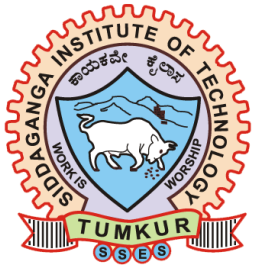
SIDDAGANGA INSTITUTE OF TECHNOLOGY, TUMAKURU-572103
(An Autonomous Institute under Visvesvaraya Technological University, Belagavi)



ADIP Project Report

on

"Multi-Thresholding Image Segmentation Using Genetic Algorithm"

submitted in partial fulfillment of the requirement for the completion of VII semester
of

BACHELOR OF ENGINEERING
in
ELECTRONICS & COMMUNICATION ENGINEERING

submitted by

Abhishek Kumar	1SI16EC004
Ankit Kumar Singh	1SI16EC010
Avinash M	1SI16EC011
Shubham Kumar Baranwal	1SI16EC093
Aquib Manzar	1SI16EC116

submitted to
Y Harshalatha
Assistant Professor
Department of E&CE
SIT, Tumakuru-03

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING
2019-20

1 Introduction

Segmentation is a fundamental task in image analysis and understanding. The goal of image segmentation is to partition an image into a set of separate regions that have similar characteristics such as intensity, color and texture. Due to the increasing amount of available data and the complexity of features of interest, it is becoming vital to advance automated segmentation methods to assist and speed-up image-processing and analysis tasks. The thresholding method is one of the most common methods for image segmentation. Several algorithms have been suggested for the bi-level and also for the multi-level thresholding problem including 'Otsu criterion' which is most adequate for bi-level thresholding but it is computationally expensive for mutli-level thresholding. To overcome the above mentioned problem, population based optimization algorithms has been put forward.

2 Genetic Algorithm

Genetic algorithms (GAs) are population based methods on principles of natural selection and genetics. One of the most successful areas of application has been the use of GAs to solve a wide variety of difficult numerical optimization problems. GAs complement existing optimization methods nicely in that they require no gradient information and are much less likely to get trapped in local minima on multimodal surfaces. In GAs, each possible solution within the population of a biological individual is coded in so called "chromosome" (i.e. individual). A number of chromosomes generate what is called a "population". The chromosomes share data with other, and each chromosome is assigned a fitness score according to how good a solution to the problem based on a given fitness function. The solutions are taken according to their fitness values and used to construct new solutions by a hope that the new solutions will be better than the old solutions and a generation is complete. Thus, the whole population moves like a one group towards an optimal area. The solutions can be evolved to the problem using the following steps:

1. *Initialization:* The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated.

2. *Evaluation:* Once the population is initialized or an offspring population. is created, the fitness values of the candidate solutions are evaluated.

3. *Selection:* Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection.

4. *Recombination:* Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this (some of which are discussed in the next section), and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner.

5. *Mutation:* While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.

6. *Replacement:* The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.

7. Repeat Steps 2–6 until a terminating condition is met.

3 Source Code

3.1 Initialization

ga_segmentation.m

```
% MULTI-THRESHOLDING IMAGE SEGMENTATION USING GENETIC ALGORITHMS
pkg load image
pkg load statistics
clc;
% Default variables
n_population =  100;
n_iterations =  25;
n_bins       = 256;
n_thresholds =  5;
% Ratios of all GA operations
p_selection = 0.1;
p_crossover = 0.8;
p_mutation  = 0.1;
assert(sum([p_selection, p_crossover, p_mutation]) == 1, 'Total sum of
    proportions have to be 1!');
% Read image
image = imread("images/MRI-of-knee-Univ-Mich.tif");
%image = imread("images/tungsten.png");
disp("loading image");
% Convert image to gray levels
if (size(image, 3) == 3)
    image_gray = rgb2gray(image);
else
    image_gray = image;
endif
% Display image
figure(1),subplot(2,2,1),imshow(image),title('Original Image');
figure(1),subplot(2,2,2),imhist(image),title('Histogram');
```

```
28 % Initialization
29 population = initialization(n_population, n_bins, n_thresholds);
30 printf("Running algorithm");
31 for i = 1:n_iterations
32     new_population = [];
33     printf(".");
34     % Evaluation of fitness
35     ranking = fitness(image, population, n_thresholds);
36     %% Reproduction
37     % Selection
38     % TODO create more strategies (like roulette wheel)
39     new_population = first_best(ranking, population, p_selection,
           new_population);
40     % Crossover
41     new_population = crossover(population, p_crossover, new_population
           );
42     % Mutation
43     new_population = mutation(population, p_mutation, new_population);
44     population = new_population;
45 endfor
46 % Accepting the solution
47 accept_solution(image_gray, population, n_thresholds);
```

initialization.m

```
1 % Randomly generates the first population.
2 function population = initialization(n_population, n_bins,
      n_thresholds)
3     population = round(unifrnd(0, 1, [n_population ceil(log2(n_bins))
          *n_thresholds]));
4 endfunction
```

