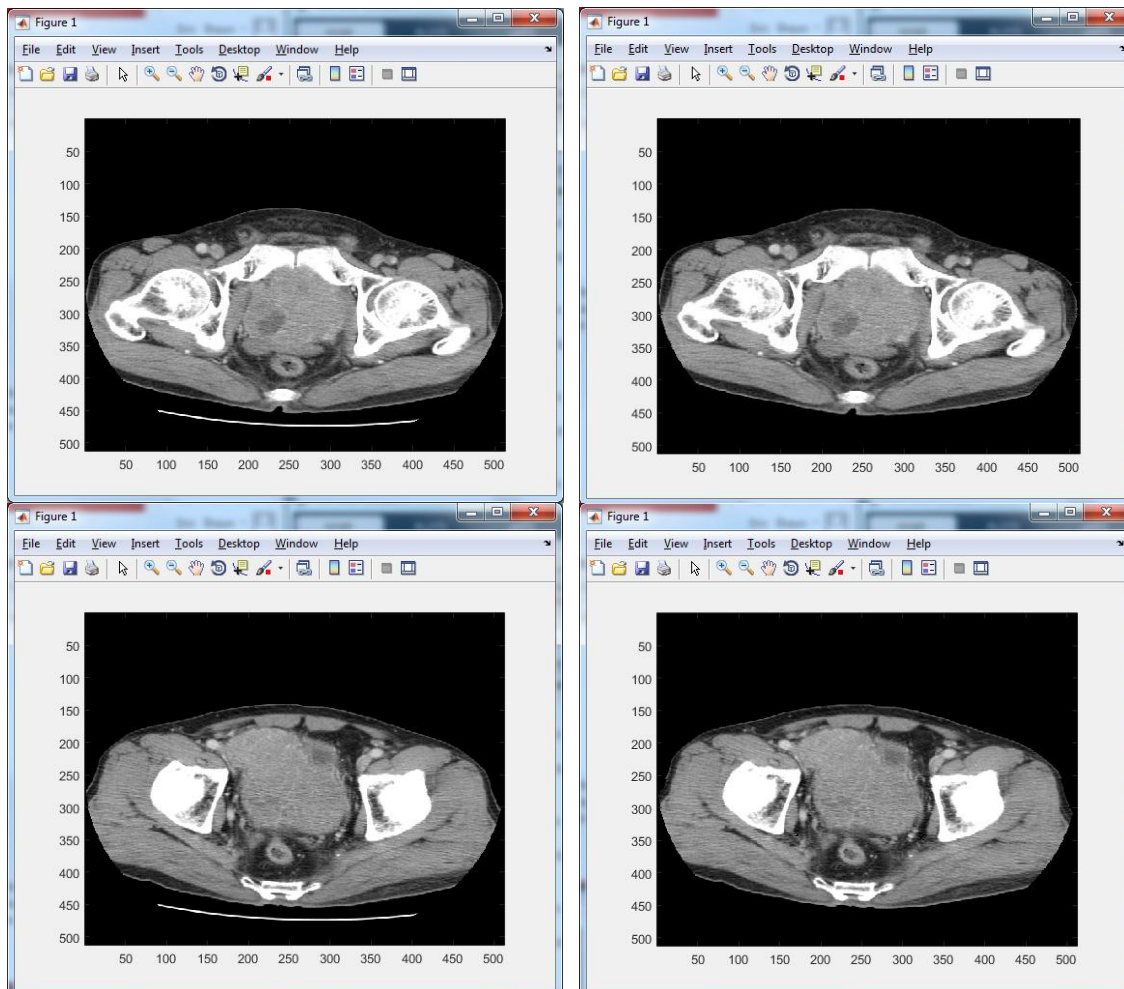


1. Artifact Removal

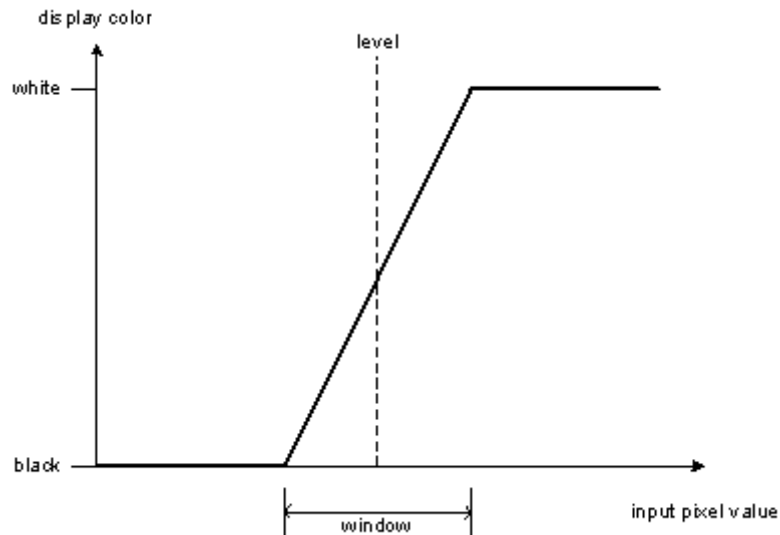
To remove the table from the prostate images, I checked all pixels across the image in the bottom half. First, I would check if the pixel was greater than 0 (not black), then I checked if the pixel 7 up from it was less than 7 (because of random pixels 0-5) to represent the black. If the pixel satisfied both these conditions, I set its third component to 0 (made it black). This takes an image (.jpg) as an input and outputs the matrix of the image with the table removed as well as saving ProstateNoTable.jpg to the workspace.

