ELEC4630 Assessment Task 3

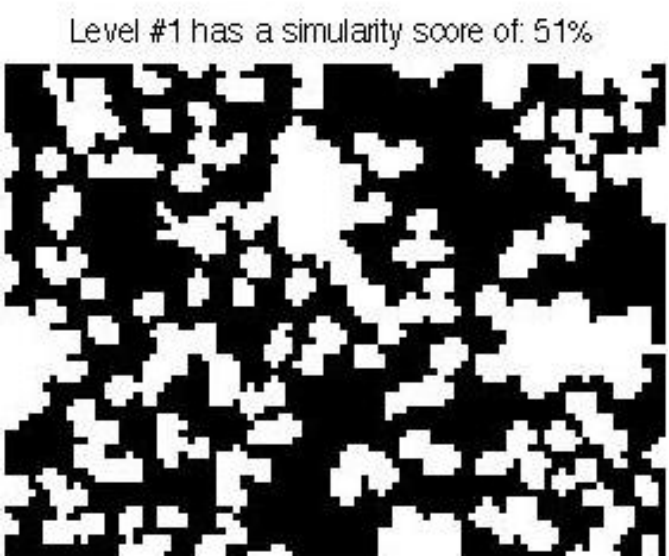
Segmenting Cells and How long is that piece of string?

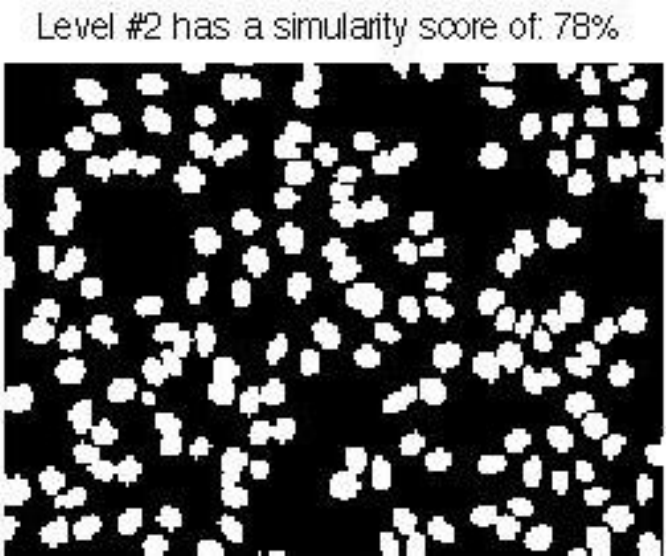
Part 1: Segmenting Cells

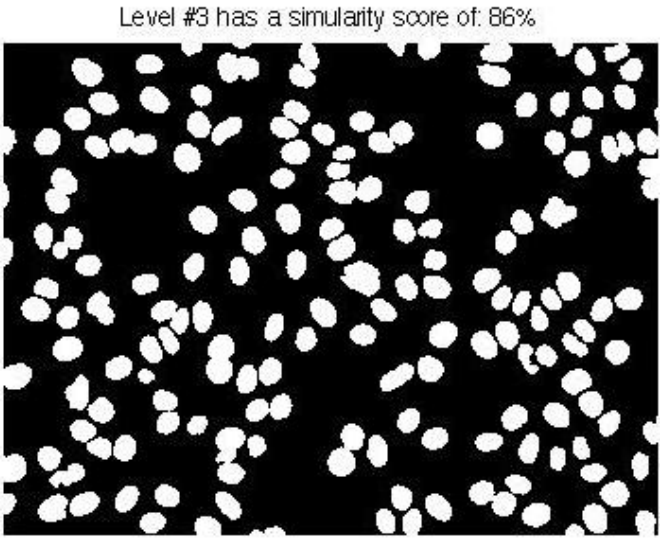
To find each of the cells in the hi_FITC.tif picture provided a submersion method was used. Starting at the darkest, or 'deepest' part of the image, adjacent pixels of less or equal value are flooded. The 'water level' of the flood rises with each iteration, eventually overcoming local maxima that prevent the flooding from reaching partially blocked areas of the image.

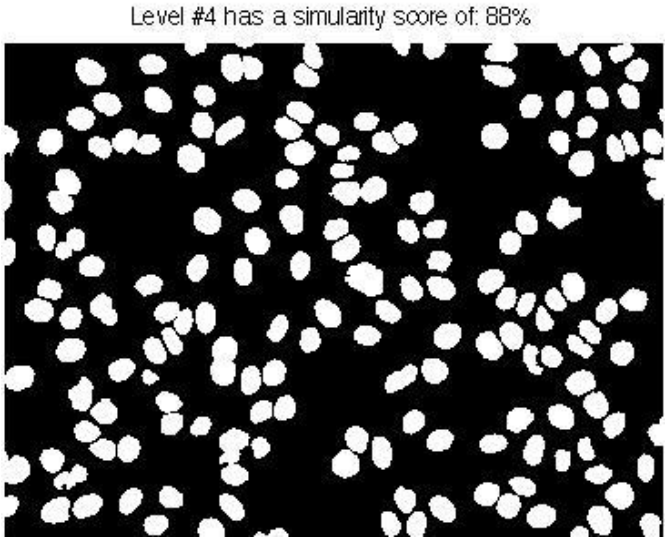
The flooding process is performed on 5 different sized images starting on a picture 1/16th the original size and ending on the original image. The first image is flooded until all pixels in the generated image mask are identified as either part of an island (white) or water (black). The following iterations use a modified version of the previous iteration's mask, resizing to fit the new size and eroding the dark areas of the image to ensure that the difference in image size does not force the flooding into unwanted areas.

