VEIN ROUTE TRACING

JENNIFER GIDDINGS, NEN HUYNH,
ZAC SCHOENROCK, MARILYN VAZQUEZ

Abstract. We had twelve weeks to design and develop a program that would automatically identify veins on an image of a placenta. In order to accomplish this task, we intended to write a program in MATLAB to identify the edges of the veins in the image, find points in the center of the veins based on their edges, identify key characteristics of the points in the center of the veins, and then use those characteristics to correlate the points and rebuild the veins mathematically. By visualizing the image as an intensity map in MATLAB, we developed a strong and workable plan to accomplish these goals. We succeeded on the first three of these four steps before running out of time and are confident that we would have been able to complete the final step given enough time. Our method has a strong chance of making connections often lacking in other automated processes, especially in the case of smaller veins. In order to finish the program, we would next need to correlate the characteristics of the points identified to be in the center of the veins to reconstruct each vein.

1. Introduction

The placenta is the unsung hero of pregnancy, keeping the fetus alive, healthy, and well fed while in the mother's womb. It is the temporary organ that provides nutrients and removes waste. The placenta has recently come into focus as a potential predictor of adolescent pathologies and adult medical conditions. How much of someone's future can be foretold by their placenta is still an open question. We have studied the veins of the placenta and hope that our work will help doctors plan preventative maintenance to keep people healthy.

The structure of the veins on the surface of the placenta may tell us more about the baby's future. We are focusing on building an automated program to make a schematic of the vein structure from the placenta image. We intend to develop an algorithm that will automatically identify the positions of the veins. Furthermore, our algorithm will determine a mathematical representation of each vein, so that we can fill in the gaps left by other image processing techniques. This will hopefully build complete veins instead of segments to be connected by hand. Almoussa et al. has already developed a program that identifies and draws a picture of the veins [1]. Their vein structure has a large number of gaps in veins that would clearly be connected

Date: May 20, 2011.

if the veins were traced by hand, as can be seen in figure 1. Unfortunately, generating maps of vein structures by hand is extremely time consuming and expensive. Our mathematical representations of the veins should enhance the others' results [1] without consuming the time and resources required to go through each of thousands of images by hand.

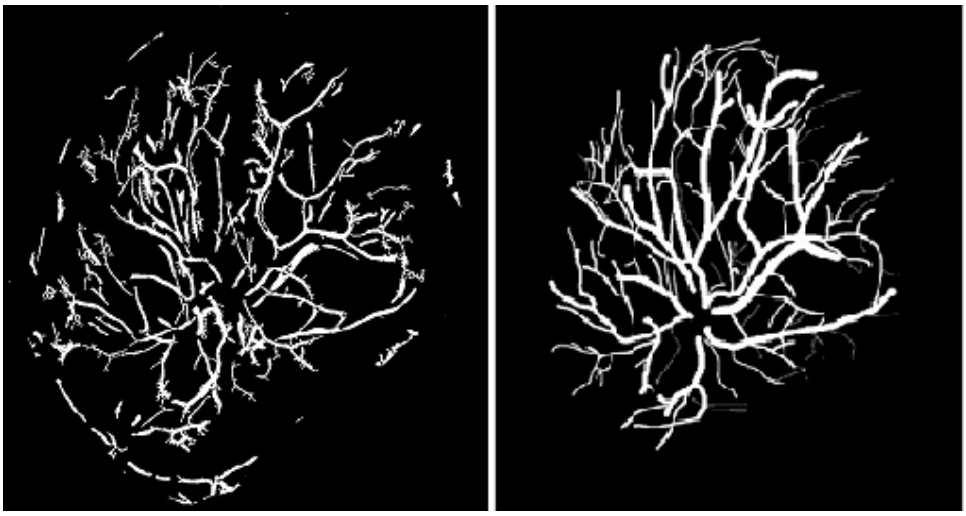

FIGURE 1. Left is an output image from the Almoussa, et al. [1]. Many of the veins are broken and do not connect as a clear system. To the right is their hand traced version of the same image.

2. METHODS

2.1. **Outline of Methods.** The programs and ideas behind this project were all designed and built from the ground up rather than building off of the work of others. We did this for a variety of reasons, but primarily because of the time constraint we had to complete this work. We chose to use a simple methodology rather than spend time patching together what others have done.

Our sample image is an 81 by 81 pixel section of the original 1024 by 768 placenta image in figure 2. We chose to use this particular cropped image because it has at least one large and one small vein that needed to be identified, as well as an interesting feature of the veins crossing each other. An intensity map of the image was made in MATLAB. We used Canny Edge Detection to locate the edges of the veins. Then we determined criteria for and located "strong points" as pixels in the middle of veins. Several characteristics of these strong points were identified to be used to connect the strong points together into a complete vein based on a correlation of these characteristics.

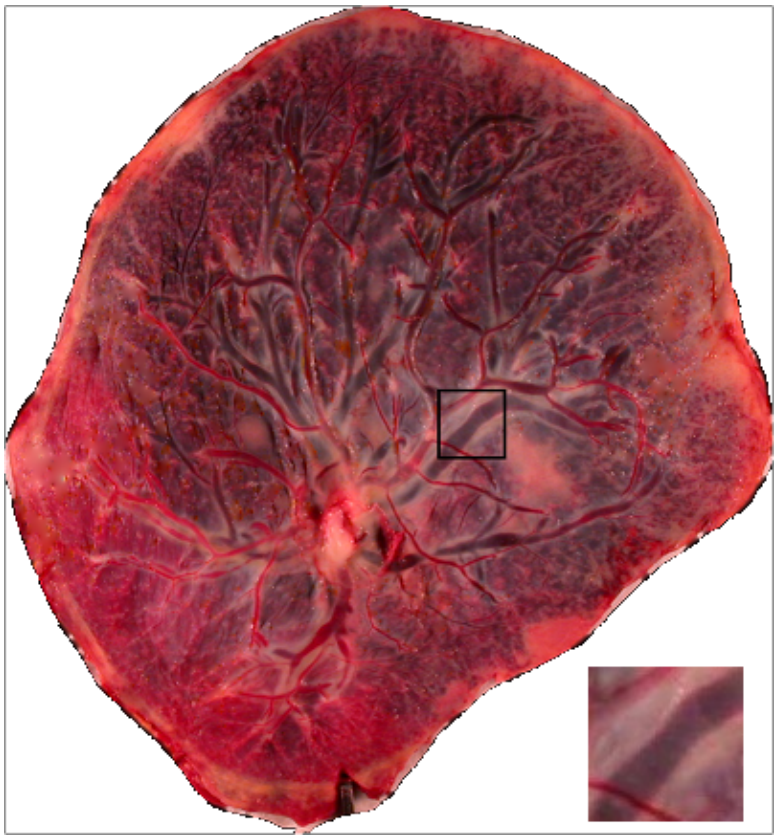

FIGURE 2. Original placenta image that we took our test image from, as seen below and to the right.

2.2. **Intensity Map.** MATLAB takes the sample image and converts into as a 3-dimensional topographical map, also known as an intensity map, figure 3, using the "surf" function. Looking at the image this way was inspired by Meth and Chellappa [5]. As we compared the intensity map to the original image, we noticed a correlation between the valleys of the intensity map and the veins that can be seen with the naked eye in the original image. We decided to focus on extracting information about the points in the valleys. By extracting enough information from each point in the valley, we can then use that information to automatically connect points in each valley, reconstructing each vein.