I limited the y component to only half to stop the algorithm from removing the lighter pixels on the top of the prostate images. However, this would limit the code from working if the image had somehow flipped and the table was above. Below are the original pictures (left) followed by the images after being called by RemoveTable.m



2. Window-Level Filtering

To create the window-level filter, I made two slides along the bottom of the figure that allow you to adjust the window and level values. The window value represents the width of the range of values allowed in the pixels (also known as contrast). The level value represents the middle point of the range of values (also known as brightness).

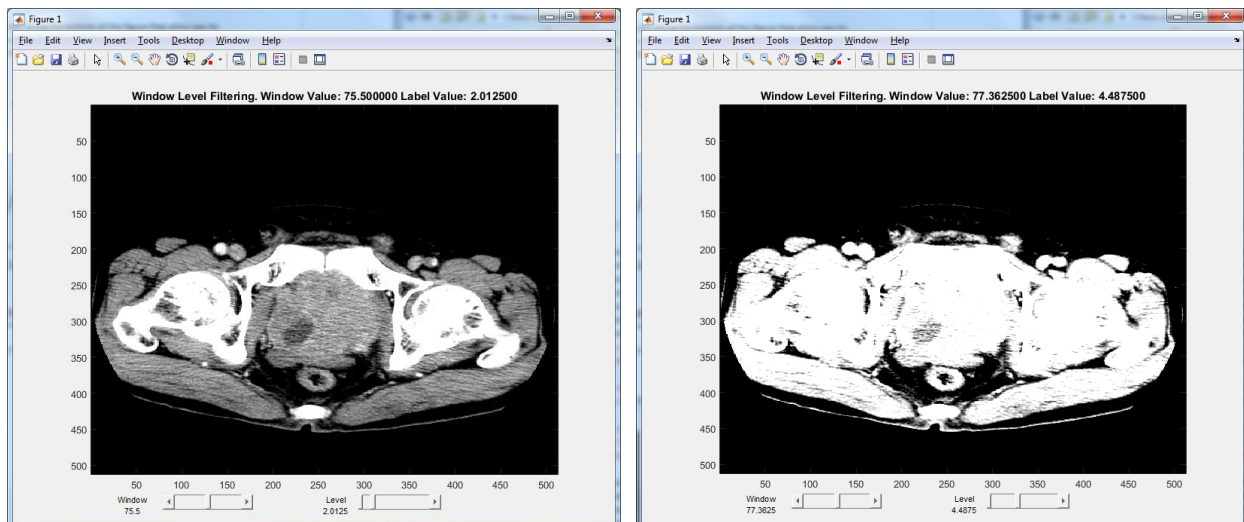


This figure shows the window and level values.

