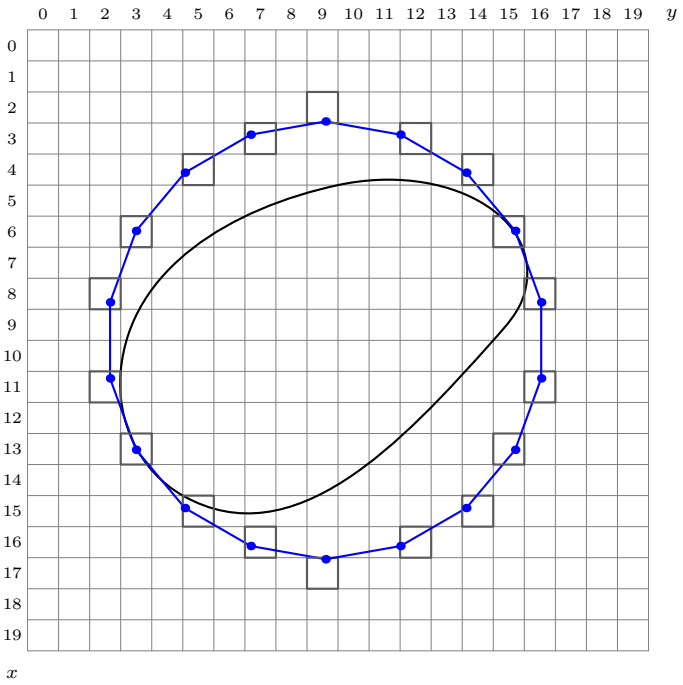
Makerere University, Summer 2015

1 How to Use This Document

Read section 2 and 4. Use section 3 as reference. Ask me questions if something is confusing.

2 General Overview

1. We first aim to find the general focus of the inputted image by using `cv2.threshold` and `cv2.findContours`. The former function converts the image into black/white given a threshold t, while the second returns a list of all detected contours. We select the t for which the change in the area of the corresponding longest contour is lowest.
2. The minimum enclosing circle (MEC) of this region is inexpensive to compute. We assume that the entirety of the root is contained within this circle. Note that the circle is encoded as 360-sided polygon.



3. We first create a enlarged circle (EC), concentric to the MEC (with center C) and with a 10% larger radius. For each of the 360 vertices V of the EC, we compute all pixels of the image that lie on the line segment $C - -V$. The computation is done in the class `bresenham`'s init function.

4. For each line segment $C--V_i$ (for $i \in \{0, 1, \cdots 359\}$), we iterate through the corresponding list of pixels, starting from the outer pixels. For each pair of adjacent pixels P_0 and P_1 in the list, we compute a score involving the vector distance between the color coordinates (L*a*b) of P_0 and P_1 and return the pixel corresponding to the highest score. There are several restrictions: (1) the jump must lie within the MEC (2) the distance from C to the pixels chosen along segment $C--V_i$ must be reasonably close to the distance from C to the pixels chosen along $C--V_{i-1}$ (3) the score for each pair of pixels along segment $C--V_i$ is weighted by comparing color similarity to the selected pair of pixels for $C--V_{i-1}$. This returns two approximations for the perimeter of the root, one obtained by iterating clockwise and the other iterating counter-clockwise.
5. The two perimeter approximations are combined together, returning a single contour of the root. The current approach is to select the pixel with a higher score for each line segment $C--V_i$.
6. We use the EM-clustering algorithm to classify the L*a*b coordinates for each pixel inside of the contour detected above.
7. It is currently unclear what the appropriate number of clusters is and how to classify each cluster as healthy/necrotized/other sections.

3 Methods

`[18]` `def pointDistance(P, Q):`

- Returns the distance between vectors `P` and `Q`

`[21]` `def display(image, msg=""):`

- Short for displaying `image`, with a default title message of `msg`; does not return anything

`[24]` `def dot(image, center, color=(255,200,0), radius=2):`

- Draws a dot on image; default color is cyan

`[28]` `def readImage(pathNo):`

- Reads image in the **same directory** with a pathname in the form of `"image`X`.jpg"` for some number X. Also resizes to 1/3 size.

`[33]` `def miscFunction():`

- Inactive
- Old code; does nothing.

`[67]` `def slowestChange(array):`

- This method computes the element in an array at which the numbers change slowest; i.e. minimizes the first derivative. If the elements of the array are denoted as $a_1, a_2, \cdots, a_n$, it returns the pair (i, a_i) for which $|a_{i-1}/a_i|$ is minimized.

`[75]` `def getLongestContour(image, t):`

- For a given threshold value `t` ($t \in [0, 256)$), the image is mapped into a binary image with `cv2.threshold`. The function returns the longest contour found with `cv2.findContours`.