2.3. **Identifying Edges.** In the intensity map, the edges of the veins are the walls of the valleys. We were able to use Canny Edge Detection (CED) to identify these walls because our image has "strong intensity contrasts, a jump in intensity from one pixel to the next [3]." Green gives three criteria why CED is the most effective technique to identify edges. Canny Edge Detection has a "low error rate" compared to other edge detectors, the

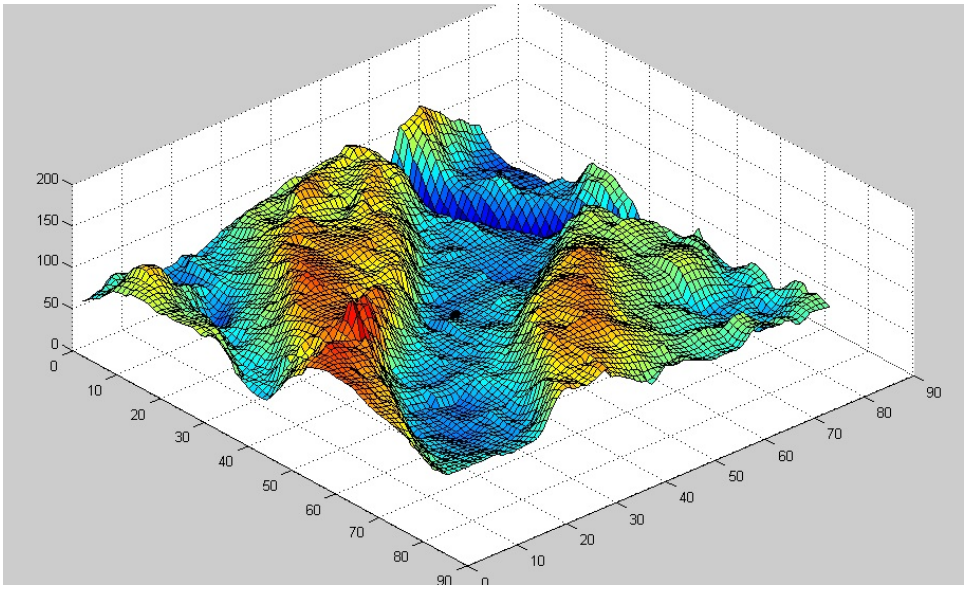

FIGURE 3. Test image intensity map from above point of view.

edges found are "well localized" to the locations of the actual edges, and the marked edges are only one pixel wide, rather than identifying a region as an edge. All of this means that we will get a clear set of edges to work with.

The Canny Edge Detector begins by smoothing the original image using the Gaussian normal distribution as a filter. Next, CED takes the gradient of the intensity map in the x and y directions. It uses the magnitude of the gradient to measure the slopes of the curves on the map. Pixels with gradients of large magnitude indicate a rapid change in intensity, which is an edge. Once it has built a list of the slopes from the intensity map, it chooses a set of thresholds unique to each image, designating the top thirty percent of the slopes as high, the next thirty percent as medium, and the remaining points as low, as categorized in figure 4. The low points are ignored because they are obviously not edges. There is then a set of decision criteria to sort through the high and medium points to identify them as edges or not. This is largely based on the gradients of the neighboring points, where medium points that are isolated from high points will not be identified as edges, but all high points and medium points connected to high points will be marked as edges. Figure 4 shows how the Canny Edge Detector might operate on a small sample of points.

2.4. **Build List of Strong Points Within Veins.** Now that we have found all the edges within the image, we need to identify what we call "strong points." Strong points are defined as points in the middle of a vein with "well-defined walls" and fit other criteria as described below. The program will filter out points that are not inside two walls or are too close

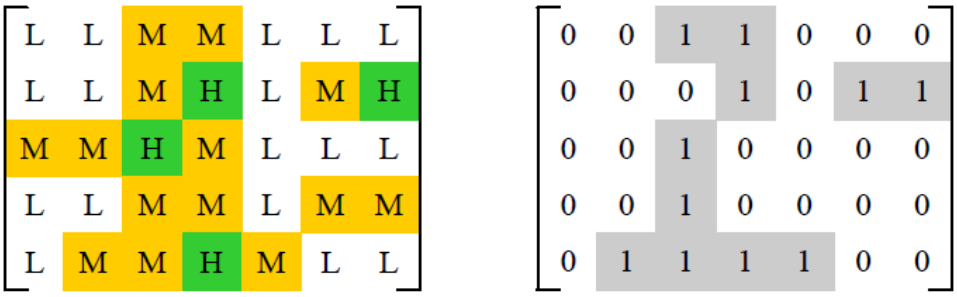

FIGURE 4. The matrix on the left represents what an intensity map might look like after the Canny Edge Detecter has designated all of the points as high (H), medium (M), or low (L). The matrix on the right represents the final output of the Canny Edge Detector on the left matrix. All of the edges become 1 and everything else is 0.

to the vein walls. We begin by testing every point that has not already been identified as a wall by CED.

We are interested in finding the distance between the point and the CED identified edges surrounding it. To find this distance as a function of angle we use Bresenham's Line Algorithm (BLA) [2, 6] to project rays out from the point in question to the nearest edge point at all angles, as seen in figure 5. We compile these distances and angles into figure 6. We apply the Gaussian normal distribution to this graph as a smoothing filter to help extract the local minima, which improves our accuracy by reducing the influence of noise on the sometimes jagged sides of the veins. The local minima indicate the distances to the nearest edges. If there are one or no local minima, we discard the point in question and move on to the next point. If there are two local minima, we use those distances as the distances to the nearest walls. If there are three local minima, x_1, x_2, and x_3, then we check the difference between x_1 and x_2. If that difference is smaller than a certain threshold, we discard x_2 and use x_1 and x_3 as the two nearest edges.

An example of this output, after smoothing with the Gaussian normal distribution filter, can be seen in figure 7, which shows us the distance from the selected point to its the nearest edges detected by CED as a function of angle. The minima of this plot tell us the distance to the nearest edge. In this example the walls are at approximately 1 and 4 radians, and about 12 and 8 pixels away, respectively.

In order to reduce the effect of irregularly shaped edges, we average the distance to the nearest edge with the distance to the edge pixels on either side along the same wall. If the difference in the magnitudes of these averages is less than or equal to 3 pixels, then the point is likely halfway between two walls: in the middle of a vein. If the difference is greater than 3 pixels, then the point is rejected.

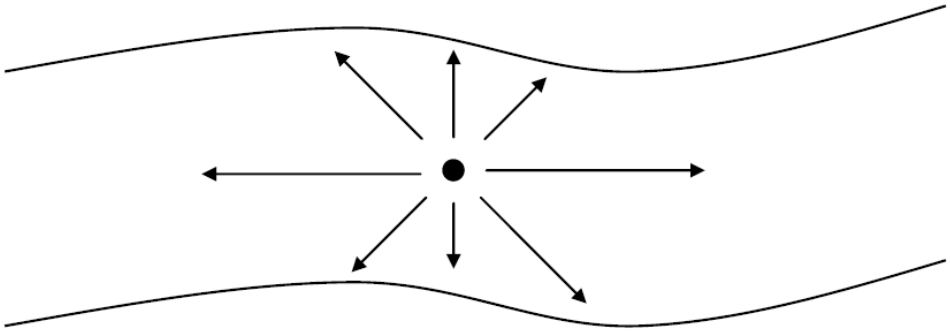

FIGURE 5. To find the distance of a point to its nearst edges, rays are sent out at every angle from that point. A sample graph to represent that can be seen in figure 6.