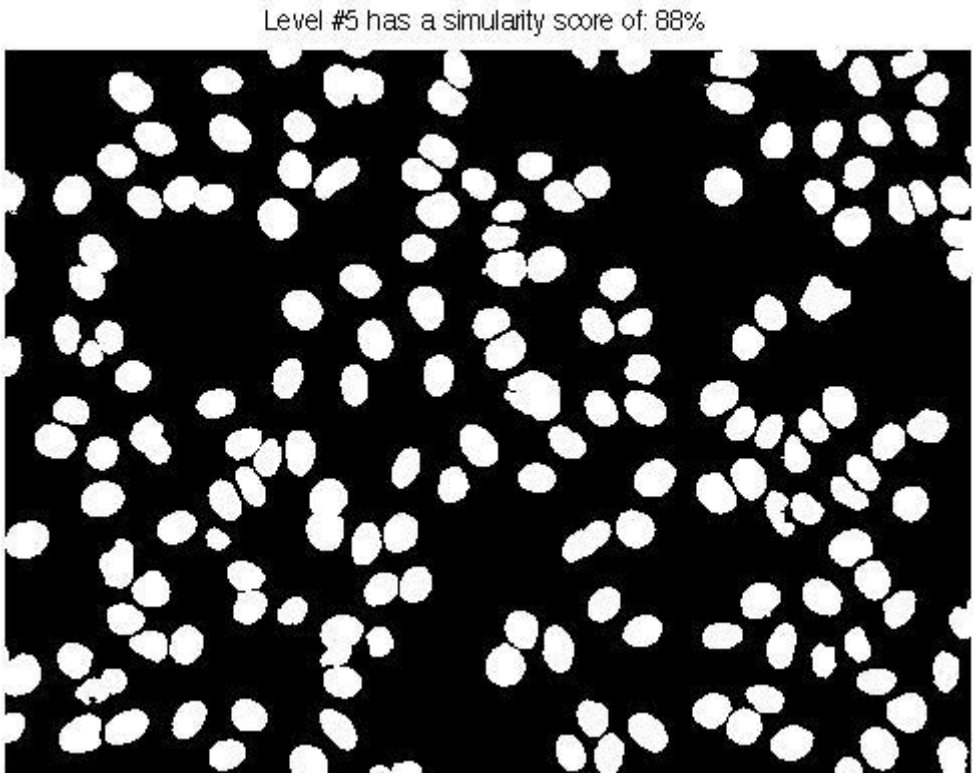
The following images show the mask produced by each iteration as well as their similarity to the mask provided.





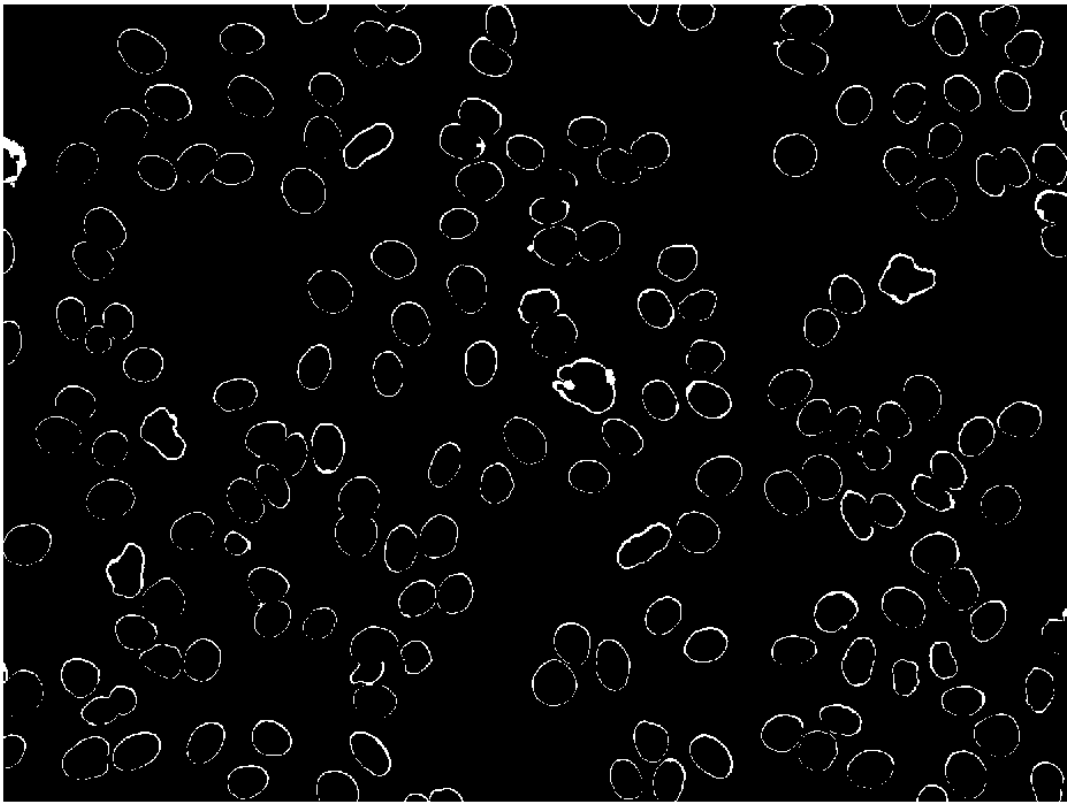






Further improvement could be done on each segmented area using, for example, a trellis and pathing method such as that used in the last assignment.

