ELEC4630 Assignment 3

Samuel Eadie - 44353607

May 13, 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 K-means Clustering

K-means clustering is an unsupervised machine learning technique used to group data into K clusters in N-dimensional space. It attempts to place K centroids in the data set, these are initialised as a random point in the data. It then iteratively finds the true location of these centroids by:

1. Associate each point in the data set with its closest centroid
2. Move each centroid to the mean of all data points associated with it

K-means clustering can be used for image quantisation (each RGB pixel corresponds to a point in three dimensional space) and grayscale binarisation (each pixel corresponds to a point in one dimensional space). K-means clustering not only yields the final association for each data point, but also the centroid value for each cluster.

2 HEP2 Segmentation

2.1 Introduction

Segmentation was performed on ANA IFF HEP2 cell images. These were either borderline or high positive sample images. Following this, a suitable performance metric was devised.

2.2 Method

The method involved:

1. Binarising the image through one-dimensional k-means clustering
2. Morphological filling of cells

The pixel-wise XOR between the segmented image and a comparison image was used as a performance metric. This metric provides a percent error between these two images.

The method is explicated in the following Listing and shown in Figure 1:

Listing 1: HEP2 Segmentation Procedure

```
figure();

%Load the images
difficulty = 'easy';
fileName = 'hi';
image = imread(sprintf('HEP-2 segmentation/%s/%s_FITC.tif', difficulty,
    fileName));
comparisonImage = imread(sprintf('HEP-2 segmentation/%s/%s_Mask.tif',
    difficulty, fileName));
subplot(3,2,1); imshow(image); title('Original Image');
subplot(3,2,2); imshow(comparisonImage); title('Comparison Image');

%Perform 1D k-means clustering
[indexes, centroids] = bwKmeansImage(image, 2);
subplot(3,2,3); imshow(indexes); title('K-means clustering');

%Evaluate intermediary performance
kmeansDifference = xor(indexes, comparisonImage == 255);
kmeansError = 100 * sum(kmeansDifference(:)) / (size(kmeansDifference, 1)
    * size(kmeansDifference, 2))
subplot(3,2,4); imshowpair(indexes, comparisonImage); title('K-means
    comparison');

%Fill cells
filled = imfill(indexes, 'holes');
subplot(3,2,5); imshow(filled); title('Filled image');

%Evaluate final performance
filledDifference = xor(filled, comparisonImage == 255);
filledError = 100 * sum(filledDifference(:)) / (size(filledDifference, 1)
    * size(filledDifference, 2))
```

```
subplot(3,2,6); imshowpair(filled, comparisonImage); title('Filled image
    comparison');
```

The K-means clustering algorithm was adapted for binarization of grey scale images, as shown in the following listing:

Listing 2: K-means clustering for grey scale binarisation

```
function [indexes, centroids] = bwKmeansImage(image, k)
    [numRows, numCols] = size(image);
    [classifications, centroids] = kmeans(double(image(:)), k);

    indexes = reshape(classifications, numRows, numCols) - 1;
    centroids = uint8(centroids);

    if(centroids(1) > centroids(2))
        indexes = ~indexes;
    end
end
```

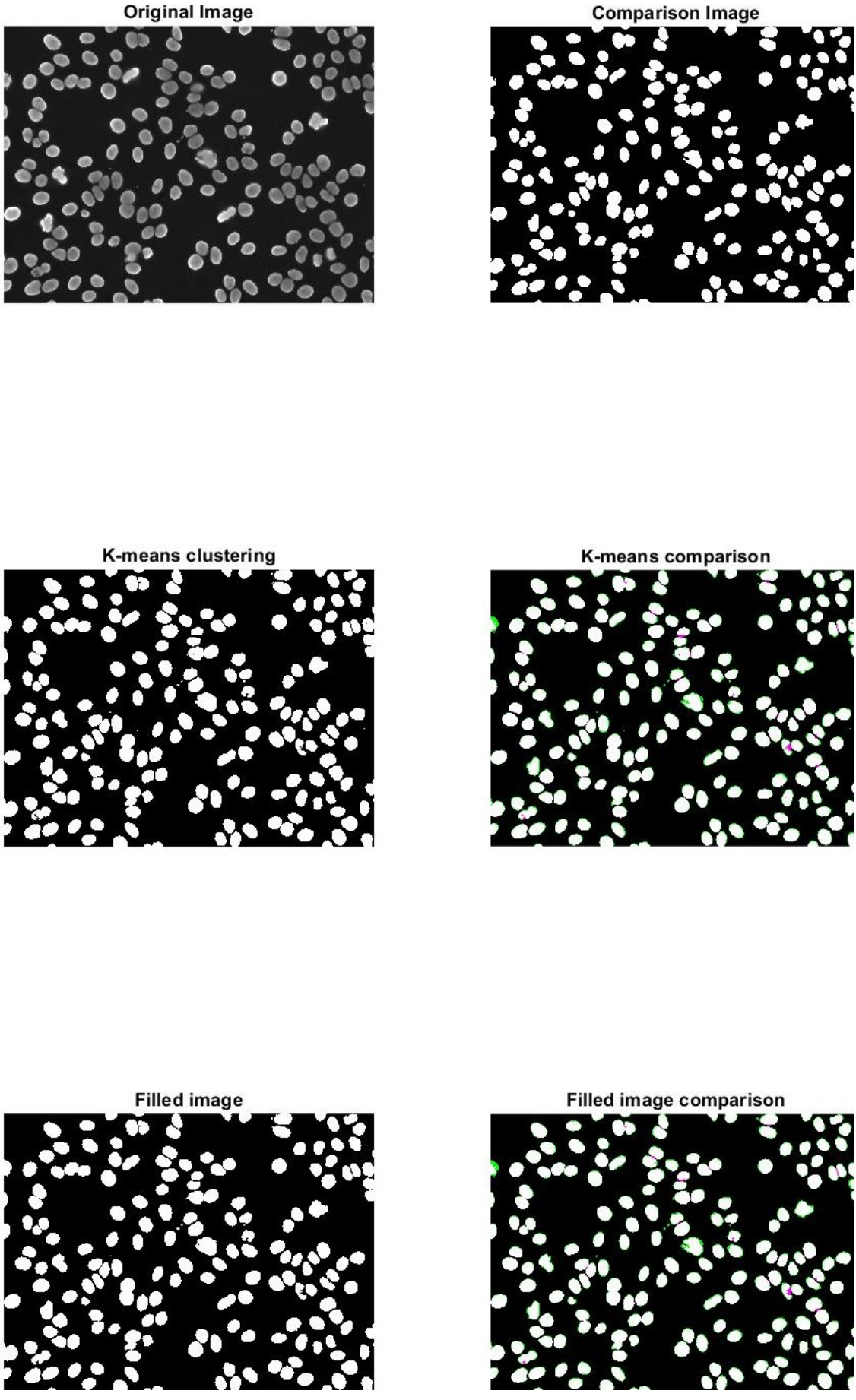

Figure 1: HEP2 Segmentation Process

2.3 Results

The HEP2 segmentation of the *easy* images resulted in errors of 2.697% and 1.389% respectively. These segmentations are overlain with the comparison images for reference in Figure 2. As an extension, this segmentation process was trialled on the *hard* and *extreme* scans, and these results are shown in Figure 3.