I created the sliders and their labels using uicontrol, which referenced a separate function for each value. These functions would change the image using the equation:

$$\text{newImage} = (\text{image} - \text{window}) * \text{level}$$

Then, output the new image.



3. Auto Segmentation

- i. For my first method, I used a radial search to segment the image. This function, RadialSegmentation.m, takes in an image as an input and will output the set of points along the contour as well as displaying a plot of the data.

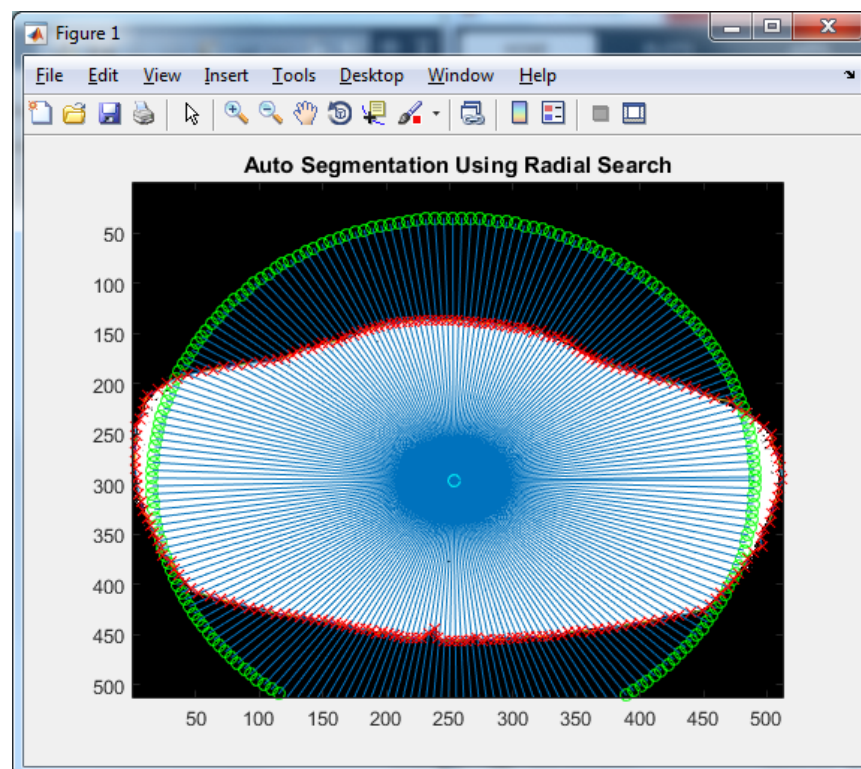
The first step was to determine the middle of the shape, to do this I contrasted the image to make the process easier, then I traversed from each of the four sides, in the middle of the side, till it hit a non-black pixel. I took the midpoint of both the sets of points to determine the center point of the organ.

Next, I created points along a circle centered at the midpoint. The amount of points used in the circle, and therefore the amount of contour points, can be controlled using the second value in the for angle statement. I used increment = 0.03: $2 * \pi / 0.03 = 209$ points. Using each of these points, I created a direction vector between the two the set my point as the equation of a line:

$$\text{Point} = \text{floor}(\text{center} + \text{length} * \text{direction vector})$$

Floored because we need real, integer values.

Using this equation of a line, I walked my point down the direction vector until it hit a zero pixel (black). Also checks for a large jump in points (catching a dead pixel) and removes.

