Image Understanding – Assignment #3

Finding Curves

1. Part I-Boundary Curve Detection

1.1 Organization of the code

The main function is BoundaryDetection_Main.m. The information of this function is shown in Table 1.1.

Table 1. 1 Description of BoundaryDetection_Main Function

Name	BoundaryDetection_Main		
Description	Calculate the boundary from (x1, y1) to (x2, y2) by calculating the shortest path between those two pixels. Show the image and the boundary		
	Name	Type	Description
Input	*path*	string	the path and name of the given image
	x1	int	the x coordinate of the starting pixel
	y1	int	the y coordinate of the starting pixel
	x2	int	the x coordinate of the ending pixel
	y2	int	the y coordinate of the ending pixel

Also, there are several helper functions which are called by the main function. Below is the description of those functions.

Table 1.2 Description of BoundaryDetection Function

Name	BoundaryDetection		
Description	Calculate the boundary from (x1, y1) to (x2, y2) by calculating the shortest path between those two pixels.		
Input	Name	Type	Description
	I	matrix	the matrix storing the input image
	x1	int	the x coordinate of the starting pixel
	y1	int	the y coordinate of the starting pixel
	x2	int	the x coordinate of the ending pixel
	y2	int	the y coordinate of the ending pixel
Output	*p*	vector	the shortest path from starting point to ending point

Table 1.3 Description of GenerateGraph Function

Name	GenerateGraph		
Description	Generate the adjacent matrix of the graph based on the input image and the gradient matrix		
Input	Name	Type	Description
	I	matrix	the matrix storing the input image
	J	matrix	the matrix storing the gradient magnitude of each pixel
	mode	bool	determine how to calculate the weight of an edge
Output	*E*	cell	the adjacent matrix standing for the graph

Table 1.4 Description of DrawLines Function

Name	DrawLines		
Description	Adding the calculated path onto the original image		
Input	Name	Type	Description
	I	matrix	the matrix storing the input image
	p	vector	the shortest path from starting point to ending point
Output	*I*	matrix	the input image with the pixels on the path marked by 255

The function to calculate the shortest path, “shortestpaths” and function to trace the shortest path, “trace” is provided by Prof. Felzenszwalb.

1.2 Generate the graph based on the input image

To find the shortest path between two pixels, the image should be transformed into a graph firstly. Several steps are used:

Step 1: Change the coordinate of the pixels in the image into index. Let *rownum_I* and *colnum_I* be the size of the input image, then the vertex index of a pixel*(i,j)*

$$Index = (i-1) * rownum\_I + j$$

Step 2: Loop all the neighbors of the pixel *(i,j)*, and calculate the weight the edge connecting the pixel and its neighboring pixel. In this implementation, 8 neighboring pixels are used.

Step 3: Calculate the weight of the edge according to the gradient magnitude of the two ending points of the edge. If a boundary curve is needed, then we should encourage the shortest path to go along where pixels have high gradient. Let matrix G store the gradient magnitude of each pixel of the input image. The weight of edge between pixel (i,j) and (m,n) can be calculated by :

$$weight = 1/[G(i,j)-G(m,n)]^2$$

Step 4: Store the adjacent matrix of the graph in a cell of matrix. *E{v}* is the set of the edges starting from the *v* th vertex. *E{v}(i,1)* is the *i* th neighbor `s vertex index; *E{v}(i,2)* is the weight of the edge between *v* and *i.*

1.3 Result of the boundary curve detection

Below are the results of the boundary curve detection.