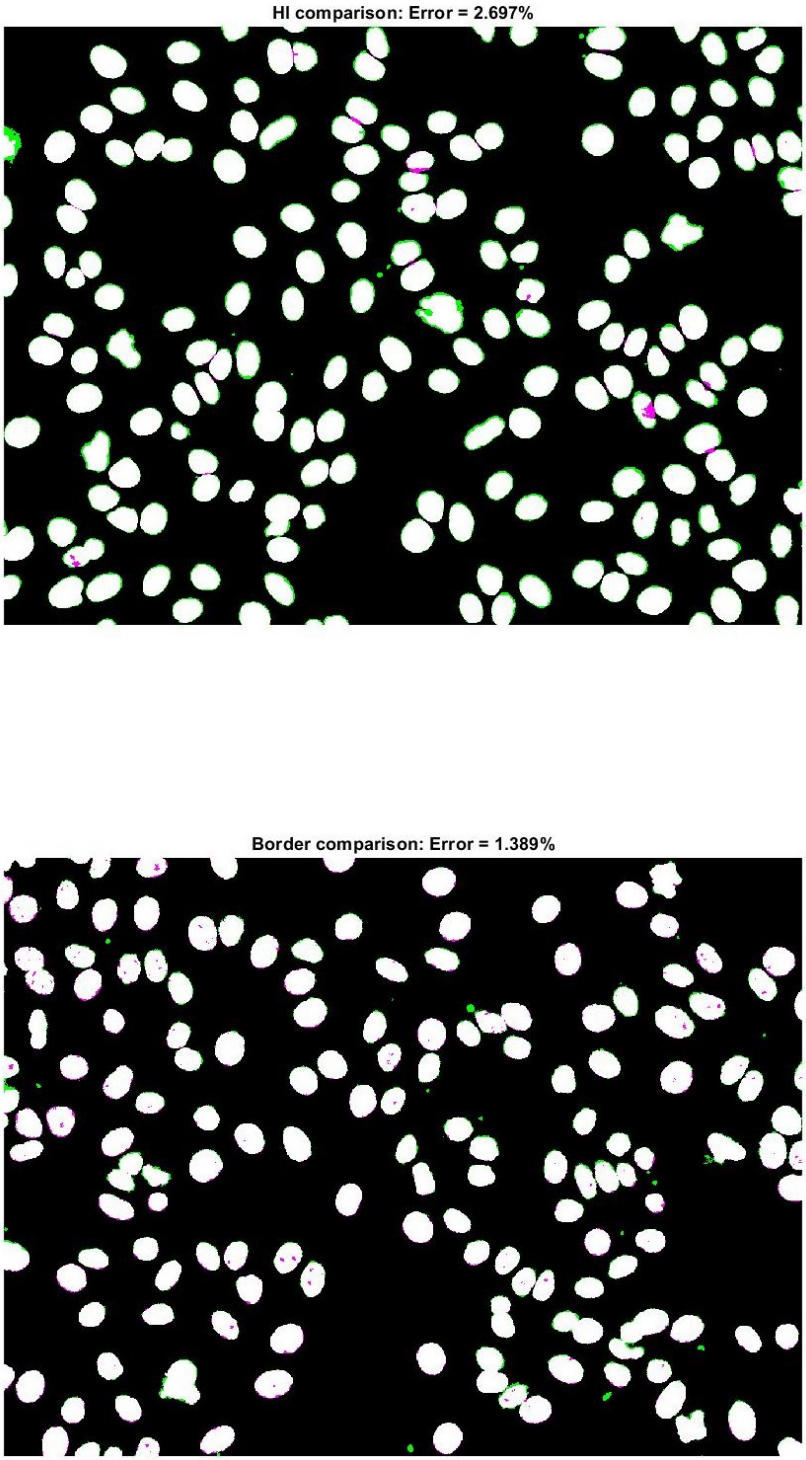

Figure 2: HEP2 Comparisons

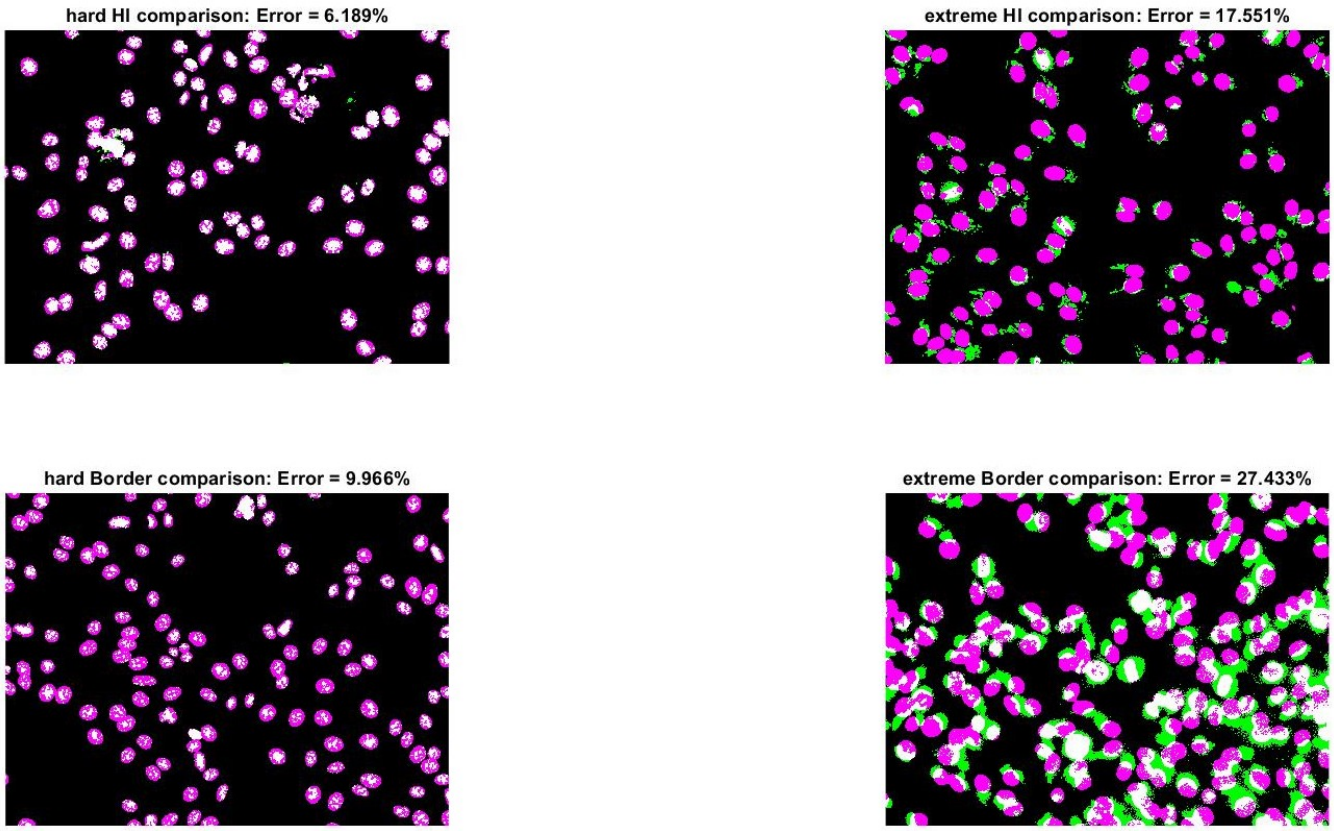

Figure 3: HEP2 Extension Comparisons

2.4 Discussion

While the devised algorithm worked on *easy* images, it struggled on the *hard* and *extreme* cases. A poor segmentation attempt on a *hard* image is shown in Figure 4.