[87] `def getBestContour(im, range_of_t=[...]):`

- For each value of `t` in range_of_t, the longest contours returned by `getLongestContour`are stored in an array. The value of `t` for which the length of the longest contour changes slowest is computed with `slowestChange`. The value of `t` and the contour are both returned.

[114] `def circleToContour(imsize, (x,y), r):`

- Inactive
- Converts a `cv2.circle` with center `(x,y)` and radius `r` into a contour/polygon.

[122] `def circleToRadialContour(imsize, (x,y), r, steps=360):`

- Given a `cv2.circle` with center C =`(x,y)` and radius `r`, the image returns a `steps`-sided polygon to approximate the circle. Two adjacent vertices V_i and V_{i+1} should satisfy that $\angle V_i C V_{i+1} = 2\pi/$`steps` for all i. If parts of the circle are cut off by the boundaries of the image, we use pixels on the edge of the image.

[132] `def getRegion(image, contour):`

- Returns a list of all pixels inside of `contour`.
- Fairly high complexity

[145] `def nearestColor(center, validColors):`

- Inactive
- Returns the color c in `validColors` that is geometrically closest (i.e. vector distance) to the given color `center`.

[148] `def replaceCenters(labels, centers, smallest=5):`

- Inactive
- If an iteration of k-means returns a cluster with less than or equal to `smallest` pixels, then the colors of the centers of the corresponding clusters are reassigned to the color of most similar center-color, using `nearestColor`.

[155] `def hls2bgr((h,l,s)):`

- Returns the equivalent BGR color given an HLS color.

[158] `def bgr2hls((b,g,r)):`

- Returns the equivalent HLS color given a BGR color.

[161] `def lab2bgr((L,a,b)):`

- Returns the equivalent BGR color given a L*a*b color.

[165] `def replaceSmallClusters(focusColors, labels, centers, smallest=5):`

- Inactive
- If an iteration of k-means returns a cluster with less than or equal to `smallest` pixels, then the colors of the pixels in such clusters are reassigned to the modified cluster color, determined by `replaceCenters`.

[181] `def getHLScoordinates(image, val=100):`

- Inactive
- Eliminates lightness component for pixels in `image`. Returns colors either as BGR or HLS.

`[189] def getLABcoordinates(image):`

- Converts image from BGR to L*a*b.

`[192] def getKmeansClusters(k, focusColors, maxInterations=...):`

- Inactive
- Clusters the pixel values given in `focusColors` into `k` clusters using the k-means algorithm. Tteratively eliminates small clusters using `replaceSmallClusters`.

`[209] def getEMClusters(k, focusColors):`

- Clusters pixel values given in `focusColors` into `k` clusters using the Expectation-Maximization (EM) algorithm. Returns the log-likelihood value, cluster labels, probabilities of each pixel belonging to each cluster, and the centers of the clusters.

`[218] def noSmallCentroids(image, k, focusColors, focusRegion):`

- Inactive

`[241] def getMedianCenterColor(image,x,y,s=10):`

- Inactive
- Does nothing

`[245] def bresenham.__init__(self, image, start, end):`

- Initializes instance of a `bresenham` object, which encodes all pixels along a certain line segment. Object properties include `self.image`, `self.path` (geometric coordinates of all relevant pixels), and `self.pixels` (pixel color values corresponding to the coordinates in `self.path`). Because this object is only used for the purpose of getting pixels along a radius, after initialization, the properties `self.start`,`self.center`, and `self.length` correspond to the pixel on the circumference, the center, and the radius respectively.

`[302] def colorDiff(P,Q):`

- Returns the vector difference between two colors `P` and `Q`, raised to the 3/2 power.

`[305] def findPixelJump(radius, r, winMin=0, winMax=-1, outer=None, inner=None, prevPoint=-1, eFactor=1.15, t=3.0):`

- Given an instance of a `bresenham` object `radius` with radius length `r`, this function iterates through the pixels comprised of `radius` and detects the pair of adjacent pixels with the biggest "jump".
- The variables `pixels` and `coords` inside of the function definition are used to denote the pixel color values and the image coordinates respectively, with each list having `length` elements. The constant `k1` affects the number of pixels on the outer part of the radius to be considered a valid jump point. Because the input radius is the radius of an enlarged circle as opposed to that of the minimum enclosing circle, it is important to cut off the outermost `eFactor` pixels. Increasing the value of `k1` from 0 cuts off fewer pixels. The constant `k2` restricts the search for the jump point from the inner part of the root, as the edge of the root is unlikely to be

near the center of the circle. A `k2` value of 0.7 means that the inner 30% of each radius will not be considered. Default value is 0.5.

