ELEC4630 Assignment 2

Samuel Eadie - 44353607

April 18, 2018

1 Relevant Background Theory

1.1 Thresholding

Thresholding is a simple quantisation method to segment images based on a property. Often, thresholding is performed on the grayscale of an image to binarize it. All pixels in an image that are above the threshold value for the given property are replaced with white, otherwise they are turned black. Thresholding can be performed globally with a constant thresholding value, or locally where the thresholding value is dependant on the position of the pixel. The latter is generally useful in images with uncontrolled or uneven lighting (e.g. outdoors).

1.2 Morphological Transformations

Morphological transformations cover a set of operations based on shapes in an image. In these transformations, each pixel is assigned a value depending on the pixels in the neighbourhood of the former. The neighbourhood is defined by a structuring element which characterises the shape the transformation relates to. Several common morphological transformations are:

- **Dilation:** A pixel is set if any of the pixels in its neighbourhood are set. This results in an expansion of objects in the shape of the structuring element.
- **Erosion:** A pixel is set if all of the pixels in its neighbourhood are set. This results in a shrinking of objects in the shape of the structuring element.
- **Closing:** A dilation followed by an erosion is performed with a common structuring element. This removes holes that are a similar shape to the structuring element from objects.
- **Opening:** An erosion followed by a dilation is performed with a common structuring element. This expands holes and removes noise that are a similar shape to the structuring element from objects.
- **Gradient:** The difference between the dilation of an image and the erosion of an image with a common structuring element. This outlines objects.

1.3 Hough Transformation

The Hough Transform is a feature extraction technique well-known for identifying lines in images. That said, the Hough transform can also identify other shapes. The transform maps from image space to a discretised parameter space using a type of voting scheme. Once transformed, the maximum intensity points in the parameter space most likely correspond to the parameter shape in image space.

For line detection, a polar line representation is used for the parameter space to avoid an infinite parameter domain caused by the Cartesian representation of vertical lines. In polar coordinates, a line is represented as $x \cos\theta + y \sin\theta = \rho$. Therefore, every point in image space contributes a sinusoid in this polar coordinate parameter space. If a line is present in the image, the sinusoidal contribution of each point on the line will intersect at a point of maximum intensity. This intersection point in parameter space corresponds to a line in image space. This is shown in Figure 1.

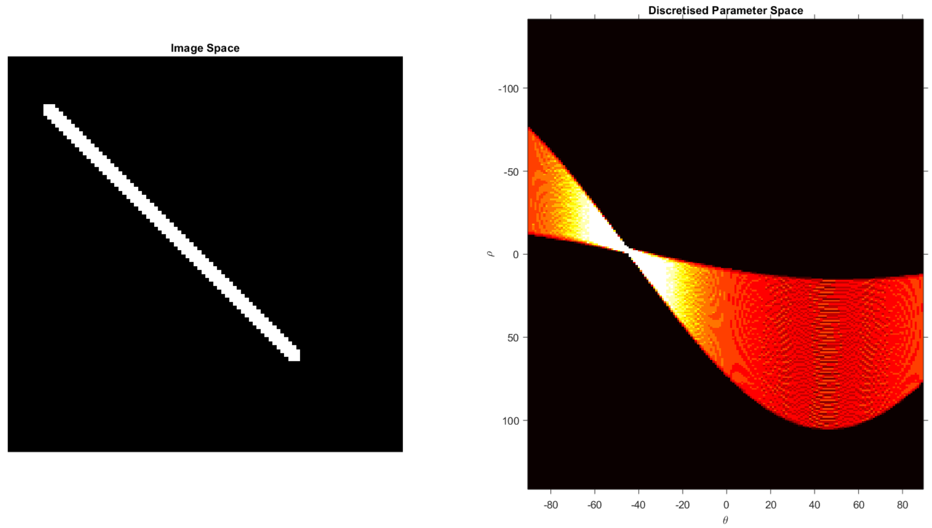


Figure 1: Hough Transform of a Line to Parameter Space

1.4 Viterbi Algorithm

The Viterbi Algorithm is a dynamic programming technique used to find the most probable sequence of states from a sequence of observations. It can be applied to image processing to find the minimum cost path for traversals/segmentations. The inherent shape to the traversal can be reflected by the coordinate system. For example, a Viterbi search space from polar coordinates is shown in Figure 2.

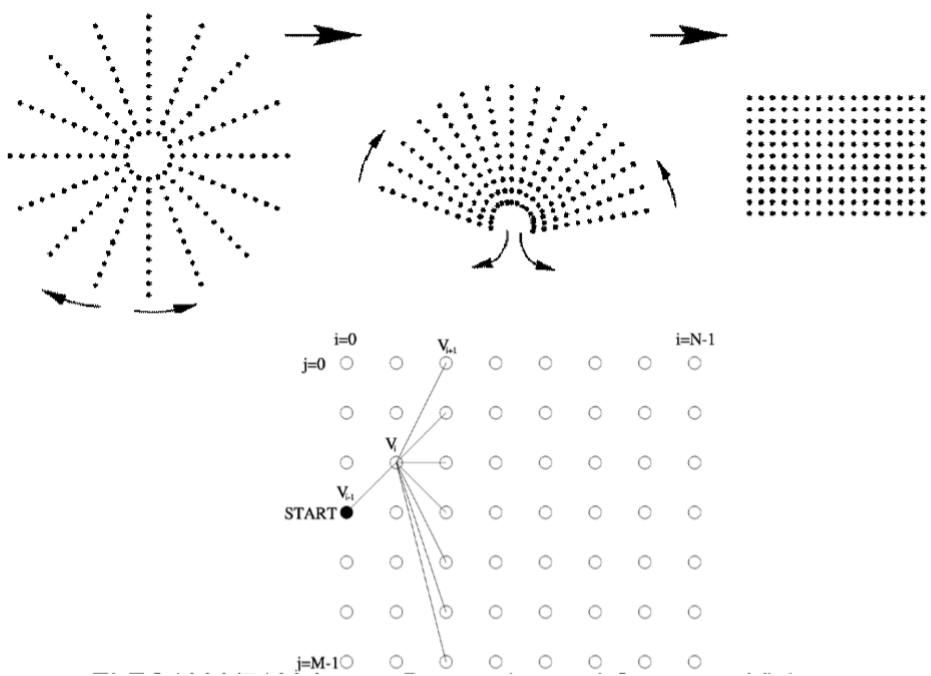


Figure 2: Viterbi Search Space in polar coordinates

2 Taj Mahal and the Hough Transform

2.1 Introduction

The Hough Transform can be used to identify shapes in images. In this section, the Hough Transform is used to outline the edges of a water feature and determine the angles of minarets in a photograph of the Taj Mahal (Figure 3). Additionally, an alternative implementation to MATLAB's Hough Transform implementation is developed and a comparison is made.

Figure 3: Taj Mahal Image

2.2 Method

The final approach begins with segmenting the image into the foreground (e.g. water feature) and background (e.g. minarets) so to allow local processing/thresholding in these distinct regions. Following this, each section was processed individually but with a similar approach: thresholding in the Lab colour space, conversion to black and white, morphological processing and application of the Hough Transform. The Lab colour space was chosen since the a axis which encodes the blue-yellow colour component is able to distinguish the Taj Mahal and water feature pavement from the sky and water respectively. An alternative implementation of the Hough Transform and MATLAB's corollary Hough functions (i.e hough, houghpeaks, houghlines) were developed and compared.

2.2.1 Image Segmentation

The Taj Mahal image was segmented into the foreground (i.e. water feature) and background (i.e. Taj Mahal) by:

1. Converting the image to grayscale
2. Thresholding the grayscale image into black and white. Due to a stark difference in illumination between background and foreground, a property which initially necessitated this segmentation, thresholding effectively segmented these regions.
3. Extracting the largest component in the image to isolate the background mask.
4. Inverting the background mask to isolate the foreground mask.
5. Applying the background and foreground masks to the image respectively/

This process is outlined in Figure 4.

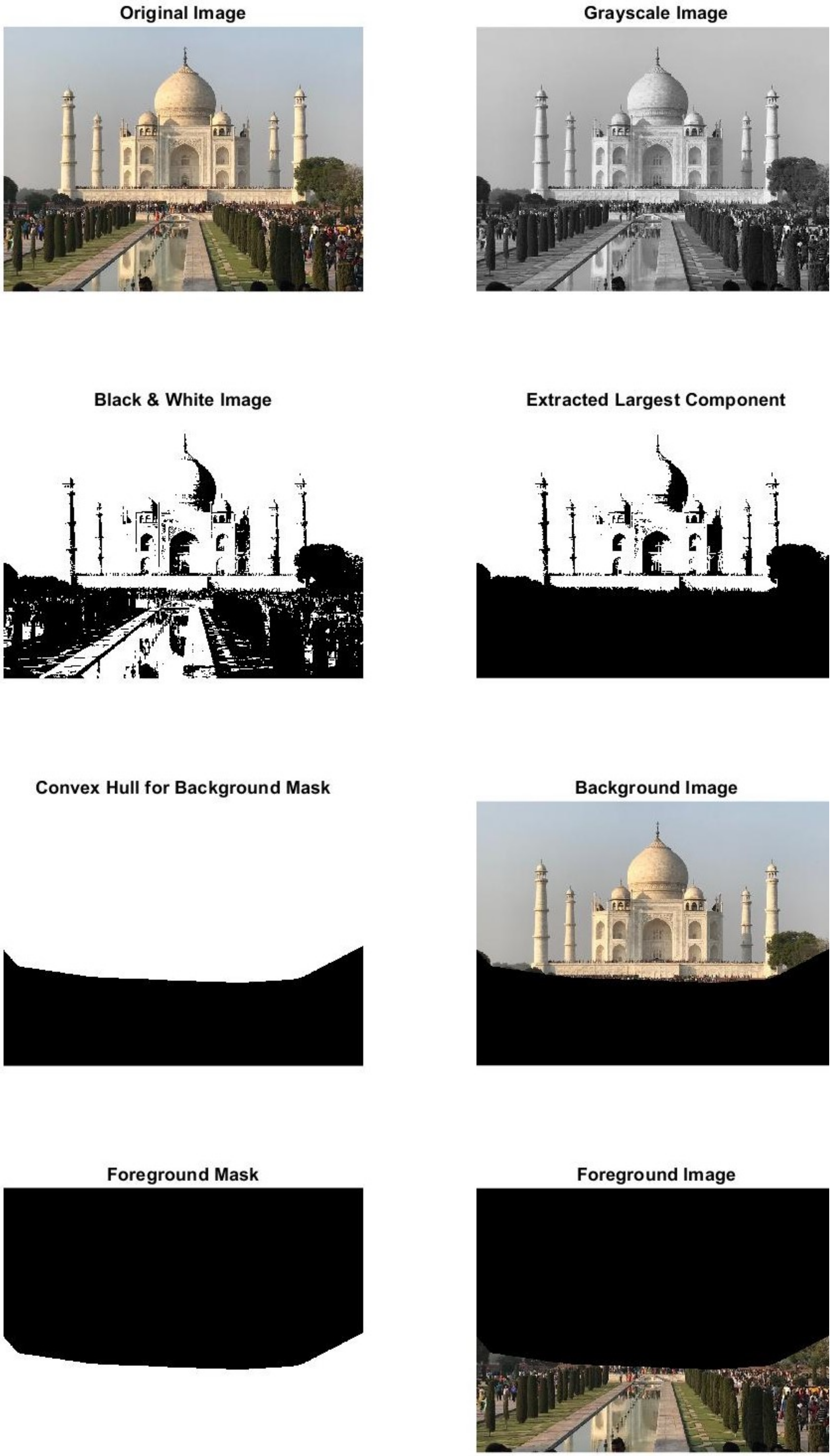


Figure 4: Separation of Taj Mahal into Foreground and Background

This process is explicated in the listing below.

Listing 1: Segmentation into Background and Foreground

```
grayImage = rgb2gray(image);
imshow(grayImage);

bwImage = imbinarize(grayImage);
imshow(bwImage);

%Remove all but largest component
largestComponent = bwImage;
stats = regionprops('table', bwImage, 'Area', 'PixelIdxList');
[sortedAreas, sortingIndex] = sort(stats.Area, 'descend');

for index = 2:length(sortingIndex)
    largestComponent(stats.PixelIdxList{sortingIndex(index)}) = 0;
end

%Find convex hull
minaretsMask = bwconvhull(largestComponent);
waterMask = ~minaretsMask;

%Apply Masks
minaretsImage = image .* uint8(repmat(minaretsMask,[1,1,3]));
waterImage = image .* uint8(repmat(waterMask,[1,1,3]));
imshow(minaretsImage);
imshow(waterImage);
```

2.2.2 Minaret Processing

After the background has been segmented, the following processing occurs to outline both edges of each minaret tower:

1. The background image is thresholded in the Lab colour space. The Lab colour space was chosen since the a axis, which encodes blue-yellow colour, clearly separates the sky from the minarets.