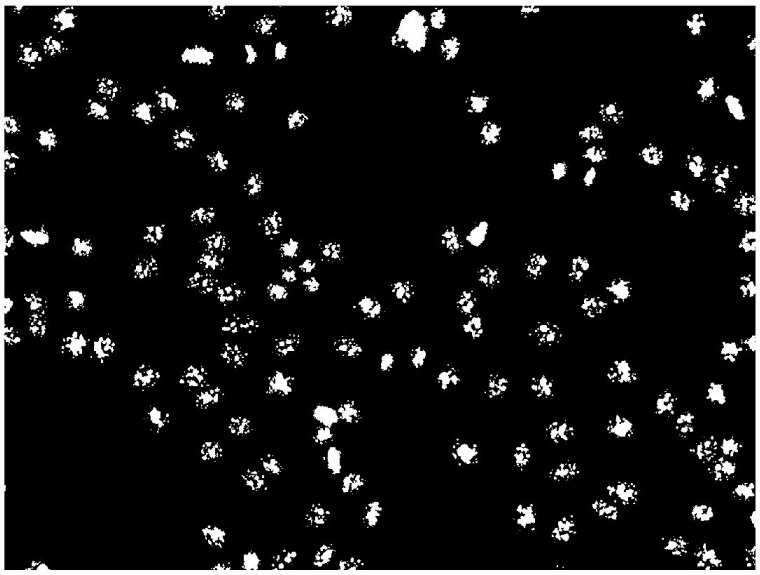

Figure 4: HEP2 *Hard* Example Binarisation

2.4.1 Alternative Methods

Alternative methods were also trialled, most notably, a histogram-based binarisation technique. This method selects a binary threshold by finding the minimum histogram value between the two highest histogram peak values. The peak values were assumed to represent the foreground and background respectively, and hence the minimum value between these was selected as a threshold. This is shown in Figure 5. This method was less accurate than K-means and hence not used, and had problems with *hard* and *extreme* cases where the grayscale histogram did not present this dual-peak structure (Figure 6). This method is explicated in the below listing:

Listing 3: Alternative histogram-based binarisation method

```
%Load the image
difficulty = 'easy';
fileName = 'hi';
image = imread(sprintf('HEP-2 segmentation/%s/%s_FITC.tif', difficulty,
    fileName));
comparisonImage = imread(sprintf('HEP-2 segmentation/%s/%s_Mask.tif',
    difficulty, fileName));
figure();
subplot(2,2,1); imshow(image); title('Original Image');

%Create grayscale histogram
subplot(2,2,2); histogram(image(:)); title('Grey Scale Histogram');
[N,edges] = histcounts(image(:));

%Find two highest peaks
[values,locs] = findpeaks(N);
[sortedValues, sortingIndex] = sort(values, 'descend');
sortedLocs = locs(sortingIndex);
```

```
topTwo = sortedLocs(1:2);

%Find minimum value between peaks
[minValue, minLocation] = min(N(min(topTwo):max(topTwo)));
minIndex = min(topTwo) + minLocation;
threshold = edges(minIndex);
peaks = edges(topTwo);

hold on; plot(peaks, N(topTwo), 'r*', 'MarkerSize', 18); plot(threshold,
   N(minIndex), 'g*', 'MarkerSize', 18); hold off;

%Binarize Image
histSegmented = image > threshold;
subplot(2,2,3); imshow(histSegmented); title('Thresholded Image');

%Compute performance metrics and comparison
histDifference = xor(histSegmented, comparisonImage == 255);
errorPercentage = 100 * sum(histDifference(:)) / (size(histDifference, 1)
    * size(histDifference, 2));
subplot(2,2,4); imshowpair(histSegmented, comparisonImage); title(sprintf
   ('Comparison Image: Error = %1.3f%%', errorPercentage));
```