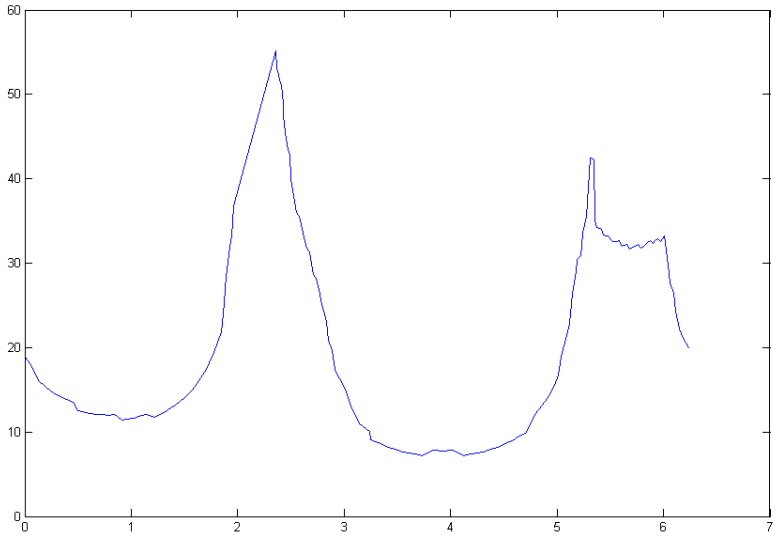

FIGURE 6. This figure shows the distance to the nearest edge from the test point at all angles. The x-axis is the angle in radians and the y-axis is the distance away from the test point in pixels.

Next, we test the quality of the walls surrounding the point. We start with the direction to the nearest wall, then add and subtract 45° to that direction. All of the points on that edge that fall within that 90° range are fit to a best fit line. We then find the standard deviation of the points from the best fit line. This procedure is repeated 180° away, on the other side of the

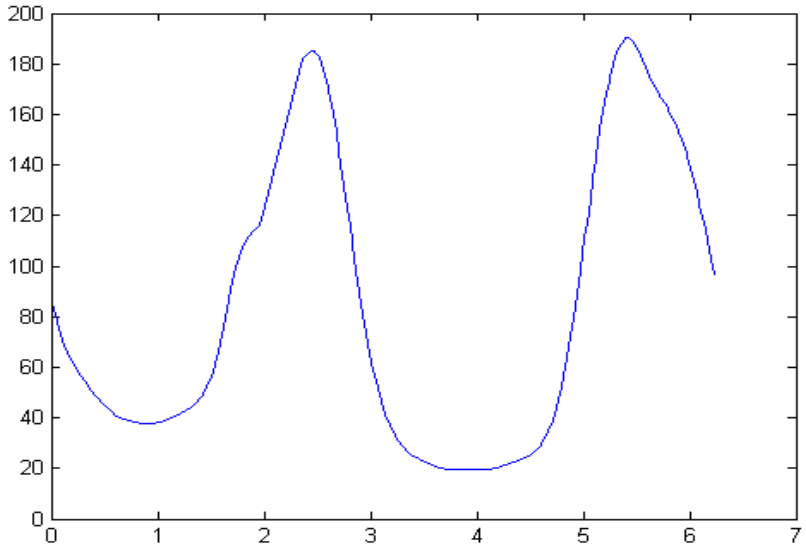

FIGURE 7. Gaussian Filter applied to figure 6. The x-axis is the angle in radians and the y-axis is the distance away from the test point in pixels.

vein, to make a best fit line on the other vein wall, shown in figure 8. If the sum of the standard deviations is small, less than or equal to 0.8, then the angle and width we measured will be accurate and we will consider the point to be a strong point. If the sum of the standard deviations is larger than 0.8, measurements made on this point may be unreliable, so we will discard the point. This has the effect of filtering out points surrounded by irregularly shaped sides. Irregular sides are often an indicator of over-identified edges that may not be vein walls.

2.5. **Characterize Strong Points.** For every strong point we identify, we then determine the characteristics of that point: position, vein width, vein direction angle, and depth. The position is the (i, j) coordinate of the pixel. The width and angle are found from figure 7. We determine the width of the vein to be the sum of the distances to the two nearest edges, as demonstrated in figure 9, 20 pixels wide according to the example illustrated in figure 7. The angle, θ, is calculated to be the average of the angles to the two nearest side walls, about 2.5 radians in figure 7. The angle is determined this way should always point down the length of the vein. To find the depth of each strong point, we look at the intensity of that point along with its eight nearest neighbors. We first compare the intensity of each neighbor to the strong point. If the difference in intensity is smaller than some threshold, then we count that point, if not it is discarded. By averaging the intensity

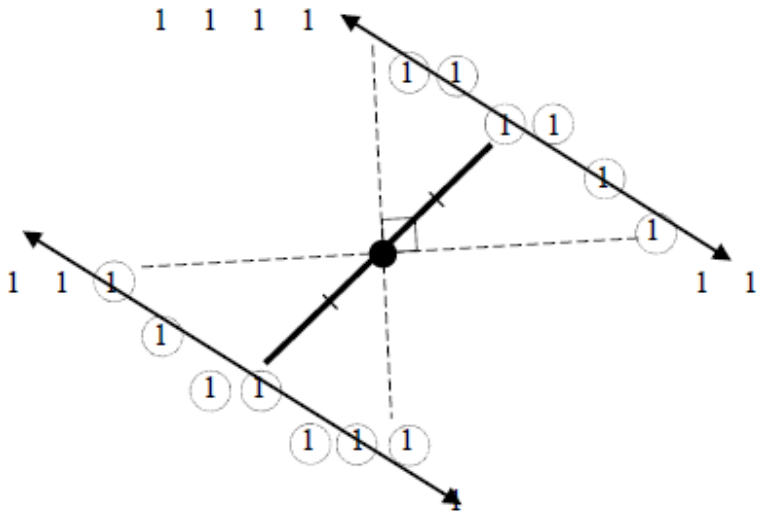

FIGURE 8. Example of checking standard deviation for a strong point. Edges on either side of test point are fit to a line of best fit, then the standard deviation between the edge points and that best fit is found.

of all of the good points around the strong point, as well as the strong point itself, we are able to find a value for the "depth" of the vein at that point.

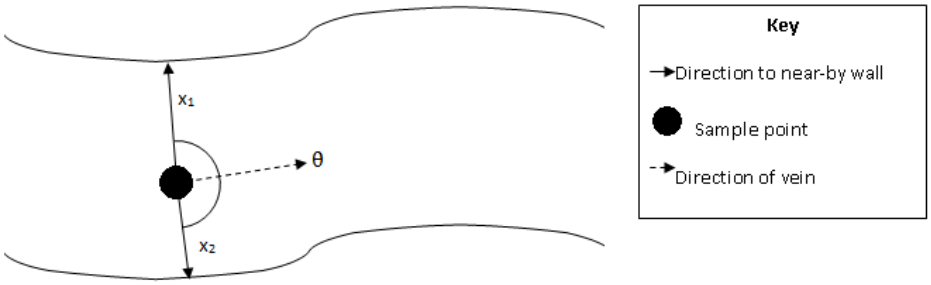

FIGURE 9. Schematic demonstrating the width and angle of each test point. x_1 and x_2 are the distances to the two nearest edges. The differences in the magnitudes of these distances will tell us how close the test point is to the center of a vein. θ is the average of the angles of x_1 and x_2.