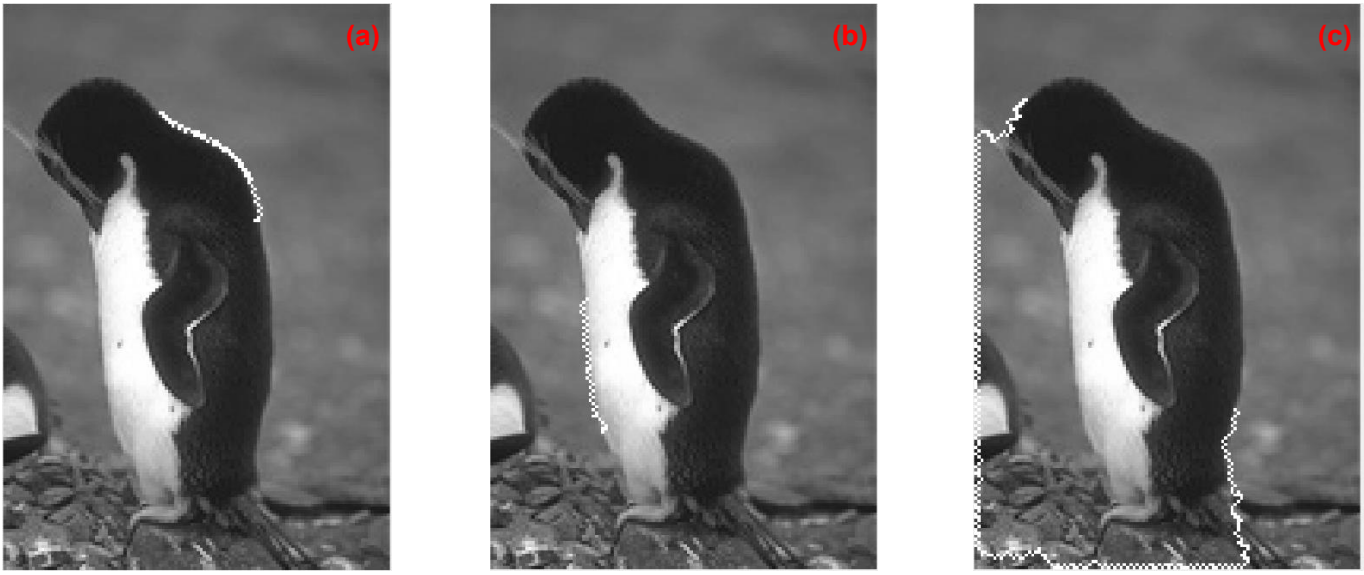

Fig 1.1 Boundary detection of "penguin". Fig 1.1(a): starting pixel (50,35) and ending pixel of (81,69); Fig 1.1 (b) starting point (31,94) and ending point (36,136); Fig 1.1 (c) starting point (17,31) and ending point (83,129)

According to the result, this boundary curve detection can work quite well when the starting point and ending point are near the boundary. However, when the path is long, the method tends to generate path going along the boundary. This shortcoming will be improved in Part3, where interaction with user is added.

2. Part II-Centerline Detection

2.1 Organization of the code

The main function is CenterlineDetection_Main.m. The information of this function is shown in Table 2.1.

Table 2. 1 Description of CenterlineDetection_Main Function

Name	CenterlineDetection_Main		
Description	Calculate the centerline from (x1, y1) to (x2, y2) by calculating the shortest path between those two pixels. Show the image and the centerline		
	Name	Type	Description
Input	*path*	string	the path and name of the given image
	x1	int	the x coordinate of the starting pixel
	y1	int	the y coordinate of the starting pixel
	x2	int	the x coordinate of the ending pixel
	y2	int	the y coordinate of the ending pixel

Also, there are several helper functions which are called by the main function. Below is the description of those functions.

Table 2.2 Description of CenterlineDetection Function

Name	CenterlineDetection		
Description	Calculate the centerline from (x1, y1) to (x2, y2) by calculating the shortest path between those two pixels.		
Input	Name	Type	Description
	I	matrix	the matrix storing the input image
	x1	int	the x coordinate of the starting pixel
	y1	int	the y coordinate of the starting pixel
	x2	int	the x coordinate of the ending pixel
	y2	int	the y coordinate of the ending pixel
Output	*p*	vector	the shortest path from starting point to ending point

The function to generate the graph`s adjacent matrix and draw the centerline on the input image is the same function introduced in Part 1. The function to calculate the shortest path, "shortestpaths" and function to trace the shortest path, "trace" is provided by Prof. Felzenszwalb. Besides, the given input image is smoothed by gaussian filter. The GaussianSmooth function and Convolution Function are the exactly same functions implemented in Assignment 1.

2.2 Difference from boundary curve detection

For centerline detection, there are two major difference from boundary curve detection.

