KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Problem Solving and Engineering Design part 3

ESAT1A1

Max Beerten
Brent De Bleser
Wouter Devos
Ben Fidlers
Simon Gulix
Tom Kerkhofs

Counting and recognizing non-moving objects by means of image processing

MATLAB CODE

Co-titular
Tinne Tuytelaars

Coaches
Xuanli Chen
José Oramas

A C A D E M I C Y E A R 2 0 1 8 - 2 0 1 9

Declaration of originality

We herebydeclare that thissubmitted draft isentirely ourown, subject tofeedback andsupportgiven usbythe didacticteam, and subjectto lawful cooperation which was agreed with thesamedidacticteam. Regarding this draft, we also declare that:

1. *Note has been taken of the text on academic integrity (https://eng.kuleuven.be/studeren/masterproef-en-papers/documenten/20161221-academischeintegriteit-okt2016.pdf).*
2. *No plagiarism has been committed as described on https://eng.kuleuven.be/studeren/masterproef-en-papers/plagiaat.*
3. *All experiments, tests, measurements, …, have been performed as described in this draft, and no data or measurement results have been manipulated.*
4. *Allsourcesemployedinthisdraft-includinginternetsources-havebeencorrectlyreferenced.*

Contents

1 Initialising functions

```
1
2  h = 900;
3
4  min_y = 120;
5  max_y = 460;
6  min_x = 75;
7  max_x = 310;
8
9  % Threshold values
10 min_thresh = 30;
11 max_thresh = 500;
12
13 % Get image from depth sensor
14 colorVid = videoinput('kinect',1);
15 depthVid = videoinput('kinect',2);
16 depth = getsnapshot(depthVid);
17 color = getsnapshot(colorVid);
18 raw_matrix = depth;
19 %%
20 %Run the sobel operator
21
22 depth = sobel_operator(depth);
23 shapes_after_sobel = depth;
24 %Run the threshold filter
25 depth = threshold(depth, min_thresh, max_thresh);
26 depth = print(depth, min_x, max_x, min_y, max_y);
27 depth_after_threshold = depth;
28
29 %%%%%%outline
30 depth = outline(depth);
31 final_img = only_outline_visible(depth);
32 edged_matrix = only_edge(depth);
33
34 new_depth = crop_depth_to_basket(edged_matrix, depth_after_threshold);
35 depth_tester = new_depth;
36
37 %OVERLAP
38 %%%%%%%%%%%%%%%%%%%%%%%%%
39
40 %color: 1920x1080 met 84.1 x 53.8
41 %depth: 512x424  met 70.6 x 60
42
43 [reformed_depth,reformed_color, res_height_angle, res_width_angle] =
       reform(depth, color);
44 [pipemm_depth_H, pipemm_depth_W, pipemm_color_H, pipemm_color_W] =
       get_pipemm(res_height_angle, res_width_angle, h, reformed_depth,
       reformed_color);
45
46
47 [prop,nb_rows_color , nb_columns_color,nb_rows_depth, nb_columns_depth] =
        proportion(reformed_depth , reformed_color);
48
49 tot_size = size_matching(prop);
50
```

```
51 total = overlap_depth_to_RGB(reformed_depth, reformed_color,
       pipemm_depth_H , pipemm_depth_W , pipemm_color_H , pipemm_color_W,
       tot_size,nb_rows_color , nb_columns_color);
52
53 new_RGB = crop_RGB_to_basket(total);
54 image(new_RGB);
55
56 img = new_RGB;
57
58 THRESHOLD_VALUE = 2;
59
60 MIN_ROW_LINES_BETWEEN_GROUPS = 10; %25 %15
61 % Once the groups are found, the algorithm searches for groups too close
62 % near each other
63 % This is defined as the min distance between two groups (only searched
64 % vertical)
65 SAME_PIXELS_SEARCH_GRID_SIZE = 10;%25
66 % Grid size = this variable *2, it searches for pixels with the same
       value
67 % in this grid.
68 GROUP_SEARCH_GRID_SIZE = 15; %25
69 % Grid size = this variable * 2, it searches for pixels with a group
       number
70 % (not 0) in this grid.
71 SURROUDING_PERCENTAGE = 10;% %
72 MIN_NB_SURROUNDING_PIXELS = floor((SAME_PIXELS_SEARCH_GRID_SIZE * 2)^2 *
       SURROUDING_PERCENTAGE/100) ;%125 % 50
73 % The minimum number of pixels with the same value that are in the grid
74 % size defined by SAME_PIXELS_SEARCH_GRID
75 % The pixels that have a less number of surrounding pixels, are not
       defined
76 % as a group but as noise.
77
78 % CROPPING: Defining rectangle
79 %top_row = 290 ; top_col = 760; bottom_row = 690 ; bottom_col = 1440;
80 %top_row = 150; top_col = 750; bottom_row = 950; bottom_col = 1900;
81 %top_row = 200; top_col = 850; bottom_row = 750; bottom_col = 1850; % For
       pictues with x2_RGB_... in name
82 %top_row = 100, top_col = 100; bottom_row = 980; bottom_col = 1820; % For
       pictues with RGB in name.
83
84 disp("Step 1: loading the image...");
85 disp("Minimum distance between 2 objects (only straight vertical or
       straight horizontal = " + max([MIN_ROW_LINES_BETWEEN_GROUPS
       SAME_PIXELS_SEARCH_GRID_SIZE MIN_NB_SURROUNDING_PIXELS;]) + " pixels")
       ;
86
87 disp("Step 2: converting the image to greyscale...");
88
89 A = greyscale(img); % Convert image to grayscale
90
91 %top_left_row, top_left_col, bottom_right_row, bottom_right_col
92 disp("Step 3: cropping the image...");
93
94 %A = simon_crop(A, top_row, top_col, bottom_row, bottom_col);
95 imshow(A, []);
```

```
%%
%A = simon_crop(A, 100,100,980,1820, 1); % USE FOR foto RGB X
%A = simon_crop(A, top_row,top_col,bottom_row, bottom_col,1); % USE FOR foto XX RGB

disp("Step 4: blurring the image...");
A = gaussian_blur(mean_blur(A)); % Filters
% Method 3: First greyscale, then blur, then edge detect then threshold and then noise removal
disp("Step 5: edge detecting...");
first_edge_detect = edge_detect(A); % Laplacian edge detection
disp("Step 6: thresholding edge");
without_noise_removal = threshold_edge(remove_boundary(first_edge_detect, 15), THRESHOLD_VALUE); % Remove boundary around image & threshold the edges.
disp("Step 7: noise removing...");
%with_noise_removal = noise_deletion(without_noise_removal,5); % Noise removal
with_noise_removal = without_noise_removal;
disp("Step 8: grouping...");
[grouped, nb_of_groups] = group(~with_noise_removal, SAME_PIXELS_SEARCH_GRID_SIZE, GROUP_SEARCH_GRID_SIZE, MIN_NB_SURROUNDING_PIXELS); % Group pixels together

disp("Step 9: regrouping...");
[regrouped, nb_of_groups2] = regroup(grouped, nb_of_groups, MIN_ROW_LINES_BETWEEN_GROUPS); % Regroup (nessicary because group function works from top left to bottom right

%Find corner points of object (not really corner points on the boundary,
%but corner points for the boundary box)
disp("Step 10: calculating corner points...");
corner_points = find_corner_points(regrouped, nb_of_groups); % Make sure to use nb_of_groups and not groups 2 because some groups don't exist anymore!

disp("Step 11: removing objects within objects...");
%[updated_corner_points, nb_of_groups3] = remove_corner_points_within_corner_points(corner_points, nb_of_groups2); % To remove objects within objects
[updated_corner_points, nb_of_groups3] = remove_box_edge(corner_points, nb_of_groups2);
[updated_corner_points, nb_of_groups3] = remove_corner_points_within_corner_points(updated_corner_points, nb_of_groups3);
%updated_corner_points = corner_points;
%nb_of_groups3 = nb_of_groups2;

disp("Step 12: drawing boundary boxes...");
boundary_box = draw_boundary_box(A, updated_corner_points);
disp("Step 13: drawing red boundary boxes on full image...");
red_boundary_box = draw_red_boundary_box(reformed_color, updated_corner_points, 1,1);
disp("Step 13: Done!!!");
imshow(red_boundary_box, []);
title("# objects: "+ nb_of_groups3);
```

```
%%
% Starting the packaging pocess

% Gathering every object from the original image
objects = get_objects(updated_corner_points, gray_image, regrouped);
% Creating the total package
total_package = smallest_package(objects);

% Showing the end package
imshow(total_package,[]);
```