2. Small components are removed from the image to remove artefacts

3. The image is filled in to remove artefacts

4. A morphological close is performed with a small vertical line structuring element to fill in the towers

5. A morphological erosions is performed to remove small edges/bumps on the minaret towers

6. The image is outlined by calculating the gradient.

7. A Hough Transform is performed on the image and lines between 80^o and 100^o from the horizontal with strong peak values were found. Lines with similar positions were grouped together to make strong outlines of the minarets. The four grouped lines furthest to the left and right sides respectively were chosen and plotted over the original image

The angles of these lines were used to determine the angle of each minaret. The process is outlined in Figure 5.

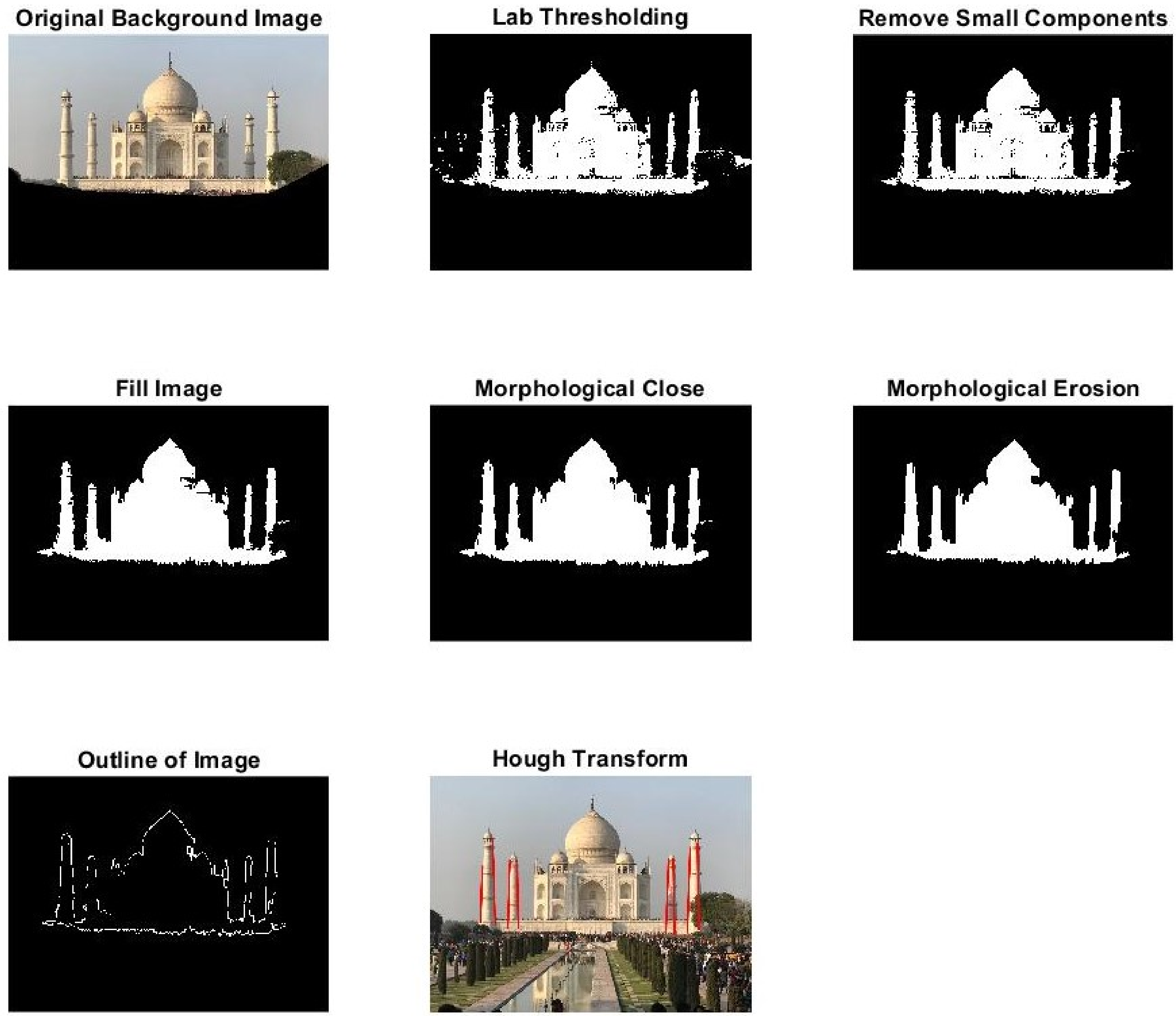


Figure 5: Processing of Minarets

This process is explicated in the listing below.

Listing 2: Processing of Minarets

```
%Threshold minarets
bwMinaret = minaretThresholder(minaretsImage);
imshow(bwMinaret);

%Remove noise
minarets = bwMinaret;
stats = regionprops('table', minarets, 'Area', 'PixelIdxList');
[sortedAreas, sortingIndex] = sort(stats.Area, 'descend');

for index = 2:length(sortingIndex)
```

```
    minarets(stats.PixelIdxList{sortingIndex(index)}) = 0;
end

%Fill Image
filled = imfill(minarets, 'holes');
imshow(filled);

%Close Image
closed = imclose(filled, strel('line', 10, 90));
imshow(closed);

%Erode image
eroded = imerode(closed, strel('line', 10, 90));
imshow(eroded);

%Outline Image
gradient = imgradient(eroded);
imshow(gradient);

%Hough Transform
[H,T,R] = hough(gradient);
H1 = H(:, find(T==70-90):find(T==89-90));
H2 = H(:, find(T==91-90):find(T==110-90));

T1 = T(find(T==70-90):find(T==89-90));
T2 = T(find(T==91-90):find(T==110-90));

P1  = houghpeaks(H1,5, 'Threshold', 0.3*max(H1(:)));
P2  = houghpeaks(H2,5, 'Threshold', 0.3*max(H1(:)));
L1 = houghlines(eroded,T1,R,P1,'FillGap',5,'MinLength', 15);
L2 = houghlines(eroded,T2,R,P2,'FillGap',5,'MinLength', 15);
myLines = [L1 L2];

%Group Hough Lines by rho
groupedLines = groupLines(myLines, 6, 3);

%Select left/right most four lines respectively
[sortedX, sortingIndex] = sort(groupedLines(:,1), 'ascend');
leftLines = groupedLines(sortingIndex(1:4), :);
rightLines = groupedLines(sortingIndex(size(groupedLines, 1) - 3:size(
    groupedLines, 1)), :);
minaretLines = [leftLines; rightLines];

plotted = insertShape(image, 'line', minaretLines, 'LineWidth',6,'Color',
    'red');
imshow(plotted);

%Calculate angles
angles = zeros(size(minaretLines, 1), 1);
for index = 1:size(minaretLines, 1)
```

```
    angle = atand((minaretLines(index, 4) - minaretLines(index, 2)) ./
       ...
                  (minaretLines(index, 3) - minaretLines(index, 1)));
    if(angle >= 0)
        angles(index) = angle;
    else
        angles(index) = angle + 180;
    end
end

angles
```

2.2.3 Water Feature Processing

After the foreground has been segmented, the following processing occurs to outline the edges of the water feature:

1. The foreground image is thresholded in the Lab colour space. The Lab colour space is used since the a axis, which encodes blue-yellow colour, easily distinguishes the water from the adjacent pavement.

2. Small components are removed from the image to remove artefacts.

3. A morphological close is performed with a small line structuring element to fill in the water feature edges.

4. The image is outlined by calculating the gradient.

5. A Hough Transform is performed on the image and lines between 30^o and 100^o from the horizontal with strong peak values were found. Lines with similar angles and positions were grouped together to make strong outlines of the water feature.

This process is outlined in Figure 6.

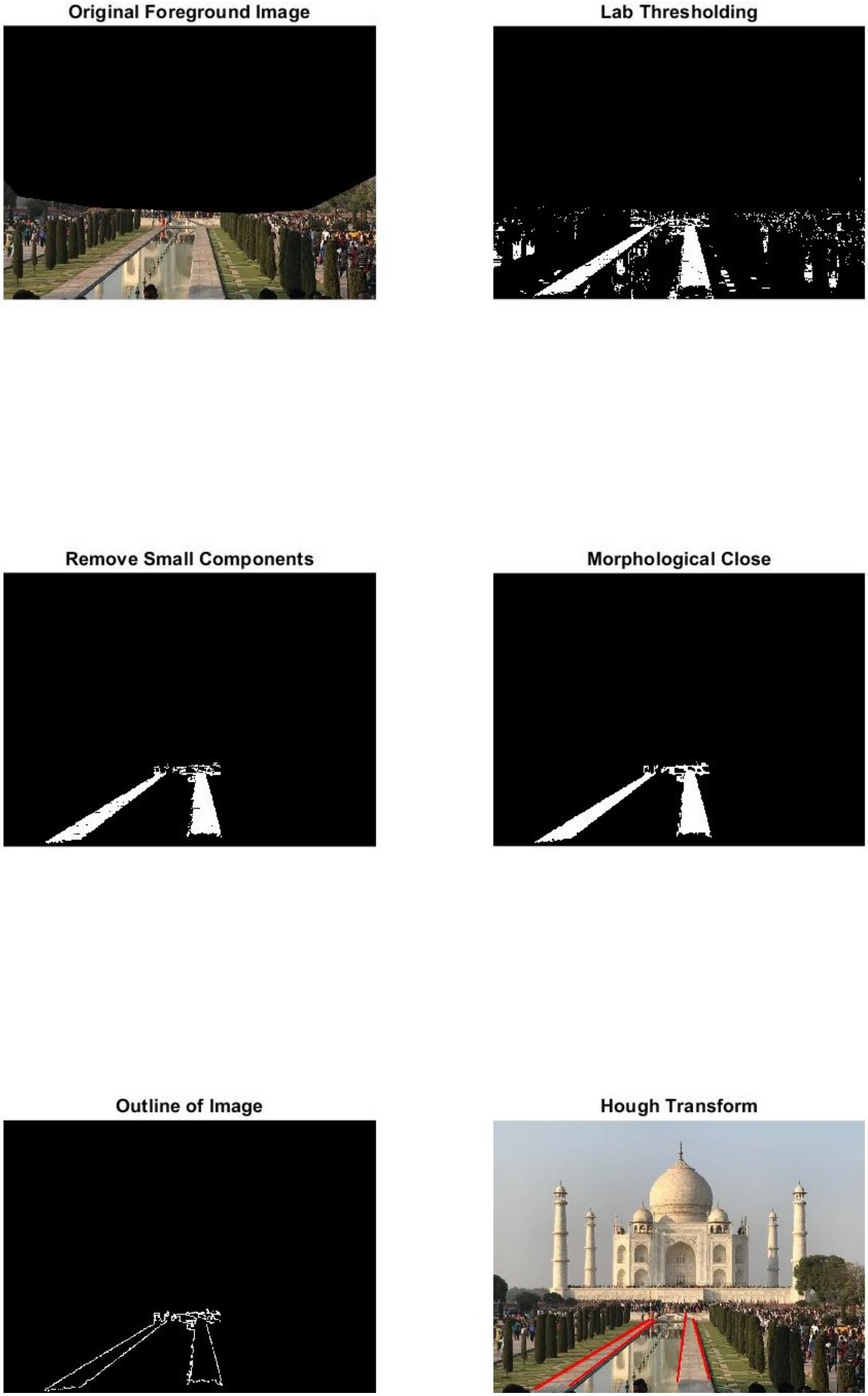


Figure 6: Processing of Water Feature

This process is explicated in the listing below.