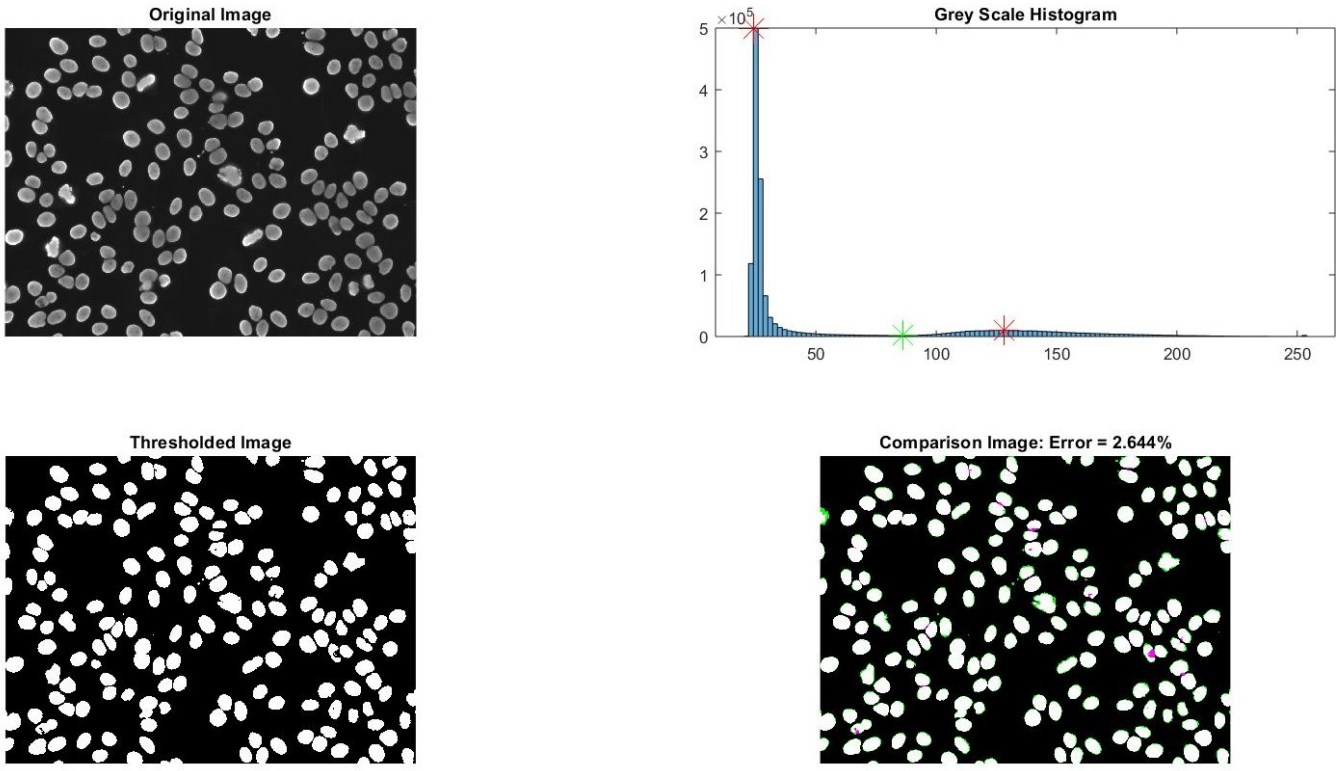

Figure 5: HEP2 Alternative Histogram Method

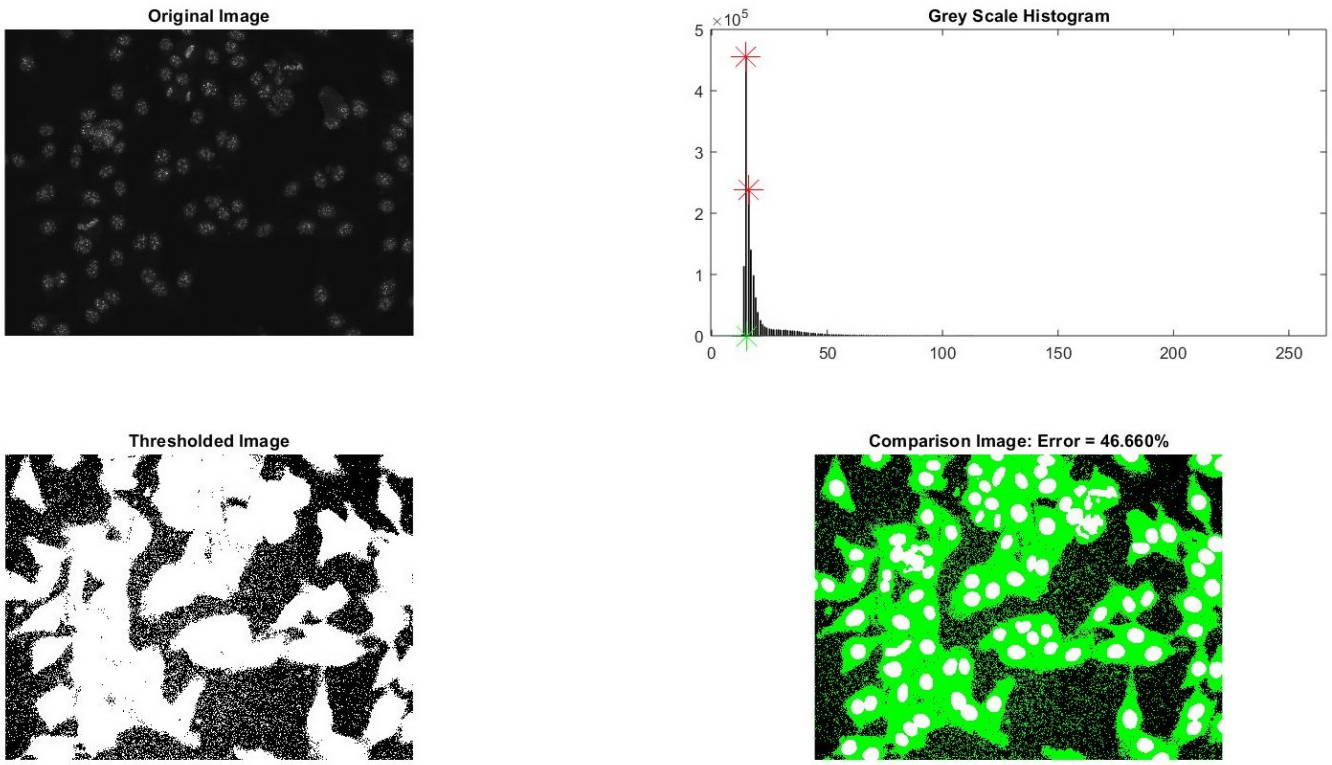

Figure 6: HEP2 Alternative Histogram Method on a *Hard* Case

2.4.2 Further Considerations

Possible avenues for further development of the binarisation algorithm includes the exploitation of:

- **Spatial properties:** a centroid could be calculated for each spatial group of white components and an ellipse estimated using the components in each group
- **Local colour value properties:** the use of colour values in neighbourhood regions could used since currently only global values characteristics are used

3 String Length Determination

3.1 Introduction

An algorithm to determine the lengths of strings was developed.

3.2 Method

The algorithm to determine the length of a string consisted of:

1. **Conversion to grayscale:** The image was converted from RGB space to grayscale
2. **Binarisation through an adaptive threshold:** A locally-adaptive threshold was used to binarise the image because of the inconsistent lighting in these images. The locally-adaptive threshold computes the mean intensity in the neighbourhood around each pixel and uses this for thresholding the pixel. The neighbourhood used was rectangular, and an eighth of the image size.
3. **Removing boundary objects:** Objects around the boundary of the image were removed since the string was generally centred in these images.
4. **Morphological skeletonisation:** The remaining string object was skeletonised so as to be one pixel wide. This ensures a relatively linear relationship between the number of pixels and the string length.
5. **Pixel enumeration:** The number of pixels in the string's skeleton was calculated.
6. **Conversion from pixels to distance:** Using a conversion factor from labelled string data, the string skeleton pixel count was converted to a physical distance.

This process is shown in Figure 7 and explicated in the listing below:

Listing 4: String length determination algorithm

```
%Load image
image = imread('string/String3_1.jpg');
grayImage = rgb2gray(image);
bw = ~imbinarize(grayImage, adaptthresh(grayImage, 0.70));

%Remove boundary objects
boundaryThreshold = 10;
boundaryRemoved = bw;

stats = regionprops('table', boundaryRemoved, 'Centroid', 'PixelIdxList')
    ;
for component = 1:size(stats, 1)
    if(stats.Centroid(component, 1) < boundaryThreshold || stats.Centroid
       (component, 1) > size(bw, 1) - boundaryThreshold || ...
       stats.Centroid(component, 2) < boundaryThreshold || stats.Centroid
          (component, 2) > size(bw, 2) - boundaryThreshold)
            boundaryRemoved(stats.PixelIdxList{component}) = 0;
    end
```

```
end

%Skeletonise Image
skeleton = bwskel(boundaryRemoved);

%Enumerate pixel count
numPixels = sum(skeleton(:));

%Convert pixel count to distance
PIXEL_TO_LENGTH_CONVERSION = 8.0670;
cmDistance = numPixels / PIXEL_TO_LENGTH_CONVERSION;

subplot(2,3,1);imshow(image);title('Original Image');
subplot(2,3,2);imshow(grayImage);title('Grayscale Image');
subplot(2,3,3);imshow(bw);title('Binarized Image');
subplot(2,3,4);imshow(boundaryRemoved);title('Remove Boundary Objects');
subplot(2,3,5);imshow(skeleton);title('Skeletonised Image');
```