2 Functions for depth

2.1 Sobel operator

```
function shapes = sobel_operator(img)
    % use the sobel-operator on the raw depth image
    % this function returns a matrix of the same size as the original
    % matrix with on every position the gradint

    X = img;
    Gx = [1 +2 +1; 0 0 0; -1 -2 -1]; Gy = Gx';
    temp_x = conv2(X, Gx, 'same');
    temp_y = conv2(X, Gy, 'same');
    shapes = sqrt(temp_x.^2 + temp_y.^2);
end
```

2.2 Threshold for depth

2.2.1 threshold in values

```
function thresholded = threshold(img, min_thresh, max_thresh)
    % run the image through a threshold to get rid of impossible values
    % this function returns a binary matrix with a 1 on the edges

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    for row = 1 : MAX_ROW
        for col = 1: MAX_COLUMN
            if (img(row, col) > min_thresh) && (img(row, col)< max_thresh)
                img(row, col) = 1;
            else
                img(row, col) = 0;
            end
        end
    end
    thresholded = img;
end

function printed = print(img, min_x, max_x, min_y, max_y)
    % this function uses a threshold to cut of part of the edges to get
        rid
    % of noise that appears in every image and replace them by '0'
```

```
    % it returns a binary image

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    mat = zeros(MAX_ROW,MAX_COLUMN,1);

    for row = 1:MAX_ROW

        for col = 1: MAX_COLUMN
            if (row>min_x) && (row<max_x) && (col> min_y) && (col<max_y)
                mat(row, col) = img(row, col);
            end
        end
    end
    printed = mat;
end
```

2.2.2 threshold in edges

```
function printed = print(img, min_x, max_x, min_y, max_y)
    % this function uses a threshold to cut of part of the edges to get
        rid
    % of noise that appears in every image and replace them by '0'
    % it returns a binary image

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    mat = zeros(MAX_ROW,MAX_COLUMN,1);

    for row = 1:MAX_ROW

        for col = 1: MAX_COLUMN
            if (row>min_x) && (row<max_x) && (col> min_y) && (col<max_y)
                mat(row, col) = img(row, col);
            end
        end
    end
    printed = mat;
end
```

2.3 Outline objects

2.3.1 Main outline

```
  function outlined_matrix = outline(img)
    % the main outline function, given a binary matrix, this function
    % outlines every shape defined by '1'
    % it returns a matrix with '-1' as value for the outlines


```

```
    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    x = 0;

    for row = 1: MAX_ROW
        col = 1;
        while col <= MAX_COLUMN
             position = img(row, col);
            if position == 0
                col = col + 1;
            elseif position == -1
                col = skip(img, row, col, MAX_COLUMN);
            elseif position == 1
                x = x + 1;
                img = outline_shape(img, row, col-1, MAX_ROW, MAX_COLUMN)
                    ;
                col = col - 1;
            end
        end
    end
    disp(x);
    outlined_matrix = img;
end
```

2.3.2 Skip column

```
function new_col = skip(img, row, col, MAX_COLUMN)
    % this function skips the part of the row that is defined to be
        inside
    % a shape
    % it returns the first column number outside a shape

    good_value = 0;
    while (good_value ~= 1) && (col < MAX_COLUMN)
        col = col+ 1;
        if img(row, col) == -1
            good_value = 1;
        end
    end
    new_col = col +1;
end
```

2.3.3 Outline the shape

```
function outlined_objects = outline_shape(img, row, col, MAX_ROW,
    MAX_COLUMN)
    %Given a binary matrix and a position that is connected to a '1',
        this
    %recursive function outlines the object and returns a matrix with the
    %value '-1' surrounding the object

    img(row, col) = -1;
    matrix = surrounded_matrix(img, row, col, MAX_ROW, MAX_COLUMN);
    for i = 1:3
```

```
        for j = 1:3
            if (matrix(i, j, 1) == 0) & (connected_to_one(img, matrix(i,j
                ,2), matrix(i,j,3), MAX_ROW, MAX_COLUMN) == 1)
                img = outline_shape(img, matrix(i,j,2), matrix(i,j,3),
                    MAX_ROW, MAX_COLUMN);
            end
        end
    end
    outlined_objects = img;
end
```

2.3.4 Check if a one is connected

```
function is_connected_to_one = connected_to_one(img, row, col, MAX_ROW,
    MAX_COLUMN)
    % given a position that is equal to '0', this function checks in a
    % cross shape if a '1' is present

    position = [row, col];
    T = top(position, img, MAX_ROW, MAX_COLUMN);
    R = right(position, img, MAX_ROW, MAX_COLUMN);
    B = bottom(position, img, MAX_ROW, MAX_COLUMN);
    L = left(position, img, MAX_ROW, MAX_COLUMN);

    matrix = [0, T(1), 0; L(1), -1, R(1); 0, B(1), 0];
    is_connected  = 0;
    for i = 1:3
        for j = 1:3
            if matrix(i,j) == 1
                is_connected = 1;
            end
        end
    end
    is_connected_to_one = is_connected;


end
```