As part of the task we were asked to design a suitable metric to measure how well we can segment the cells from the image. The function segCheck() handles this by performing a logical subtraction of the segmented image from the provided, 'perfect' mask. This gives the difference between the images and, by counting the number of white pixels left over I extracted a percentage value of how accurate the segmentation was. The segCheck() function can be found with the code.



Part 1 – Code

A3p1.m – The main script

```
% s4262468's solution to Part1 of ELEC4630's 3rd Assessment task
close all, clear, clc;

% load images
hi = imread('hi_FITC.tif');
hi_m = imread('hi_Mask.tif');

bord = imread('borderline_FITC.tif');
bord_m = imread('borderline_Mask.tif');

i = hi; % make us a nice variable

disp('Pre-scaling images');
% Make a structure containing different the image at different scales.
levels = {};
levels{5} = struct('z', 1, 'p', i, 'm', mode(double(i(:))), 't', zeros(size(i)));
for j = 1:4,
    scale = 1/(2^j);
    tmp = struct('z', scale, 'p', imresize(i, scale));
    tmp.t = zeros(size(tmp.p));
    tmp.m = mode(double(tmp.p(:))); % find the most common value in the image
    levels{5-j} = tmp;
end

disp('Processing First level using submersion');

% Starting at the lowest level of pixelage
cur = levels{1};
dims = size(cur.p);

% Border matrices
cb(1, 1:dims(2)) = 1; rb(1:dims(1), 1) = 1;

% Submerge the lowest value
sub = zeros(dims);
depth = min(cur.p(:)); fdepth = depth;
[ty, tx] = ind2sub(dims, find(cur.p==depth));
for point = 1:size(ty,1), sub(ty(point), tx(point)) = 1; end

% iteratively submerge until all local maxima are found.
mask = cur.p;
while min(min((mask==255) + (mask==0))) ~= 1, % while we still have nonbinary values
    depth = depth+1;
    disp(['     Processing depth: ' int2str(depth)]);
%     mask = cur.p;
    % for each corner of the currently selected points tx,ty
    for newpt = 1:size(tx, 1),
        px = tx(newpt); py = ty(newpt); % shorthand points
        mask(py, px) = 0;
        % North
        if py>1 && mask(py-1,px)~=0 && mask(py-1,px)~=255 && cur.p(py-1, px)<=depth,
            mask = flood(px, py-1, mask, 0);
            mask(py-1, px) = 0;
        end
        % South
        if py<dims(1) && mask(py+1,px)~=0 && mask(py+1,px)~=255 && cur.p(py+1, px)<=depth,
```

```
            mask = flood(px, py+1, mask, 0);
            mask(py+1, px) = 0;
        end
        % East
        if px<dims(2) && mask(py,px+1)~=0 && mask(py,px+1)~=255 && cur.p(py, px+1)<=depth,
            mask = flood(px+1, py, mask, 0);
            mask(py, px+1) = 0;
        end
        % West
        if px>1 && mask(py,px-1)~=0 && mask(py,px-1)~=255 && cur.p(py, px-1)<=depth,
            mask = flood(px-1, py, mask, 0);
            mask(py, px-1) = 0;
        end
    end
    sub(mask==0 - sub>0) = depth-fdepth;

    % Look at the outermost pixels
    outerPixels = mask==0 - imerode(mask==0, strel('disk', 1));
    [ty, tx] = ind2sub(dims, find(outerPixels));

    % Check for enclosed areas
    enclosed = imfill([rb sub>0 rb], 'holes') - [rb sub>0 rb];
    enclosedRow = enclosed(:, 2:size(enclosed,2)-1);
    enclosed = imfill([cb;sub>0;cb], 'holes') - [cb;sub>0;cb];
    enclosedCol = enclosed(2:size(enclosed,1)-1, :);

%     imshow(enclosed)
    mask(logical(enclosedRow + enclosedCol)) = 255;
end

clearvars -except levels mask hi hi_m bord_m bord
disp('We just cleared most of the variables')

 % clean up the mask
mask = bwareaopen(mask,5);
imshow(mask); score = segCheck(mask, imread('hi_Mask.tif'))
title(['Level #1 has a simularity score of: ' int2str(score) '%']);
 disp(['Our base score is: ' int2str(score) '%']);
input('Enter-Key to take to the next level');
clc, close all

curlvl = 2; % we just did level 1
while curlvl <= 5 % for each level leading up to full size
    disp(['Now processing level No.' int2str(curlvl)]);
    % setup all the vars for this time
    cur = levels{curlvl};
    dims = size(cur.p);

    % Border matrices
    cb(1, 1:dims(2)) = 1; rb(1:dims(1), 1) = 1;

    % Bring mask up to size from last level
    mask = imresize(mask, dims, 'box');

    % Bring back the edges to be safe;
    mask = ~imerode(~mask, strel('disk', 1));

    % Remove the white from the mask so we can refill things in.
    tmp = cur.p; tmp(mask==0) = cur.m;
    mask = tmp; clearvars tmp % kill tmp after we're finished
```

```
% % Begin the submersion
% Submerge the lowest value
sub = zeros(dims);
depth = min(cur.p(:)); fdepth = depth;
[ty, tx] = ind2sub(dims, find(cur.p==depth));
for point = 1:size(ty,1), sub(ty(point), tx(point)) = 1; end
while min(min((mask==255) + (mask==0))) ~= 1, % while we still have nonbinary values
    depth = depth+1;
    disp(['Processing depth: ' int2str(depth)]);

    % for each corner of the currently selected points tx,ty
    for newpt = 1:size(tx, 1),
        px = tx(newpt); py = ty(newpt); % shorthand points
        mask(py, px) = 0;
        % North
        if py>1 && mask(py-1,px)~=0 && mask(py-1,px)~=255 && cur.p(py-1, px)<=depth,
            mask = flood(px, py-1, mask, 0);
            mask(py-1, px) = 0;
        end
        % South
        if py<dims(1) && mask(py+1,px)~=0 && mask(py+1,px)~=255 && cur.p(py+1, px)<=depth,
            mask = flood(px, py+1, mask, 0);
            mask(py+1, px) = 0;
        end
        % East
        if px<dims(2) && mask(py,px+1)~=0 && mask(py,px+1)~=255 && cur.p(py, px+1)<=depth,
            mask = flood(px+1, py, mask, 0);
            mask(py, px+1) = 0;
        end
        % West
        if px>1 && mask(py,px-1)~=0 && mask(py,px-1)~=255 && cur.p(py, px-1)<=depth,
            mask = flood(px-1, py, mask, 0);
            mask(py, px-1) = 0;
        end
    end
    sub(mask==0 - sub>0) = depth-fdepth;

    % Look at the outermost pixels next time
    outerPixels = mask==0 - imerode(mask==0, strel('disk', 1));
    [ty, tx] = ind2sub(dims, find(outerPixels));

    if depth-fdepth > 60, % 50 is a random number :(
        % Check for enclosed areas
        enclosed = imfill([rb sub>0 rb], 'holes') - [rb sub>0 rb];
        enclosedRow = enclosed(:, 2:size(enclosed,2)-1);
        enclosed = imfill([cb;sub>0;cb], 'holes') - [cb;sub>0;cb];
        enclosedCol = enclosed(2:size(enclosed,1)-1, :);

    %     imshow(enclosed)
        mask(logical(enclosedRow + enclosedCol)) = 255;
    end
end

mask = bwareaopen(mask,5);
score = segCheck(mask, imread('hi_Mask.tif'));

disp(['Finished proc-ing lvl No.' int2str(curlvl)]);
disp(['This is the result, it scored: ' int2str(score) '%']);
imshow(mask);
```

```
    title(['Level #' int2str(curlvl) ' has a simularity score of: ' int2str(score) '%']);
    if curlvl ~= 5,
        input('Use the Enter key to take this to the next level');
        clc, close all
    else
        disp('This is the final result!');
    end

    curlvl=curlvl+1; % NEXT LEVEL!
end
```