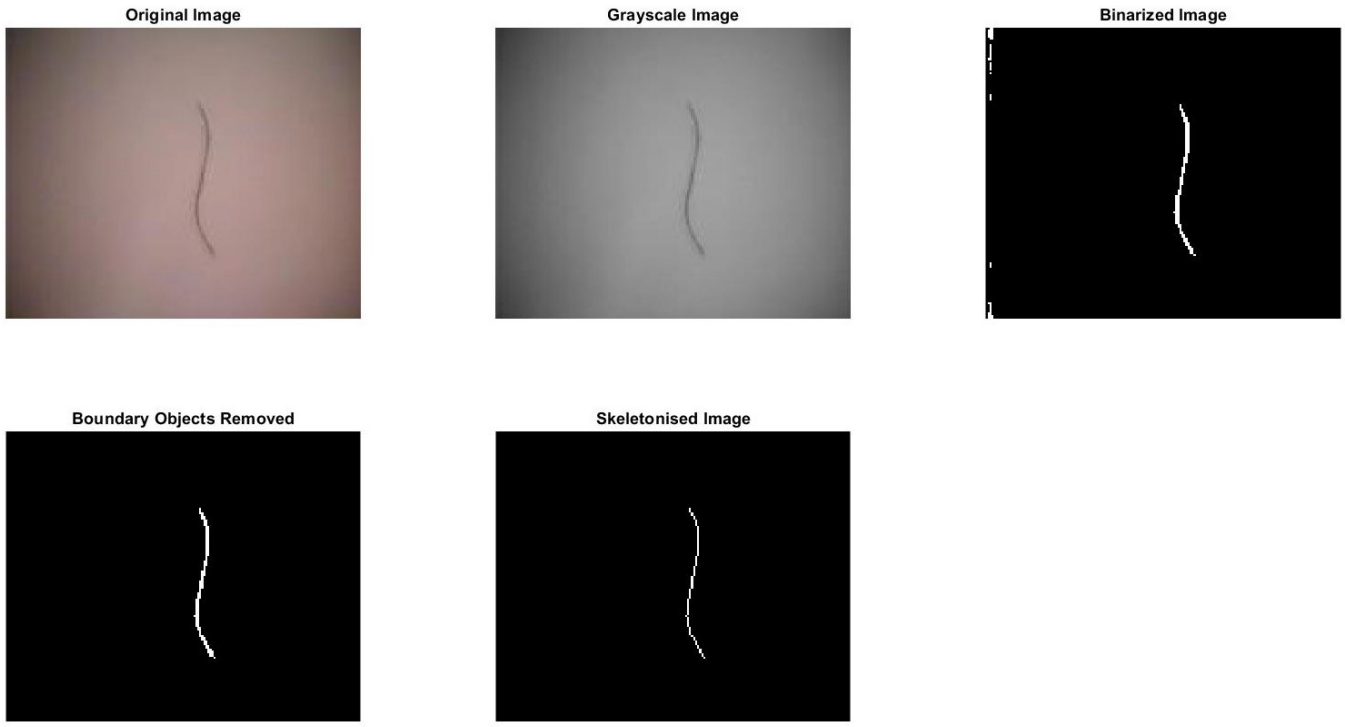

Figure 7: String Length Determination Method

3.3 Results

This algorithm was run on all images of strings 1 and 2 to calculate a pixel count to distance conversion factor (Figure 8). The ten calculated conversion factors were [8.0000, 8.3846, 7.3077, 8.6154, 7.8462, 7.8710, 8.9032, 7.4839, 7.8065, 8.4516]. This gives a mean conversion factor of $8.0670 px/cm$. This conversion factor was used to calculate the length of the third string in the given images (Figure 9). The calculated string length for these images were [9.2971, 7.5617, 8.4294, 8.4294, 7.4377]. This gives a mean length of $8.231 cm$ for the third string.

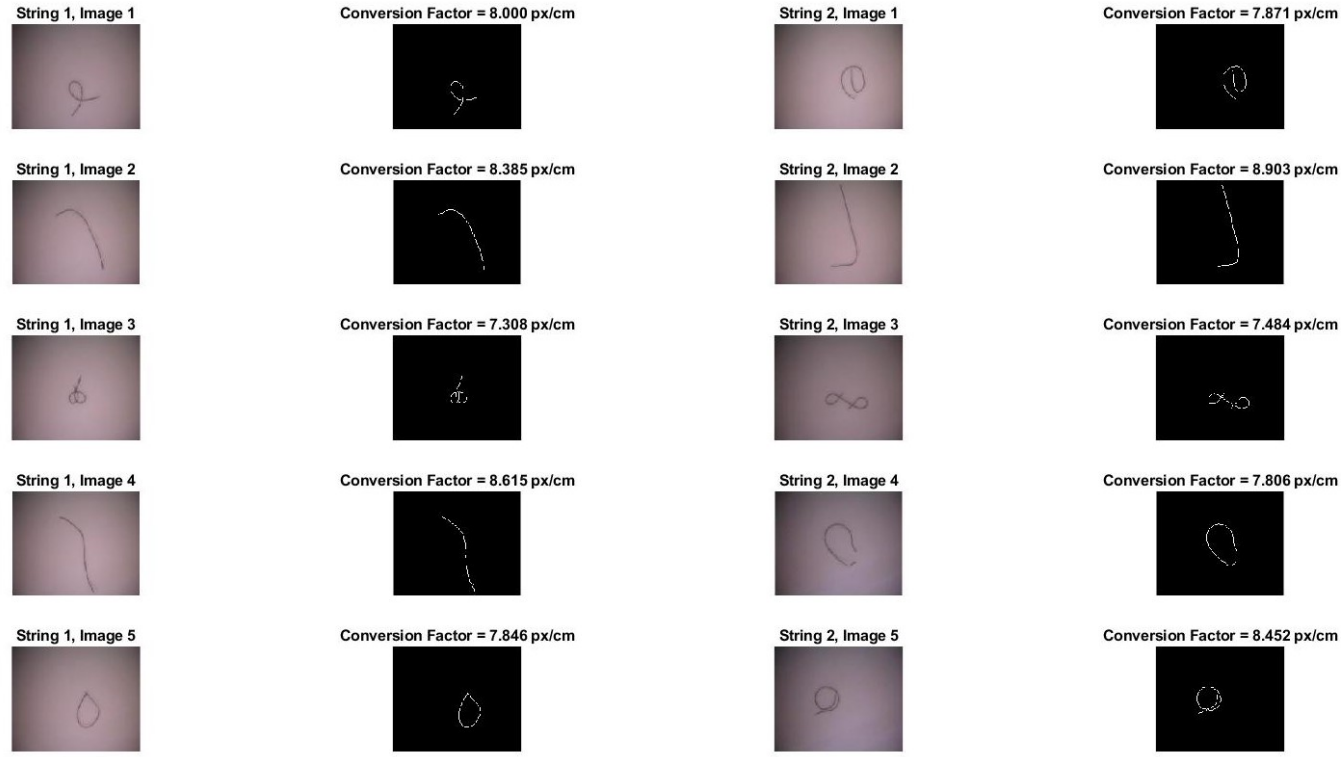

Figure 8: String 1 and 2 Results

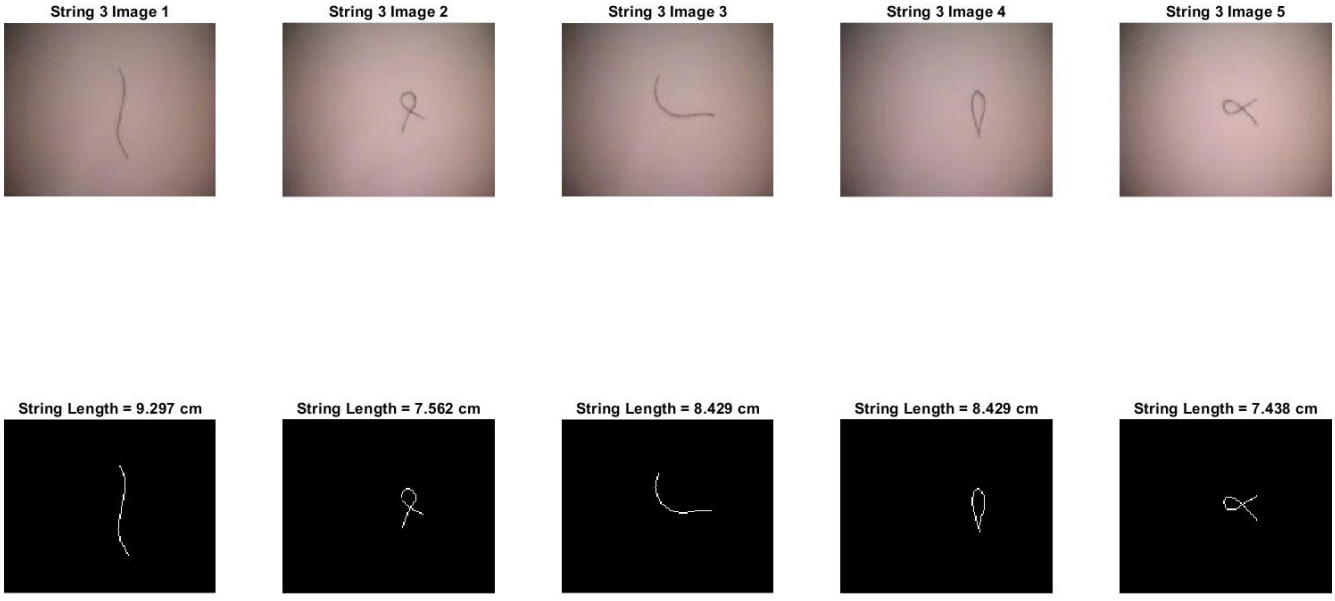

Figure 9: String 3 Results

3.4 Discussion

The primary drawback of the developed algorithm is that, by simply enumerating the pixel count and converting linearly into a distance, it does not take into consideration intersection points: where the line overlaps itself. These intersection points yield a single pixel count after skeletonisation, but

should yield at least two since the underneath string is not visible. Therefore, future development could focus on traversal algorithms for string length determination. Using this, string intersections could be properly accounted for.