2.3.5 Create surrounding matrix

```
function created_matrix = surrounded_matrix(img, row, col, MAX_ROW,
    MAX_COLUMN)
    % given a position in a matrix, this matrix returns the value and
    % position of the 9 surrounding positions

    position = [row, col];
    TL = top_left(position, img, MAX_ROW, MAX_COLUMN);
    T = top(position, img, MAX_ROW, MAX_COLUMN);
    TR = top_right(position, img, MAX_ROW, MAX_COLUMN);
    R = right(position, img, MAX_ROW, MAX_COLUMN);
    BR = bottom_right(position, img, MAX_ROW, MAX_COLUMN);
    B = bottom(position, img, MAX_ROW, MAX_COLUMN);
    BL = bottom_left(position, img, MAX_ROW, MAX_COLUMN);
    L = left(position, img, MAX_ROW, MAX_COLUMN);

    matrix_1 = [TL(1), T(1), TR(1); L(1), -1, R(1); BL(1), B(1), BR(1)];
    matrix_2 = [TL(2), T(2), TR(2); L(2), row, R(2); BL(2), B(2), BR(2)];
    matrix_3 = [TL(3), T(3), TR(3); L(3), col, R(3); BL(3), B(3), BR(3)];
```

```
18
19      matrix_total = matrix_1;
20      matrix_total(:,:,2) = matrix_2;
21      matrix_total(:,:,3) = matrix_3;
22
23      created_matrix = matrix_total;
24
25  end
```

2.3.6 all surrounding positions

```
1   function placing = top_left(position, img, MAX_ROW, MAX_COLUMB)
2       % returns the position top left of the given position
3       x = position(1) -1;
4       y = position(2) -1;
5
6       if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
7
8           value = img(x, y);
9           placing = [value, x, y];
10      else
11
12          value = -2;
13          placing = [value, x, y];
14      end
15  end
16  function placing = top(position, img, MAX_ROW, MAX_COLUMB)
17      % returns the position above the given position
18      x = position(1) -1;
19      y = position(2) ;
20
21      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
22          value = img(x, y);
23          placing = [value, x, y];
24      else
25
26          value = -2;
27          placing = [value, x, y];
28      end
29  end
30  function placing = top_right(position, img, MAX_ROW, MAX_COLUMB)
31      % returns the position top right of the given position
32      x = position(1) - 1;
33      y = position(2) + 1;
34
35      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
36
37          value = img(x, y);
38          placing = [value, x, y];
39      else
40
41          value = -2;
42          placing = [value, x, y];
43      end
44  end
45  function placing = right(position, img, MAX_ROW, MAX_COLUMB)
46      % returns the position to the right of the given position
```

```
47      x = position(1) ;
48      y = position(2) +1;
49
50      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
51
52          value = img(x, y);
53          placing = [value, x, y];
54      else
55
56          value = -2;
57          placing = [value, x, y];
58      end
59  end
60  function placing = bottom_right(position, img, MAX_ROW, MAX_COLUMB)
61      % returns the position bottom right of the given position
62      x = position(1) +1;
63      y = position(2) +1;
64
65      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
66
67          value = img(x, y);
68          placing = [value, x, y];
69      else
70
71          value = -2;
72          placing = [value, x, y];
73      end
74  end
75  function placing = bottom(position, img, MAX_ROW, MAX_COLUMB)
76      % returns the position below the given position
77      x = position(1) +1;
78      y = position(2) ;
79
80      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
81
82          value = img(x, y);
83          placing = [value, x, y];
84      else
85
86          value = -2;
87          placing = [value, x, y];
88      end
89  end
90  function placing = bottom_left(position, img, MAX_ROW, MAX_COLUMB)
91      % returns the position bottom left of the given position
92      x = position(1) +1;
93      y = position(2) -1;
94
95      if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)
96
97          value = img(x, y);
98          placing = [value, x, y];
99      else
100
101         value = -2;
102         placing = [value, x, y];
```

```
    end
end
function placing = left(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position to the left of the given position
    x = position(1);
    y = position(2) -1;

    if (0 < x) && (x <= MAX_ROW) && (0 < y) && (y <= MAX_COLUMB)

        value = img(x, y);
        placing = [value, x, y];
    else

        value = -2;
        placing = [value, x, y];
    end
end
```

3 Functions for overlap

3.1 Get the needed values

3.1.1 Crop depth and RGB to the same aspect ratio

```
function [reformed_depth,reformed_color,resulting_height_angle,
    resulting_width_angle] = reform(depth, color) %met h= height camera
    % this function modifies the incomming color and depth matrices to
        give
    % them the same aspect ratio

    %breedte van color naar 70.6 brengen
    width_color_angle = 84.1;
    height_color_angle = 53.8;

    width_depth_angle = 70.6;
    height_depth_angle = 60;

    resulting_height_angle = height_color_angle;
    resulting_width_angle = width_depth_angle;

    [~, nb_columns_color,~]=size(color);
    nb_pixels_color_per_degree_width = nb_columns_color /
        width_color_angle;


    nb_width_pixels_removed_color = (width_color_angle-width_depth_angle)
        * nb_pixels_color_per_degree_width ;
        %totaal aantal pixels dat in de breedte weggehaald moeten worden
            bij color

    reformed_color = color(:,80 + round(nb_width_pixels_removed_color
        /2,0): round(nb_columns_color-(nb_width_pixels_removed_color/2),0)
        ,:);
        %Dit is een 1080 x (aangepaste breedte) matrix


    % hoogte van depth naar 53.8 brenge
```

```
27      [nb_rows_depth,~]=size(depth);
28
29      nb_pixels_depth_per_degree_height = nb_rows_depth /
            height_color_angle;
30
31      nb_height_pixels_removed_depth = (height_depth_angle-
            height_color_angle)*nb_pixels_depth_per_degree_height;
32      reformed_depth = depth(round(nb_height_pixels_removed_depth/2,0):
            round(nb_rows_depth -(nb_height_pixels_removed_depth/2),0),:);
33  end
```

3.1.2 Get the pixels per mm

```
1   function [pipemm_depth_H, pipemm_depth_W, pipemm_color_H, pipemm_color_W]
        = get_pipemm(res_height_angle, res_width_angle, h, reformed_depth,
        reformed_color)
2       % this function returns the pixels per millimeter for the given depth
3       % and color matrices
4
5       depth_size = size(reformed_depth);
6
7       MAX_ROW_DEPTH = depth_size(1);
8
9       MAX_COLUMN_DEPTH = depth_size(2);
10
11      color_size = size(reformed_color);
12
13      MAX_ROW_COLOR = color_size(1);
14
15      MAX_COLUMN_COLOR = color_size(2);
16
17      tot_width = 2*h*tan(((res_width_angle)/2)*(pi/180));
18
19      tot_height = 2*h*tan(((res_height_angle)/2)*(pi/180));
20
21      pipemm_depth_H = MAX_ROW_DEPTH/tot_height;
22
23      pipemm_depth_W = MAX_COLUMN_DEPTH/tot_width;
24
25      pipemm_color_H = MAX_ROW_COLOR/tot_height;
26
27      pipemm_color_W = MAX_COLUMN_COLOR/tot_width;
28
29  end
```

3.1.3 Get the proportion between depth and RGB pixels

```
1   function [prop,nb_rows_color , nb_columns_color,nb_rows_depth,
        nb_columns_depth] = proportion(reformed_depth , reformed_color)
2       % this function returns the size of the given color and depth
            matrices,
3       % and the proportion between the depth and color pixels
4
5       [nb_rows_color , nb_columns_color,~]=size(reformed_color);
6       [nb_rows_depth, nb_columns_depth]= size(reformed_depth);
7
8       nb_pixels_color=nb_rows_color * nb_columns_color;
9       nb_pixels_depth=nb_rows_depth * nb_columns_depth;
```

```
10
11      x= max(nb_pixels_color,nb_pixels_depth);
12      y= min(nb_pixels_color,nb_pixels_depth);
13
14      prop = x/y;
15
16  end
17
18  function the_size=size_matching(prop)
19      the_size= round(sqrt(prop));
20  end
```