3.2 Fitness Function

fitness.m

```
% Computes fitness ranking for all population.
function ranking = fitness(image, population, n_thresholds)
    ranking = [];
    % Convert thresholds to decimal representation
    thresholds =  convert_thresholds(population, n_thresholds);
    % Vectorize image
    image_vec = image(:);
    % Computes fitness ranking for all thresholds in population
    for i = 1:size(thresholds, 1)
        ranking = [ranking; fitness_one(image_vec, thresholds(i,:))];
    endfor
endfunction
```

convert_thresholds.m

```
% Converts binary thresholds of all population to decimal
   representation.
function thresholds = convert_thresholds(population, n_thresholds)
    thresholds = [];
    population_size = size(population, 1);
    for i = 1:population_size
        thresholds = [thresholds; threshold_bin2dec(population(i,:),
           n_thresholds)];
    endfor
endfunction
```

threshold_bin2dec.m

```
% Converts thresholds of one genome from binary to decimal format.
function dec_thresholds = threshold_bin2dec(bin_thresholds,
   n_thresholds)
```

```
    dec_thresholds = [];
    threshold_length = (size(bin_thresholds, 2)/n_thresholds);
    for i = 1:threshold_length:size(bin_thresholds, 2)
        dec_thresholds = [dec_thresholds, bi2de(bin_thresholds(i:i+
            threshold_length-1))];
    endfor
endfunction
```

bi2de.m

```
% Binary to Decimal
function d = bi2de (b, p, f)
  switch (nargin)
    case 1,
      p = 2;
      f = 'right-msb';
     case 2,
      if (isstr(p))
        f = p;
        p = 2;
      else
        f = 'right-msb';
      endif
     case 3,
      if (isstr(p))
        tmp = f;
        f = p;
        p = tmp;
      endif
   otherwise
      error ("usage: d = bi2de (b, [p])");
  endswitch
  if ( any (b (:) < 0) || any (b (:) > p - 1) )
```

```
24      error ("bi2de: d must only contain value in [0, p-1]");
25    endif
26    if (strcmp(f,'left-msb'))
27      b = b(:,size(b,2):-1:1);
28    elseif (!strcmp(f,'right-msb'))
29      error("bi2de: unrecognized flag");
30    endif
31    if (length (b) == 0)
32      d = [];
33    else
34      d = b * ( p .^ [ 0 : (columns(b)-1) ]' );
35    endif
36  endfunction;
```

fitness_one.m

```
1  % Computes fitness ranking for given chromosome.
2  function ranking = fitness_one(image_vec, thresholds_vec)
3      ranking = 1;
4      inter_var = 0;
5      intra_var = 0; % TODO implement
6      % Sort thresholds
7      thresholds_vec = sort(thresholds_vec);
8      end_i = size(thresholds_vec, 2) + 1;
9      for i = 1:end_i
10         if ((i == 1 && end_i == 2) || i == 1)
11         % One threshold or the first threshold
12             left = 0;
13             right = thresholds_vec(i);
14         elseif (i == end_i)
15         % The last threshold
16             left = thresholds_vec(i-1);
17             right = max(image_vec);
```

```
18          else
19          % More thresholds
20              left = thresholds_vec(i-1);
21              right = thresholds_vec(i);
22          endif
23              % <0; x) <x; y) <y; max(image_vec))
24              left_mask = image_vec >= left;
25              right_mask = image_vec < right;
26              mask = left_mask .* right_mask;
27              object = image_vec(find(mask));
28          % TODO better way to relate all variances within objects?
29          if (length(object) == 0)
30              variance = 1;
31          else
32              variance = var(object);
33          endif
34          ranking = ranking + variance;
35      endfor
36  endfunction
```

3.3 Reproduction

3.3.1 Selection

first_best.m

```
1  % Selecting the first best solutions from current population and
2  % transfering to new population.
3  function new_population = first_best(ranking, population, p_selection,
       new_population)
4      population_size = size(population, 1);
5      [best, best_i] = sort(ranking);
6      for i = 1:round(p_selection*population_size)
```

```
7           new_population = [new_population; population(best_i(i), :)];
8       endfor
9   endfunction
```