segCheck() – Compares and returns a percentage of similarity

```
function [ value ] = segCheck( src, mask)
% Run this to compare 2 bw images and return a value from 1-100 showing how
% close they are to being the same. 100 implies a perfect match.

mask = logical(mask);
src = logical(src);

resrc = imresize(src, size(mask), 'box');
i = abs(resrc - mask);

pixInResrc = size(find(resrc),1);
pixInDiff = size(find(i),1);

value = 100-(pixInDiff/pixInResrc)*100;

end
```

Part 2: How long is that piece of string?

Provided with the lengths of 2 pieces of string and 15 pictures matching those 2 and a third string, we were asked to calculate the length of the string in each of the 15 images in both pixels and centimetres.

My script can be split into 4 main sections: Setup, Preprocessing, Processing and the Conclusion. During the setup each string's images are loaded as grayscale images into an array for each string. During preprocessing I even the colour of the background and crop the image to where I suspect the string is. The background evening takes an extremely blurred version of the image (to remove fine detail) and subtracts it from the original image. This works, as the natural gradient of the background remains similar during the blurring process and this process provides less dynamic background to work from.

The cropping simply uses edge detection to determine where the most interesting pixels are and then uses the outmost pixel values to determine the cropped area. This method does clash with the background filtering method as the Gaussian filter used for the background creates small, but pickup-able edges to the left and top of an image, giving the badly cropped images found in some of the examples included later in this report.

Processing is the most involved part of the script as it integrates thresholding with skeletonisation, pruning and rejoining. To convert the cropped image into something to threshold I use a mixture of morphology and edge detection. The mix was needed due to the un-reliability of simply thresholding the image. Multiple calls to bwareaopen() are used to remove small errors in the image and dilation is performed to further reduce the holes inside the thresholded string. Unfortunately this thresholding destroys the small gaps created by the small loops in str1-3 and str2-5. These images must be treated as outliers as a result.

Skeletonisation or thinning of the blob is then performed use the hit and miss method. This is the standard way to perform thinning and uses 4 structured elements, or 2 composite structured elements. Thinning is run until convergence before being pruned. Unlike thinning, if pruning can be destructive if used too many times. I've opted for 2 runs of pruning in order to trim most of the small extra ends without influencing the size of the actual string too much. If pruning is left to run until convergence the only shapes left would be loops.

The final step in processing an image is to attempt to re-connect any parts of the string that became unjoined, or were never captured properly. Using another composite structured element the image is checked for line endings. A string can have 1 or 2 depending on how it's laid, and if there's an extra line-end (making 3) then it's probably a badly pruned edge. Given this, if there are 4 or more detected line ends then the script joins the 2 closest ends with a line. The 2 points that are joined are recorded and then cannot be used to draw another line in this manner. This method of repairing the image works will in most cases and in the case of str1-3 helps repair some of the damage caused by the during the thresholding. The best example of a repaired image is str3-2, images of which you can find later in this document.