3. Results and Discussion

3.1. **Intensity Map.** When we first examined the intensity map and compared it to the original image, we had to choose which color channel would

best help us outline the veins. The image itself is a combination of reds, therefore the red color channel did not bring out enough detail. The blue color channel was very shallow and when the Canny Edge Detection algorithm was run, it had over-identified edges. The green color channel gave a good balance of contrast, so we chose to continue working with green. There was some discussion of how much the occasional glare spots in the image may affect our results. We used cropped, de-glared images, but as you can see in figure 10, the glare in our image created a peak that may alter the results of the CED's thresholds. After running CED it seemed that the initial filtering on the image took care of most of those issues.

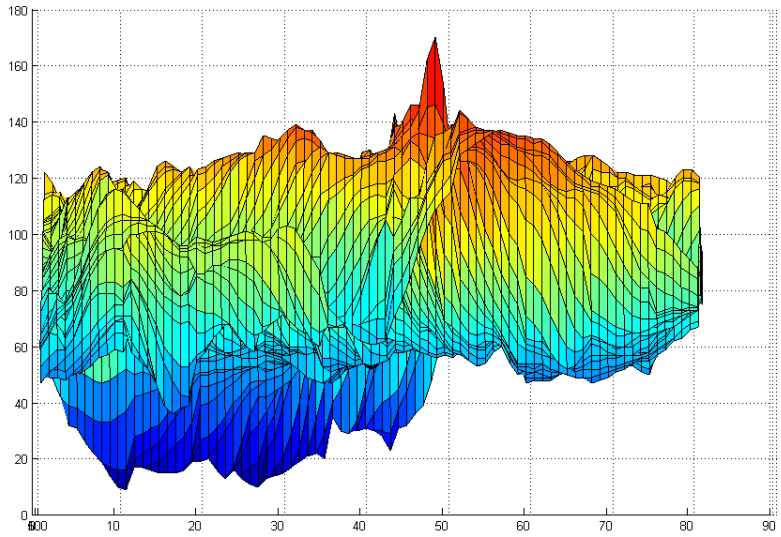

FIGURE 10. A side view from the lower end of the intensity map of the test image from figure 2 (lower right corner). Note the high peak associated with a glare spot on the orginal image.

3.2. **Identifying Edges.** Figure 11 is the final output of the CED overlaid on the original intensity map. The Canny Edge Detection did have a low error rate, but it still detected too many edges around one of the glare spots in the image at about (50,15) in figure 11.

3.3. **Build List of Strong Points Within Veins.** Figure 12 shows our output of strong points as determined by this subroutine. The grey stars are the edges determined by the Canny Edge Detector and the black circles are the points identified as strong points. We had several issues when initially debugging this program and then polishing it in order to try to complete

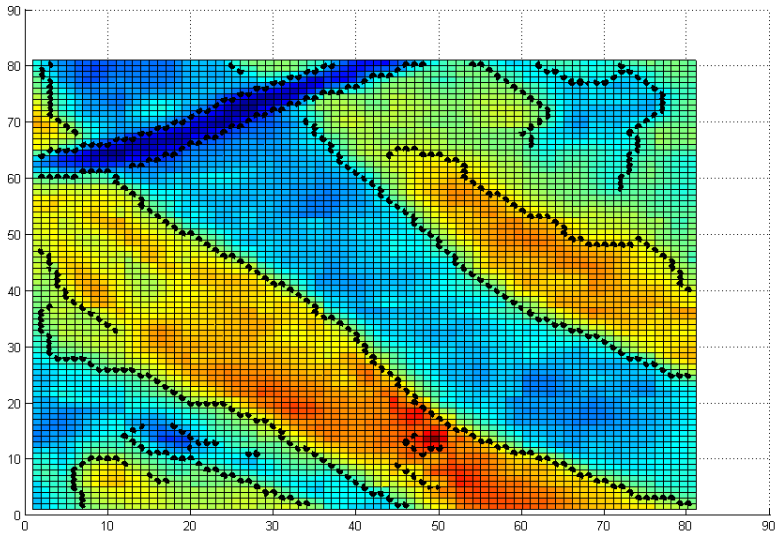

FIGURE 11. Intensity map of veins with results of Canny edge detector overlaid as dark spots

as much of the project as possible within our time constraint. Initially, the program only detected a few strong points along the thin vein at the top left hand side of the image. Part of our polishing was to adjust thresholds accordingly to try to get the result we were looking for.

Figure 13 highlights five regions of our image where our code had issues. Region A shows an example of anomalous disconnects that needs to be explored and resolved. It is possible that the bend in the vein caused the standard deviation of the walls around the middle points to be too high, so we are getting this random output. Region B came about where two edges intersect. In this case, the cornering occurs when two veins cross each other. Ideally, region B should be a line. To establish the line, depth must be taken into account, not just the edges as the current algorithm does. Region C should not have much marking because it is not part of any vein. To remove region C, the algorithm needs to take into account concavity. Veins are concave up whereas region C is concave down. By eliminating regions with downward concavity, regions such as C will not be marked. Region D arose when the same edge is identified as the sections of angles x_1, x_2 when we are checking for the standard deviations of the walls surrounding the point. No solution has been implemented to resolve this error. Region E occurred when the edge walls are not well-formed. One option is to remove strong points with few or no neighbors, which would remove most of region E.

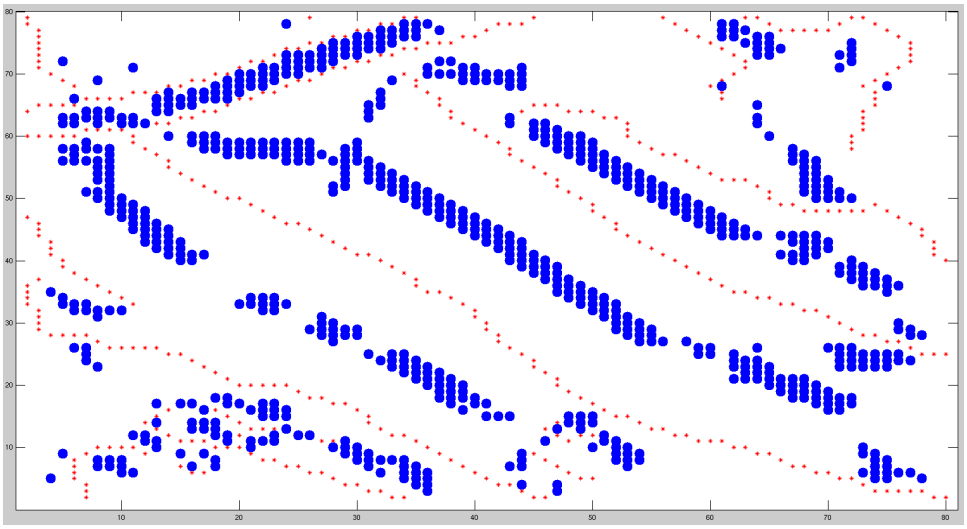

FIGURE 12. Image of Canny Edge Detection results as small light stars and the identifed strong points as darker circles.