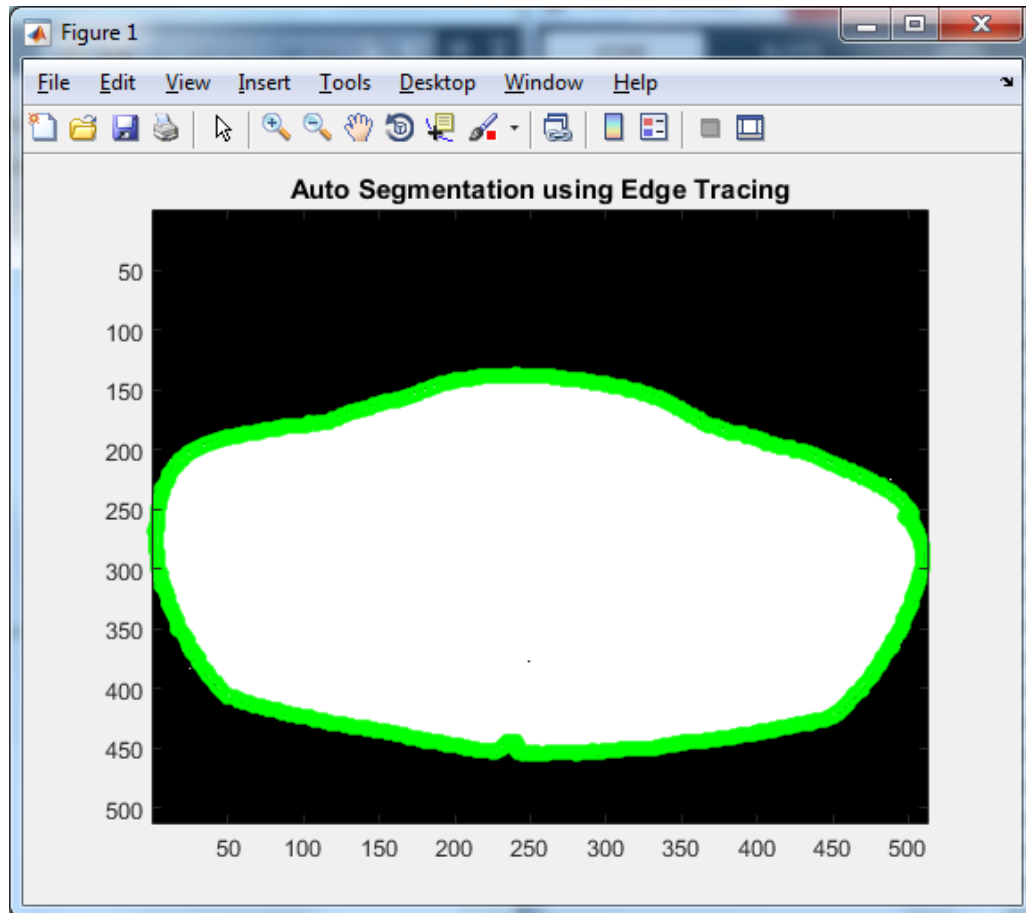


This is the radial segmentation of 'ProstateNoTable.jpg' from 'Prostate.jpg' using an increment amount about the circle of 0.03. Red crosses are the contour points.

- ii. For my second segmentation method I used edge tracing. For the first point, I found the left most point of the organ in the middle of the image. Then the function, EdgeSegmentation.m, checks each of the neighboring pixels and follows the outline of the image (which has been contrasted) in a clockwise motion.

A couple problems I found I ran into was if the point found itself isolated and could not move in any of my given directions, or if the point got stuck in a literal or coded circle where it would just go back and forth between pixels. I solved these by having an else statement to check if it couldn't move in any direction and another if statement that checked if the points were repeating. These would both remove the point that it was stuck on, clears the points list, and start again.

My function stops when the new point added is the same as the first point that was calculated from the left most. A limitation could arise if part of the image is separated, this function will only calculate neighbors. I used $>$ or $<$ 150 to distinguish white/black to help ignore grey pixels. Also, in some directions I checked over/up/down one OR two to help the case of dead pixels interfering. Too many dead pixels or large loops could break the algorithm