Listing 3: Processing of Water Feature

```
% Threshold in LAB domain
bwWater = waterThresholder(waterImage);
imshow(bwWater);

%Denoise image
denoised = bwareaopen(bwWater, 1000);
imshow(denoised);

%Close Image
closed = imclose(denoised, strel('line', 5, 90));
imshow(closed);

%Outline Image
gradient = imgradient(closed);
imshow(gradient);

%Hough Transform
[H,T,R] = hough(gradient);
H1 = H(:, find(T==140-90):find(T==160-90)); %-140 to -160 degrees
H2 = H(:, find(T==93-90):find(T==100-90));  %-93  to -100 degrees
H3 = H(:, find(T==70-90):find(T==80-90));   %-70  to -80  degrees

T1 = T(find(T==140-90):find(T==160-90));
T2 = T(find(T==93-90):find(T==100-90));
T3 = T(find(T==70-90):find(T==80-90));

P1 = houghpeaks(H1,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H1(:)));
P2 = houghpeaks(H2,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H2(:)));
P3 = houghpeaks(H3,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H3(:)));
L1 = houghlines(closed,T1,R,P1,'FillGap',5,'MinLength', 15);
L2 = houghlines(closed,T2,R,P2,'FillGap',5,'MinLength', 15);
L3 = houghlines(closed,T3,R,P3,'FillGap',5,'MinLength', 15);
myLines = [L1 L2 L3];

%Group lines by theta
waterLines = groupLines(myLines, 5, 5);
plotted = insertShape(image, 'line', waterLines, 'LineWidth',6,'Color','red');
imshow(plotted);
```

2.2.4 Alternative Hough Transform Implementation

An alternative implementation to MATLAB's Hough Transform was developed. It is a naive implementation that follows the aforementioned theory. It converts each point in image space to the discretised parameter space for straight lines. Polar coordinates were used to ensure a finite parameter domain; hence, each point in image space is transformed by $x \cos\theta + y \sin\theta = \rho$.

Listing 4: Hough Transform Implementation

```
function [hough, theta, rho] = myHough(BW)
    [numRows, numCols] = size(BW);

    %Calculate theta, rho ranges
    theta = linspace(-90, 89, 180);
    D = sqrt((numRows - 1)^2 + (numCols - 1)^2);
    nrho = 2*ceil(D) + 1;
    rho = linspace(-ceil(D), ceil(D), nrho);

    %Initialise discretised hough space
    hough = double(zeros(length(rho), length(theta)));

    %Find white pixels in image
    [y, x] = find(BW);

    %Transform image to parameter space
    for index = 1:numel(x)
        calculatedRho = int16(x(index) * cos(pi * theta / 180) + y(index)
            * sin(pi * theta / 180));
        hough = hough + double(transpose(rho) == calculatedRho);
    end
end
```

Furthermore, MATLAB's houghpeaks and houghlines functions were also implemented. The Hough peaks function selects the *numpeaks* largest values in parameter space and returns the associated *theta* and *rho*. However, after each peak selection, all values in the peak's *proximity* are zeroed to avoid selecting multiple points from the same peak region. The Hough lines function finds the longest line segment in the *bw* image corresponding to the peak *theta* and *rho* values.

Listing 5: Hough Peaks Implementation

```
function [P] = myHoughPeaks(hough, numPeaks, proximity)
    P = zeros(numPeaks, 2);

    for index = 1:numPeaks
        %Find first peak
        [row, col] = find(hough == max(max(hough)));
        P(index, :) = [row(1) col(1)];

        %Set cells in proximity to zero
        distance = round(proximity / 2);
        hough(max(1,row(1)-distance):min(size(hough,1),row(1)+distance),
           ...
```

```
                max(1,col(1)-distance):min(size(hough,2),col(1)+distance))
                   = 0;
    end
end
```

Listing 6: Hough Lines Implementation

```
function [lines] = myHoughLines(bw, theta, rho, peaks)
    lines = zeros(size(peaks, 1), 4);

    for index = 1:size(peaks, 1)
        %Get peak theta, rho
        T = theta(peaks(index,2));
        R = rho(peaks(index, 1));

        %Convert polar to cartesian
        xs = R .* cos(pi .* (T) ./ 180);
        ys = R .* sin(pi .* (T) ./ 180);
        m = tan(pi.*(T + 90) ./ 180);
        c = ys - (m .* xs);

        %Calculate line points
        xs = 1:size(bw,2);
        ys = round(c + (m * xs));
        xs = xs((ys >= 1) & (ys <= size(bw, 1)));
        ys = ys((ys >= 1) & (ys <= size(bw, 1)));

        %Find longest line in points
        whites = bw(sub2ind(size(bw), ys, xs));
        start = find(whites, 1, 'first');
        finish = start;
        while whites(finish)
            finish = finish + 1;
        end

        lines(index,:) = [xs(start) ys(start) xs(finish - 1) ys(finish -
           1)];
    end
end
```

2.3 Results

This approach outlined above was successful in outlining the water feature, outlining the minarets and calculating their angle. The combined outlined image is shown in Figure 7. The outlines of the minarets were used to calculate the minaret angles anti-clockwise from the horizontal. From left to right, these are 90.1850^o, 90.41815^o, 89.4822^o and 89.4499^o respectively.

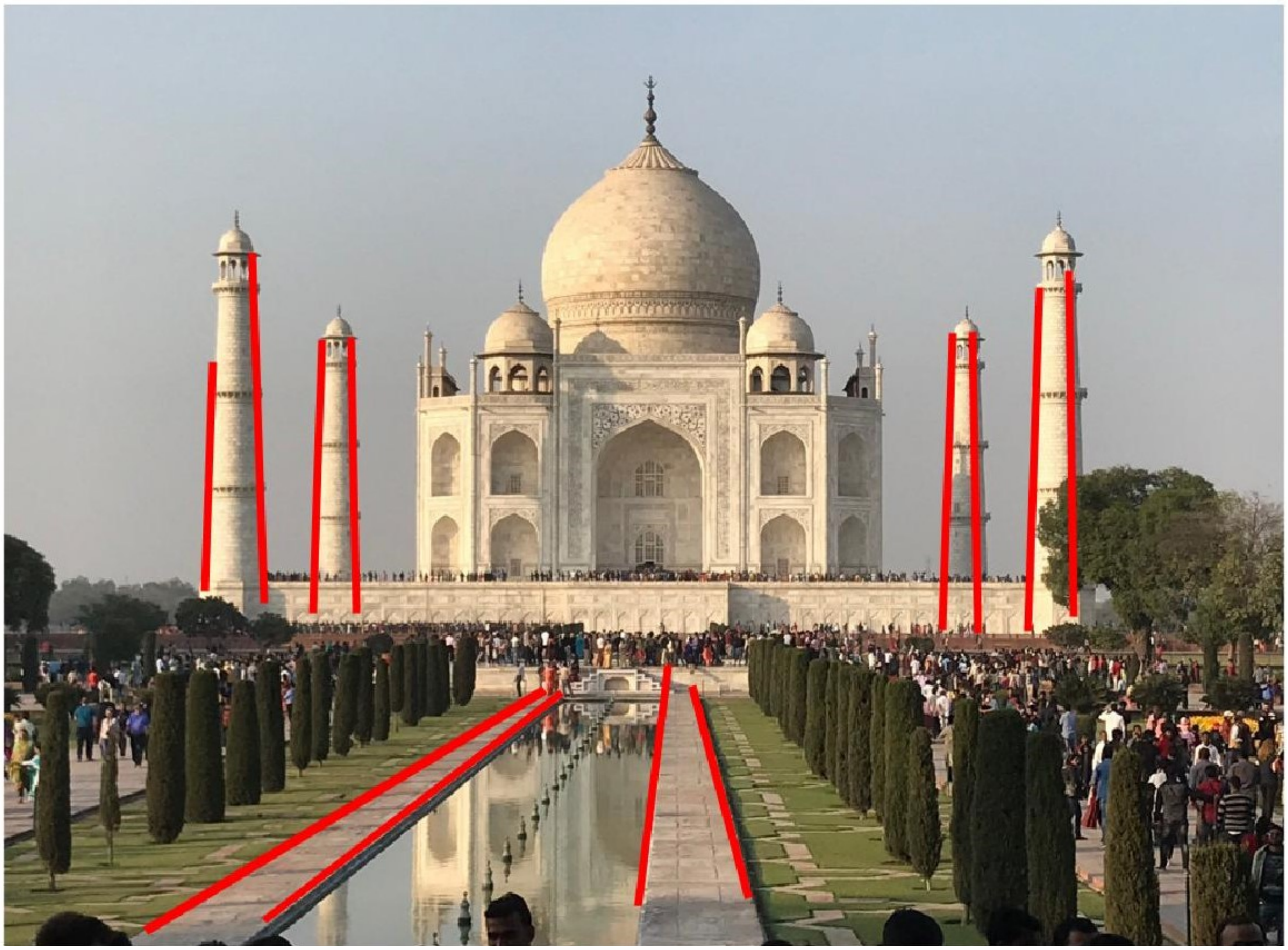


Figure 7: Outline of Minarets and Water Feature

The primary source of error in the segmentation and hence outlining of the Taj Mahal stemmed from shadows on objects (e.g. the right side of minarets, the bottom of the water feature). This caused issues with thresholding the image, and by extension, the outlining of these sections. This is exemplified by the water feature lines ceasing at the shadow, and the right minarets' right outline being off.

2.3.1 Alternative Hough Transform Implementation Results

The alternative Hough Transform developed was compared against MATLAB's implementation. Three sample black and white images were used in this comparison. These comparisons are shown in Figures 8 - 10.