3.1.4 Get the exact positions from depth to RGB

```
1   function [row_start, row_stop, col_start, col_stop]= depth_to_color(
        pipemm_depth_H , pipemm_depth_W , pipemm_color_H , pipemm_color_W,row,
        col,the_size,nb_rows_color , nb_columns_color)
2
3       mm_width_from_left = col/pipemm_depth_W;
4       mm_height_from_top = row/pipemm_depth_H;
5
6       corr_pixel_col_color = round(mm_width_from_left * pipemm_color_W);
7       corr_pixel_row_color = round(mm_height_from_top * pipemm_color_H);
8
9       steps = floor(the_size/2);
10      %steps=5;
11
12      row_start=corr_pixel_row_color-steps;
13      row_stop=corr_pixel_row_color+steps;
14
15      col_start=corr_pixel_col_color-steps;
16      col_stop=corr_pixel_col_color+steps;
17
18      if row_start<1
19          row_start = 1;
20      end
21
22      if row_stop > nb_rows_color
23          row_stop=nb_rows_color;
24      end
25
26      if col_start<1
27          col_start = 1;
28      end
29
30      if col_stop > nb_columns_color
31          col_stop = nb_columns_color;
32      end
33
34
35
36
37  end
```

3.2 Overlap from depth to RGB

```
1   function overlapped_matrix = overlap_depth_to_RGB(reformed_depth,
        reformed_color, pipemm_depth_H , pipemm_depth_W , pipemm_color_H ,
```

```
    pipemm_color_W,the_size,nb_rows_color  , nb_columns_color)

    depth_size = size(reformed_depth);

    MAX_ROW_DEPTH = depth_size(1);

    MAX_COLUMN_DEPTH = depth_size(2);

    for row = 1:MAX_ROW_DEPTH
        for col = 1:MAX_COLUMN_DEPTH
            if(reformed_depth(row, col, 1) == -1)
                [row_start, row_stop, col_start, col_stop] =
                    depth_to_color(pipemm_depth_H , pipemm_depth_W ,
                    pipemm_color_H , pipemm_color_W,row, col,the_size,
                    nb_rows_color , nb_columns_color);
                reformed_color(row_start:row_stop, col_start:col_stop, 1)
                    = 255;
                reformed_color(row_start:row_stop, col_start:col_stop, 2)
                    = 0;
                reformed_color(row_start:row_stop, col_start:col_stop, 3)
                    = 0;
            end
        end
    end
    overlapped_matrix = reformed_color;
end
```

3.3 Crop RGB to basket

```
function usefull_matrix = crop_RGB_to_basket(img)

    z = 20;

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    row = 1;
    col = 1;
    %thicken the edge
    for i = (1+z): (MAX_ROW-z)
        for j = (1+z) : (MAX_COLUMN-z)
            if (img(i, j, 1) == 255) && (img(i, j, 2) == 0) && (img(i, j,
                3) == 0)
                img(i-z:i+z, j-z:j+z, 1) = 0;
                img(i-z:i+z, j-z:j+z, 2) = 0;
                img(i-z:i+z, j-z:j+z, 3) = 255;
            end
        end
    end
    %go from left to right
    while (row ~= MAX_ROW)
        if col == MAX_COLUMN
            col = 1;
            row = row + 1;
```

```
28
29          elseif (img(row, col, 1) == 0 ) && (img(row, col, 2) == 0) && (
                img(row, col, 3) == 255)
30              col = 1;
31              row = row + 1;
32
33          else
34              img(row, col, 1) = 255;
35              img(row, col, 2) = 255;
36              img(row, col, 3) = 255;
37              col = col + 1;
38          end
39      end
40      %go from right to left
41      row = MAX_ROW;
42      col = MAX_COLUMN;
43      while (row ~= 1)
44          if col == 1
45              col = MAX_COLUMN;
46              row = row - 1;
47
48          elseif (img(row, col, 1) == 0) && (img(row, col, 2) == 0) && (img
                (row, col, 3) == 255)
49              col = MAX_COLUMN;
50              row = row - 1;
51
52          else
53              img(row, col, 1) = 255;
54              img(row, col, 2) = 255;
55              img(row, col, 3) = 255;
56              col = col - 1;
57          end
58      end
59      %go from top to bottom
60      row = 1;
61      col = 1;
62      while (col ~= MAX_COLUMN)
63          if row == MAX_ROW
64              row = 1;
65              col = col + 1;
66
67          elseif (img(row, col, 1) == 0) && (img(row, col, 2) == 0) && (img
                (row, col, 3) == 255)
68              row = 1;
69              col = col + 1;
70
71          else
72              img(row, col, 1) = 255;
73              img(row, col, 2) = 255;
74              img(row, col, 3) = 255;
75              row = row + 1;
76          end
77      end
78      %go from bottom to top
79      row = MAX_ROW;
80      col = MAX_COLUMN;
```

```
    while (col ~= 1)
        if row == 1
            row = MAX_ROW;
            col = col - 1;

        elseif (img(row, col, 1) == 0) && (img(row, col, 2) == 0) && (img
            (row, col, 3) == 255)
            row = MAX_ROW;
            col = col - 1;

        else
            img(row, col, 1) = 255;
            img(row, col, 2) = 255;
            img(row, col, 3) = 255;
            row = row - 1;
        end
    end
    %add in the white edge
    for i = 1: MAX_ROW
        for j = 1 : MAX_COLUMN
            if (img(i, j, 1) == 0) && (img(i, j, 2) == 0) && (img(i, j,
                3) == 255)
                img(i,j, 1) = 255;
                img(i,j, 2) = 255;
                img(i,j, 3) = 255;
            end
        end
    end



   usefull_matrix = img;

end
```

4 Functions for colour

4.1 Greyscale

```
function grey = greyscale(img)

    grey = img(:,:,1) * 0.2989 + img(:,:,2) * 0.5870 + img(:,:,3) *
        0.1140;

end
```

4.2 Blurring the image

4.2.1 Mean blur

```
function mean_blurred = mean_blur(img)
    mean = (1/9) * [ 1 1 1; 1 1 1; 1 1 1];
    mean_blurred = conv2(img, mean);
end
```

4.2.2 Gaussian blur

```
function gaussian_blurred = gaussian_blur(img)
    gaussian = (1/159) * [2 4 5 4 2; 4 9 12 9 4; 5 12 15 12 5; 4 9 12 9
        4; 2 4 5 4 2;];
    gaussian_blurred = conv2(img, gaussian);
end
```

4.3 Laplacian edge detect

```
function edge = edge_detect(img)
    klaplace=[0 -1 0; -1 4 -1; 0 -1 0];              % Laplacian filter
        kernel
    edge=conv2(img,klaplace);                         % convolve test img
         with
end
```