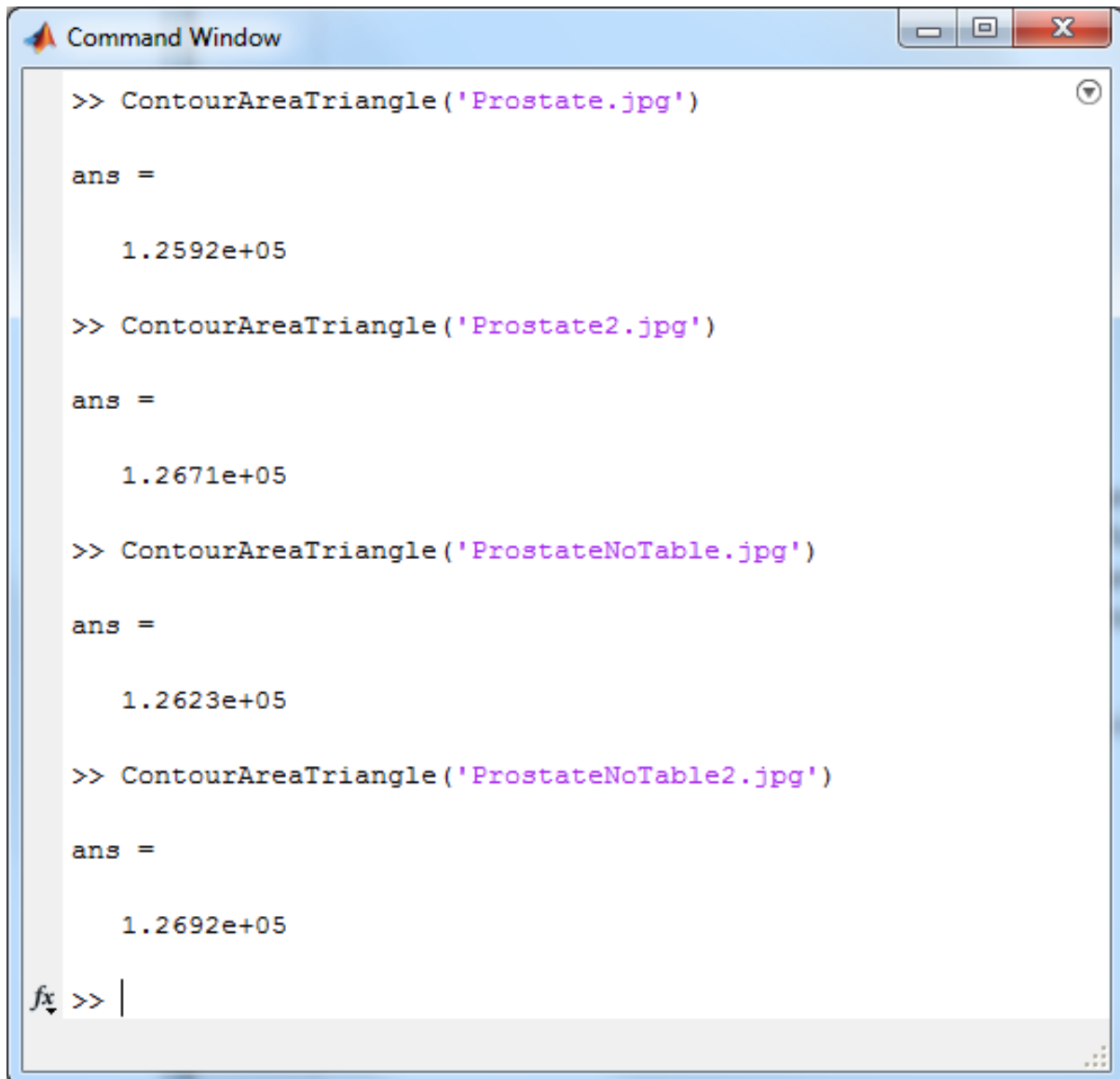


This is the segmentation using edge tracing, where the green circles represent the contour points

4. Contour Area

- i. The first method I used to find the area of the contour was finding the area of each of the triangles made by my radial search function (ContourAreaTriangle.m). I called my function to generate the points, found the center of the image, then used the shoelace formula for a triangle to find all the areas of the triangles. More is explained in the header.

$$A_{\text{tri.}} = \frac{1}{2} |x_1y_2 + x_2y_3 + x_3y_1 - x_2y_1 - x_3y_2 - x_1y_3|$$



```
Command Window

>> ContourAreaTriangle('Prostate.jpg')

ans =

    1.2592e+05

>> ContourAreaTriangle('Prostate2.jpg')

ans =

    1.2671e+05

>> ContourAreaTriangle('ProstateNoTable.jpg')

ans =

    1.2623e+05

>> ContourAreaTriangle('ProstateNoTable2.jpg')


ans =

    1.2692e+05

fx >> |
```