3.3.2 Crossover

crossover.m

```
1  % Crossovers desired part of population.
2  function new_population = crossover(population, p_crossover,
       new_population)
3       population_size = size(population, 1);
4       % Random permutation of genomes order
5       parent_first = randperm(population_size);
6       parent_second = randperm(population_size);
7       % Number of couples used for crossover
8       n_crossovers = round(p_crossover*population_size)/2;
9       for i = 1:n_crossovers
10          % Crossovers parents
11          [desc_first desc_second] = crossover_one(population(
                parent_first(i), :), ...
12              population(parent_second(i), :));
13          % Add crossover descendants
14          new_population = [new_population; desc_first; desc_second];
15      endfor
16  endfunction
```

crossover_one.m

```
1  % Crossover of two parents creating new descendants by one-point
       crossover.
2  function [desc_first desc_second] = crossover_one(parent_first,
       parent_second)
```

```
3       parent_size = size(parent_first, 2);
4       % Randomly generated number between 1 and the length of parent's
            genome.
5       point = round(unifrnd(1, parent_size-1));
6       % Crossover
7       desc_first  = [parent_first(1:point)    parent_second(point+1:
            parent_size)];
8       desc_second = [parent_second(1:point)   parent_first(point+1:
            parent_size)];
9   endfunction
```

3.3.3 Mutation

mutation.m

```
1   %Mutation
2   function new_population = mutation(population, p_mutation,
        new_population)
3       population_size = size(population, 1);
4       % Random permutation of genomes order
5       mutation_order = randperm(population_size);
6       for i = 1:round(p_mutation*population_size);
7           new_population = [new_population; mutate_one(population(
                mutation_order(i), :))];
8       endfor
9   endfunction
```

mutate_one.m

```
1   % Mutates given chromose at one randomly generated position.
2   function new_chromosome = mutate_one(chromosome)
3       new_chromosome = chromosome;
4       chromosome_size = size(chromosome, 2);
```

```
    gene = round(unifrnd(1, chromosome_size));
    % Mutate one gene
    if (chromosome(gene) == 1)
        new_chromosome(gene) = 0;
    else
        new_chromosome(gene) = 1;
    endif
endfunction
```

3.4 Multiple Thresholding

accept_solution.m

```
% Employs the best solution and displays segments of examined image.
function accept_solution(image, population, n_thresholds)
    segmentation = zeros(size(image));
    segmentation_value = 1/n_thresholds;
    genome_size = size(population, 2)/n_thresholds;
    % Retrieve the best solution
    [b, b_i] = sort(fitness(image, population, n_thresholds));
    best_genome = population(b_i(1),:);
    threshold = sort(threshold_bin2dec(best_genome, n_thresholds));
    printf("\nThresholds:\n");
    disp(threshold);
    value = 0;
    end_i = size(threshold, 2) + 1;
    for i = 1:end_i
        if (i == 1)
        % The first threshold
            left = 0;
            right = threshold(i);
        elseif (i == end_i)
```

```
20          % The last threshold
21              left = threshold(i-1);
22              right = max(image(:));
23          else
24          % Regular threshold
25              left = threshold(i-1);
26              right = threshold(i);
27          endif
28          % <0; x) <x; y) <y; max(image))
29          left_mask = image >= left;
30          right_mask = image < right;
31          mask = left_mask .* right_mask;
32          segmentation += value*mask;
33          %Display segments
34          if (i >= 2)
35              figure
36              mask_value = value*mask;
37              figure(i),subplot(1,2,1),imshow(mask_value),title("
                    Thresholded segment ");
38              figure(i),subplot(1,2,2),imhist(mask_value),title('
                    Histogram');
39             %imwrite(mask_value, strcat(num2str(i), ".png"));
40          endif
41          value += segmentation_value;
42      endfor
43      figure(1),subplot(2,2,3),imshow(segmentation),title('Multi-
            threshold segmented Image');
44      figure(1),subplot(2,2,4),imhist(segmentation),title('Histogram');
45  endfunction
```

4 Output

The output is taken on a grayscale gradient image with the genetic algorithm configuration as

- n_population = 50;
- n_iterations = 18;
- n_bins = 256;
- n_thresholds = 10;