1. The input image is needed to be smoothed by the difference-of-gaussian (DOG) filter.
2. The edge weight calculation is modified to encourage path to go through the center. Let matrix G store the gradient magnitude of each pixel of the input image. The weight of edge between pixel (i, j) and (m, n) can be calculated by :

$$weight = [G(i,j)-G(m,n)]^2$$

2.3 Implementation of difference-of-gaussian filter

The difference-of-gaussian filter is implemented by smoothing the input image by two gaussian filters which have different standard deviation. Several steps are used:

Step 1: Smooth the image with gaussian filter 1, which has a standard deviation σ1. Store the result in matrix G1.

Step 2: Smooth the image with gaussian filter 2, which has a standard deviation σ2. Store the result in matrix G2.

Step 3: Calculate the final result G, which equals G1-G2.

The retina image smoothed by a difference of gaussian(dog) filter is shown as below.

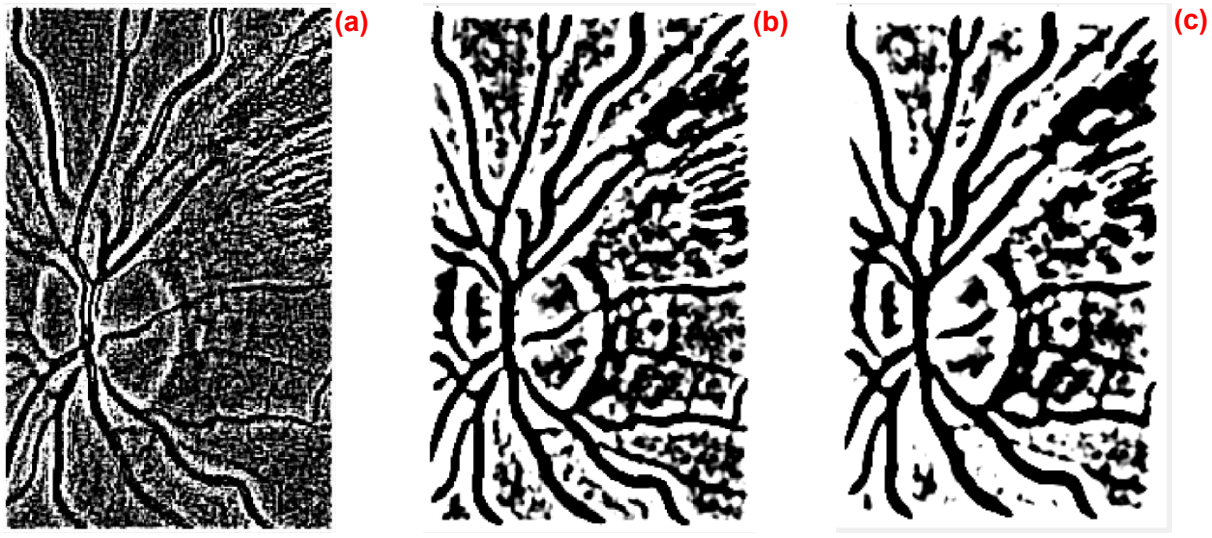

Fig 2.1 Result of dog smooth. Fig 2.1(a): σ1 = 0.5, σ2 = 1; Fig 2.1(b): σ1 = 1, σ2 = 3; Fig 2.1(c): σ1 = 1, σ2 = 5.

In the implementation, σ1 = 1, σ2 = 3 is chosen for the difference of gaussian filter.

2.4 Result of the centerline detection

Below are the results of the centerline detection.

Fig 2.2 Centerline detection of "retina". Fig 2.2(a): starting pixel (123,14) and ending pixel of (87,78); Fig 2.2 (b) starting point(9,9) and ending point(23,48); Fig 2.2 (c) starting point(9,9) and ending point (25,54)

According to the result, this centerline curve detection can work well when the starting point and ending point are in the center of the artery. However, when the path is long, the method tends to generate path wrongly. This shortcoming will be improved in Part3, where interaction with user is added.

3. Part III-Interactive Tool

3.1 GUI of the tool

An interactive tool based on guide toolbox of matlab is developed to interacted with the used to give starting pixel and ending pixel by clicking on the figure.

Below is the GUI of the interactive tool and the description of each control on the GUI.