4.4 Threshold for the edge

```
function thresholded_img = threshold_edge(img, THRESHOLD_VALUE)
    THRESHOLD_VALUE = 2;
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);
    THICKNESS = 1; % 3

    thresholded_img = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if img(row, col) > THRESHOLD_VALUE
                value = 1;
                for i=1:THICKNESS
                    % Create thicker edges (edges of THICKNESS pixels
                        thick)
                    if (col - i) > 0
                        thresholded_img(row, col-i) = 1;
                    end

                    if (col + i) <= MAX_COLUMN
                        thresholded_img(row, col+i) = 1;
                    end

                    if (row - i) > 0
                        thresholded_img(row -i, col) = 1;
                    end

                    if (row + i) <+ MAX_ROW
                        thresholded_img(row +i, col) = 1;
                    end

                end
            else
                value = 0;
            end
            thresholded_img(row, col) = value;
        end
    end
end
```

4.5 Group the edges

```
function [result, nb_of_groups] = group(img, SAME_PIXEL_SEARCH_GRID_SIZE,
    GROUP_SEARCH_GRID_SIZE, MIN_NB_SURROUNDING_PIXELS)
    % Goal, group pixels.
    % First loop from left to right to find an object
    % Check if it's connected
    % Number connected pixels in the second dimension
    WHITE = 1;
    BLACK = 0;
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    groups = 0;

    result = zeros(MAX_ROW,MAX_COLUMN,2); % Dimension 2 is for the group
        number.
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            pixel_value = img(row, col);
            result(row, col,1) = pixel_value; % Transfer picture to result
                variable (in dim 1)
            if pixel_value == BLACK
                % This is an edge
                connecting_pixels = same_pixels_in_range(img, row, col,
                    SAME_PIXEL_SEARCH_GRID_SIZE);
                %connecting_pixels = real_connecting_pixels(img, row, col);


                if connecting_pixels > MIN_NB_SURROUNDING_PIXELS
                    % This is defined as an object outline.
                    group_number = find_group_in_range(result, row, col,
                        GROUP_SEARCH_GRID_SIZE);

                    if group_number == 0
                        % assign new group
                        groups = groups + 1;
                        group_number = groups;
                    end
                    %disp("connecting pixels=" + connecting_pixels + "
                        group number=" + group_number + " pos=" + row + ", "
                        + col);
                    result(row, col, 2) = group_number;
                end
            end
            %imagesc(result(:,:,2));

        end
    end
    nb_of_groups = groups;
end
```

4.6 Regroup the edges

```
function [result, nb_groups] = regroup(grouped_img, nb_of_groups,
    MIN_ROW_LINES_BETWEEN_GROUPS)
```

```
    % Loop from (right)top to (left)bottom
    % Check if there are connecting groups.

    matrix_size = size(grouped_img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    nb_groups = nb_of_groups;
    for col_i=1:MAX_COLUMN
        for row=1:MAX_ROW
            col = MAX_COLUMN - col_i+1;
            group_nb = grouped_img(row, col, 2);
            if group_nb ~= 0
                for row_i=1:MIN_ROW_LINES_BETWEEN_GROUPS
                    if is_valid_position(MAX_ROW, MAX_COLUMN, row + row_i
                        , col) == 1 && grouped_img(row + row_i, col, 2) ~=
                         0 && grouped_img(row+row_i, col,2) ~= group_nb
                        % Found a different group in the next 5 pixels
                        % below this one
                        % Replace next group with previous group number
                        grouped_img = group_replace(grouped_img,
                            grouped_img(row+row_i, col, 2), group_nb);
                        nb_groups = nb_groups - 1;
                        break;
                    end
                end
            end
        end
    end


    result = grouped_img;
end
```

4.7 Find the corner points

```
function result = find_corner_points(img, nb_groups)
    % Loop through grouped image
    % find MIN_ROW & MIN_COL and MAX_ROW & MAX_COL
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    GROUP_MAX_ROW = zeros(1,nb_groups);
    GROUP_MAX_COL = zeros(1,nb_groups);
    GROUP_MIN_ROW = zeros(1,nb_groups);
    GROUP_MIN_COL = zeros(1,nb_groups);

    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            group_nb = img(row, col, 2);
            if group_nb ~= 0
                % Group found (==0 means nothing is set)
                if GROUP_MAX_ROW(1,group_nb) == 0 || GROUP_MAX_ROW(1,
                    group_nb) < row
                    GROUP_MAX_ROW(1,group_nb) = row;
                end
```

```
21
22                  if GROUP_MAX_COL(1,group_nb) == 0 || GROUP_MAX_COL(1,
                        group_nb) < col
23                      GROUP_MAX_COL(1,group_nb) = col;
24                  end
25
26                  if GROUP_MIN_ROW(1,group_nb) == 0 || GROUP_MIN_ROW(1,
                        group_nb) > row
27                      GROUP_MIN_ROW(1,group_nb) = row;
28                  end
29                  if GROUP_MIN_COL(1,group_nb) == 0 || GROUP_MIN_COL(1,
                        group_nb) > col
30                      GROUP_MIN_COL(1,group_nb) = col;
31                  end
32              end
33
34          end
35          result = [GROUP_MIN_ROW; GROUP_MIN_COL; GROUP_MAX_ROW;
                GROUP_MAX_COL];
36      end
37
38  end
```

4.8 Remove objects within objects

4.8.1 Remove box edge

```
1  function [result, new_nb_of_groups] = remove_box_edge(corner_points,
       nb_of_groups)
2      mat_size = size(corner_points);
3      groups = mat_size(2);
4      surfaces = zeros(groups); % Every column is a group, the value is the
           distance
5
6      for i=1:groups
7
8          min_row = corner_points(1,i);
9          min_col = corner_points(2,i) ;
10         max_row = corner_points(3,i);
11         max_col = corner_points(4,i);
12
13         surfaces(i) = (max_row - min_row) * (max_col - min_col);
14     end
15
16     %Now find biggest surface
17     [max_value, max_col] = max(surfaces);
18     for i=1:4
19         % Set the coordinates of the outer points to 0
20         corner_points(i, max_col) = 0;
21     end
22
23     result = corner_points;
24     new_nb_of_groups = nb_of_groups-1;
25 end
```

4.8.2 Remove corner points within corner points

```
function [updated_corner_points, nb_of_groups] =
    remove_corner_points_within_corner_points(corner_points, nb_groups)
    mat_size = size(corner_points);
    groups = mat_size(2); % This is the original number_of_groups
    nb_of_groups = nb_groups; % This is the number_of_groups after
        regroup
    updated_corner_points = corner_points;

    for first=1:groups
        % Loop through every group
        % Now draw boundary box
        min_row_first = corner_points(1,first);
        min_col_first = corner_points(2,first);
        max_row_first = corner_points(3,first);
        max_col_first = corner_points(4,first);
        for second = 1:groups
            if first ~= second && max_row_first ~= 0 && corner_points(4,
                second) ~= 0 % If the max values would be 0, this won't be
                 a group
                % Same groups, cant lay within eachother
                min_row_second = corner_points(1,second);
                min_col_second = corner_points(2,second);
                max_row_second = corner_points(3,second);
                max_col_second = corner_points(4,second);

                % Check if second lays within first

                if min_row_second >= min_row_first && min_col_second >=
                    min_col_first && max_row_second <= max_row_first &&
                    max_col_second <= max_col_first
                    % Second object lays within first object
                    % Remouve this object
                    updated_corner_points(:, second) = zeros(4,1);

                    nb_of_groups = nb_of_groups - 1;
                end
            end
        end
    end
end
```