5. Hausdorff Distance

My function, HausdorffDistance.m, calculates the scalar value of the hausdorff distance between two .jpg images. The function takes in two images, and calculates their respective contour points using the RadialSegmentation.m code. Then, the code will calculate the distance between each point in the first image to every point in the second, square these values, then take the square root of each. The function will then take the minimum distance for every point of image 1 matched to a point from image 2. Next, we will do the same process but for calculating the distances from points in image 2 to image 1. Then, the algorithm takes the maximum values from both the vectors of distances. The Hausdorff Distance is then the maximum of the two maximum values. This is the furthest distance of any of the points to one another and is a measure of how similar two contour sets are.



```
>> HausdorffDistance('Prostate.jpg','Prostate.jpg')

ans =

    0

>> HausdorffDistance('Prostate.jpg','Prostate2.jpg')

ans =

   13.4164

>> HausdorffDistance('Prostate.jpg','ProstateNoTable.jpg')

ans =

    5.0990

>> HausdorffDistance('Prostate.jpg','ProstateNoTable2.jpg')

ans =

    13

>> HausdorffDistance('Prostate2.jpg','ProstateNoTable2.jpg')

ans =

   12.8062

fx >> |
```

6. Dice Similarity

My function, DiceSimilarity.m, computes the dice similarity coefficient between two images. The function will read in two images into their respective matrices to represent them using imread. It will also brighten the pixels so we can look at the whole shape of the image not the differences in shades (evaluate black vs white instead of black vs grey vs white).

We will use the dice similarity coefficient to calculate:

$$\text{Dice} = 2 * \text{AnB} / (\text{A}) + (\text{B})$$

Where the numerator is twice the intersection of the two images and the denominator is the sum of the size of the two images.

The closer the dice coefficient is to one, the more similar the two images are. If it is one, it means one image falls exactly on top of the other. If it is zero, that means there is no intersection between the two images. The two images must be the same dimensions.

More about the code is explained in the header and comments.

A screenshot of a MATLAB Command Window. The window has a title bar with the text "Command Window" and standard minimize, maximize, and close buttons. The command history shows five calls to the DiceSimilarity function. Each call is followed by the output "ans =" and a numerical value. The values are 1, 0.9764, 0.9823, 0.9706, and 0.9821. At the bottom of the window, there is a prompt "fx >>" with a cursor.

```
>> DiceSimilarity('Prostate.jpg','Prostate.jpg')

ans =

    1

>> DiceSimilarity('Prostate.jpg', 'Prostate2.jpg')

ans =

    0.9764

>> DiceSimilarity('Prostate.jpg', 'ProstateNoTable.jpg')

ans =

    0.9823

>> DiceSimilarity('Prostate.jpg', 'ProstateNoTable2.jpg')

ans =

    0.9706

>> DiceSimilarity('Prostate2.jpg', 'ProstateNoTable2.jpg')

ans =

    0.9821

fx >>
```


7. Mesh Reconstruction

My function, MeshReconstruction.m, reads in an .nrrd sequence of images and converts it into a 3D visual mesh representation.

First, the function reads in the images. Then, it runs through each of the images in the sequence, computing the contour points using MeshSegmentation.m (which is a copy of RadialSegmentation.m without plotting or creating a figure, also with a smaller increment value).

It also computes the contour points of the image above it, then adds the z-components to each of these sets of contour points.

The resolution is determined in MeshSegmentation.m where I choose a 0.1 increment: $2\pi/0.1 = 62$ points for each layer. These are multiplied by the values given in .ijkToLpsTransform.m

The function plots the points at each image, connects the ending so there is no gap, then connects the images in sequence by their z value going until the 5th to last image. (Because there is no image in the last 5 images and this makes the reconstruction have a tip sticking out).

I multiplied all x values by the x-pixel size, y values by y-pixel size and z-values by the slice thickness given in the .ijkToLpsTransform.m function.