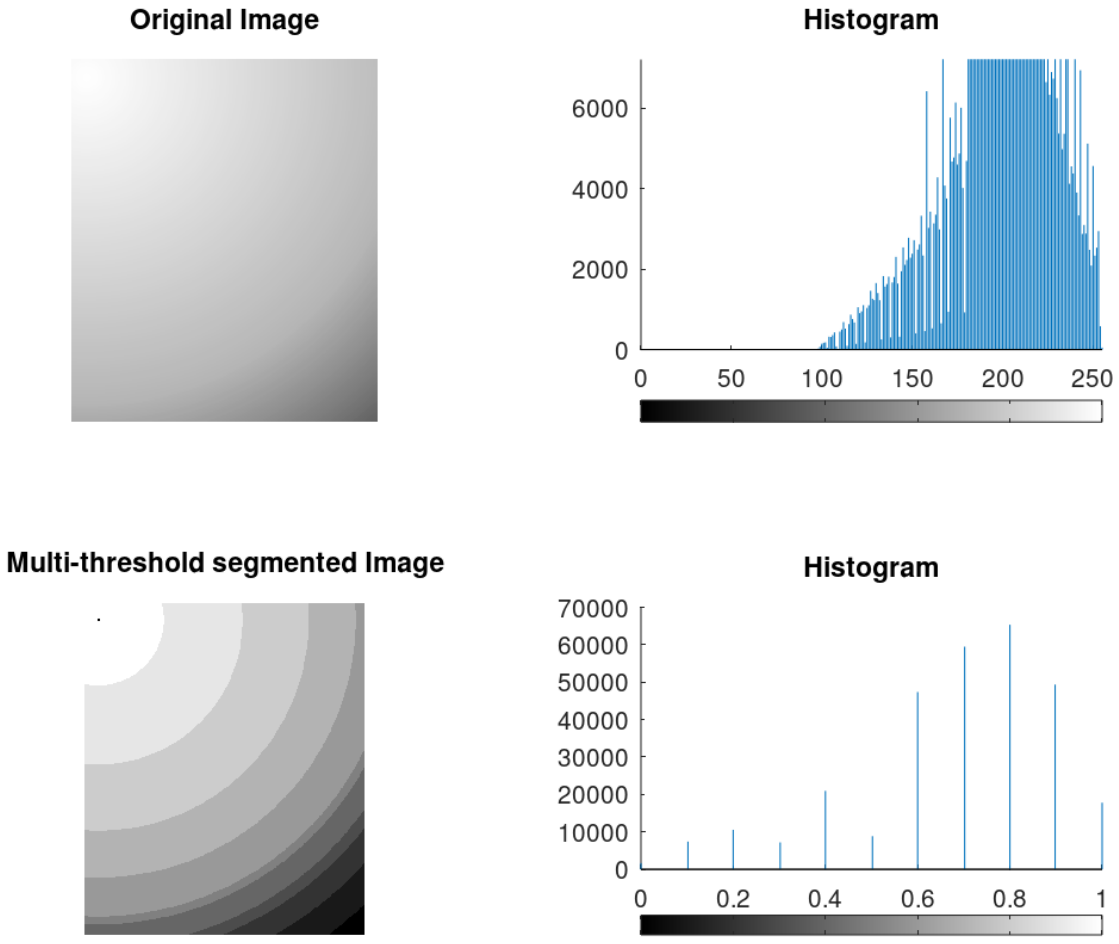


Figure 1: Original and multi-thresholded image with histograms

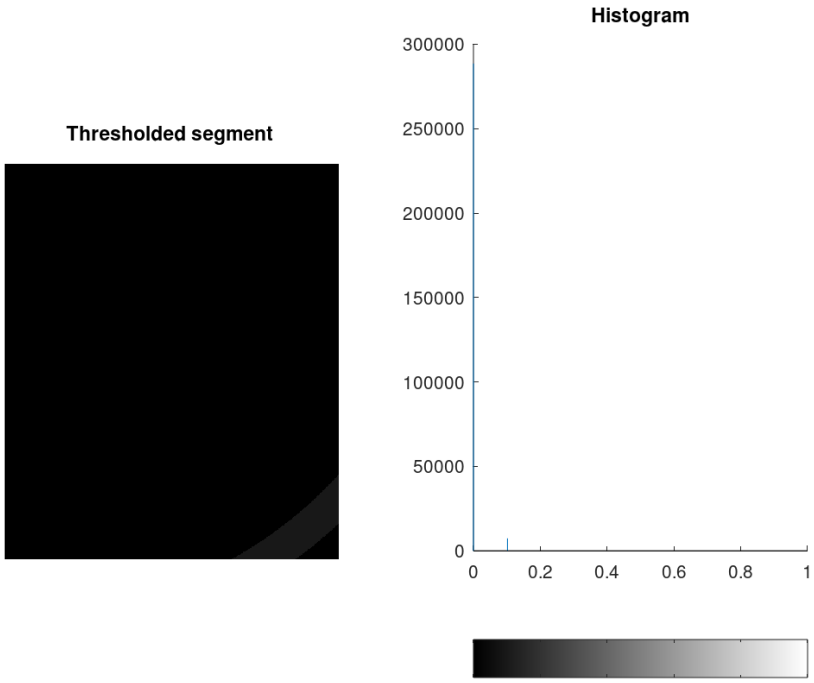


Figure 2: Threshold segment for threshold value = 116