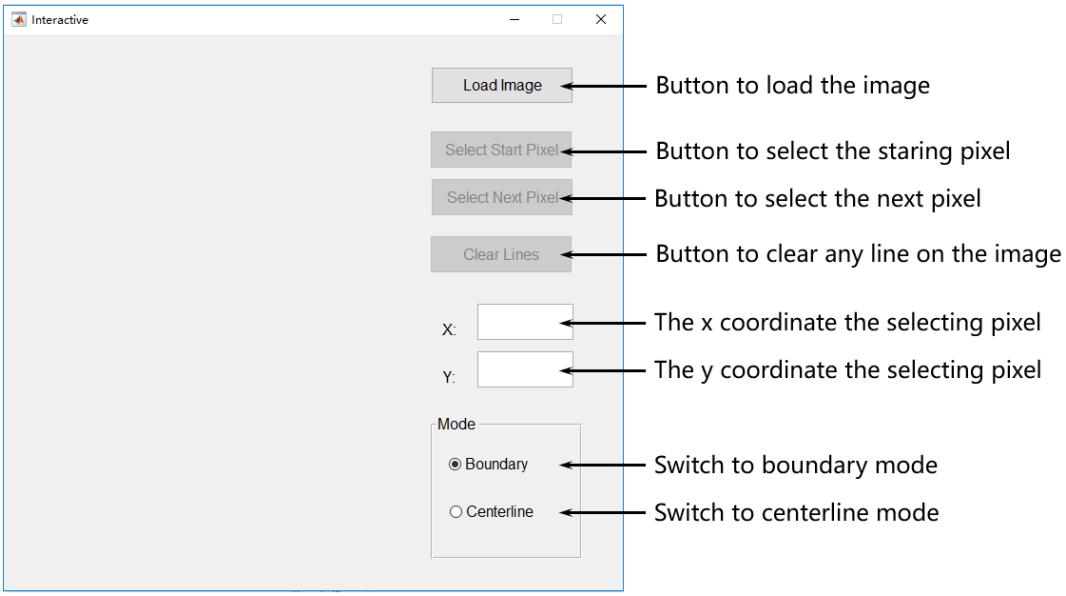

Fig 3.1 GUI of interactive tool and description of each control

3.2 Steps for usage

The steps for usage are shown as below:

Step 1: Click "Load Image" button to open the open file dialog form to select the image as shown in Fig 3.2. After an image is selected, the image will be shown in the left part of the GUI.

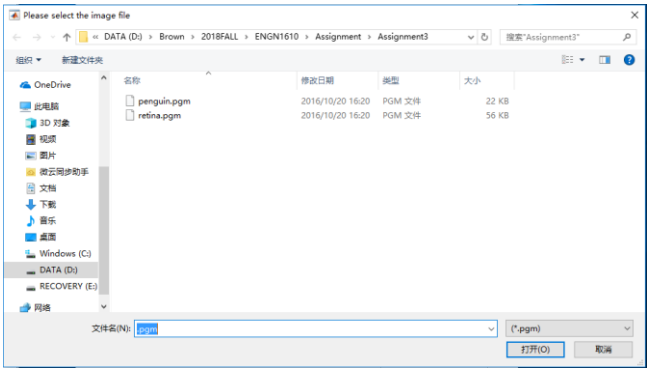

Fig 3.2 Dialog form to select image

Step 2: Click "Select Start Pixel" button and then a cross appears on the image, user can click on the image to select the starting point as shown in Fig 3.3.

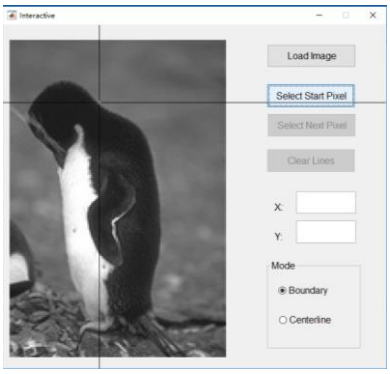

Fig 3.3 Select a pixel from image

Step 3: Click "Select Next Pixel" button to select the ending pixel. User can keep selecting next pixel to form a whole boundary. Each time, the last ending pixel will become the current starting pixel, as shown in Fig 3.4.