4.9 Draw the boundary box

```
function img = draw_red_boundary_box(img, corner_points, top_row, top_col
    )
    mat_size = size(corner_points);
    groups = mat_size(2);
    THICKNESS = 5;

    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);
    for i=1:groups
        % Loop through every group
        % Now draw boundary box
        min_row = corner_points(1,i) + top_row;
        min_col = corner_points(2,i) + top_col;
```

```
14          max_row = corner_points(3,i) + top_row;
15          max_col = corner_points(4,i) + top_col;
16          % First draw horizontal lines
17          for col=min_col:max_col
18              for e=0:THICKNESS
19                  if is_valid_position(MAX_ROW, MAX_COLUMN, min_row+e, col)
                        == 1
20                      img(min_row+e, col, 1) = 255;
21                      img(min_row+e, col, 2) = 1;
22                      img(min_row+e, col, 3) = 1;
23                  end
24                  if is_valid_position(MAX_ROW, MAX_COLUMN, max_row-e, col)
                        == 1
25                      img(max_row-e, col,1) = 255;
26                      img(max_row-e, col,2) = 1;
27                      img(max_row-e, col,3) = 1;
28                  end
29              end
30          end
31
32          % Vertical lines
33          for row=min_row:max_row
34              for e=0:THICKNESS
35                  if is_valid_position(MAX_ROW, MAX_COLUMN, row, min_col +
                        e) == 1
36                      img(row, min_col+e, 1) = 255;
37                      img(row, min_col+e, 2) = 1;
38                      img(row, min_col+e, 3) = 1;
39                  end
40                  if is_valid_position(MAX_ROW, MAX_COLUMN, row, max_col -
                        e) == 1
41                      img(row, max_col-e, 1) = 255;
42                      img(row, max_col-e, 2) = 1;
43                      img(row, max_col-e, 3) = 1;
44                  end
45              end
46
47          end
48      end
49  end
```

5 Implementation: packaging code

5.1 Gathering objects

5.1.1 Get objects

```
1   function objects = get_objects(updated_corner_points, original_img,
        regrouped)
2       % Returns a list of all objects in the given image counted by the
3       % counting system.
4       % Each object is placed in the smallest possible package.
5       % The list is sorted from largest to smallest object.
6
7       % Initializing variables
8       mat_size = size(updated_corner_points);
9       groups = mat_size(2);
```

```
    unsorted_objects = {};
    last_added_img = 1;

    % Gathering every group.
    for groupnb=1:groups

        % Updated_cornern_points can contain zeros as corner points, these
        % aren't valid.
        if updated_corner_points(1,groupnb) ~= 0

            % Getting the object cut out of the full image.
            [objAlone, min_row_i, min_col_i, max_row_i, max_col_i,] = single_object(original_img,updated_corner_points, groupnb);
            obj_points = [min_row_i; min_col_i; max_row_i; max_col_i];

            % Making the whole image white except the object itself.
            objAlone = object_highlighter(objAlone, obj_points, regrouped, groupnb);
            % Fitting the object in the smallest possible package
            img = boundaryBoxedImgRotator(objAlone);

            % Adding the object to the list of objects
            unsorted_objects{last_added_img} = img;
            last_added_img = last_added_img + 1;
        end
    end

    % Sort the list of objects
    objects = imgs_InsertionSort(unsorted_objects);

end
```

5.1.2 Object highlighter

```
function newImg = object_highlighter(img, obj_points, regrouped, groupnb)
    % Makes everything besides the object with a given group number white.

    % Initializing variables
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COL = matrix_size(2);
    newImg = ones(MAX_ROW,MAX_COL);

    % Iterate over every row
    for row = 1:MAX_ROW
        first_pixel = -1;
        last_pixel = -1;

        % First iteration over the row
        % Mark for each row the first and the last pixel of the object, as
        % seen by the regrouping algortihm
        for col = 1:MAX_COL
            if regrouped(row + obj_points(1)-1,col + obj_points(2)-1,2)
```

```
                == groupnb
                    last_pixel = col-2;
                    if first_pixel == -1
                        first_pixel = col+2;
                    end
                end
            end

            % Second iteration over the row
            % If the pixel falls inbetween the two marked points, it belongs to
            % the object and is given the same value as the original image. If
            % the pixel is outside those points or there are no marked points
            % at all it is coloured white (255).
            for col = 1:MAX_COL

                if first_pixel == -1
                    newImg(row,col) = 255;
                end

                if col >= first_pixel && col <= last_pixel
                    newImg(row,col) = img(row,col);
                else
                    newImg(row,col) = 255;
                end
            end
        end
    end
```

5.1.3 Insertion sort

```
function sorted_imgs = imgs_InsertionSort(listed_imgs)
    % Insurtion sort based fucntion to sort a list of images from biggest
    % to smallest surface area.

    listSize = size(listed_imgs);

    % Iterate over every image
    for i=1:listSize(2)

        % Calculating the surface size of images i
        img_i = listed_imgs{i};
        img_i_sizes = size(img_i);
        img_i_surface_size = img_i_sizes(1)*img_i_sizes(2);

       % Run through the all images before i
       for j=1:i

           % Calculating the surface size of image j
            img_j = listed_imgs{j};
            img_j_sizes = size(img_j);
            img_j_surface_size = img_j_sizes(1)*img_j_sizes(2);

           % Swap the two if the image i is bigger than image j
            if img_i_surface_size > img_j_surface_size
                temp = listed_imgs{j};
```

```
                listed_imgs{j} = listed_imgs{i};
                listed_imgs{i} = temp;
                img_i_surface_size = img_j_surface_size;
            end

        end
    end

    sorted_imgs = listed_imgs;
end
```

5.1.4 Single object

```
function [objAlone, min_row_group, min_col_group, max_row_group,
    max_col_group] = single_object(img, corner_points, groupnb)
    % Returns a part of the image which fully contains the group
        associated with the given group number.

    min_row_group = corner_points(1,groupnb);
    min_col_group = corner_points(2,groupnb);
    max_row_group = corner_points(3,groupnb);
    max_col_group = corner_points(4,groupnb);
    % Crop around the group
    objAlone = simon_crop(img, min_row_group,min_col_group,max_row_group,
        max_col_group);
end
```