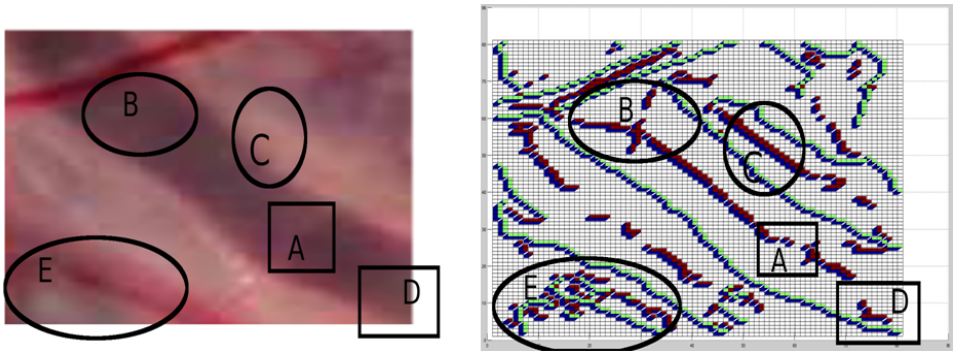

FIGURE 13. These figures highlight some of the issues we noticed from our final output to the right and the corresponding regions in the original image on the left. Region A: unexpectedly sparse strong points. Region B: cornering issues. Region C: is not part of a vein. Region D: same wall identified twice. Region E: poorly formed edges.

We decided to reduce the number of strong points to a more manageable level by using a function in MATLAB known as "bwmorph" which thinned the strong points wherever they were clustered. The result of this process can be seen in figure 14.

3.4. **Characterize Strong Points.** We successfully characterized the strong points. All of the strong points were put into a list along with the following

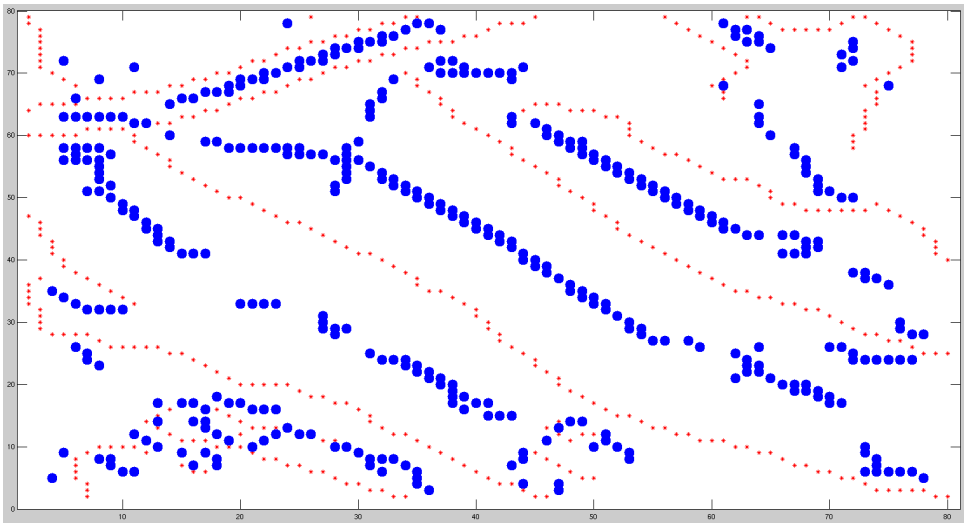

FIGURE 14. Final output after thinning the strong points.

characteristics: position, width, angle, and depth. We did not have any issues compiling this list of characteristics.

4. Conclusion

Our group chose to develop a program that would automatically pick out and draw the vein structure of an image of a placenta. We did not reach this goal, even just with our test image, because we simply ran out of time to complete the work before the end of the semester. We did get a positive initial set of strong points that we would like to further narrow down to only the strong points we want to use. Once establishing a complete list of strong points, we believe we would be capable of finishing the work to connect the strong points to recreate the veins. Based on what we have accomplished so far we believe this method shows promise to reaching a more connected vein network than others [1].

5. Future Work

The last part of the project, which we were unable to get to, is to connect the strong points together to generate the vein network. This section will summarize our intended path to completing the task as well as other means to improve the overall method.

After identifying the strong points, we began working on an algorithm that would connect them. This algorithm analyzes one strong point at a time, using the characteristic angle of the strong point, it would travel along the direction of that angle until it found another strong point or an edge. The algorithm we decided to use to travel down the angular direction from the point is Bresenham's Line Algorithm. We chose this algorithm because

of its simplicity in determining a line of pixels given a direction and starting point.

The algorithm would travel from one strong point to another. If the algorithm intercepts an edge before finding a strong point, then we will simply move on to the next strong point. To make sure to cover our tracks, we want to mark the places we will travel by placing dashed lines on the edge graph. This way, we can see the point we started from, the path we took, if we hit any other strong points, and what happened if we hit these points. We had high hopes for this technique, allowing us to perform precise error checking on our code, but we ran into some difficulties when coding the algorithm.

One trouble was incorporating BLA into our code. BLA requires some inputs which interfere with our inputs. Fitting BLA into our iterative code interfered with some of the variables we had created. Also, we are having trouble determining how to iterate the procedure over many strong points. We figured out how to check one point, but have not yet expanded the program to check all strong points.

Once we identify additional strong points along the angular direction of the strong point in question, we need to determine if there is a correlation between these points and the original one. If so, we would connect those points as being part of the same vein and eventually connect all points in the same vein.

Our test image did not include any veins with forks in them. Future effort would be required to adjust our program to find the vein structure when there are obstacles such as splitting veins.

We would like to increase the efficiency of our program. At the moment we search through every point to find the distance to the nearest edge as a function of angle. Our algorithm spends about 95% of its time on this part of the process, making it impractical for full sized images. Afterwards, the two side walls must be identified by finding local minima and choosing the correct minima. This step is difficult to do accurately, as seen in figure 7 where the minima are broad, making it hard to pinpoint a specific direction. It is also difficult when more than two minima are identified, as discussed above. We would ideally like to develop an alternate algorithm to identify the strong points without the rays and local minima.

One alternative method we wish to attempt to implement we call the "Echo Method." In this method, we first select a section of the edge of a vein, labeled $\overline{AB}$ in figure 15. This section is projected in the direction of most negative gradient which should be directly across the vein. When this projection reaches the far wall of the vein, we will mark the new section of vein $\overline{CD}$. $\overline{CD}$ is then projected back across the vein according to its gradient direction, marking its projection on the original wall as $\overline{EF}$. On the second wall, where $\overline{CD}$ is located, we label G as the point that projects onto A on the original wall. We will then find the midline between $\overline{AE}$ and $\overline{GD}$, which we will label $\overline{UV}$. $\overline{UV}$ should be a line down the center of the

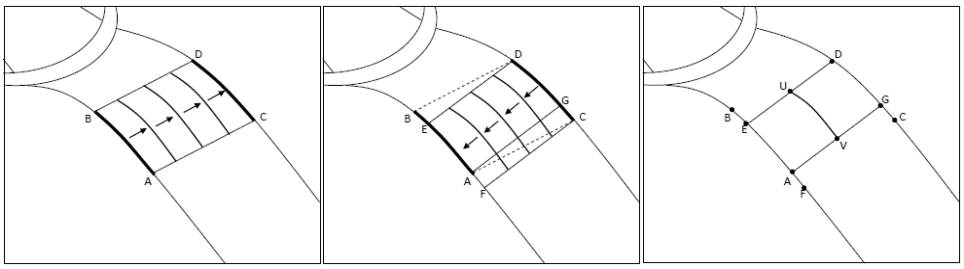

FIGURE 15. Left: Points A and B are selected on an edge. The line segment $\overline{AB}$ is projected to the other side of the vein in the direction of largest negative gradient, which determines segment $\overline{CD}$. Center: The projection procedure is repeated on segment $\overline{CD}$ to generate points E and F. Point G is also generated as the point on $\overline{CD}$ that projected back onto point A. Right: The midpoints of segments $\overline{ED}$ and $\overline{AG}$ are used to find segment $\overline{UV}$, which should lie down the center of th vein.

vein. This method would allow us to run tests on the points identified as edges by Canny Edge Detection, rather than all 700,000+ pixels in each full sized image.