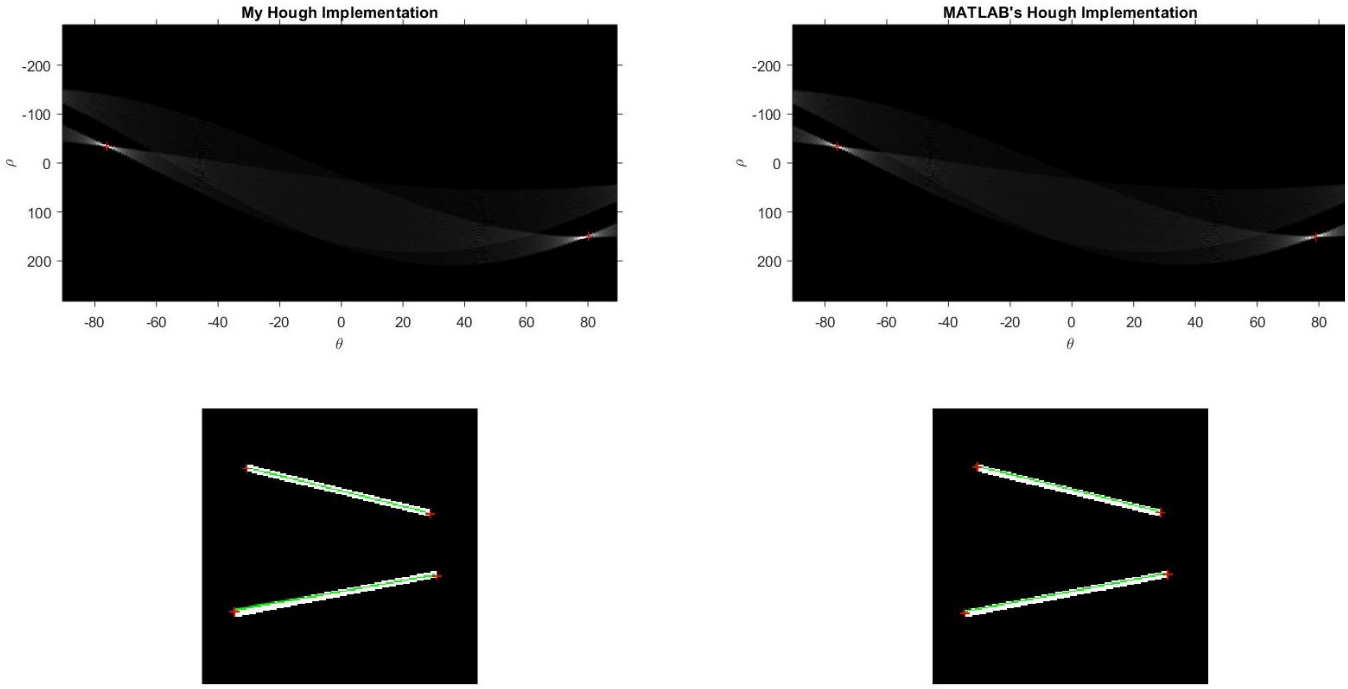


Figure 8: Comparison 1

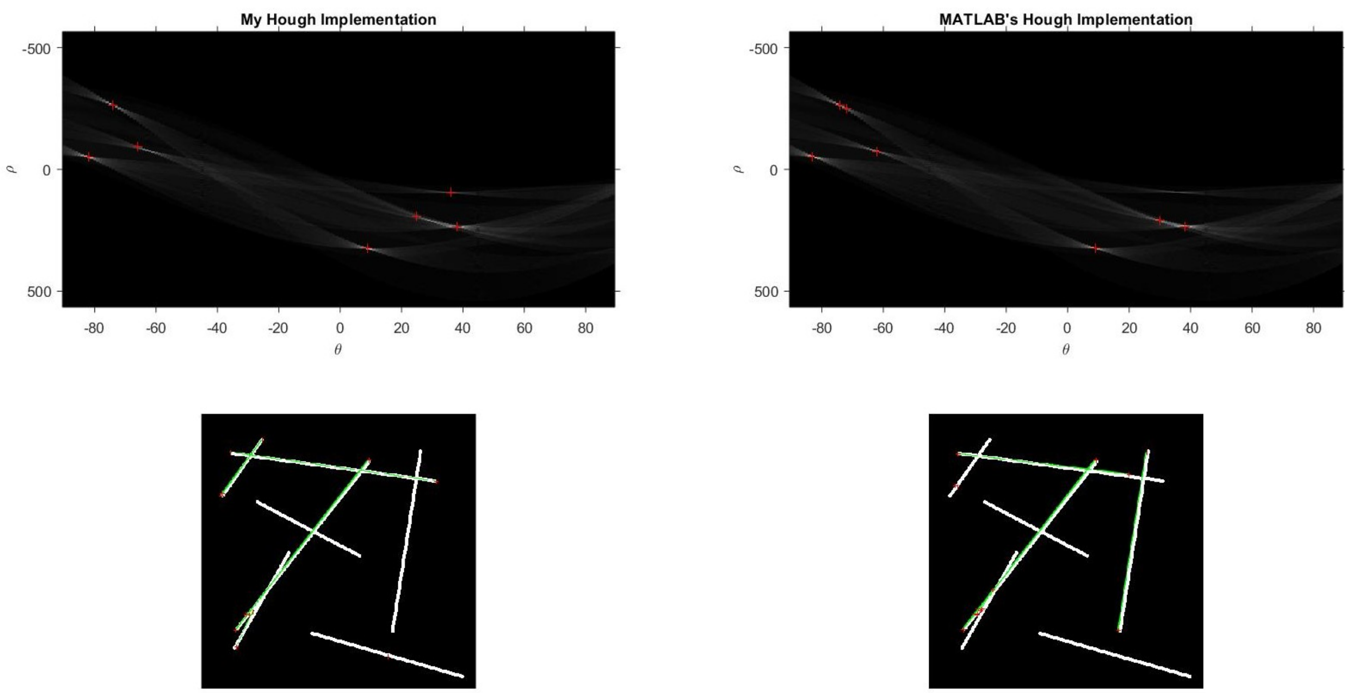


Figure 9: Comparison 2

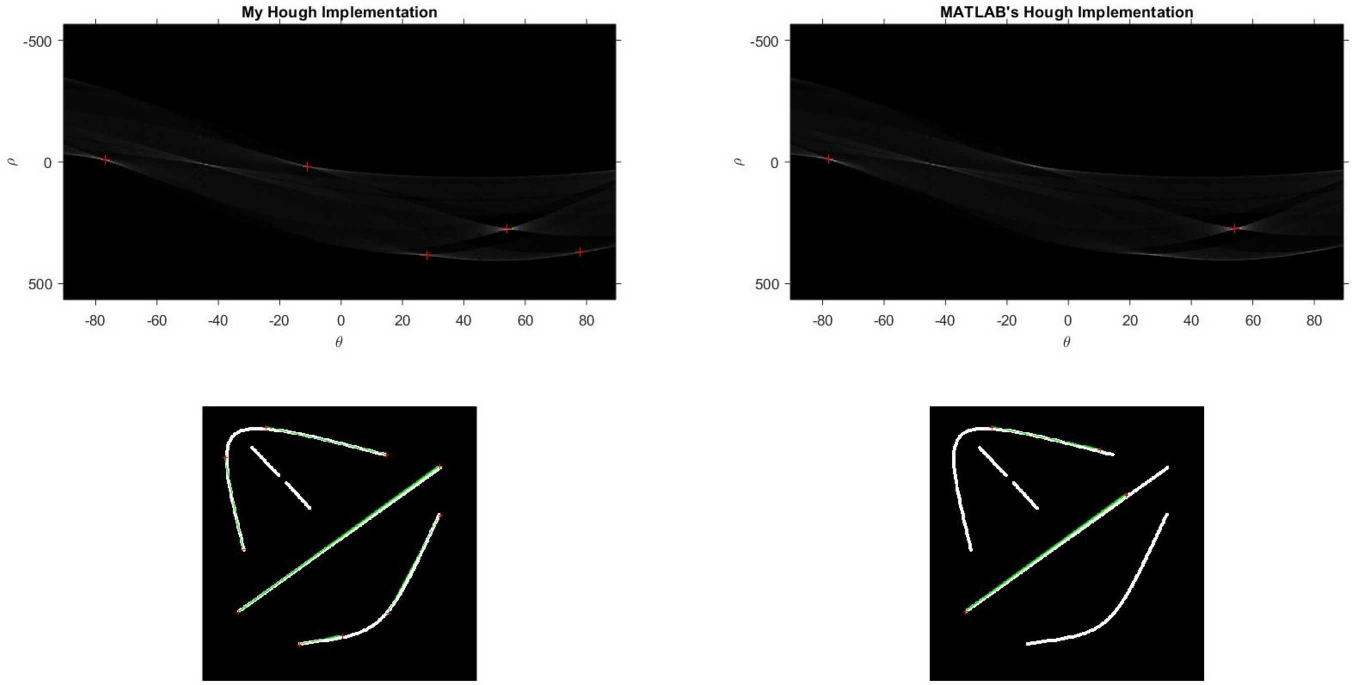


Figure 10: Comparison 3

Evidently, this alternative Hough Transform implementation matched MATLAB's, but their are discrepancies in the houghpeaks and houghlines implementations. In addition, the alternative implementation is notably less computationally efficient and has fewer optional arguments than MATLAB's. For these reasons, the MATLAB implementation was used.

3 Contouring Heart MRIs through Cost Minimisation in the Viterbi Trellis Search Space

3.1 Introduction

Medical imaging is an ever increasing application of image processing. In this section, the inner and outer walls of the left ventricle are automatically outlined in a time-sequence of MRI scans of the heart. Through this process, the area of the left ventricle was determined over time, providing important information to cardiologists.

3.2 Method

The overarching approach is outlined in Figure 11. It involves

- Thresholding scan to find a heart mask
- Locating the centroid of the heart mask
- Establishing polar coordinates centred at the heart's centroid
- Finding the initial starting points for the inner and outer ventricle walls
- Performing a minimum cost traversal around the ventricle on the inner and outer walls
- Calculating area between inner and outer ventricle walls

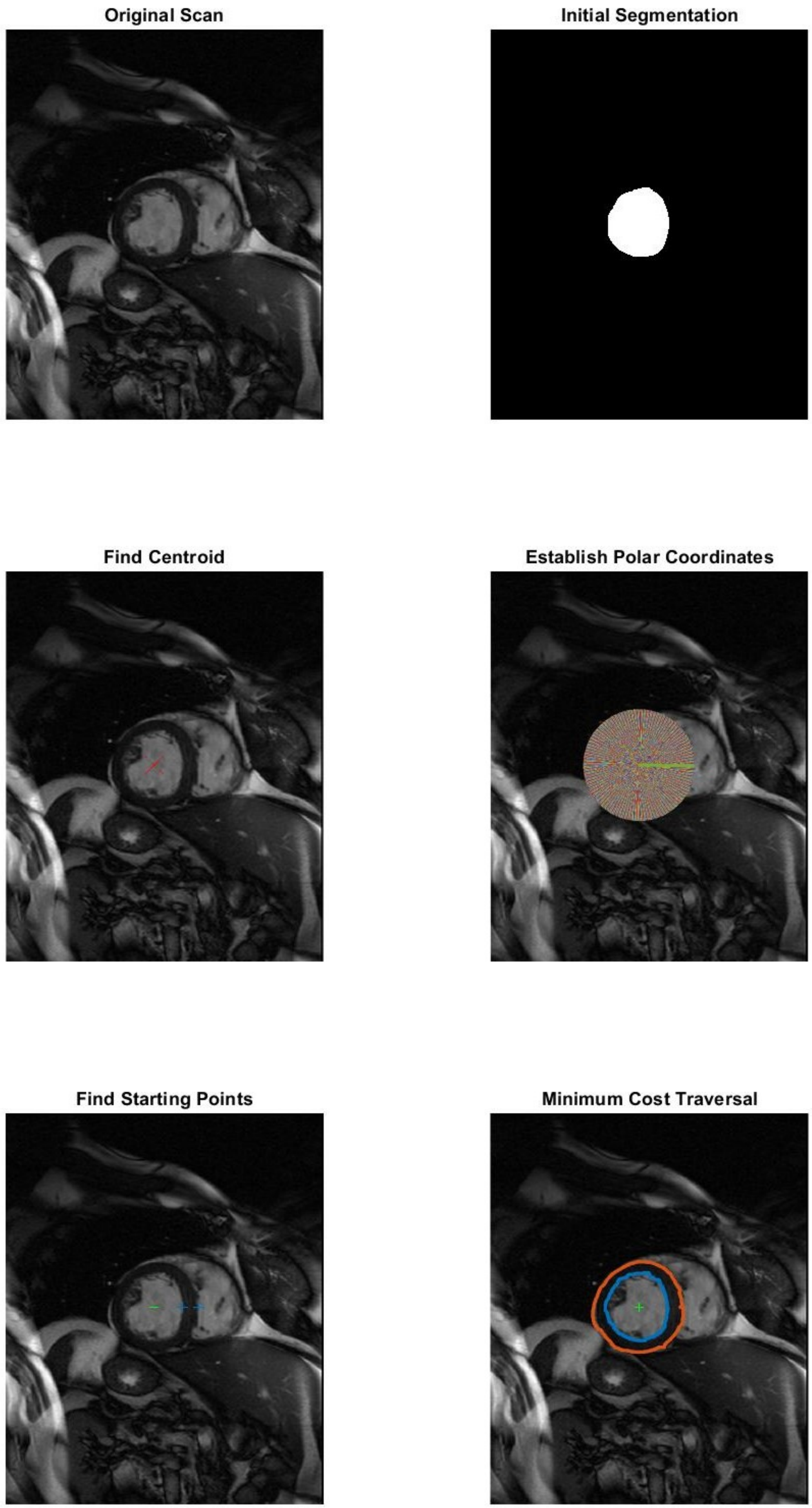


Figure 11: Heart Processing Overview

This process is explicated in the listing below.

Listing 7: Processing of Heart Scans

```
        function [innerPath, outerPath] = getOutlines(self, innerLambda,
           outerLambda, radiiThreshold)
            if nargin < 4
                radiiThreshold = self.theDefaultRadiiThreshold;
            end
            if nargin < 3
                outerLambda = self.theDefaultOuterLambda;
            end
            if nargin < 2
                innerLambda = self.theDefaultInnerLambda;
            end

            %Initial Segmentation
            mask = self.getHeartMask();

            %Find Centroid
            [centroid, diameter] = self.getCentroid(mask);

            %Establish Polar Coordinates
            [radii, thetas, xs, ys] = self.getPolarCoordinates(centroid);

            %Find starting points
            startingRadii = self.getStartingRadii(xs, ys);
            innerRadius = min(startingRadii);
            outerRadius = max(startingRadii) - 5;

            %Minimum Cost Traversal
            [innerPath, innerRadii] = self.findMinimumPath(centroid,
               innerRadius, innerLambda, radiiThreshold);
            [outerPath, outerRadii] = self.findMinimumPath(centroid,
               outerRadius, outerLambda, radiiThreshold);
        end
```

The following subsections provide further detail into each processing section.