The conclusion wraps the length and ratio calculations in with making an output statement for each picture. Firstly I use the hit and miss method with a composite structured element of my own design to count all the diagonally connect pixels in the image. This allows the script to compensate for the difference in distance between directly and diagonally adjacent pixels with diagonal distances are $\sqrt{2}$ times more than direct distances.

$$Length = \left((\sqrt{2} - 1) \sum (Diagonalpx) + \sum (Foregroundpx) \right) \times Ratio$$

The equation above shows how I calculate both the length of the strings and the ratio of Length to pixels for the 3rd string. The following table shows the small difference that is created by compensating for the diagonal distances as well as the format of my script's output. Values are in millimetres and the d:# at the end of each line shows the number of diagonals found in that particular string's skeleton.

All the code written for this assignment is written by me or included in a MATLAB installation except for the func_Drawline() function used during re-joining part of the processing. This function simply draws a line between 2 points on an image and was found on the official MATLAB script sharing website.

Without compensation for Diagonals	**Compensating for Diagonals**
Actual len of str#1 is: 130 -- d:34 Actual len of str#2 is: 129 -- d:42 Actual len of str#3 is: 122 -- d:27 Actual len of str#4 is: 134 -- d:37 Actual len of str#5 is: 126 -- d:47 Average Length: 128 . Actual len of str#1 is: 155 -- d:44 Actual len of str#2 is: 166 -- d:31 Actual len of str#3 is: 155 -- d:48 Actual len of str#4 is: 149 -- d:56 Actual len of str#5 is: 118 -- d:35 Average Length: 149 . Actual len of str#1 is: 86 -- d:14 Actual len of str#2 is: 80 -- d:19 Actual len of str#3 is: 78 -- d:16 Actual len of str#4 is: 79 -- d:17 Actual len of str#5 is: 75 -- d:25 Average Length: 80 .	Actual len of str#1 is: 130 -- d:34 Actual len of str#2 is: 132 -- d:42 Actual len of str#3 is: 120 -- d:27 Actual len of str#4 is: 134 -- d:37 Actual len of str#5 is: 132 -- d:47 Average Length: 130 . Actual len of str#1 is: 155 -- d:44 Actual len of str#2 is: 159 -- d:31 Actual len of str#3 is: 157 -- d:48 Actual len of str#4 is: 155 -- d:56 Actual len of str#5 is: 119 -- d:35 Average Length: 149 . Actual len of str#1 is: 82 -- d:14 Actual len of str#2 is: 79 -- d:19 Actual len of str#3 is: 75 -- d:16 Actual len of str#4 is: 77 -- d:17 Actual len of str#5 is: 77 -- d:25 Average Length: 78 .

The following screenshots of the figures produced by my script for each string. From left to right the subject of the images are: Picture after pre-processing + skeleton overlay, picture after pre-processing, thresholded image before processing, full processed image.

String1-#1 Pixel Length: 111 - Est. Length: 130

String1-#2 Pixel Length: 110 - Est. Length: 132

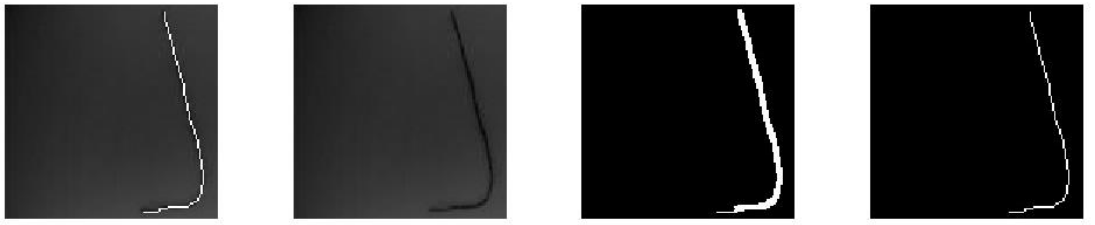
String1-#3 Pixel Length: 104 - Est. Length: 120

String1-#4 Pixel Length: 114 - Est. Length: 134

String1-#5 Pixel Length: 108 - Est. Length: 132

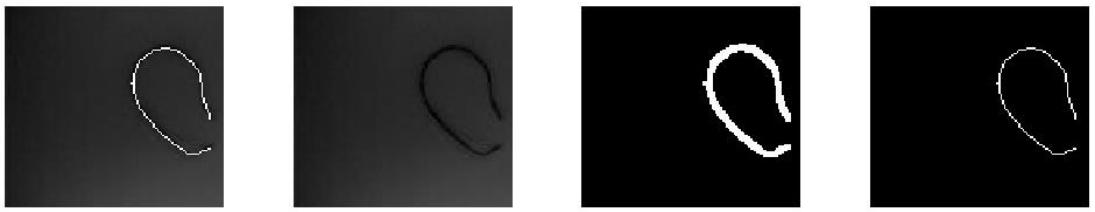
String2-#1 Pixel Length: 131 - Est. Length: 155