We have successfully identified small veins, as shown in figure 14, which was one of the more important features missing from previous research [1]. This means our technique may be successful in identifying the smaller structure that others have missed.

6. Acknowledgements

We would like to thank Professor Chang, Doctor Carolyn Salafia, and Matthew Aggleton, Ph.D. for their invaluable contributions. We also appreciate the opportunity to borrow code from various others [4, 7].

References

[1] N Almoussa, B Dutra, B Lampe, P Getreuer, T Wittman, C Salafia, and L Vese. Automated vasculature extraction from placenta images. *UCLA Summer 2009 REU Report*, 2009.

[2] Jack E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4:25–30, 1965.

[3] Bill Green. Canny edge detection tutorial. `http://www.pages.drexel.edu/~weg22/can_tut.html`, 2002.

[4] MathWorks. R2010b mathworks documentation. http://www.mathworks.com/help/, Feb 2011.

[5] R. Meth and R. Chellappa. Stability and sensitivity of topographic features for synthetic aperture radar target characterization. *J. Opt. Soc. Am. A*, 16:396–413, 1999.

[6] (Various). Bresenham's line algorithm. `http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm`, 2011.

[7] Zhiqiang Zhang. List, queue, stack. Matlab Central, Oct 2010.

Appendix A. MATLAB Code

A.1. **clusterLine.m.**

```
function [T,cluster] = clusterLine(E,sigma,stdThresh)

% The Function: clusterLine examines the edge map at each point.
  %If the point is close enough to the closest walls, has enough
  %points to build some walls modeled after the actual walls, and
  %has a small standard deviation, this point will be marked as 1
  %in cluster. Otherwise, it will be disregarded.
% Input: E         - The edge map of the topographical map of the
  %image
%        sigma     - the standard deviation of the gaussian
  %smoothing
%        stdThresh - the threshold of the sum of the standard
  %devation allowed.
% Output: clustered points(C) and a line (T) formed from the
  %cluster points

[m,n] = size(E);

cluster = zeros(m,n);

%x's are the angles and r's are the distances
for i=2:m-1
 %For debugging:
 %i

 for j=2:n-1

   %Only taking into account points that are not edges
   if ~E(i,j) == 1

     % Getting all the x's and r's from point (i,j)
     [detailarrPlot,arrPlot] = MM_GetRayEnds(double(E),i,j);

     % Making sure the matrix has enough sample of points
     [length,l] = size(arrPlot);
     if length < 4
         break;
     end

%Doing the smoothing of arrPlot and getting the local minimum x's
     S = MM_Smooth( arrPlot,sigma );
     [x1,x2,x3,found] = MM_GetMinWDeriv( S, sigma );
```

```
    if found
     %Getting the r's corresponding to the x's previously found
      z = arrPlot(:,1) == x1;
       r1 = arrPlot(z,2);
       z = arrPlot(:,1) == x2;
       r2 = arrPlot(z,2);

 % Switching between x2 and x3 if x1 is too close to x2.
 % This is used to improve the likelyhood that x1 and x2
 % represent different edge walls.
    if ~(x3 == x2)
        if (abs(x2-x1) < min([pi/3*(1/r1+1) pi/3*(1/r2+1)]))
            x2 = x3;
            z = arrPlot(:,1) == x3;
            r2 = arrPlot(z,2);
        end
    end

    %Average distance from its neighbors
    d1 = MM_AvgDist(arrPlot,x1,pi/3*(1/r1+1));
    d2 = MM_AvgDist(arrPlot,x2,pi/3*(1/r2+1));

%Check if they are good points:

 %Check if the point is in the middle of two edge
 %walls and if the edge walls have enough point to sample.
    if (abs(d1-d2) <= 3 ...
     && MM_CheckIfEnoughPoints(detailarrPlot,x1,pi/3*(1/d1+1)) ...
     && MM_CheckIfEnoughPoints(detailarrPlot,x2,pi/3*(1/d2+1)))

  %Take a sample of x1's wall section.
  aPlot = MM_GetHorizLine(arrPlot,x1,pi/3*(1/d1+1));
  [length,l] = size(aPlot);
  % Check if there is emough points in the sample
  if length >= 3

   %Calculate the standard deviations from the built
   %wall to the true wall
   [std1,l,l] = MM_stdofLine(aPlot);

   %Take a sample of x2's wall section.
   aPlot = MM_GetHorizLine(arrPlot,x2,pi/3*(1/d2+1));
   [length,l] = size(aPlot);
   % Check if there is enough points in the sample
```

```
      if length >= 3
      %Calculate the standard deviations from the built
       %wall to the true wall
       [std2,1,1] = MM_stdofLine(aPlot);

        %If they are good points, meaning they have small standard
        %deviations, then mark the location with the value of  1.
         if (std1 + std2) <= stdThresh
         cluster(i,j)=1;
       end
      end
     end
    end
   end
  end
 end
end

%Getting the thin line from the cluster points
T = bwmorph(cluster,'thin',1);

end
```