3.2.1 Initial Heart Segmentation

An initial segmentation was performed to outline the heart. The approach is outlined in Figure 12. It involves:

- The scan is thresholded and binarized
- A circularity metric was devised to threshold components based on circularity. This metric is the ratio of the component's major axis length to its equivalent diameter. Component with similar major axis lengths and equivalent diameters are more deemed more circular.
- The largest component is then selected
- The convex hull of the selected component is calculated and used as a mask

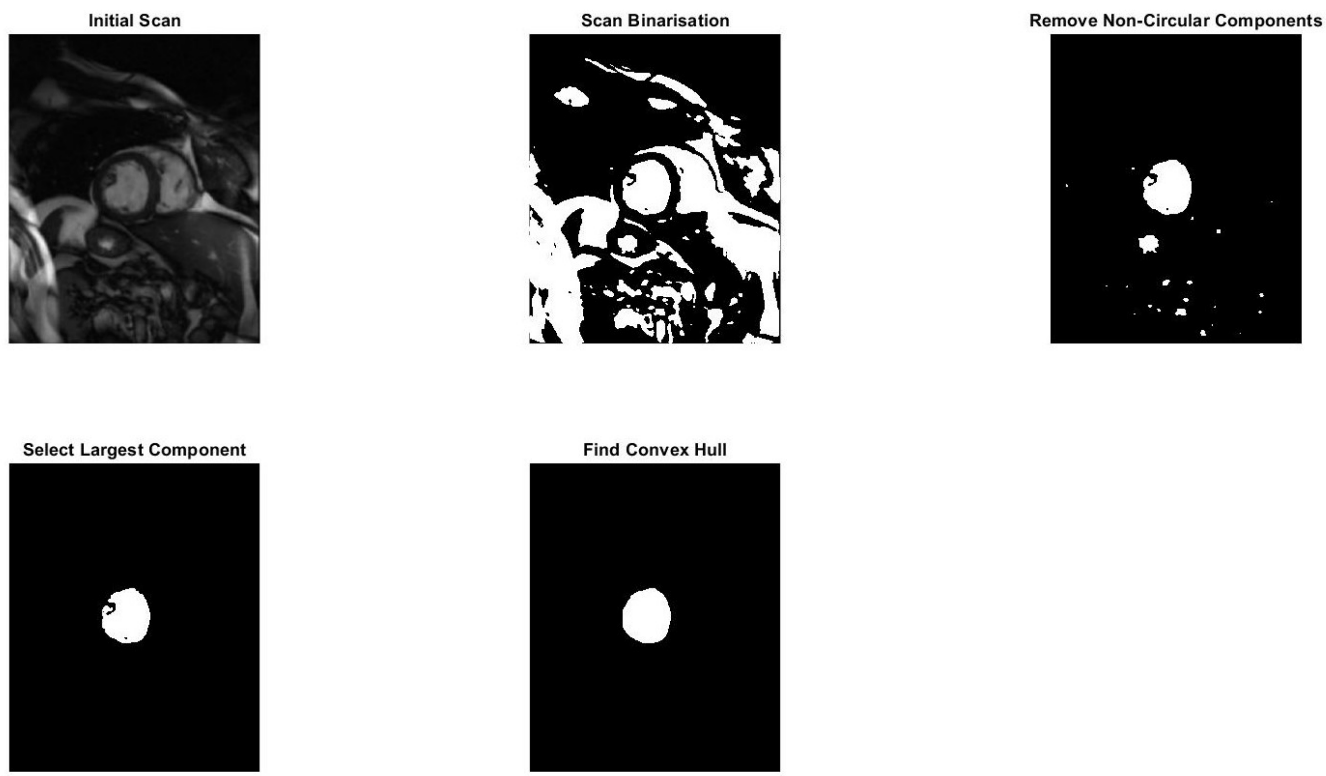


Figure 12: Outline of Processing for Heart Mask

This process is explicated in the listing below.

Listing 8: Initial Segmentation

```
        function mask = getHeartMask(self)
            mask = imbinarize(self.theScan, self.theBinaryThreshold);

            %Threshold on circularity
            stats = regionprops('table', mask, 'PixelIdxList', '
               MajorAxisLength', 'EquivDiameter');
            circularity = stats.MajorAxisLength ./ stats.EquivDiameter;

            for index = 1:length(circularity)
                if circularity(index) < self.theMinCircularity || ...
                   circularity(index) > self.theMaxCircularity
                    mask(stats.PixelIdxList{index}) = 0;
                end
            end

            %Redefine components in image and their size
            connectedComponents = bwconncomp(mask);
            numPixels = cellfun(@numel,connectedComponents.PixelIdxList);

            [largestNum, idx] = max(numPixels);

            %Remove all but largest component
            for index = 1:length(connectedComponents.PixelIdxList)
                if index ~= idx
                    mask(connectedComponents.PixelIdxList{index}) = 0;
                end
            end

            %Convert to conxex hull
            mask = bwconvhull(mask);
        end
```

3.2.2 Heart Centroid Localisation

The centroid of the heart mask found in the previous section is calculated. An example is shown in Figure 13 and the process is explicated in the listing below.

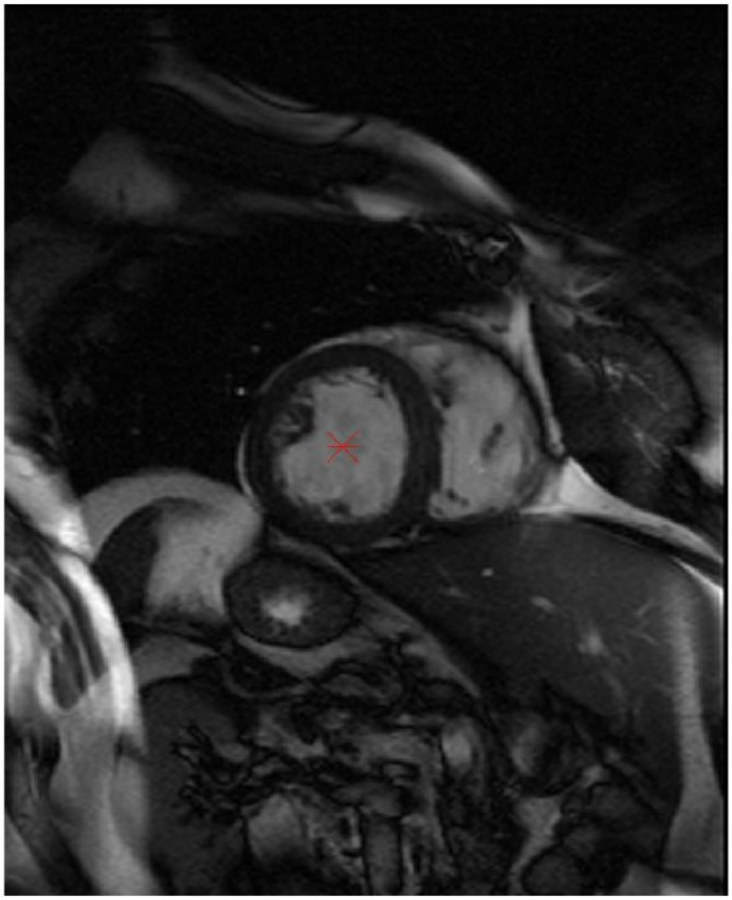


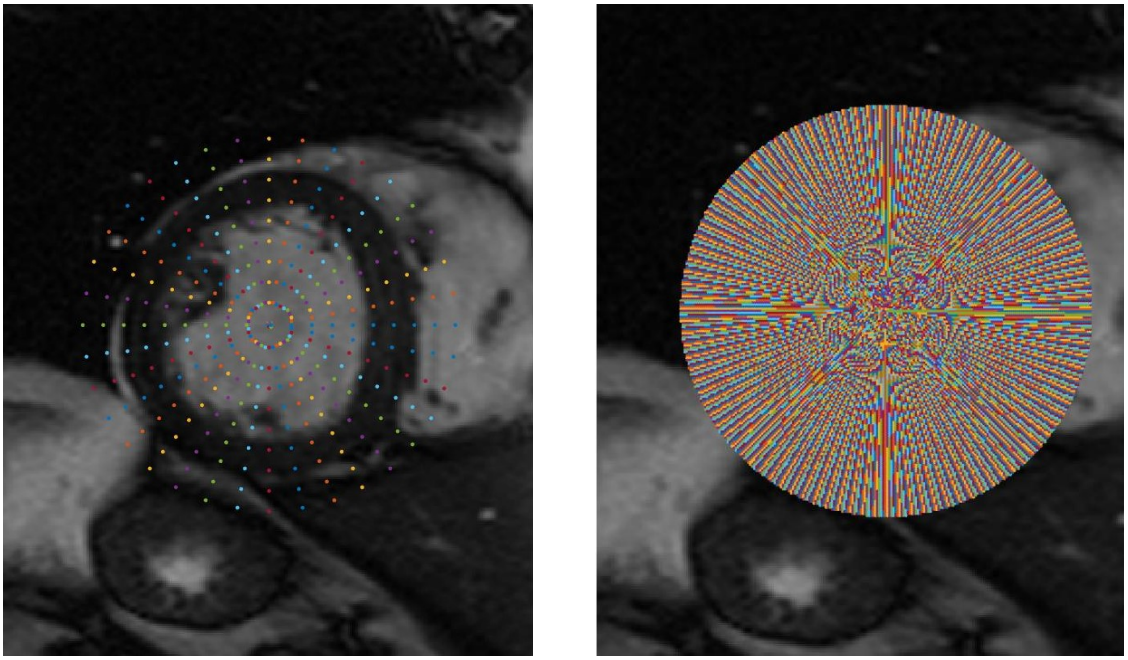
Figure 13: Centroid of Heart Mask

Listing 9: Centroid Localisation

```
        function [centroid, diameter] = getCentroid(self, mask)
            stats = regionprops('table', mask, 'centroid', 'EquivDiameter
                ');
            centroid = stats.Centroid;
            diameter = stats.EquivDiameter;
        end
```

3.2.3 Establish Polar Coordinates

The centroid found in the previous section is used as the origin in the establishment of a polar coordinate system. An example of such a polar coordinate system, discretised to 36 spokes and 10 radii points from 0 to 100 pixels, is shown in Figure 14. A more detailed polar coordinate system was used for the minimum cost traversal to improve accuracy: 720 spokes and 200 radii points from 0 to 100 pixels. This is shown alongside in Figure 14.



(a) 36 spokes, 10 radii points (b) 720 spokes, 200 radii points

Figure 14: Polar Coordinates Established at Heart Centroid

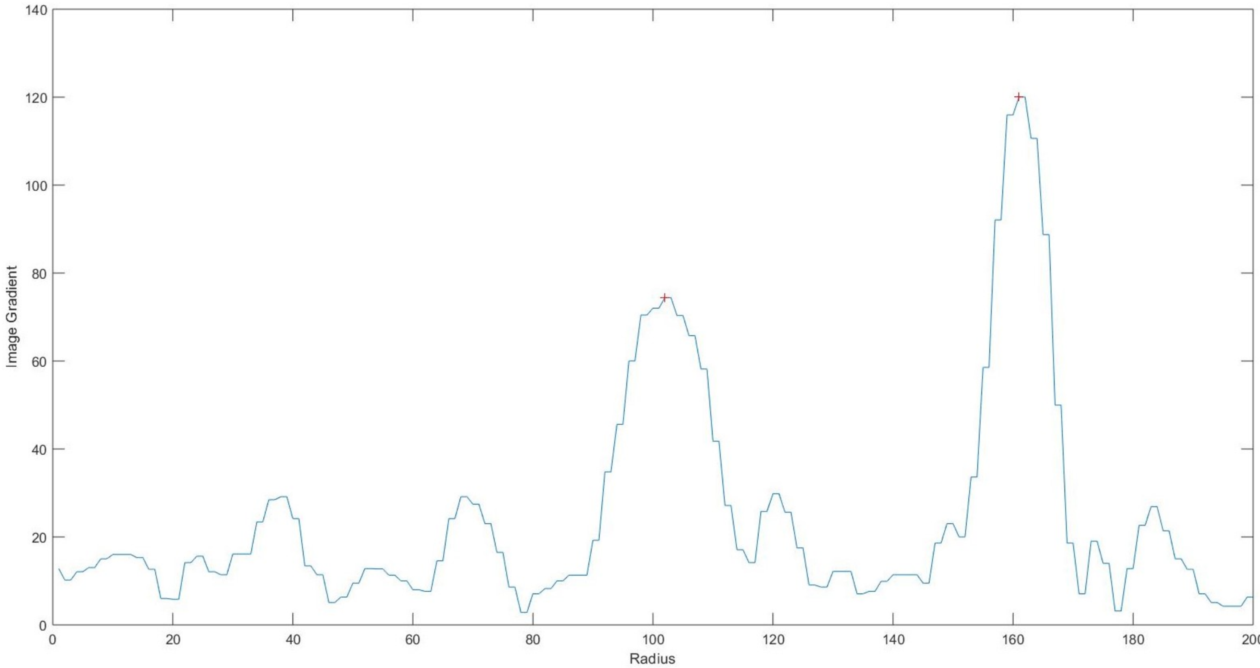
This process is explicated in the listing below.