4 String Tracing

4.1 Introduction

A line tracing algorithm was used to solve the children's game in Figure 10.

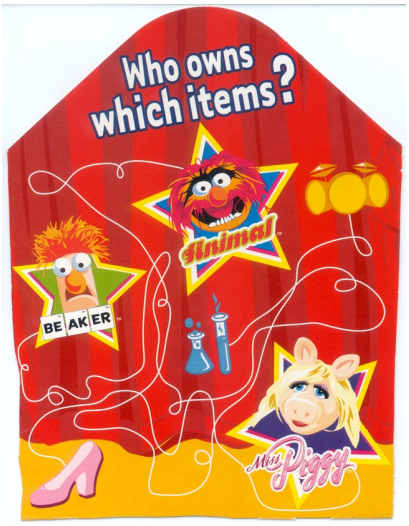

Figure 10: Sesame Street Children's Game

4.2 Method

The algorithm comprises of image preprocessing to extract the lines, and a line tracing algorithm. In addition, a GUI was developed.

4.2.1 Preprocessing Algorithm

The preprocessing algorithm consists of:

1. Colour quantisation through K-means clustering
2. Conversion to grayscale
3. Binarisation into black and white
4. A morphological opening to remove the desired lines
5. Differencing the opened image to recover just the desired lines
6. Denoising by extracting the largest component
7. Morphological skeletonisation

This process is shown in Figure 11 and explicated in the listing below:

Listing 5: String tracing preprocessing algorithm

```
%Load image
image = imread('sesame.tif');

```

```
%Image quantisation using K-means
quantisedImage = kmeansImage(image, 4);

%Conversion to grayscale
grayImage = rgb2gray(quantisedImage);

%Binarisation
bw = imbinarize(grayImage, 0.8);

%Morphological removing line
linesRemoved = imopen(bw, strel('diamond', 3));

%Difference gives back line
lines = linesRemoved ~= bw;

%Remove noise
justLine = zeros(size(lines));
stats = regionprops('table', lines, 'Area', 'PixelIdxList');
[sortedComponentAreas, sortingIndex] = sort(stats.Area, 'descend');
justLine(stats.PixelIdxList{sortingIndex(1)}) = 1;
justLine = justLine == 1;

%Skeletonisation
skeleton = bwskel(justLine);

subplot(4,2,1); imshow(image); title('Original Image');
subplot(4,2,2); imshow(quantisedImage); title('K-means quantisation');
subplot(4,2,3); imshow(grayImage); title('Grayscale Image');
subplot(4,2,4); imshow(bw); title('Binarised Image');
subplot(4,2,5); imshow(linesRemoved); title('Morphological Removal of
   Line');
subplot(4,2,6); imshow(lines); title('Removed Components');
subplot(4,2,7); imshow(justLine); title('Removing Noise');
subplot(4,2,8); imshow(skeleton); title('Skeletonisation');
```

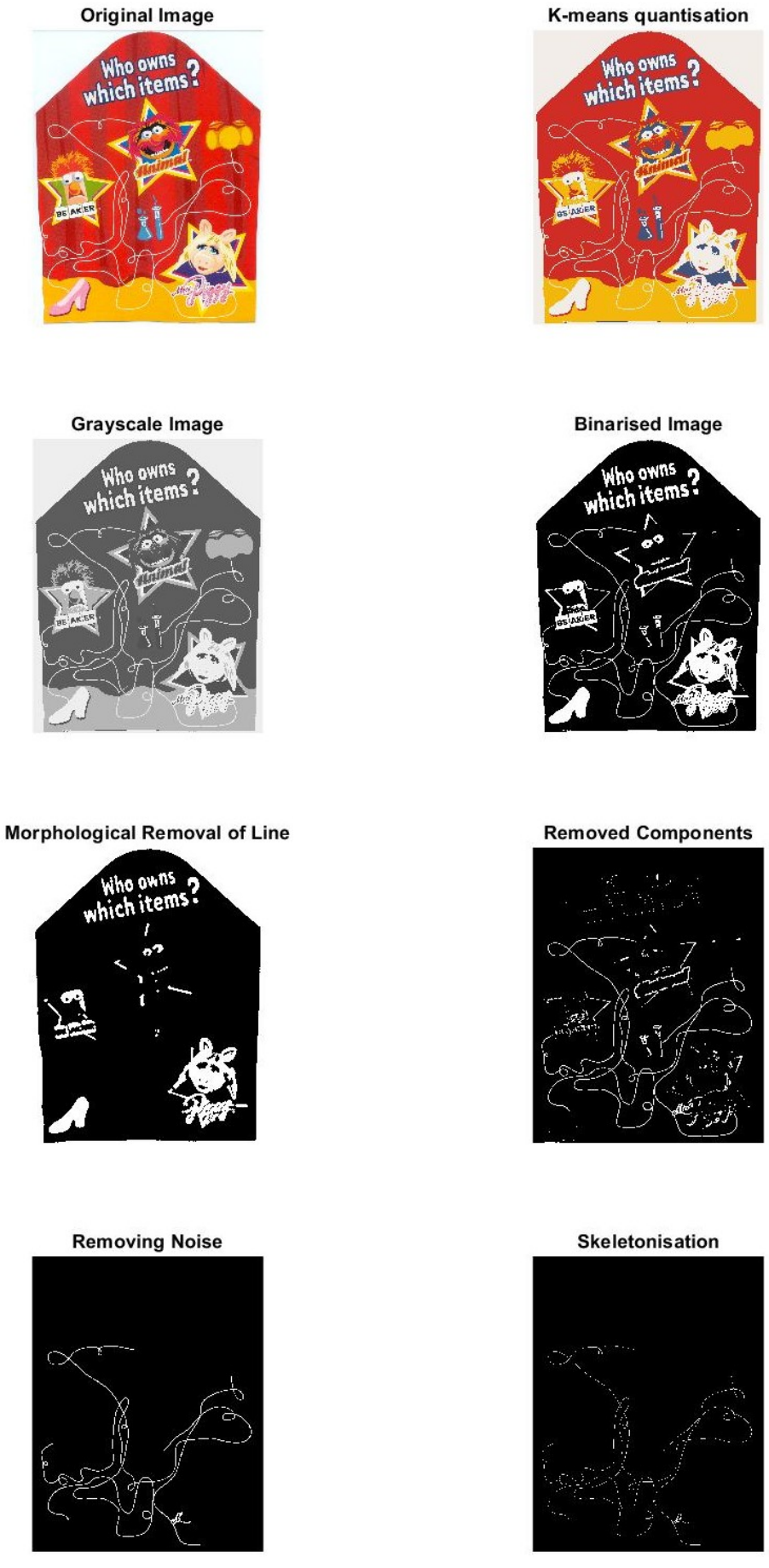

Figure 11: Preprocessing Algorithm

4.2.2 Line Tracing

The line tracing algorithm was applied to the preprocessed skeleton image. Since skeletonised, at each non-intersection point along the traversal there were only two other non-zero points connected to the current point. At each intersection point there was greater than two other non-zero points connected and at the end point there was only one other non-zero point connected. These cases were handled separately.