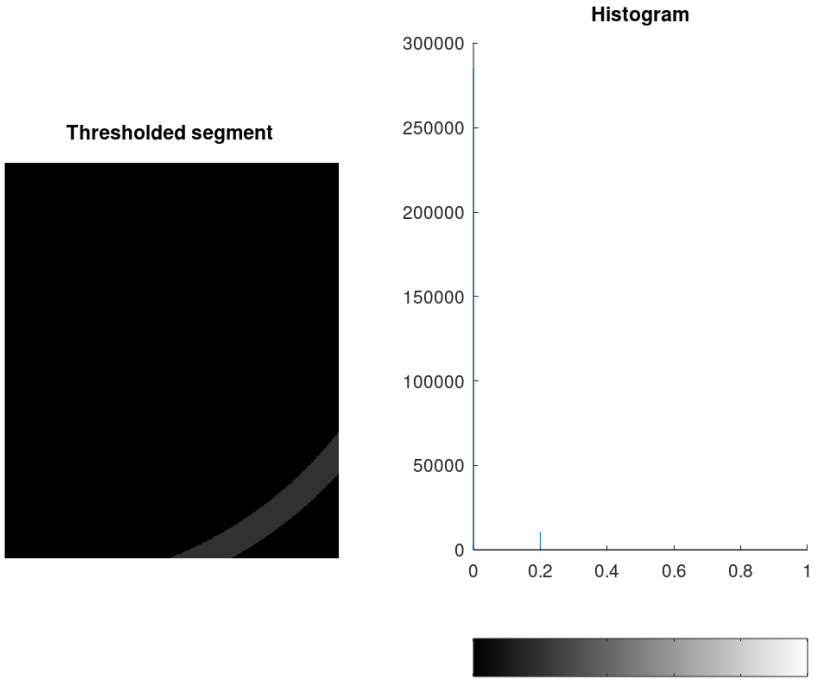


Figure 3: Threshold segment for threshold value = 137

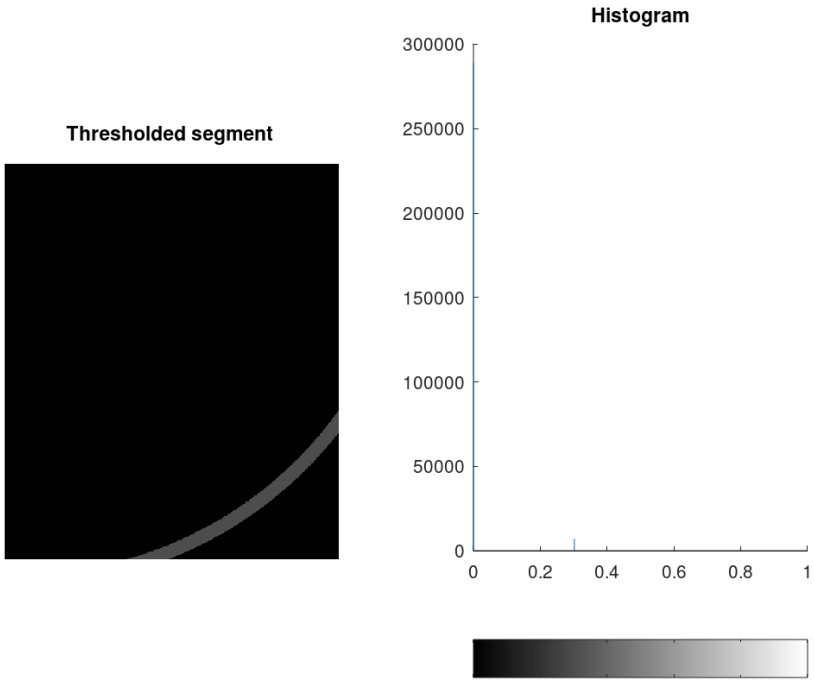


Figure 4: Threshold segment for threshold value = 154

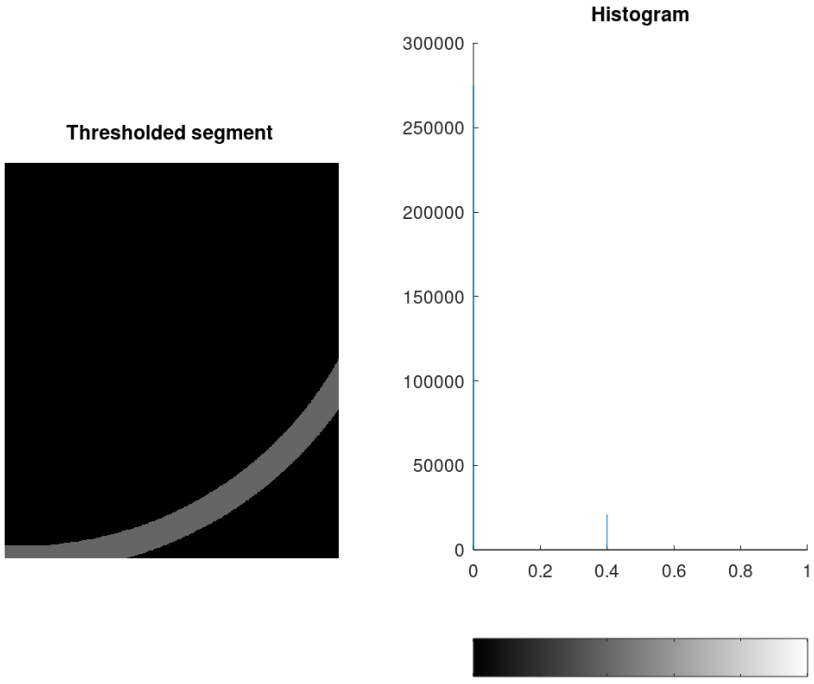