Listing 10: Establishment of Polar Coordinates

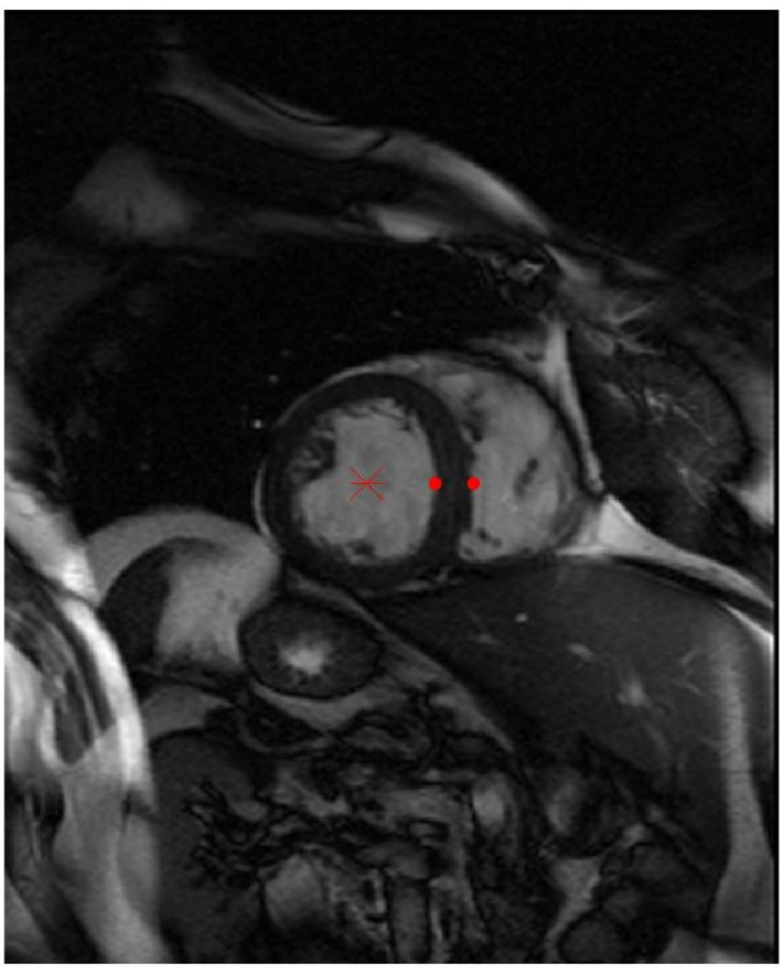
```
        function [radii, thetas, xs, ys] = getPolarCoordinates(self, centroid, numRadii, thetaSpacing)
            if nargin < 4
                thetaSpacing = self.theDefaultThetaSpacing;
            end
            if nargin < 3
                numRadii = self.theDefaultNumRadii;
            end

            radii = double(linspace(1, self.theMaxRadius, numRadii));
            thetas = 0:thetaSpacing:359;
            xs = round(centroid(1) + transpose(radii) * cos(pi*thetas/180));
            ys = round(centroid(2) - transpose(radii) * sin(pi*thetas/180));
        end
```

3.2.4 Starting Point Localisation

The minimum cost traversal starts from 0^o. The starting radii for the inner and outer walls are the two maximum peak image gradients along the 0^o spoke in the polar coordinate system. The results of this approach is shown in Figure 15.



(a) Locating Maximum Image Gradient Peaks along Initial Spoke



(b) Maximum Image Gradient Peaks on Scan

Figure 15: Starting Point for Minimum Cost Traversal

This process is explicated in the listing below.

Listing 11: Localisation of Starting Points

```
        function [startingRadii] = getStartingRadii(self, xs, ys)
            theStartingMags = self.theGradMags(sub2ind(size(self.
               theGradMags), ys(:,1), xs(:,1)));
            [peakValues, peakIndexes] = findpeaks(theStartingMags);

            [sortedVals, sortedIndex] = sort(peakValues, 'descend');
            sortedPeakIndexes = peakIndexes(sortedIndex);

            startingRadii = xs(sortedPeakIndexes(1:2), 1);
        end
```

3.2.5 Minimum Cost Traversal

A novel cost function was developed to allow for a minimum cost traversal around the inner and outer ventricle walls. The cost function consists of three components:

- **Circularity Cost:** The difference between a point's radius and the mean of all previously traversed radii. This cost is normalised by the maximum radius. This cost is also scaled by the number of points comprising the mean to ensure that, at the beginning of the traversal, it is not skewed from a lack of points. This cost aims to retain the circularity inherent in the ventricle walls.

- **Momentum Cost:** The difference between a point's radius and the previous traversed radius. This cost is normalised by the maximum radius. This cost aims to protect the traversal from sudden spikes in radii across consecutive points. Due to the density of spokes, consecutive points should be located nearby.

- **Gradient Cost:** The gradient of the image at a point. This cost is normalised by the maximum gradient in the image. This cost aims to incorporate the ventricle's outline in the cost function.

A result of this minimum cost traversal is shown on an example scan in Figure 16.

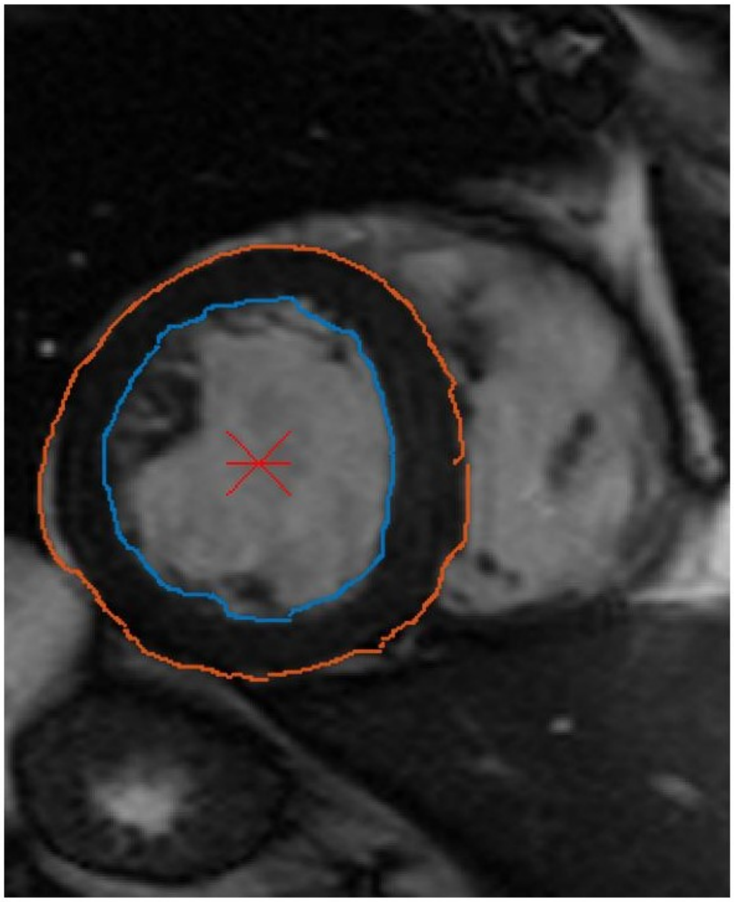


Figure 16: Minimum Cost Traversal around Inner and Outer Walls

This process is explicated in the listing below.

Listing 12: Minimum Cost Traversal

```
        function [minCartesianPath, minRadiiPath] = findMinimumPath(self,
            centroid, startingRadius, lambda, radiiThreshold)
            if nargin < 6
                radiiThreshold = self.theDefaultRadiiThreshold;
            end
            if nargin < 5
                lambda = self.theDefaultLambda;
            end

            %Initialise polar coordinate system
            [radii, thetas, xVals, yVals] = self.getPolarCoordinates(
               centroid);

            %Pre-allocate memory for minimum radius path
            minRadiiPath = zeros(length(thetas), 1);

            %Find starting point
            [value, index] = min(abs(xVals(:,1) - startingRadius));
            minRadiiPath(1) = index;
```

```

            %Preallocate memory for metric tracking
            self.theCircularityCosts = zeros(length(thetas), 1);
            self.theGradientCosts = zeros(length(thetas), 1);
            self.theWeightedCosts = zeros(length(thetas), 1);
            self.theMomentumCosts = zeros(length(thetas), 1);

            minRadiiPath(2) = minRadiiPath(1);
            %Traverse around
            for t = 3:length(thetas)
                %Calculate costs
                circularityCost = transpose(abs(radii - mean(radii(minRadiiPath(1:t-1))))) / max(radii);
                momentumCost = transpose(abs(2.*radii - radii(minRadiiPath(t-1)) - radii(minRadiiPath(t-2)))) / max(radii);
                gradientCost = self.theGradMags(sub2ind(size(self.theGradMags), yVals(:,t), xVals(:, t))) ./ self.theMaxGradMag;
                weightedCost = lambda.*(momentumCost + (t / length(thetas) .* circularityCost)) + (1 - gradientCost);

                %Choose minimum cost radii
                [value, index] = min(weightedCost(max(minRadiiPath(t-1) - radiiThreshold, 1): ...
                                                  min(minRadiiPath(t-1) + radiiThreshold, length(radii))));
                index = index(1);
                minRadiiPath(t) = minRadiiPath(t-1) - radiiThreshold + index - 1;
                minRadiiPath(t) = min(max(minRadiiPath(t), 1), length(radii));

                %Record metrics
                self.theCircularityCosts(t-1) = circularityCost(index);
                self.theMomentumCosts(t-1) = momentumCost(index);
                self.theGradientCosts(t-1) = gradientCost(index);
                self.theWeightedCosts(t-1) = weightedCost(index);
            end

            %Convert polar to cartesian
            minCartesianPath = [xVals(sub2ind(size(xVals), minRadiiPath, transpose(1:length(thetas)))) ...
                                yVals(sub2ind(size(yVals), minRadiiPath, transpose(1:length(thetas))))];
        end
```

3.2.6 Calculate Ventricle Area

The ventricle area was calculated as the difference in area between the inner and outer ventricle walls. The area of the inner and outer walls was calculated as the area contained in a polygon comprising of all points in the respective traversal.

This process is explicated in the listing below.

Listing 13: Calculation of Cross-Sectional Area

```
        function [area, innerArea, outerArea] = getArea(self, innerPath,
            outerPath)
            [innerBoundary, innerArea] = boundary(innerPath);
            [outerBoundary, outerArea] = boundary(outerPath);
            area = outerArea - innerArea;
        end
```

3.3 Results

The outline of the inner and outer ventricle walls on each of the 16 scans respectively is shown in Figure 17.