- The method `weightDist` assigns a weight value to each pair of adjacent pixels, comprised of several factors. If the pixels are too close to the circumference (adjusted by `k1`), they are automatically assigned a weight of 0. Otherwise, each pair of adjacent pixels is weighted by their color similarity using the `colorDiff` function, computed and stored in `diff` (line 320). There are several optional weighting options. The optional inputs `outer` and `inner` denote color values of the two pixels with the highest score in the iteration of this function on the previous `radius`. For example, if the previous `radius` found a jump from a brown to a white pixel, we would want to prioritize another jump from a brown to white pixel over a jump from a blue to red pixel. This weighting factor is expressed in `color_sim`. Additionally, we expect pixel jumps to be geometrically near the previously detected pixel jump. For example, for an enlarged circle of radius 100, if we find a pixel jump at 90 units from the center for a certain radius, it would be surprising if the pixel jump for the next radius were 50 units from the center; we'd expect it to be between 85 and 95 units. The `dist_weight` factor takes care of this.

[352] `def similarity(P,Q):`

- Computes the normalized dot product of the vectors `P` and `Q`.

[357] `def cosWRTcenter(C,P,Q):`

- Given the three geometric coordinates of points `C`, `P`, and `Q`, this function computes $\cos \angle PQC$.

[366] `def cosine(C,P,Q):`

- Currently equivalent to `cosWRTcenter`.

[369] `def getMedianIndex(edgePoints, center):`

- Inactive
- This method computes the distances from each point in `edgePoints` to `center` and returns the point for which its distance is the median.

[380] `def aS1(C,P,Q):`

- Inactive

[386] `def aS2(C,P,Q,s):`

- Inactive

[394] `def mediocreContour(C, contour, pm=3):`

- Inactive
- Given a center `C` and approximate contour `contour`, this method seeks to smooth the contour. Suppose this contour is an n-sided polygon with vertices $V_1V_2V_3\cdots V_n$ and let $p =$`pm`. For each $i \in \{1, 2, \cdots, n\}$, we compute the distances from each of $V_{i-p}, V_{i-(p-1)}, \cdots, V_i, \cdots, V_{i+(p-1)}, V_{i+p}$ from `C` and take the median distance m. We replace V_i with the point V_i' that lies in the same direction but with a distance of m from `C`.

[407] `def getWholeContour(radii, windowPercent=0.1):`

- Automates the procedure of using `findPixelJump` to get an entire contour. The first part of the code starts with the first radius in the list `radii`. For each radius thereafter, we collect information concerning the previous radius, such as the colors of the two pixels in its jump and the jump's distance from the center. The `windowPercent` factor restricts the radial distance of each successive jump. If a certain jump was found to occur 50 units from the center for a radius of length 100, then the next jump must lie between 40 and 60 units from the center.

[435] `def getBothContours(radii):`

- Because `getWholeContour` iterates through the radii in an explicitly specified direction (clockwise by default), there is some inherent bias in the detected contour. This method first computes the contours detected by iterating both clockwise and counter-clockwise. Thus, each of the 360 different radial directions has two candidate points. We select the appropriate candidate by iteratively considering their weighted scores (from the `getWholeContour`process), the angle formed with the adjacent jump points, and distances from the adjacent jump points. Only one candidate is chosen for each radial direction.
- This function is still far from perfect. It still assumes a certain direction in the process.

[459] `def partialReverse(array):`

- Returns the array `[array[0],array[-1],array[-2],...,array[-(n-1)]]`, where `n = len(array)`.

[462] `def mostLikelyEdge(radii):`

- Reorders the list of radii `radii` such that the radius with the highest score from `findPixelJump` is the first element in the array.

[467] `def evalThreeClusters(labels, probs, labcenters):`

- The post processing phase in which the pixels inside of the determined root perimeter are clustered (by color) into three groups. Currently uses a bag of hacks to determine which cluster is white (lowest standard deviation in BGR) and brown (unsure). This function needs more work.

4 Main Function (starts on line 504)

4.1 Outline of methods used

- `readImage`
- `getBestContour`
  - `getLongestContour`
    - `cv2.threshold`
    - `cv2.findContours`
  - `slowestChange`
- `cv2.minEnclosingCircle`
- `cv2.resize`
- `circleToRadialContour`
- `getLABcoordinates`

- bresenham.__init__
- mostLikelyEdge
    - findPixelJump
        - pointDistance
- getBothContours
    - getWholeContour
        - findPixelJump
            - pointDistance
            - findPixelJump.weightDist
    - partialReverse
    - cosine
    - pointDistance
- getLABcoordinates
- getRegion
    - cv2.pointPolygonTest
- getEMClusters
    - cv2.EM
- lab2bgr
- evalThreeClusters

5 Citations

- OpenCV 2.4
- NumPy
- bresenham class