Figure 5: Threshold segment for threshold value = 161

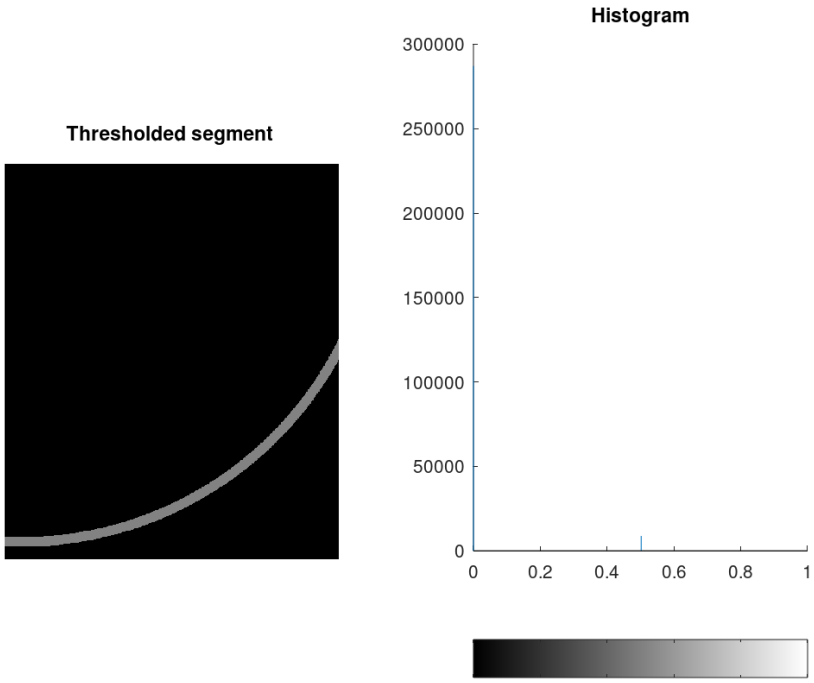


Figure 6: Threshold segment for threshold value = 177

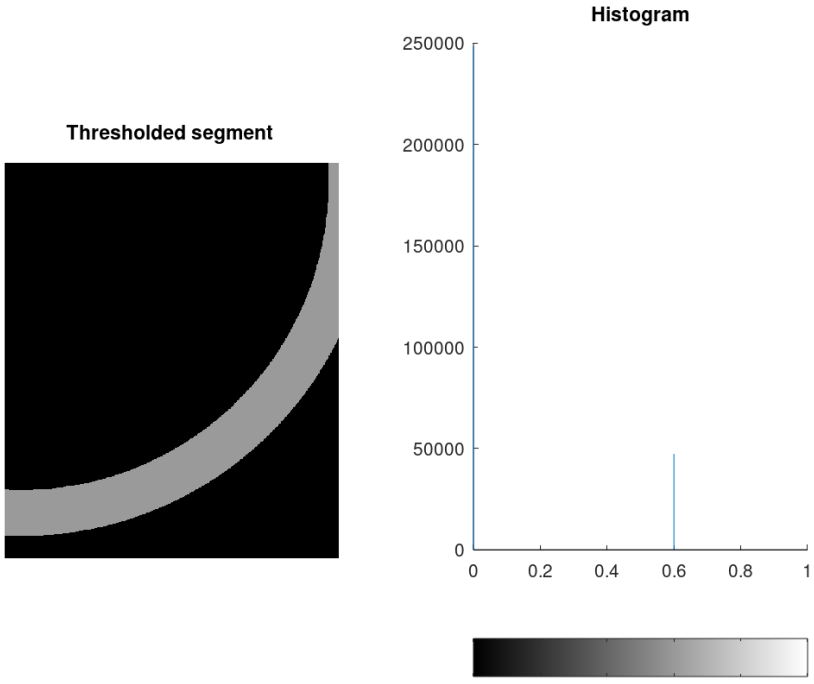