A.2. **MM_GetRayEnds.m.**

```
function [ detailarrPlot,arrPlot ] = MM_GetRayEnds( E, x0, y0 )

% The Function: MM_GetRayEnds takes in a matrix E and shoots out rays
  %from (x0,y0) in different directions (using Bresenham's line algorithm)
  %recording when it hits an element of matrix E of value 1, which we will
  %call a "wall". Inputs: The edge map(E), coordinates of a point(x0,y0)
% Outputs: arrPlot is a n by 2 matrix containing in
  %the first column the angles where a "wall" was hit and the
  %corresponding distances in the second column.
  %detailarrPlot is the matrix arrPlot with two extra columns containing
  %the coordinates of the "wall points". The x coordinate is placed its
  %third column and the corresponding y coordinate in its last column.

[m,n] = size(E);

list = CList();

% Set direction. (x,y).
dir = [ 1 0 ];

% EN
```

```
while dir(2)/dir(1) < 1
 for x = x0 + 1 : m
   % Bias towards larger value.
   y = round(dir(2)*(x-x0)/dir(1))+y0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x-1,y) == 1 && E(x,y-1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
               norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(x-x0) 2*(y-y0)+1 ];
end

% Switch representation. (y,x) instead.
dir = [ dir(2) dir(1) ];

% NE
while dir(2)/dir(1) > 0
 for y = y0 + 1 : n
   % Bias towards smaller value.
   x = fix(dir(2)*(y-y0)/dir(1))+x0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x-1,y) == 1 && E(x,y-1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
              norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
```

```
 end
    % Update direction.
    dir = [ 2*(y-y0)+1 2*(x-x0) ];
end

% Moves to next direction to prevent repeating same point
if (x == x0)
    dir = [ 2*(y-y0) 2*(x-x0)-1 ];
end

% NW
while dir(2)/dir(1) > -1
 for y = y0 + 1 : n
   % Bias towards larger value.
   x = round(dir(2)*(y-y0)/dir(1))+x0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x+1,y) == 1 && E(x,y-1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
              norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(y-y0) 2*(x-x0)-1 ];
end

% Switch representation. (x,y) instead.
dir = [ dir(2) dir(1) ];

% WN
while dir(2)/dir(1) < 0
 for x = x0 - 1 : -1 : 1
   % Bias towards smaller value.
   y = fix(dir(2)/dir(1)*(x-x0))+y0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
```

```
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x+1,y) == 1 && E(x,y-1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
               norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(x-x0)-1 2*(y-y0) ];
end

% Moves to next direction to prevent repeating same point
if (y == y0)
    dir = [ 2*(x-x0) 2*(y-y0)-1 ];
end

% WS
while dir(2)/dir(1) < 1
 for x = x0 - 1 : -1 : 1
   % Bias towards larger value.
   y = round(dir(2)*(x-x0)/dir(1))+y0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x+1,y) == 1 && E(x,y+1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
              norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(x-x0) 2*(y-y0)-1 ];
end
```

```
% Switch representation. (y,x) instead.
dir = [ dir(2) dir(1) ];

% SW
while dir(2)/dir(1) > 0
 for y = y0 - 1 : -1 : 1
   % Bias towards smaller value.
   x = fix(dir(2)*(y-y0)/dir(1))+x0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if found wedged between to marked pixels.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x+1,y) == 1 && E(x,y+1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
               norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(y-y0)-1 2*(x-x0) ];
end

% Moves to next direction to prevent repeating same point
if (x == x0)
    dir = [ 2*(y-y0) 2*(x-x0)+1 ];
end

% SE
while dir(2)/dir(1) > -1
 for y = y0 - 1 : -1 : 1
   % Bias towards larger value.
   x = round(dir(2)*(y-y0)/dir(1))+x0;
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if out of bounds.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x-1,y) == 1 && E(x,y+1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
               norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
```

```
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(y-y0) 2*(x-x0)+1 ];
end

% Switch representation. (x,y) instead.
dir = [ dir(2) dir(1) ];

% ES
while dir(2)/dir(1) < 0
 for x = x0 + 1 : m
   % Bias towards smaller value.
   y = fix(dir(2)*(x-x0)/dir(1))+y0;
   %if out of bounds.
   if ~(1 <= x && x <= m && 1 <= y && y <= n)
     list.pushtorear([GetPolarAngle(x-x0,y-y0) Inf 0 0 ]);
     break;
   %if out of bounds.
   elseif ~(sign(dir(2)/dir(1)) == 0) && E(x-1,y) == 1 && E(x,y+1) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0)...
               norm([x-x0 y-y0])-1/sqrt(2) x y ]);
     break;
   % if found at location.
   elseif E(x,y) == 1
     list.pushtorear([GetPolarAngle(x-x0,y-y0) norm([x-x0 y-y0]) x y]);
     break;
   end
 end
    % Update direction.
    dir = [ 2*(x-x0)+1 2*(y-y0) ];
end


elem = list.front();
temp = list.back();
% Remove the last element if it's the same as the first element.
if temp{1}(1:2) == elem(1:2)
    list.poprear();
end
```

```
len = list.size();
detailarrPlot = zeros(len,4);

% Copy the list into detailarrPlot
for i = 1 : len
    elem = list.front();
    detailarrPlot(i,:) = elem;
    list.popfront();
end

% Copy only in-bound angle and distance from detailarrPlot to arrPlot
idx = ~(detailarrPlot(:,2) == Inf);
arrPlot = detailarrPlot(idx,1:2);

end
```

A.3. **GetPolarAngle.m.**

```
function [ angle ] = GetPolarAngle( x,y )

% The Function: GetPolarAngle obtains the angle formed by the x axis
  %and the vector [x,y].
% Inputs: The coordinates of the point(x,y) that is being analyzed
% Output: The "polar angle" as described above

if sign(x) == 1
    %If the point is in the first quadrant
    if sign(y) == 1
        angle = atan(y/x);

    %If the point is in the fourth quadrant
    elseif sign(y) == -1
        angle = atan(y/x)+2*pi;

    %If the point is in the poisitive x-axis
    else %that is, if y == 0
        angle = 0;
    end

elseif sign(x) == -1
    %If the point is in the second or third quadrant
    angle = atan(y/x) + pi;

else % that is if x == 0
    %If the point is in the positive y-axis
    if sign(y) == 1
```

```
        angle = pi/2;

    %If the point is in the negative y-axis
    elseif sign(y) == -1
        angle = 3*pi/2;

    %If the point is at the origin
    else %y == 0 ERROR
        angle = 0;
    end
end

end
```

A.4. **MM_Smooth.m.**

```
function [ smooth ] = MM_Smooth( arrPlot,s )

% The Function: MM_Smooth uses the arrPlot, the n by 2 matrix
  %containing angles and distances, as a list of coordinate points
  %to make a "function." Using the gaussian filtering and the
  %sigma provided, it proceeds to use Gaussian Filtering to
  %smooth out the "function."
%Inputs: An n by 2 matrix containing angles and distances
  %(arrPlot), and sigma
% Outputs: A "smooth" arrPlot

[m,l]=size(arrPlot);

smooth = zeros(m,1);

step = floor(m/2);
new_m = 2*floor(m/2)+1;

% Center the point
A = [arrPlot(m+1-step:m,:); arrPlot(1:1+step,:)];
A(:,1) = [A(1:step,1) - 2*pi*ones(step,1) ; A(step+1 : new_m,1)]
                            - arrPlot(1,1)*ones(new_m,1);

% Create gaussian filtering
gau = exp(-(A(:,1).*A(:,1))/(2*s^2))/(2*pi*s^2);
gau = gau/(norm(gau));

% Apply Filter
smooth(1) = gau'*A(:,2);
```

```
for i = 2:m
    % Center the point
    A = [A(2:new_m,:) ; A(1,:)];
    A(m,1) = A(new_m, 1) + (2*pi);

    B = A(:,1)-arrPlot(i,1)*ones(new_m,1);

    % Create gaussian filtering
    gau = exp(-(B.*B)/(2*s^2))/(2*pi*s^2);
    gau = gau/(norm(gau));

    % Apply Filter
    smooth(i) = gau'*A(:,2);

    %Debugging: plot(1:new_m,A(:,1));
end

smooth = [arrPlot(:,1) smooth];

end
```

A.5. **MM_GetMinWDeriv.m.**

```
% It's best that X be smoothed.
function [ x1,x2,x3,found ] = MM_GetMinWDeriv( X, sigma )

% The Function: MM_GetMinWDeriv goes inside arrPlot and finds the
  %smallest distances. Note that all the distances are located in the
  %second column of arrPlot.
% Input: A matrix and a sigma
% Output: The three angles corresponding to the three smallest
  %distances from the point to the wall.

found = true;

% Debuggin elements:
% D = MM_GetGaussWeightedDerivative(X,sigma);
% D = [D(:,1) abs(D(:,2))];
% M = min([ [D(size(X,1),2);D(1:size(X,1)-1,2)]-D(:,2)...
                                [D(2:size(X,1),2);D(1,2)]-D(:,2) ],[],2);

% dX = x_(i+1) - x_i
dX = [X(2:size(X,1),1) ; X(1,1)+2*pi] - X(:,1);
% dY = f(x_(i+1)) - f(x_i)
dY = [X(2:size(X,1),2) ; X(1,2)] - X(:,2);
```

```
% Derivative
M = dY./dX;

% min(f(x_(i+1)) - f(x_i),f(x_(i-1)) - f(x_i))
N = min([ [X(size(X,1),2);X(1:size(X,1)-1,2)]-X(:,2)...
                              [X(2:size(X,1),2);X(1,2)]-X(:,2) ],[],2);

% Get index of f'(x_i)>=0 && f(x_(i-1)) <= f(x_i) <= f(x_(i+1)),
  %hence the index that are local min.
idx = find((M >= 0).*(N >= 0));

if isempty(idx)
    x1 = 0;
    x2 = 0;
    x3 = 0;
    found = false;
else
    % Get first local min
    [l,id] = min(N(idx));
    x1 = X(idx(id),1);

    % Mark that location to not pick the same local min.
    N(idx(id),1) = Inf;

    if size(idx,1) > 1
        % Get second local min
        [l,id] = min(N(idx));
        x2 = X(idx(id),1);
        % Mark that location to not pick the same local min.
        N(idx(id),1) = Inf;

        if size(idx,1) > 2
            % Get second local min
            [l,id] = min(N(idx));
            x3 = X(idx(id),1);
        else
            x3 = x2;
        end
    else
        % x2 is guessed to be pi radians away from x1. The result is
        % unreliable.
        x2 = x1 - pi;
        if x2 < 0
            x2 = x2 + 2*pi;
        end
```

```
        x3 = x2;

        found = false;
    end
end

end
```