String2-#2 Pixel Length: 140 - Est. Length: 159

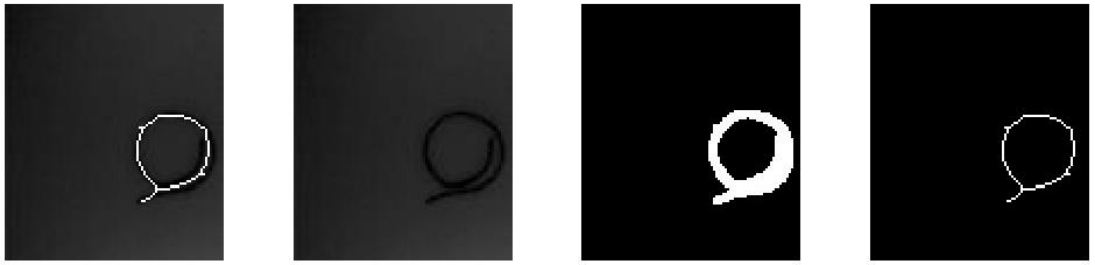


String2-#3 Pixel Length: 131 - Est. Length: 157

String2-#4 Pixel Length: 126 - Est. Length: 155

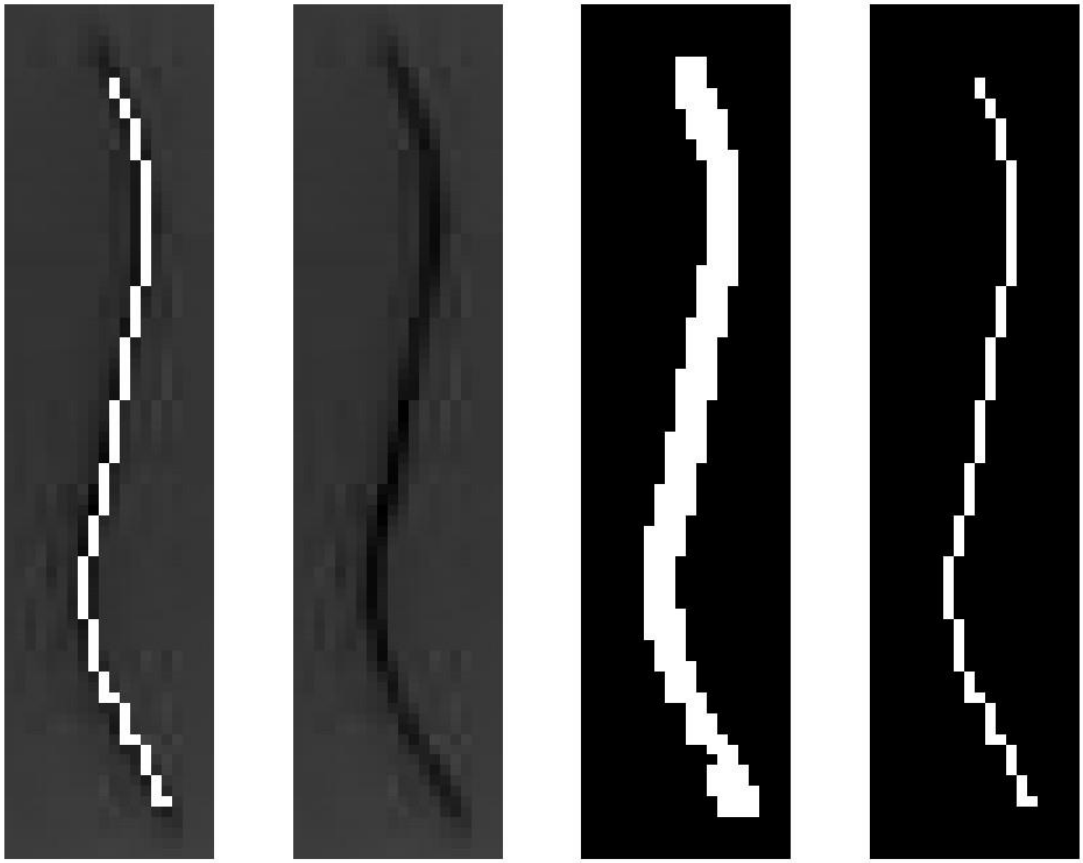


String2-#5 Pixel Length: 100 - Est. Length: 119



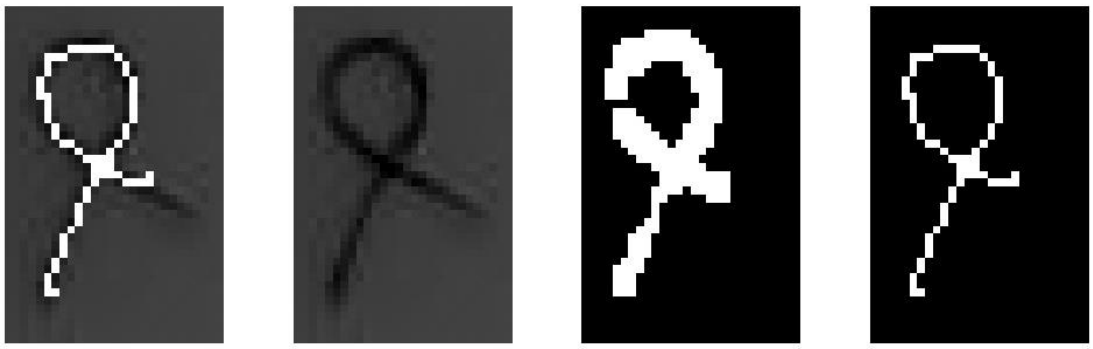
String3-#1

Pixel Length: 73 - Est. Length: 82



String3-#2

Pixel Length: 68 - Est. Length: 79

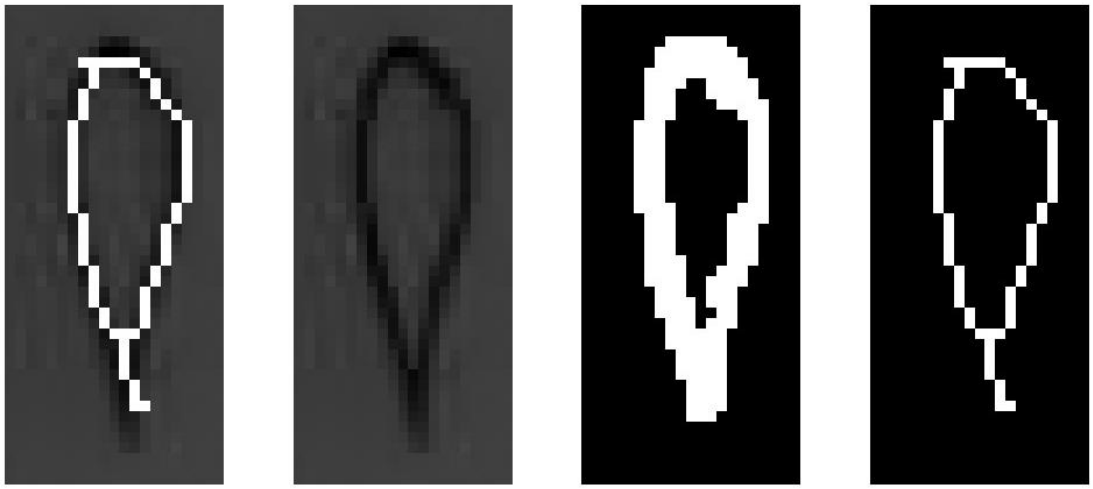


String3-#3

Pixel Length: 66 - Est. Length: 75

String3-#4

Pixel Length: 67 - Est. Length: 77



String3-#5

Pixel Length: 64 - Est. Length: 77

Part 2 – Code

```
% s4262468's solution to Part1 of ELEC4630's 3rd Assessment task
close all, clear, clc;

% How long is a piece of string?

% Load the strings!
str1 = {struct()}; str2 = {struct()}; str3 = {struct()};
disp('Loading Strings');
addpath('string');
for itr = 1:15,
    if itr <= 5,
        disp(['String1_' int2str(itr) '.jpg']);
        str1{itr}.o = (rgb2gray(imread(['String1_' int2str(itr) '.jpg'])));
    elseif itr <= 10,
        disp(['String2_' int2str(itr) '.jpg']);
        str2{itr-5}.o = (rgb2gray(imread(['String2_' int2str(itr-5) '.jpg'])));
    else
        disp(['String3_' int2str(itr) '.jpg']);
        str3{itr-10}.o = (rgb2gray(imread(['String3_' int2str(itr-10) '.jpg'])));
    end
end

% Loading saved lengths
str1{1}.len = 130; str2{1}.len = 155;
clc
disp('Pre-processing images')
% Crop all the images to the interesting bits by looking at edges.
buf = 3; f = fspecial('gaussian', [500 500], 50);
for itr = 1:15,
    if itr <= 5,
        disp(['> Str1_' int2str(itr)]);
        i = str1{itr}.o;
        i = i-(imfilter(i,f));
        [r, c] = ind2sub(size(i), find(edge(i, 'sobel')));
        rmin = min(r)-buf; rmax = max(r)+buf; cmin = min(c)-buf; cmax = max(c)+buf;
        % Make sure we don't try to go off the edge.
        if rmin<1, rmin=1; end, if rmax>size(i,1), rmax=size(i,1); end
        if cmin<1, cmin=1; end, if cmax>size(i,2), cmax=size(i,2); end
        t(:,:) = i(rmin:rmax, cmin:cmax);
        str1{itr}.p = t;
    elseif itr <= 10,
        disp(['> Str2_' int2str(itr-5)]);
        i = str2{itr-5}.o;
        i = i-(imfilter(i,f));
        [r, c] = ind2sub(size(i), find(edge(i, 'sobel')));
        rmin = min(r)-buf; rmax = max(r)+buf; cmin = min(c)-buf; cmax = max(c)+buf;
        % Make sure we don't try to go off the edge.
        if rmin<1, rmin=1; end, if rmax>size(i,1), rmax=size(i,1); end
        if cmin<1, cmin=1; end, if cmax>size(i,2), cmax=size(i,2); end
        t(:,:) = i(rmin:rmax, cmin:cmax);
        str2{itr-5}.p = t;
    else
        disp(['> Str3_' int2str(itr-10)]);
        i = str3{itr-10}.o;
        i = i-(imfilter(i,f));
        [r, c] = ind2sub(size(i), find(edge(i, 'sobel', 0.02)));