Figure 7: Threshold segment for threshold value = 182

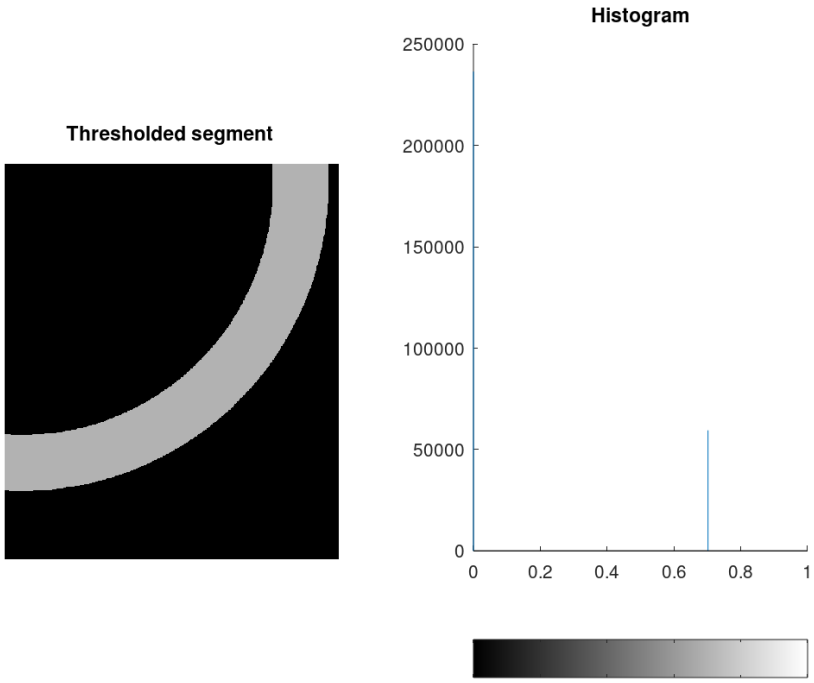


Figure 8: Threshold segment for threshold value = 193

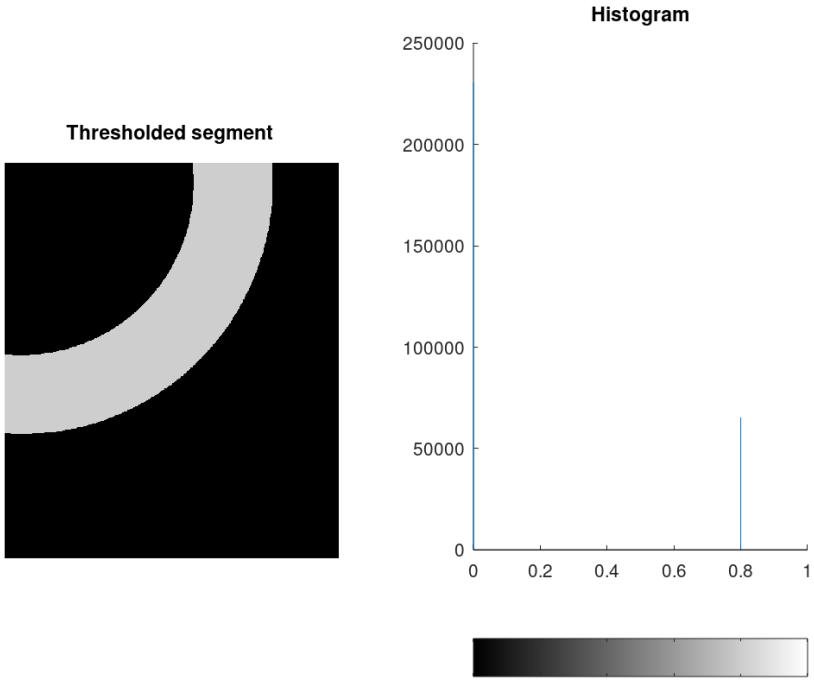


Figure 9: Threshold segment for threshold value = 204