A.6. MM_AvgDist.m.

```
function [ avg ] = MM_AvgDist( arrPlot, angle, intervalSize )

% The Function: MM_AvgDist finds the average distance from the point
  %to the section of the wall where an angles is pointing.
% Input: The n by 2 matrix containing angles and distances (arrPlot), the
  %direction angle, and an angle of partition (intervalSize)
% Output: The average distance.

%Getting a portion of arrPlot to work with
I = MM_GetInterval(arrPlot,angle,intervalSize);

[m,l] = size(I);

%Finding the average distance for that portion of arrPlot
if m == 0
    avg = -1;
else
    avg = sum(I(:,2))/m;
end

end
```

A.7. MM_CheckIfEnoughPoints.m.

```
function [ b ] = MM_CheckIfEnoughPoints( detailarrPlot, angle,...
                                  intervalSize )

% The Function: MM_CheckIfEnoughPoints checks if there are
  %enough points to work with.
% Input: detailarrPlot is the arrPlot with two extra columns
  %telling the end points of the direction the angle is going,
  %the direction angle from where we start, and angle of
  %partition (intervalSize)
% Output: Whether ('true') or not ('false') there are enough
  %points to construct the line
```

```
[m,l] = size(detailarrPlot);

idx = find(detailarrPlot(:,1) == angle);
idx = idx(1);
b = true;

stop = false;

% Get the points having less radians than angle but still in the interval
% of angle.
if (idx > 1)
    for i = idx - 1 : -1 : 1
        if (detailarrPlot(idx,1)-detailarrPlot(i,1) > intervalSize/2)
            stop = true;
            break;
        elseif detailarrPlot(i,2) == Inf
            b = false;
            return;
        end
    end
end

% Rap around
if ~stop
    for i = m : -1 : idx + 1
        if (detailarrPlot(idx,1)-detailarrPlot(i,1) + 2*pi > intervalSize/2)
            break;
        elseif detailarrPlot(i,2) == Inf
            b = false;
            return;
        end
    end
end

stop = false;

% Get the points having more radians than angle but still in the interval
% of angle.
if (idx < m)
    for i = idx + 1 : m
        if (detailarrPlot(i,1)-detailarrPlot(idx,1) > intervalSize/2)
            stop = true;
            break;
        elseif detailarrPlot(i,2) == Inf
            b = false;
```

```
            return;
        end
    end
end

% Rap around
if ~stop
    for i = 1 : idx - 1
        if (detailarrPlot(i,1)+2*pi-detailarrPlot(idx,1) > intervalSize/2)
            break;
        elseif detailarrPlot(i,2) == Inf
            b = false;
            return;
        end
    end
end

end
```

A.8. **MM_GetHorizLine.m.**

```
function [ aPlot ] = MM_GetHorizLine( arrPlot, angle,...
                                            intervalSize )

% The Function: MM_GetHorizLine grabs the arrPlot information
  %to build a wall. We pick an interval size from that point's
  %angle and highlight a part of the actual wall to compare.
% Input: The n by 2 matrix containing angles and distances
  %(arrPlot), the angle you are working with, and an an angle
  %of partition (intervalSize)
% Output: A horizontal line

% Get Interval centered around angle of size intervalSize
I = MM_GetInterval(arrPlot,angle,intervalSize);

[m,1] = size(I);

% Shift the angles so centered angle becomes 0.
I(:,1) = I(:,1)-angle*ones(m,1);

% Convert to Cartesian
aPlot = [I(:,2).*sin(I(:,1)) I(:,2).*cos(I(:,1))];

% Debugging:
% figure, plot(aPlot(:,1),aPlot(:,2));
```

```
end
```

A.9. **MM_GetInterval.m.**

```
function [ A ] = MM_GetInterval( M, angle, intervalSize )

% The Function: MM_GetInterval goes inside arrPlot, takes the
  %angle of interest, and  goes back a little more that
  %intervalSize/2 radians backward and foward. Then, it grabs
  %angles withing that interval from the arrPlot, puts the angle
  %of interest at the center, and subtracts its value from all the
  %angles (transposing).
% Input: The direction angle, and an angle of partition (intervalSize)
% Output: aPlot before centering transposing.

[m,l] = size(M);

[l,index] = min(abs(M(:,1) - angle*ones(m,1)));

% Get Left
i = index;
while (i > 1)
    if M(i-1,1) < angle - intervalSize / 2
        break;
    end
    i = i-1;
end

A = M(i:index,:);

if (i == 1)
    for i = m : -1 : index + 1
        if M(i-1,1) < angle + 2*pi - intervalSize / 2
            break;
        end
    end

    A = [ [M(i:m,1)-2*pi*ones(m-i+1,1) M(i:m,2)]; A];
end

% Get Right
i = index;
while (i < m)
    if M(i+1,1) > angle + intervalSize / 2
        break;
    end
```

```
    i = i + 1;
end

A = [A ; M(index+1:i,:)];

if (i == m)
    for i = 1 : 1 : index - 1
        if M(i+1,1) > angle - 2*pi + intervalSize / 2
            break;
        end
    end

    A = [A ; [M(1:i,1)+2*pi*ones(i,1) M(1:i,2)] ];
end

end
```

A.10. MM_stdofLine.m.

```
function [ std,m,b ] = MM_stdofLine( arrPlot )

% The Function: MM_stdofLine gives you a value (standard deviation)
  %of how good or bad a given line is compared to a "standard line"
  %made from arrPlot using polyfit.
% Input: The n by 2 matrix containing angles and distances (arrPlot)
% Output: The standard deviation, the first element in C, and the
  %second element in C. C is the best fit line from
  %MM_GetHorizontalLine.

[n,l] = size(arrPlot);

%Fitting a line through the points in arrPlot
C = polyfit(arrPlot(:,1),arrPlot(:,2),1);

%The first and last value of this standard line
m = C(1);
b = C(2);

%Getting the standard deviation
std = sqrt(sum(((m*arrPlot(:,1)+b)-arrPlot(:,2)).^2)/n);

end
```

Department of Mathematics and Statistics, California State University, Long Beach, 1250 Bellflower Blvd., Long Beach, CA 90840-1001

E-mail address: jmeyersg@gmail.com, nvhuynh16@gmail.com, zacschoenrock@hotmail.com, vazquez.marilyn.y@gmail.com