        rmin = min(r)-buf; rmax = max(r)+buf; cmin = min(c)-buf; cmax = max(c)+buf;
```

```
        % Make sure we don't try to go off the edge.
        if rmin<1, rmin=1; end, if rmax>size(i,1), rmax=size(i,1); end
        if cmin<1, cmin=1; end, if cmax>size(i,2), cmax=size(i,2); end
        t(:,:) = i(rmin:rmax, cmin:cmax);
        str3{itr-10}.p = t;
    end
%       figure, imshow(t);
    clearvars t i
end

% First step is to find the pixel length of the str1&2 in the provided
% pictures
% This then gives us something to go with for the rest of the images

disp('Clearing console');
% input('Press enter to clear the console and continue');
clc, close all
strs = {str1, str2, str3};
for numstr = 1:3,
    str = strs{numstr};
    for itr = 1:5,
        str{itr}.bw = imclearborder(~(str{itr}.p>30));
        str{itr}.bw = str{itr}.bw | (edge(str{itr}.p, 'sobel'));
        str{itr}.bw = bwareaopen(str{itr}.bw, 10, 8);

        factor = 2;
        str{itr}.bw = imresize(str{itr}.bw, factor);
        str{itr}.bw = imdilate(str{itr}.bw, strel('disk', 1));
        str{itr}.bw = imresize(str{itr}.bw, 1/factor, 'box');
        str{itr}.bw = ~bwareaopen(~str{itr}.bw, 10, 4);

        % Skeletonisation Starts here
        sk = str{itr}.bw;
        el1 = [-1 -1 -1;
                0  1  0;
                1  1  1];

        el2 = [0 -1 -1;
               1  1 -1;
               0  1  0];
        el = el1; before = zeros(size(sk));
        while(1),
            before = sk;
            counter = 8;
            while(1),
                tmp = imerode(sk, el==1) & imerode(~sk, el==-1);
                sk = sk & ~logical(tmp);

                % Switch between the 2 rotating elements
                if el==el1, el1 = rot90(el); el = el2;
                else el2 = rot90(el); el = el1; end

                counter = counter-1;
                % break if we've been through this 8 times
                if counter == 0, break; end
            end
            if isequal(before, sk), break; end
        end

        % Commence pruning!
```

```
pr1 = [-1 -1 -1;
       -1  1 -1;
       -1  0  0];

pr2 = [-1 -1 -1;
       -1  1 -1;
        0  0 -1];
pr = el1;
reps = 2;
counter = 8;
while(1),
    tmp = imerode(sk, pr==1) & imerode(~sk, pr==-1);
    sk = sk & ~logical(tmp);

    if pr==pr1, pr1 = rot90(pr); pr = pr2;
    else pr2 = rot90(pr); pr = pr1; end

    counter=counter-1;
    if counter==0, reps=reps-1; counter = 8; end;
    if reps==0, break; end
end

% Make an image of just "ending" points
% and look for closest for joining
ends = zeros(size(sk)); used = ends;
el = [-1 -1 -1;
      -1  1 -1;
       0  0  0];

while(1), r = 4;
    while r>0, % Find the ends
        ends = ends | imerode(sk, el==1) & imerode(~sk, el==-1);
        el = rot90(el); r=r-1;
    end
    ends = ends & ~used; % ignore the ones we've used already