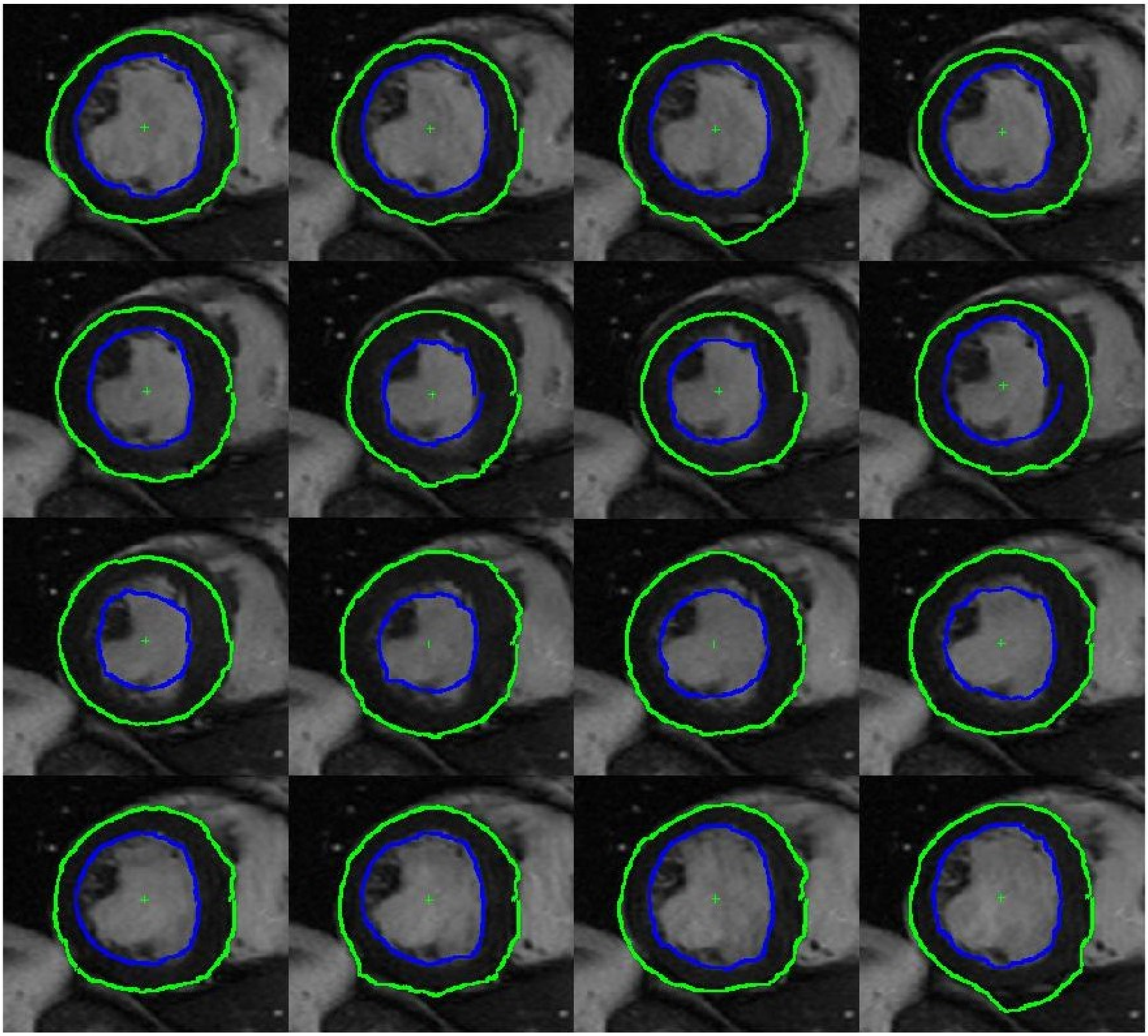


Figure 17: Result of Minimum Cost Traversal on All Scans

The cross-sectional area of the inner and outer ventricle walls is plotted alongside the net ventricular area in Figure 18.

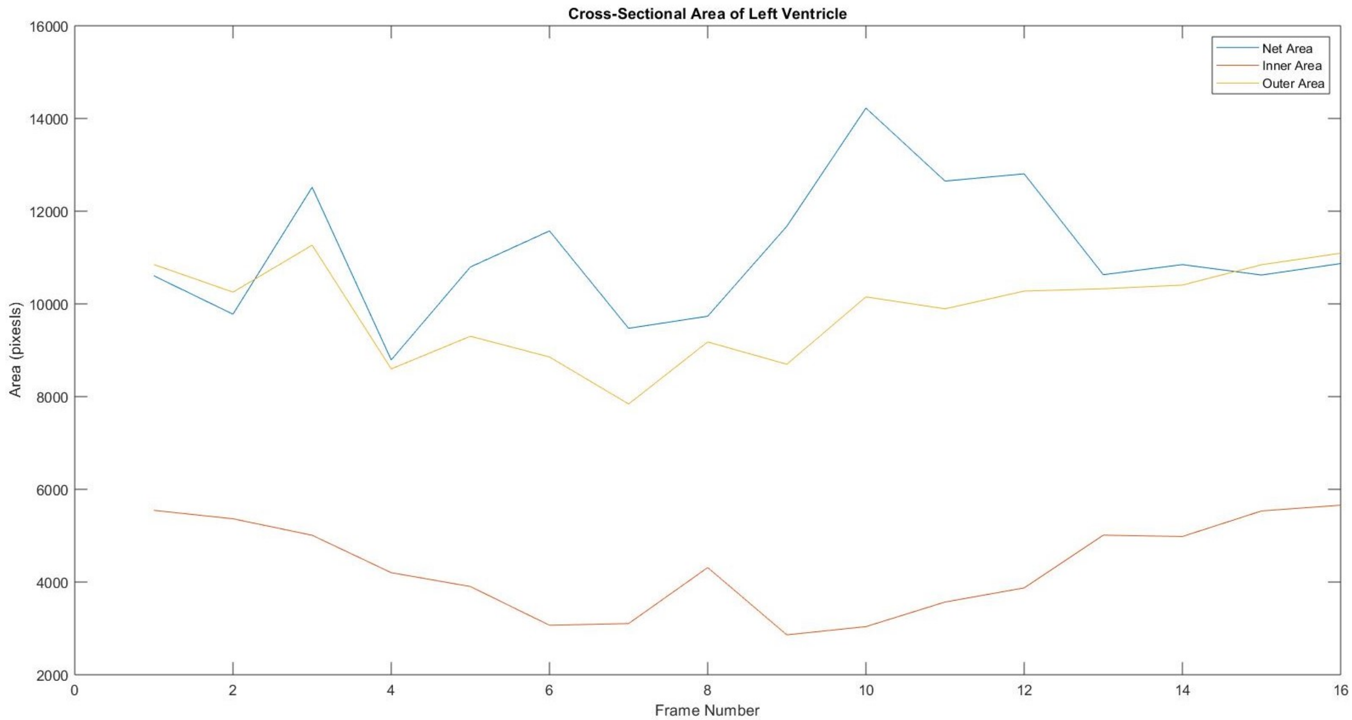


Figure 18: Cross-Sectional Area of Inner & Outer Walls, and Net Area

Throughout the process, there are several alternative approaches that could also be considered:

- **Heart Mask:** Circular Hough transform, template matching
- **Centroid Localisation:** Centre of circular Hough transform, equivalent radius of component along major axis
- **Polar Coordinates:** Cartesian coordinates with a circular traversal as opposed to a linear traversal through polar coordinates, an annulus of radii values rather than radii values from the origin to the maximum radius
- **Starting Point Localisation:** Gradient thresholding along 0^o spoke, edge detection
- **Minimum Cost Traversal:** Djikstra search algorithm, Heuristic Search
- **Ventricle Area Calculation:** Convex Hull, pixel count contained in mask

4 Appendix A: Code Listings

4.1 Taj Mahal and the Hough Transform

Listing 14: Hough Transform Implementation

```
function [hough, theta, rho] = myHough(BW)
    [numRows, numCols] = size(BW);

    %Calculate theta, rho ranges
    theta = linspace(-90, 89, 180);
    D = sqrt((numRows - 1)^2 + (numCols - 1)^2);
    nrho = 2*ceil(D) + 1;
    rho = linspace(-ceil(D), ceil(D), nrho);

    %Initialise discretised hough space
    hough = double(zeros(length(rho), length(theta)));

    %Find white pixels in image
    [y, x] = find(BW);

    %Transform image to parameter space
    for index = 1:numel(x)
        calculatedRho = int16(x(index) * cos(pi * theta / 180) + y(index)
             * sin(pi * theta / 180));
        hough = hough + double(transpose(rho) == calculatedRho);
    end
end
```

Listing 15: Hough Peaks Implementation

```
function [P] = myHoughPeaks(hough, numPeaks, proximity)
    P = zeros(numPeaks, 2);

    for index = 1:numPeaks
        %Find first peak
        [row, col] = find(hough == max(max(hough)));
        P(index, :) = [row(1) col(1)];

        %Set cells in proximity to zero
        distance = round(proximity / 2);
        hough(max(1,row(1)-distance):min(size(hough,1),row(1)+distance),
            ...
            max(1,col(1)-distance):min(size(hough,2),col(1)+distance))
                = 0;
    end
end
```

Listing 16: Hough Lines Implementation

```
function [lines] = myHoughLines(bw, theta, rho, peaks)
```

```
    lines = zeros(size(peaks, 1), 4);

    for index = 1:size(peaks, 1)
        %Get peak theta, rho
        T = theta(peaks(index,2));
        R = rho(peaks(index, 1));

        %Convert polar to cartesian
        xs = R .* cos(pi .* (T) ./ 180);
        ys = R .* sin(pi .* (T) ./ 180);
        m = tan(pi.*(T + 90) ./ 180);
        c = ys - (m .* xs);

        %Calculate line points
        xs = 1:size(bw,2);
        ys = round(c + (m * xs));
        xs = xs((ys >= 1) & (ys <= size(bw, 1)));
        ys = ys((ys >= 1) & (ys <= size(bw, 1)));

        %Find longest line in points
        whites = bw(sub2ind(size(bw), ys, xs));
        start = find(whites, 1, 'first');
        finish = start;
        while whites(finish)
            finish = finish + 1;
        end

        lines(index,:) = [xs(start) ys(start) xs(finish - 1) ys(finish -
           1)];
    end
end
```

Listing 17: Taj Mahal Processing Script

```
%%%%%%Load Image%%%%%%
image = imread('TajMahal.jpg');
image = imresize(image, 0.5);
imshow(image);

%%%%%%Separate Background and Foreground%%%%%%
grayImage = rgb2gray(image);
imshow(grayImage);

bwImage = imbinarize(grayImage);
imshow(bwImage);

%Remove all but largest component
largestComponent = bwImage;
stats = regionprops('table', bwImage, 'Area', 'PixelIdxList');
[sortedAreas, sortingIndex] = sort(stats.Area, 'descend');

```

```
for index = 2:length(sortingIndex)
    largestComponent(stats.PixelIdxList{sortingIndex(index)}) = 0;
end

%Find convex hull
minaretsMask = bwconvhull(largestComponent);
waterMask = ~minaretsMask;

%Apply Masks
minaretsImage = image .* uint8(repmat(minaretsMask,[1,1,3]));
waterImage = image .* uint8(repmat(waterMask,[1,1,3]));
imshow(minaretsImage);
imshow(waterImage);

%%%%%%Minaret Processing%%%%%%
%Threshold minarets
bwMinaret = minaretThresholder(minaretsImage);
imshow(bwMinaret);

%Remove noise
minarets = bwMinaret;
stats = regionprops('table', minarets, 'Area', 'PixelIdxList');
[sortedAreas, sortingIndex] = sort(stats.Area, 'descend');

for index = 2:length(sortingIndex)
    minarets(stats.PixelIdxList{sortingIndex(index)}) = 0;
end

%Fill Image
filled = imfill(minarets, 'holes');
imshow(filled);

%Close Image
closed = imclose(filled, strel('line', 10, 90));
imshow(closed);

%Erode image
eroded = imerode(closed, strel('line', 10, 90));
imshow(eroded);

%Outline Image
gradient = imgradient(eroded);
imshow(gradient);

%Hough Transform
[H,T,R] = hough(gradient);
H1 = H(:, find(T==70-90):find(T==89-90));
H2 = H(:, find(T==91-90):find(T==110-90));

T1 = T(find(T==70-90):find(T==89-90));
```

```
T2 = T(find(T==91-90):find(T==110-90));

P1  = houghpeaks(H1,5, 'Threshold', 0.3*max(H1(:)));
P2  = houghpeaks(H2,5, 'Threshold', 0.3*max(H1(:)));
L1 = houghlines(eroded,T1,R,P1,'FillGap',5,'MinLength', 15);
L2 = houghlines(eroded,T2,R,P2,'FillGap',5,'MinLength', 15);
myLines = [L1 L2];

%Group Hough Lines by rho
groupedLines = groupLines(myLines, 6, 3);

%Select left/right most four lines respectively
[sortedX, sortingIndex] = sort(groupedLines(:,1), 'ascend');
leftLines = groupedLines(sortingIndex(1:4), :);
rightLines = groupedLines(sortingIndex(size(groupedLines, 1) - 3:size(
    groupedLines, 1)), :);
minaretLines = [leftLines; rightLines];

plotted = insertShape(image, 'line', minaretLines, 'LineWidth',6,'Color',
    'red');
imshow(plotted);

%Calculate angles
angles = zeros(size(minaretLines, 1), 1);
for index = 1:size(minaretLines, 1)
    angle = atand((minaretLines(index, 4) - minaretLines(index, 2)) ./
        ...
                  (minaretLines(index, 3) - minaretLines(index, 1)));
    if(angle >= 0)
        angles(index) = angle;
    else
        angles(index) = angle + 180;
    end
end

angles

%%%%%%Water Feature Processing%%%%%%
% Threshold in LAB domain
bwWater = waterThresholder(waterImage);
imshow(bwWater);

%Denoise image
denoised = bwareaopen(bwWater, 1000);
imshow(denoised);

%Close Image
closed = imclose(denoised, strel('line', 5, 90));
imshow(closed);