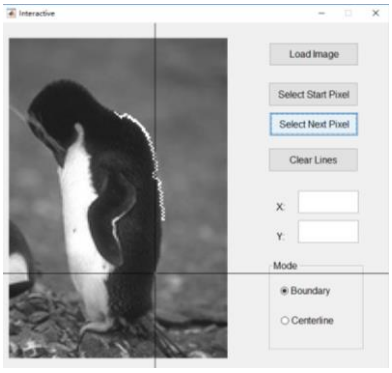

Fig 3.4 Select a pixel from image

Step 4: User can select a new starting pixel by **Step 2**, and create a new boundary by repeating **Step 2** and **Step 3** with the previous boundary still existing as shown in Fig 3.5.

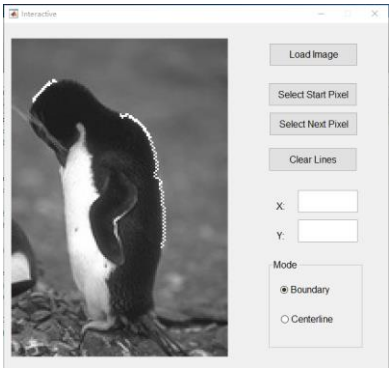

Fig 3.5 Detect multiple boundaries

Step 5: By clicking the "Clear Lines" button to clear all the boundary on the image.

Step 6: Select "Boundary" and "Centerline" mode by clicking the corresponding ratio button.

3.3 Conclusion

Basically, this program can detect a boundary or centerline based on user`s interaction. However, there is still some illegal input that this program cannot deal with properly.

4. Part IV- Dijkstra Method to Calculate Shortest Path

4.1 Organization of the code

The function using Dijkstra method to calculate shortest path is DijkstraShortestPaths.m. The information of this function is shown in Table 4.1.

Table 4.1 Description of DijkstraShortestPaths Function

Name	DijkstraShortestPaths		
Description	Adding the calculated path onto the original image		
Input	Name	Type	Description
	E	cell	the adjacent matrix of the graph
	s	int	the index of the starting vertex
Output	*prev*	vector	vector storing the shortest path from given vertex s

4.2 Implementation of Dijkstra Method

Several steps are used to implement the Dijkstra method:

Step 1: A array "*dist*" is declared with the length of the number of the vertexes. In this array, *dist[i]* is used to store the distance, which is the sum of edge weight, from the *i* th vertex to the given starting vertex *s*. In the initialization, *dist[s]* = 0 and other elements in *dist* are infinite, in this implementation 1e10.

Step 2: A array "*prev*" is declared with the length of the number of the vertexes. This vector is used to the store the vertexes which were found in the shortest path.

Step 3: A array "*visited*" is declared with the length of the number of the vertexes. This vector is used to store whether a vertex has been visited.

Step 4: Starting with the starting vertex *s*, if a vertex *u* is connected with *s* by an edge, which means *u* is a neighbor of *s*. Then *dist[u]* will be assigned by the weight of the edge connecting *s* and *u*. In vector *prev*, *prev[u]* will be assigned by *s*, because at this moment s is the nearest vertex of *u*.

Step 5: In next iteration, the *dist* vector will be traversed to find the smallest distance to starting vertex *s*, which has not been visited. This time, this vertex will be regarded as the current vertex in this iteration to find all its neighboring vertexes. The program will check whether by using the current vertex as an "intermediary", the distance from its neighboring vertexes to the starting vertex is smaller than the distance directly to starting vertex. If the distance using current vertex as an "intermediary" is smaller than the corresponding distance stored in vector *dist* will be overwritten.

Step 6: Repeat **Step 5** until all vertexes have been visited.

4.3 Big O analysis

The Dijkstra method traverse all the nodes. In each iteration, all nodes will be traversed to find smallest distance and all neighboring vertexes of the vertex founded will be checked for its distance to the starting vertex. Let n be the number of the vertexes. The time complexity of the Dijkstra method is $O(n^2)$.

According to the introduction of Dijkstra method in Wikipedia, the Dijkstra method can run in O(|E|+nlogn), in which |E| is the number of edges, if a min-priority queue implemented by a Fibonacci heap is used.

Unlike the standard "priority_queue" provided by C++, matlab didn`t provide such built-in data structure. Stack, which is used in Assignment1, or queue are simple data structures based on array, I suppose they are the limit of my ability. However, building a priority queue in matlab is beyond my capacity. So, in the assignment, I just implemented the origin Dijkstra method.

In the final version, both BoundaryDetection function and Centerline Detection used Dijkstra method to calculate the shortest path.