- **Non-Intersection Point:** The two non-zero points were the previous and next points respectively. Since the trace up to the current point was recorded, the non-zero point not already traversed was the next point.
- **Intersection Point:** Since the macroscopic behaviour of the line (e.g. direction, curvature) was not accurately reflected on the pixel scale at intersection points, the previous trace points were used to forecast ahead some distance and *skip over* the intersection point. The forecast calculated the spatial derivatives leading up to the point, filtered these spatial derivatives, linearly extrapolated the next spatial derivative, and then applied this extrapolated spatial derivative to forecast some distance ahead from the current point.

These cases are shown in the below listing:

Listing 6: Line Tracing Methods

```
%Handle non-intersection case
function handleNormalCase(self)
    currentPos = self.theTrack(self.theTrackIndex, :);
    lastPos = self.theTrack(self.theTrackIndex - 1, :);

    %Search connected points
    for i = [-1 0 1]
        for j = [-1 0 1]
            if(self.theBW(currentPos(1) + i, currentPos(2) + j) && ... %
               Dont choose a non-white point
              (i ~= 0 || j ~= 0) && ...                                %
                 Dont choose centre point - that was the last point
               sum((lastPos - currentPos - [i j]) ~= [0 0]))           %
                  Dont turn around and choose previous point
                  self.theTrack(self.theTrackIndex + 1, :) = currentPos
                      + [i j];
                  self.theTrackIndex = self.theTrackIndex + 1;
                  return
            end
        end
    end
end

%Spatial derivative forecasting
function [pointNAhead] = derivativeForecastN(self, backN, filterLength,
   forwardM)
    %Define history
    history = self.theTrack(self.theTrackIndex-backN:self.theTrackIndex,
       :);
```

```
24
25      %Compute spatial derivative
26      dHistory = diff(history);
27
28      %Filter
29      filter = ones(filterLength, 1) / filterLength;
30      smoothdHist = conv2(dHistory, filter, 'valid');
31
32      %Linearly extrapolate spatial derivatives
33      forecastDeriv = [polyval(polyfit(transpose(1:size(smoothdHist, 1)),
           smoothdHist(:,1), 1), size(smoothdHist, 1) + 1) ...
34                       polyval(polyfit(transpose(1:size(smoothdHist, 1)),
                            smoothdHist(:,2), 1), size(smoothdHist, 1) + 1)
                            ];
35
36      %Forecast point from extrapolated spatial derivative
37      pointNAhead = round(history(end, :) + (forwardM * forecastDeriv));
38  end
39
40  %Intersection case
41  function handleIntersectionCase(self)
42      %Spatial forecast some distance
43      forecastPos = self.derivativeForecastN(30, 5, 5);
44
45      %Spiral search from forecast to refind line
46      for index = 1:size(self.theSpiral, 1)
47          searchPos = forecastPos + self.theSpiral(index, :);
48          if(self.theBW(searchPos(1), searchPos(2)) && ...
49             ~sum(ismember(self.theTrack, searchPos, 'rows')))
50                futurePos = searchPos;
51                break
52          end
53      end
54
55      %Search connected points to find next point after forecast
56      index = 1;
57      for i = [-1 0 1]
58          for j = [-1 0 1]
59              if(self.theBW(futurePos(1) + i, futurePos(2) + j) && ...   %
                   Dont choose a non-white point
60                 (i ~= 0 || j ~= 0))                                    %Dont
                      choose centre point
61                    possiblePoints(index, :) = futurePos + [i j];
62                    index = index + 1;
63              end
64          end
65      end
66      self.theTrack(self.theTrackIndex + 1, :) = futurePos;
67      self.theTrackIndex = self.theTrackIndex + 1;
68
```

```
    %Choose point furthest from previous to preserve forecast direction
    [~, furthestIndex] = pdist2(possiblePoints, self.theTrack(self.
        theTrackIndex - 1, :), 'squaredeuclidean', 'Largest', 1);

    self.theTrack(self.theTrackIndex + 1, :) = possiblePoints(
        furthestIndex, :);
    self.theTrackIndex = self.theTrackIndex + 1;
end
```

4.2.3 GUI

The GUI comprised of:

- **Sesame Street Image:** Registered clicks to a callback. The callback finds the closest end point in the skeleton to the click and begins tracing the skeleton from this point. Once complete, the trace is overlain on the image.
- **Reset Button:** Clears all traces on the Sesame Street Image

This is shown in the following listing:

Listing 7: Line Tracing GUI

```
function GUI
    figHeight = 700;
    figWidth = 700;
    f = figure('Visible','off','Position',[25, 50, figWidth, figHeight]);

    % Image plot
    imageAxes = axes('Units','Pixels','Position',[25,25,figWidth-125,
        figHeight-50]);
    image = imread('sesame.tif');
    [bw, endPoints] = lineTrackingPreprocessing(image);
    tracker = LineTracker(bw);
    [endPointRows, endPointCols] = find(endPoints);
    endPoints = [endPointRows endPointCols];
    imageHandle = imshow(image, 'Parent', imageAxes);
    imageHandle.ButtonDownFcn = @imageClickCallback;
    hold on;

    % Reset Button
    resetButtonHandle = uicontrol('Style','pushbutton','String','Reset',
        ...
                                  'Position',[figWidth-100, figHeight/2,
                                      70,25], ...
                                  'Callback',{@resetButtonCallback});
    align(resetButtonHandle,'Center','None');

    % Make the UI visible.
    f.Visible = 'on';

```

```
26      function resetButtonCallback(source, eventData)
27          disp('Reset Button pressed');
28          hold off;
29          imageHandle = imshow(image, 'Parent', imageAxes);
30          imageHandle.ButtonDownFcn = @imageClickCallback;
31          hold on;
32      end
33
34      function imageClickCallback(source, eventData)
35          disp('Image was clicked');
36          clickPoint = get(gca, 'CurrentPoint');
37          xy = [clickPoint(1, 2) clickPoint(1,1)];
38
39          [~, closestIndex] = pdist2(endPoints, xy, 'squaredeuclidean', '
                Smallest', 1);
40          selectedEndPoint = endPoints(closestIndex, :);
41
42          tracker.trackFrom(selectedEndPoint);
43          scatter(tracker.theTrack(:,2), tracker.theTrack(:,1), 2, '*');
44      end
45
46
47      function [skeleton, endPoints] = lineTrackingPreprocessing(image)
48          quantisedImage = kmeansImage(image, 4);
49          grayImage = rgb2gray(quantisedImage);
50          bw = imbinarize(grayImage, 0.8);
51
52          linesRemoved = imopen(bw, ones(5));
53          lines = linesRemoved ~= bw;
54
55          justLine = zeros(size(lines));
56          stats = regionprops('table', lines, 'Area', 'PixelIdxList');
57          [~, sortingIndex] = sort(stats.Area, 'descend');
58          justLine(stats.PixelIdxList{sortingIndex(1)}) = 1;
59          justLine = justLine == 1;
60
61          skeleton = bwskel(justLine);
62          endPoints = bwmorph(skeleton, 'endpoints');
63      end
64  end
```

4.3 Results

The resulting GUI is shown in Figure 12. The line tracing was successful for two of the three paths. These are shown in Figure 13. The third trace was unsuccessful due to artefacts in the preprocessing algorithm, this is discussed later.

Figure 12: Line Tracing GUI

Figure 13: Successful Line Tracing

4.4 Discussion

The unsuccessful line tracing was caused by an artefact from the image preprocessing. This is shown in Figure 14. This artefact arose because it is connected to the desired lines, of identical colour and possesses similar neighbourhood characteristics. This inhibited separation based on component

selection, colour thresholding and morphological operations. As such, other distinguishing characteristics (e.g. density of intersection points) should be exploited in further development of the preprocessing algorithm.