```

```
%Outline Image
gradient = imgradient(closed);
imshow(gradient);

%Hough Transform
[H,T,R] = hough(gradient);
H1 = H(:, find(T==140-90):find(T==160-90)); %-140 to -160 degrees
H2 = H(:, find(T==93-90):find(T==100-90));  %-93  to -100 degrees
H3 = H(:, find(T==70-90):find(T==80-90));   %-70  to -80  degrees

T1 = T(find(T==140-90):find(T==160-90));
T2 = T(find(T==93-90):find(T==100-90));
T3 = T(find(T==70-90):find(T==80-90));

P1 = houghpeaks(H1,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H1(:)));
P2 = houghpeaks(H2,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H2(:)));
P3 = houghpeaks(H3,5, 'NHoodSize', [51, 1], 'Threshold', 0.3*max(H3(:)));
L1 = houghlines(closed,T1,R,P1,'FillGap',5,'MinLength', 15);
L2 = houghlines(closed,T2,R,P2,'FillGap',5,'MinLength', 15);
L3 = houghlines(closed,T3,R,P3,'FillGap',5,'MinLength', 15);
myLines = [L1 L2 L3];

%Group lines by theta
waterLines = groupLines(myLines, 5, 5);
plotted = insertShape(image, 'line', waterLines, 'LineWidth',6,'Color','red');
imshow(plotted);

%Combined
combinedPlot = insertShape(image, 'line', [minaretLines; waterLines], 'LineWidth', 6, 'Color', 'red');
imshow(combinedPlot);
```

Listing 18: Group Hough Lines on Arbitary Line Attribute

```
function [resultLines] = groupLines(houghLines, attribute, attributeThreshold)
     groupLines = zeros(length(houghLines), 6);

    for index = 1:length(groupLines)
        groupLines(index,:) = [houghLines(index).point1 houghLines(index).point2 houghLines(index).theta houghLines(index).rho];
    end

    %Sort lines by theta
    [sortedThetas, sortingIndex] = sort(groupLines(:, attribute), 'ascend');
    sortedLines = groupLines(sortingIndex, :);

    %Group close thetas together
    linesProcessed = 0;
```

```
    index = 0;
    while(linesProcessed < length(groupLines))
        groupedLines = sortedLines(abs(sortedLines(:, attribute) -
           sortedLines(linesProcessed + 1, attribute)) <
           attributeThreshold, :);
        groupedLine = [min([groupedLines(:,1); groupedLines(:,3)]) ...
                                            min([groupedLines(:,2);
                                               groupedLines(:,4)]) ...
                                            max([groupedLines(:,1);
                                               groupedLines(:,3)]) ...
                                            max([groupedLines(:,2);
                                               groupedLines(:,4)])];
        if groupedLines(1, 5) > 0
            groupedLine = [groupedLine(3) groupedLine(2) groupedLine(1)
               groupedLine(4)];
        end
        index = index + 1;
        resultLines(index,:) = groupedLine;
        linesProcessed = linesProcessed + size(groupedLines, 1);
    end
end
```

Listing 19: Minarets Thresholder

```
function [BW,maskedRGBImage] = minaretThresholder(RGB)
% Convert RGB image to chosen color space
I = rgb2lab(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 40.183;
channel1Max = 99.829;

% Define thresholds for channel 2 based on histogram settings
channel2Min = -13.270;
channel2Max = 32.516;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 2.835;
channel3Max = 35.441;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;
```

```

end
```

Listing 20: Water Feature Thresholder

```
function [BW,maskedRGBImage] = waterThresholder(RGB)
% Convert RGB image to chosen color space
I = rgb2lab(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 46.949;
channel1Max = 76.291;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 2.285;
channel2Max = 60.768;

% Define thresholds for channel 3 based on histogram settings
channel3Min = -6.081;
channel3Max = 28.126;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;

end
```

Listing 21: K means Image Quantization

```
function [kImage, centroids, indexes] = kmeansImage(image, k)
    [numRows, numCols, three] = size(image);
    r = image(:,:,1);
    g = image(:,:,2);
    b = image(:,:,3);
    x = [r(:) g(:) b(:)];

    [classifications, centroids] = kmeans(double(x), k);

    indexes = reshape(classifications, numRows, numCols);
    kImage = ind2rgb(indexes, centroids ./ 256);

    centroids = uint8(centroids);
    kImage = uint8(kImage .* 256);
```

```
end
```

4.2 Cost Minimisation in the Viterbi Trellis Search Space for Contouring Heart MRIs

Listing 22: Heart Segmentation Class

```
classdef HeartSegmenter
    properties
        %Constants
        theBinaryThreshold = 0.15;
        theMaxRadius = 100;
        theDefaultThetaSpacing = 0.5;
        theDefaultNumRadii = 200;
        theDefaultInnerLambda = 1.475; %1.95;
        theDefaultOuterLambda = 1.3 % 3.1;
        theDefaultRadiiThreshold = 2;
        theBlurrerSize = 7;

        theMinCircularity = 0.7;
        theMaxCircularity = 1.3;

        %Additional metrics
        theCircularityCosts;
        theGradientCosts;
        theWeightedCosts;
        theMomentumCosts;

        theOriginalScan;
        theScan;
        theGradMags;
        theGradDirs;
        theMaxGradMag;

    end

    methods
        function self = HeartSegmenter(scan)
            self.theOriginalScan = scan;

            kernel = ones(self.theBlurrerSize)./(self.theBlurrerSize.^2);
            self.theScan = uint8(conv2(scan, kernel, 'same'));

            [self.theGradMags, self.theGradDirs] = imgradient(scan);
            self.theGradDirs(self.theGradDirs < 0) = self.theGradDirs(
               self.theGradDirs < 0) + 180;
            self.theMaxGradMag = max(max(self.theGradMags));
        end

```

```
        function mask = getHeartMask(self)
            mask = imbinarize(self.theScan, self.theBinaryThreshold);

            %Threshold on circularity
            stats = regionprops('table', mask, 'PixelIdxList', 'MajorAxisLength', 'EquivDiameter');
            circularity = stats.MajorAxisLength ./ stats.EquivDiameter;

            for index = 1:length(circularity)
                if circularity(index) < self.theMinCircularity || ...
                   circularity(index) > self.theMaxCircularity
                    mask(stats.PixelIdxList{index}) = 0;
                end
            end

            %Redefine components in image and their size
            connectedComponents = bwconncomp(mask);
            numPixels = cellfun(@numel,connectedComponents.PixelIdxList);

            [largestNum, idx] = max(numPixels);

            %Remove all but largest component
            for index = 1:length(connectedComponents.PixelIdxList)
                if index ~= idx
                    mask(connectedComponents.PixelIdxList{index}) = 0;
                end
            end

            %Convert to conxex hull
            mask = bwconvhull(mask);
        end

        function [centroid, diameter] = getCentroid(self, mask)
            stats = regionprops('table', mask, 'centroid', 'EquivDiameter');
            centroid = stats.Centroid;
            diameter = stats.EquivDiameter;
        end

        function [radii, thetas, xs, ys] = getPolarCoordinates(self, centroid, numRadii, thetaSpacing)
            if nargin < 4
                thetaSpacing = self.theDefaultThetaSpacing;
            end
            if nargin < 3
                numRadii = self.theDefaultNumRadii;
            end

            radii = double(linspace(1, self.theMaxRadius, numRadii));
            thetas = 0:thetaSpacing:359;
```

```
            xs = round(centroid(1) + transpose(radii) * cos(pi*thetas/180));
            ys = round(centroid(2) - transpose(radii) * sin(pi*thetas/180));
        end

        function [startingRadii] = getStartingRadii(self, xs, ys)
            theStartingMags = self.theGradMags(sub2ind(size(self.theGradMags), ys(:,1), xs(:,1)));
            [peakValues, peakIndexes] = findpeaks(theStartingMags);

            [sortedVals, sortedIndex] = sort(peakValues, 'descend');
            sortedPeakIndexes = peakIndexes(sortedIndex);

            startingRadii = xs(sortedPeakIndexes(1:2), 1);
        end

        function [minCartesianPath, minRadiiPath] = findMinimumPath(self, centroid, startingRadius, lambda, radiiThreshold)
            if nargin < 6
                radiiThreshold = self.theDefaultRadiiThreshold;
            end
            if nargin < 5
                lambda = self.theDefaultLambda;
            end

            %Initialise polar coordinate system
            [radii, thetas, xVals, yVals] = self.getPolarCoordinates(centroid);

            %Pre-allocate memory for minimum radius path
            minRadiiPath = zeros(length(thetas), 1);

            %Find starting point
            [value, index] = min(abs(xVals(:,1) - startingRadius));
            minRadiiPath(1) = index;

            %Preallocate memory for metric tracking
            self.theCircularityCosts = zeros(length(thetas), 1);
            self.theGradientCosts = zeros(length(thetas), 1);
            self.theWeightedCosts = zeros(length(thetas), 1);
            self.theMomentumCosts = zeros(length(thetas), 1);

            minRadiiPath(2) = minRadiiPath(1);
            %Traverse around
            for t = 3:length(thetas)
                %Calculate costs
                circularityCost = transpose(abs(radii - mean(radii(minRadiiPath(1:t-1))))) / max(radii);
```

```
                momentumCost = transpose(abs(2.*radii - radii(minRadiiPath(t-1)) - radii(minRadiiPath(t-2)))) / max(radii);
                gradientCost = self.theGradMags(sub2ind(size(self.theGradMags), yVals(:,t), xVals(:, t))) ./ self.theMaxGradMag;
                weightedCost = lambda.*(momentumCost + (t / length(thetas) .* circularityCost)) + (1 - gradientCost);

                %Choose minimum cost radii
                [value, index] = min(weightedCost(max(minRadiiPath(t-1) - radiiThreshold, 1): ...
                                                  min(minRadiiPath(t-1) + radiiThreshold, length(radii))));
                index = index(1);
                minRadiiPath(t) = minRadiiPath(t-1) - radiiThreshold + index - 1;
                minRadiiPath(t) = min(max(minRadiiPath(t), 1), length(radii));

                %Record metrics
                self.theCircularityCosts(t-1) = circularityCost(index);
                self.theMomentumCosts(t-1) = momentumCost(index);
                self.theGradientCosts(t-1) = gradientCost(index);
                self.theWeightedCosts(t-1) = weightedCost(index);
            end

            %Convert polar to cartesian
            minCartesianPath = [xVals(sub2ind(size(xVals), minRadiiPath, transpose(1:length(thetas)))) ...
                                yVals(sub2ind(size(yVals), minRadiiPath, transpose(1:length(thetas))))];
        end

        function [innerPath, outerPath] = getOutlines(self, innerLambda, outerLambda, radiiThreshold)
            if nargin < 4
                radiiThreshold = self.theDefaultRadiiThreshold;
            end
            if nargin < 3
                outerLambda = self.theDefaultOuterLambda;
            end
            if nargin < 2
                innerLambda = self.theDefaultInnerLambda;
            end

            %Initial Segmentation
            mask = self.getHeartMask();

```

```
            %Find Centroid
            [centroid, diameter] = self.getCentroid(mask);

            %Establish Polar Coordinates
            [radii, thetas, xs, ys] = self.getPolarCoordinates(centroid);

            %Find starting points
            startingRadii = self.getStartingRadii(xs, ys);
            innerRadius = min(startingRadii);
            outerRadius = max(startingRadii) - 5;

            %Minimum Cost Traversal
            [innerPath, innerRadii] = self.findMinimumPath(centroid,
               innerRadius, innerLambda, radiiThreshold);
            [outerPath, outerRadii] = self.findMinimumPath(centroid,
               outerRadius, outerLambda, radiiThreshold);
        end

        function [area, innerArea, outerArea] = getArea(self, innerPath,
           outerPath)
            [innerBoundary, innerArea] = boundary(innerPath);
            [outerBoundary, outerArea] = boundary(outerPath);
            area = outerArea - innerArea;
        end
    end
end
```