    % break if we have less than 4 end points
    if size(find(ends),1)<4, break; end

    % if we have enough, lets connect the 2 points closest to
    % eachother
    [r,c] = ind2sub(size(ends), find(ends));
    closest = struct('d', 1000000);
    for j = 1:size(r,1),
        for k = 1:size(r,1),
            if j==k, continue, end;
            d = pdist([r(j) c(j); r(k) c(k)]);
            if d<closest.d,
                closest.d = d;
                closest.a = [r(j) c(j)];
                closest.b = [r(k) c(k)];
            end
        end
    end
    % Mark used points
    used(closest.a(1), closest.a(2)) = 1;
    used(closest.b(1), closest.b(2)) = 1;

    % connect the 2 closest ends for reasons
    sk = func_Drawline(sk, closest.a(1), closest.a(2), ...
        closest.b(1), closest.b(2),1);
```

```
        end

        % Count diagonals for Operation: #totes_precise length measurement
        el = [ 0  0 0; % this will trigger for every point that has a
              -1  1 0; % diagonally adjacent point.
               1 -1 0];
        tmp1 = imerode(sk, el==1) & imerode(~sk, el==-1);
        tmp2 = imerode(sk, rot90(el)==1) & imerode(~sk, rot90(el)==-1);
        diagCount = size(find(tmp1 | tmp2), 1);

        % Wrap-up
        % strLen = (diagCount * 2^0.5 + (numpx - diagCount)) * r
        numpx = size(find(sk),1); % Count pixels in foreground
        if numstr ~= 3 && itr == 1, % store the length to px ratio
            str{1}.r = str{1}.len / (diagCount * 2^0.5 + (numpx - diagCount));

        % if third string use average of last 2 ratios to find length
        elseif numstr == 3 && itr == 1,
            str{1}.r = strs{1}{1}.r/2 + strs{2}{1}.r/2;
            str{1}.len = (diagCount * 2^0.5 + (numpx - diagCount)) * str{1}.r;
        else
            str{itr}.r = str{1}.r;
            str{itr}.len = (diagCount * 2^0.5 + (numpx - diagCount)) * str{itr}.r;
        end

        t = str{itr}.p; t(sk)=255;
        figure

        subplot(141), imshow(t);
        title(['String' int2str(numstr) '-#' int2str(itr)]);
        subplot(142), imshow(str{itr}.p);
        subplot(143), imshow(str{itr}.bw);
        title(['Pixel Length: ' int2str(numpx) ' - Est. Length: '
int2str(str{itr}.len)]);
        subplot(144), imshow(sk);
        print(['str' int2str(numstr) '-' int2str(itr)], '-djpeg90')
        disp(['Actual len of str#' int2str(itr) ' is:  ' int2str(str{itr}.len) ' -- d:'
int2str(diagCount)]);

    end

    strs{numstr} = str;
    avg = 0; for j = 1:5, avg = str{j}.len + avg; end; avg=avg/5;
    disp(['      Average Length: ' int2str(avg)]);
    print(['str' int2str(numstr) '-' int2str(itr)], '-djpeg90')
    clearvars -except strs numstr itr
    disp('.')
end

i = input('enterkey with no input to close all figures');
close all;
```