Figure 14: Image Preprocessing Artefact

4.4.1 Alternative Methods

Alternative forecasting methods were developed but were less successful, these include:

- **Linear Forecasting:** Linear interpolation from previous N point
- **Momentum Forecasting:** Weighted average of the linear interpolation from the previous N points

These approaches are shown in the below listing:

Listing 8: Alternative Forecasting Methods

```
function [pointNAhead] = linearForecastN(self, N)
    nPointsAgo = self.theTrack(self.theTrackIndex - N, :);
    diff = self.theTrack(self.theTrackIndex, :) - nPointsAgo;
    pointNAhead = self.theTrack(self.theTrackIndex, :) + diff;
end

function [pointNAhead] = momentumForecastN(self, N, lookBackM)
    history = self.theTrack(self.theTrackIndex-N-lookBackM+1:self.
       theTrackIndex - N, :);

    %Linear interpolate M points
    diffs = self.theTrack(self.theTrackIndex, :) - history;
```

```
12      normalisedDiffs = diffs ./ linspace((lookBackM + N)/N, 1, lookBackM)
            ';
13      forecasts = self.theTrack(self.theTrackIndex, :) + normalisedDiffs;
14
15      %Calculate weighted average
16      adjustedForecasts = forecasts - forecasts(end, :);
17      forecastWeights = 1 .^ linspace(lookBackM - 1, 0, lookBackM)';
18      weightedForecasts = adjustedForecasts .* forecastWeights;
19
20      pointNAhead = round(forecasts(end, :) + mean(weightedForecasts(1:end
            -1, :)));
21  end
```

5 Appendix A: Additional Code Listings

Listing 9: Line Tracing Class

```
classdef LineTracker < handle
properties
        theSpiralSearchSize = 100;
        theForecastLength = 5;

        theBW;
        theTrack;
        theTrackIndex;
        isTracking;
        theSpiral;
end

methods
    function self = LineTracker(bw)
        self.theBW = bw;
        self.theTrackIndex = 0;
        self.isTracking = 0;

        %Spiral search
        spir = spiral(self.theSpiralSearchSize);
        [~, idx] = sort(spir(:));
        [spiralRow, spiralCol] = ind2sub([self.theSpiralSearchSize, self.
           theSpiralSearchSize], idx);
        spiralRow = spiralRow - round(self.theSpiralSearchSize / 2);
        spiralCol = spiralCol - round(self.theSpiralSearchSize / 2);
        self.theSpiral = [spiralRow spiralCol];
    end

    function trackFrom(self, start)
        self.theTrack = [start; self.getFirstPoint(start)];
        self.theTrackIndex = 2;
        self.isTracking = 1;

        self.trackNextPoint();
        while(self.isTracking)
            self.trackNextPoint();
        end
    end

    function [secondPosition] = getFirstPoint(self, start)
        for i = [-1 0 1]
            for j = [-1 0 1]
                if(self.theBW(start(1) + i, start(2) + j) == 0 || ...
                  (i==0 && j == 0))
                    continue;
                end
```

```
                secondPosition = start + [i j];
            end
        end
    end

    function trackNextPoint(self)

        neighbourhood = self.theBW(self.theTrack(self.theTrackIndex, 1)
           -1:self.theTrack(self.theTrackIndex, 1)+1, ...
                                   self.theTrack(self.theTrackIndex, 2)
                                      -1:self.theTrack(self.theTrackIndex
                                      , 2)+1);
        if(sum(neighbourhood(:)) <= 2)
            self.isTracking = 0;
        elseif(sum(neighbourhood(:)) == 3)
            self.handleNormalCase();
        else
            if(self.theTrackIndex <= 5)
                self.isTracking = 0;
            else
                self.handleIntersectionCase();
            end
        end
    end

    function handleNormalCase(self)
        currentPos = self.theTrack(self.theTrackIndex, :);
        lastPos = self.theTrack(self.theTrackIndex - 1, :);

        for i = [-1 0 1]
            for j = [-1 0 1]
                if(self.theBW(currentPos(1) + i, currentPos(2) + j) &&
                   ... %Dont choose a non-white point
                  (i ~= 0 || j ~= 0) && ...
                                                %Dont choose centre
                     point - that was the last point
                   sum((lastPos - currentPos - [i j]) ~= [0 0]))
                               %Dont turn around and choose old point
                    self.theTrack(self.theTrackIndex + 1, :) =
                       currentPos + [i j];
                    self.theTrackIndex = self.theTrackIndex + 1;
                    return
                end
            end
        end
    end

    function [pointNAhead] = linearForecastN(self, N)
        nPointsAgo = self.theTrack(self.theTrackIndex - N, :);
        diff = self.theTrack(self.theTrackIndex, :) - nPointsAgo;
```

```
88          pointNAhead = self.theTrack(self.theTrackIndex, :) + diff;
89      end
90
91      function [pointNAhead] = momentumForecastN(self, N, lookBackM)
92          history = self.theTrack(self.theTrackIndex-N-lookBackM+1:self.
               theTrackIndex - N, :);
93
94          diffs = self.theTrack(self.theTrackIndex, :) - history;
95          normalisedDiffs = diffs ./ linspace((lookBackM + N)/N, 1,
               lookBackM)';
96          forecasts = self.theTrack(self.theTrackIndex, :) +
               normalisedDiffs;
97
98          adjustedForecasts = forecasts - forecasts(end, :);
99          forecastWeights = 1 .^ linspace(lookBackM - 1, 0, lookBackM)';
100         weightedForecasts = adjustedForecasts .* forecastWeights;
101
102         pointNAhead = round(forecasts(end, :) + mean(weightedForecasts(1:
               end-1, :)));
103     end
104
105     function [pointNAhead] = derivativeForecastN(self, backN,
           filterLength, forwardM)
106          history = self.theTrack(self.theTrackIndex-backN:self.
                theTrackIndex, :);
107          dHistory = diff(history);
108          filter = ones(filterLength, 1) / filterLength;
109
110          smoothdHist = conv2(dHistory, filter, 'valid');
111          forecastDeriv = [polyval(polyfit(transpose(1:size(smoothdHist,
                1)), smoothdHist(:,1), 1), size(smoothdHist, 1) + 1) ...
112                          polyval(polyfit(transpose(1:size(smoothdHist,
                             1)), smoothdHist(:,2), 1), size(smoothdHist,
                              1) + 1)];
113
114          pointNAhead = round(history(end, :) + (forwardM * forecastDeriv)
                );
115     end
116
117
118     function handleIntersectionCase(self)
119         forecastPos = self.derivativeForecastN(30, 5, 5);
120
121         for index = 1:size(self.theSpiral, 1)
122             searchPos = forecastPos + self.theSpiral(index, :);
123             if(self.theBW(searchPos(1), searchPos(2)) && ...
124                ~sum(ismember(self.theTrack, searchPos, 'rows')))
125                     futurePos = searchPos;
126                     break
127             end
```

```
        end

        index = 1;
        for i = [-1 0 1]
            for j = [-1 0 1]
                if(self.theBW(futurePos(1) + i, futurePos(2) + j) && ...
                     %Dont choose a non-white point
                   (i ~= 0 || j ~= 0))                                  %Dont
                       choose centre point
                        possiblePoints(index, :) = futurePos + [i j];
                        index = index + 1;
                end
            end
        end
        self.theTrack(self.theTrackIndex + 1, :) = futurePos;
        self.theTrackIndex = self.theTrackIndex + 1;

        [~, furthestIndex] = pdist2(possiblePoints, self.theTrack(self.
           theTrackIndex - 1, :), 'squaredeuclidean', 'Largest', 1);

        self.theTrack(self.theTrackIndex + 1, :) = possiblePoints(
           furthestIndex, :);
        self.theTrackIndex = self.theTrackIndex + 1;
    end
end
end
```

Listing 10: K-means Image Colour Quantisation

```
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
```