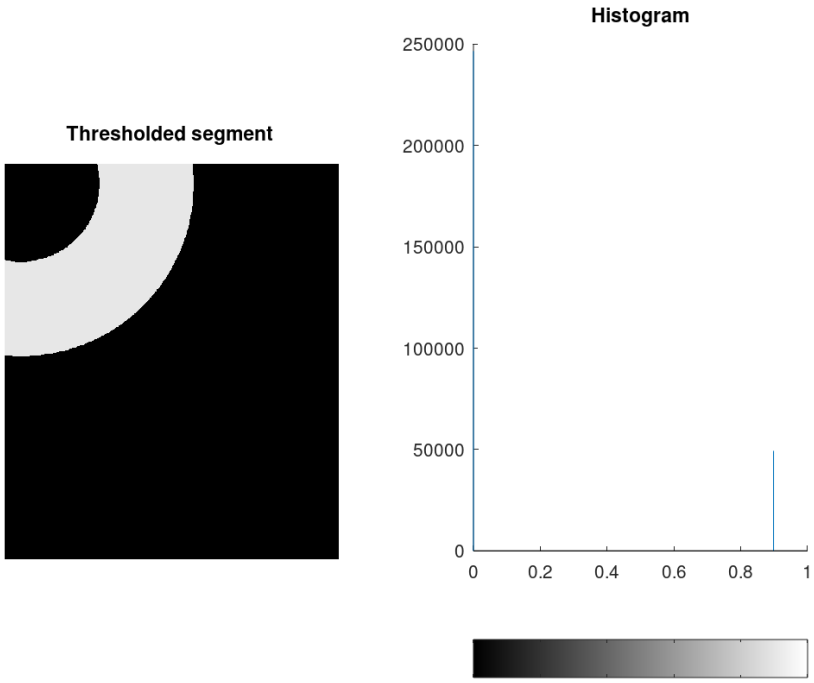


Figure 10: Threshold segment for threshold value = 221

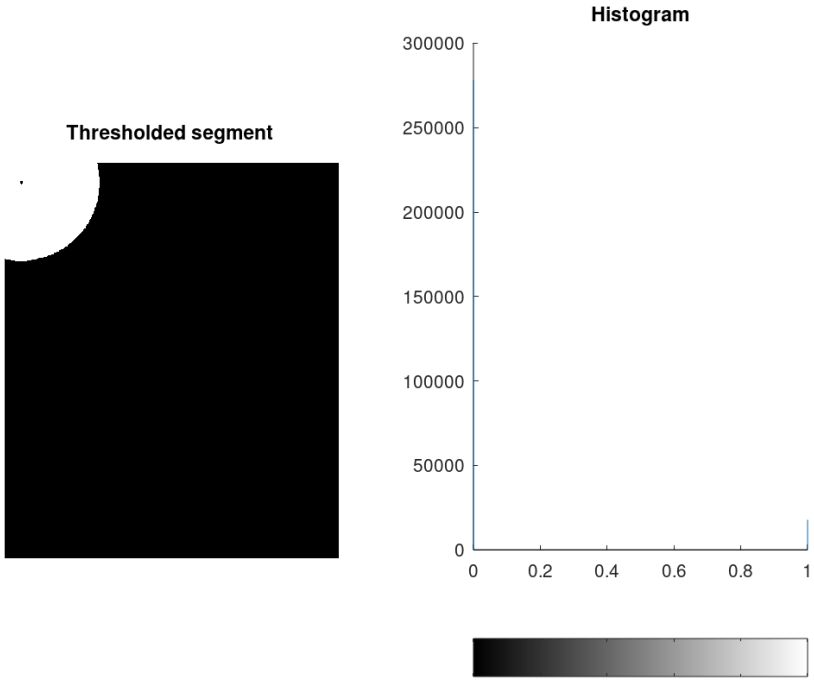


Figure 11: Threshold segment for threshold value = 240