5.2 fitting the objects

5.2.1 Boundary boxed image rotator

```
function rotated_objec = boundaryBoxedImgRotator(boxedObjec)
    % Bissection method based algorithm to find the best fitting
    % package/boundary box

    % Initialize variables
    lower_rad = 0;
    upper_rad = pi*3/8;
    lower_dim = packaged_objec(boxedObjec, lower_rad, 1);
    upper_dim = packaged_objec(boxedObjec, upper_rad, 1);

    i = 0;
    while i < 10
        % Calculating pivot values
        pivot_rad = (upper_rad-lower_rad)/2+lower_rad;
        pivot_dim = packaged_objec(boxedObjec, pivot_rad, 1);

        % Comparing the lower and upper points and changing their values
        % accordingly.
        if lower_dim <= upper_dim
            upper_rad = pivot_rad;
            upper_dim = pivot_dim;
        else
            lower_rad = pivot_rad;
            lower_dim = pivot_dim;
        end
        i=i+1;
    end
```

```
28          % Returning the the rotated image for the elevnth pivot.
29          rotated_objec = packaged_objec(boxedObjec,(abs((upper_rad-
                lower_rad)/2)+min(upper_rad,lower_rad)),0);
30
31  end
```

5.2.2 Packaged object

```
1   function boxedDim = packaged_objec(boxedObjec, angle, flag)
2       % Function which will return either the dimensions of the new
            boundary
3       % box after rotating the image (if flag == 1) or returns the whole
4       % image after it has been cropped to the new boundary box of the
5       % rotated object (if flag == 0)
6
7       % Initializing constants
8       THRESHOLD_VALUE = 2;
9       MIN_ROW_LINES_BETWEEN_GROUPS = 10;
10      SAME_PIXELS_SEARCH_GRID_SIZE = 10;
11      GROUP_SEARCH_GRID_SIZE = 15;
12      SURROUDING_PERCENTAGE = 10;
13      MIN_NB_SURROUNDING_PIXELS = floor((SAME_PIXELS_SEARCH_GRID_SIZE * 2)
            ^2 * SURROUDING_PERCENTAGE/100);
14
15      % Rotate the image
16      rotatedObj = rotator(boxedObjec, angle);
17
18      % Finding the object in the rotated image.
19      rotatedObj = gaussian_blur(mean_blur(rotatedObj));
20      first_edge_detect = edge_detect(rotatedObj);
21      without_noise_removal = threshold_edge(remove_boundary(
            first_edge_detect, 15), THRESHOLD_VALUE);
22      [grouped, nb_of_groups] = group(~without_noise_removal,
            SAME_PIXELS_SEARCH_GRID_SIZE, GROUP_SEARCH_GRID_SIZE,
            MIN_NB_SURROUNDING_PIXELS);
23      [regrouped, nb_of_groups2] = regroup(grouped, nb_of_groups,
            MIN_ROW_LINES_BETWEEN_GROUPS);
24      corner_points = find_corner_points(regrouped, nb_of_groups);
25      [updated_corner_points, nb_of_groups3] =
            remove_corner_points_within_corner_points(corner_points,
            nb_of_groups2);
26
27      % Checking whether one object is found
28      if nb_of_groups3 == 1
29
30          cropped_rotated_boxedObjec = generic_crop(rotatedObj,
                updated_corner_points);
31          % imshow(cropped_rotated_boxedObjec,[]);
32
33
34          if flag == 1
35              % Return the new boundary box dimensions.
36              matSize = size(cropped_rotated_boxedObjec);
37              boxedDim = matSize(1)*matSize(2);
38          elseif flag == 0
39              % return the whole image.
40              boxedDim = cropped_rotated_boxedObjec;
```

```
        end

    % If there are multiple objects, no rotated image is returned
    else
        disp("not one object");
        boxedDim = 0;
    end
end
```

5.2.3 Rotator

```
function rotated = rotator(img, rads)
    % Calculating the size of the padding matrix
    [ROWS, COLS] = size(img);
    diagonal = sqrt(ROWS^2 + COLS^2);
    rowPad = ceil(diagonal - ROWS) + 2;
    colPad = ceil(diagonal - COLS) + 2;

    % Creating the paddding matrix and filling it whith the original
        image
    % in the middle and everything else white.
    padding_mat = ones(ROWS+rowPad, COLS+colPad)*255;
    padding_mat(ceil(rowPad/2):(ceil(rowPad/2)+ROWS-1),ceil(colPad/2):(
        ceil(colPad/2)+COLS-1)) = img;

    % Calcularting the mid coordinates of the matrices.
    padding_size = size(padding_mat);
    midx=ceil((padding_size(2)+1)/2);
    midy=ceil((padding_size(1)+1)/2);

    % Creating the rotated image
    rotated=ones(padding_size)*255;
    rotSize = size(rotated);

    % For each position in the rotated image get the value out of the
    % padding matrix which corresponds with the position if it were
        rotated
    % by the given angle.
    for i=1:rotSize(1)
        for j=1:rotSize(2)

            x = (i-midx)*cos(rads)+(j-midy)*sin(rads);
            x=round(x)+midx;
            y=-(i-midx)*sin(rads)+(j-midy)*cos(rads);
            y=round(y)+midy;

            if x >= 1 && y >= 1 && x <= padding_size(2) && y <=
                padding_size(1)
                rotated(i,j)=padding_mat(x,y);
            end

        end
    end
end
```

5.2.4 Generic crop

```
function img_crop = generic_crop(img, fourp)
```

```
    % A function which crops the given image so that the edges are
    % definened by the four point given in fourp.
    mat_size = size(fourp);
    groups = mat_size(2);

    for i=1:groups
        if fourp(1,i)~=0
        MIN_ROW = fourp(1,i);
        MIN_COL = fourp(2,i);
        MAX_ROW = fourp(3,i);
        MAX_COL = fourp(4,i);
        end
    end

    img_crop = zeros(MAX_ROW-MIN_ROW+1,MAX_COL-MIN_COL+1,1);
        for row = MIN_ROW:MAX_ROW
            for col = MIN_COL:MAX_COL

                img_crop(row - MIN_ROW + 1,col - MIN_COL + 1,1) = img(row
                    ,col);
            end
        end

end
```

5.3 total packaging

5.3.1 Smallest Package

```
function replacedObjects = smallest_package(objects)
    % This fucntion creates a possible packaging for all the objects
       given.
    % This package has to be as small as possible, but isn't the optimal
    % packaging. This is a greedy algorithm which fills the package from
    % the biggest to smallest objects.

    % The function returns an image of the package with each individual
    % object's package outlinded in black.

    % OBJECTS HAS TO BE SORTED FROM BIGGEST TO SMALLEST BEFOREHAND


    % The optimal package for the biggest object is the object itself.
    replacedObjects = black_edger(objects{1});

    % Iterate over every object except the biggest and make a new package
    % of the old package and the new object
    listSize = size(objects);
    for i=2:listSize(2)

       % Initialize variables
        mat_size = size(replacedObjects);
        object_size = size(objects{i});
        extra_size_smallest = inf;
        smallest_col = 0;
        smallest_row = 0;
        flag = 0;

```

```
29          % Iterate over every pixel
30          for row=1:mat_size(1)+1
31              for col=1:mat_size(2)+1
32
33                  % Checking wether there is an object at this location.
34                  if row ~= mat_size(1)+1 && col ~= mat_size(2)+1
35                    % If there is an object, continue with the next pixel.
36                     if replacedObjects(row,col) ~= -1
37                       continue
38                     end
39                  end
40
41                  % Measuring the size of the package if it were appended
                        with
42                  % object at on this position.
43                  vert_diff = object_size(1)  + row - 1;
44                  hor_diff = object_size(2) + col - 1;
45
46                  if vert_diff < mat_size(1)
47                      vert_diff = mat_size(1);
48                  end
49                  if hor_diff < mat_size(2)
50                      hor_diff = mat_size(2);
51                  end
52
53                  extra_size = vert_diff * hor_diff;
54
55                  % Measuring the same size as above, but the object is
                        rotated
56                  % 90 .
57                  vert_diff_rot = object_size(2)  + row - 1;
58                  hor_diff_rot = object_size(1) + col - 1;
59
60                  if vert_diff_rot < mat_size(1)
61                      vert_diff_rot = mat_size(1);
62                  end
63                  if hor_diff_rot < mat_size(2)
64                      hor_diff_rot = mat_size(2);
65                  end
66
67                  extra_size_rot = vert_diff_rot * hor_diff_rot;
68
69
70                  % If the this position results in a smaller package than
                        the
71                  % previous one use this one as the best position.
72                  if extra_size < extra_size_smallest
73                      % Appending the object at this position may not result
74                      % in overlap of objects.
75                      bool = position_tester(replacedObjects, objects{i}, row
                            , col);
76                      if bool == 1
77                        extra_size_smallest = extra_size;
78                        smallest_row = row;
79                        smallest_col = col;
80                        flag = 0;
```

```
                    end
                end

                % Same as above but for the rotated object.
                if extra_size_rot < extra_size_smallest
                    % Appending the rotated object at this position may not
                        result
                    % in overlap of objects.
                    bool =  position_tester(replacedObjects, transpose(
                        objects{i}), row, col);
                    if bool == 1
                     extra_size_smallest = extra_size_rot;
                     smallest_row = row;
                     smallest_col = col;
                     flag = 1;
                    end
                end
            end
        end

        % Append the object on the best position to the old package.
        % If the optimal measurments are reached bij rotationg the object
        % transpose it.
        if flag == 1
             replacedObjects = package_appender(replacedObjects,
                black_edger(transpose(objects{i})), smallest_row,
                smallest_col);

        else
             replacedObjects = package_appender(replacedObjects,
                black_edger(objects{i}), smallest_row, smallest_col);
        end

        imshow(replacedObjects,[]);

    end
end
```

5.3.2 Black Edged

```
function blackEdged = black_edger(img)
    % Function which the outer one pixel in both dimensions of an image
    % black.
    % Used to see the edge of each individual package in the combined one
        .

    img_size = size(img);

    for row = 1:img_size(1)
        for col = 1:img_size(2)
            if row == 1 || row == img_size(1) || col == 1 || col ==
                img_size(2)
                 img(row,col) = 1;
            end
        end
    end

```

```
    blackEdged = img;

end
```

5.3.3 Position tester

```
function result = position_tester(package, object, smallest_row,
    smallest_col)
    % The result of this function is true if and only if appending the
    % package with the given object on the given row and col doesn't
        result
    % in overlap.

    % Remeber that every non-object pixel in package has a value of -1.

    % Initialize variables
    result = 1;
    mat_size = size(package);
    object_size = size(object);

    % Determine the boundaries of iteration. If the object fully overlaps
    % with the current package use the object dimensions + the position
        as
    % the upper-boundary else use the package dimensions as the boundary.
    if  object_size(1) + smallest_row-1 > mat_size(1)
        biggest_row = mat_size(1);
    else
        biggest_row = object_size(1) + smallest_row-1;
    end

    if object_size(2) + smallest_col - 1 > mat_size(2)
        biggest_col = mat_size(2);
    else
        biggest_col = object_size(2) + smallest_col - 1;
    end

    % Run trhough the part of the package with which the object would
    % overlap, if there is another object the result will be changed to 0
    for row=smallest_row:biggest_row
        for col=smallest_col:biggest_col
            if package(row,col) ~= -1
                result = 0;
            end
        end
    end

end
```

5.3.4 Package appender

```
function new_package = package_appender(package, object, smallest_row,
    smallest_col)
    % Appends the package with the given object on the given postion.
    % If the object dimensions exceed the dimensions of the package, a
        new
    % one will be created which wil fit both. Empty spaces in the package
    % are denoted by a value of -1.
    mat_size = size(package);
```

```
7       object_size = size(object);
8
9
10      % The old package can fit the object.
11      if mat_size(1) >= object_size(1) + smallest_row && mat_size(2) >=
            object_size(2) + smallest_col
12          % The package is appended with the object at the given position.
13          for row = smallest_row:(object_size(1)+smallest_row-1)
14              for col = smallest_col:(object_size(2)+smallest_col-1)
15                  package(row,col) = object(row - smallest_row + 1, col -
                        smallest_col + 1);
16              end
17          end
18          new_package = package;
19
20
21      % The old package can't fit the object horizontally.
22      elseif mat_size(1) >= object_size(1) + smallest_row && mat_size(2) <
            object_size(2) + smallest_col
23          % Create a new package which can fit the object.
24          new_package =  ones(mat_size(1), object_size(2) + smallest_col)
                *-1;
25
26          % Fill the new package with the old one.
27          for row = 1:mat_size(1)
28              for col = 1:mat_size(2)
29                  new_package(row,col) = package(row,col);
30              end
31          end
32
33          % The package is appended with the object at the given position.
34          for row = smallest_row:(object_size(1)+smallest_row-1)
35              for col = smallest_col:(object_size(2) + smallest_col-1)
36                  new_package(row,col) = object(row - smallest_row+1, col -
                        smallest_col+1);
37              end
38          end
39
40
41      % The old package can't fit the object vertically.
42      elseif mat_size(1) < object_size(1) + smallest_row && mat_size(2) >=
            object_size(2) + smallest_col
43          % Create a new package which can fit the object.
44          new_package =  ones(object_size(1) + smallest_row, mat_size(2))
                *-1;
45
46          % Fill the new package with the old one.
47          for row = 1:mat_size(1)
48              for col = 1:mat_size(2)
49                  new_package(row,col) = package(row,col);
50              end
51          end
52
53          % The package is appended with the object at the given position.
54          for row = smallest_row:(object_size(1) + smallest_row-1)
55              for col = smallest_col:(object_size(2)+smallest_col-1)
```

```
56                  new_package(row,col) = object(row - smallest_row+1, col -
                        smallest_col+1);
57              end
58          end
59
60
61      % The old package can't fit the object both vertically and
            horizontally.
62      else
63          % Create a new package which can fit the object.
64          new_package =  ones(object_size(1) + smallest_row, object_size(2)
                + smallest_col)*-1;
65
66          % Fill the new package with the old one.
67          for row = 1:mat_size(1)
68              for col = 1:mat_size(2)
69                  new_package(row,col) = package(row,col);
70              end
71          end
72
73          % The package is appended with the object at the given position.
74          for row = smallest_row:(object_size(1) + smallest_row-1)
75              for col = smallest_col:(object_size(2) + smallest_col-1)
76                  new_package(row,col) = object(row - smallest_row + 1, col
                        - smallest_col + 1);
77              end
78          end
79      end
80
81
82  end
```