Contents

1 Initialising functions

```

h = 900;

min_y = 120;
max_y = 460;
min_x = 75;
max_x = 310;

% Threshold values
min_thresh = 30;
max_thresh = 500;

% Get image from depth sensor
colorVid = videoinput('kinect',1);
depthVid = videoinput('kinect',2);
depth = getsnapshot(depthVid);
color = getsnapshot(colorVid);
raw_matrix = depth;
%%
%Run the sobel operator

depth = sobel_operator(depth);
shapes_after_sobel = depth;
%Run the threshold filter
depth = threshold(depth, min_thresh, max_thresh);
depth = print(depth, min_x, max_x, min_y, max_y);
depth_after_threshold = depth;

%%%%%%%outline
depth = outline(depth);
final_img = only_outline_visible(depth);
edged_matrix = only_edge(depth);

new_depth = crop_depth_to_basket(edged_matrix, depth_after_threshold);
depth_tester = new_depth;

%OVERLAP
%%%%%%%%%%%%%%%%%%%%%%%%%%

%color: 1920x1080 met 84.1 x 53.8
%depth: 512x424  met 70.6 x 60

[reformed_depth,reformed_color, res_height_angle, res_width_angle] =
    reform(depth, color);
[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);


[prop,nb_rows_color , nb_columns_color,nb_rows_depth, nb_columns_depth] =
     proportion(reformed_depth , reformed_color);

tot_size = size_matching(prop);

```

```
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);

new_RGB = crop_RGB_to_basket(total);
image(new_RGB);

img = new_RGB;

THRESHOLD_VALUE = 2;

MIN_ROW_LINES_BETWEEN_GROUPS = 10; %25 %15
% Once the groups are found, the algorithm searches for groups too close
% near each other
% This is defined as the min distance between two groups (only searched
% vertical)
SAME_PIXELS_SEARCH_GRID_SIZE = 10;%25
% Grid size = this variable *2, it searches for pixels with the same value
% in this grid.
GROUP_SEARCH_GRID_SIZE = 15; %25
% Grid size = this variable * 2, it searches for pixels with a group number
% (not 0) in this grid.
SURROUDING_PERCENTAGE = 10;% %
MIN_NB_SURROUNDING_PIXELS = floor((SAME_PIXELS_SEARCH_GRID_SIZE * 2)^2 * SURROUDING_PERCENTAGE/100) ;%125 % 50
% The minimum number of pixels with the same value that are in the grid
% size defined by SAME_PIXELS_SEARCH_GRID
% The pixels that have a less number of surrounding pixels, are not defined
% as a group but as noise.

% CROPPING: Defining rectangle
%top_row = 290 ; top_col = 760; bottom_row = 690 ; bottom_col = 1440;
%top_row = 150; top_col = 750; bottom_row = 950; bottom_col = 1900;
%top_row = 200; top_col = 850; bottom_row = 750; bottom_col = 1850; % For pictues with x2_RGB_.... in name
%top_row = 100, top_col = 100; bottom_row = 980; bottom_col = 1820; % For pictues with RGB in name.

disp("Step 1: loading the image...");
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");

disp("Step 2: converting the image to greyscale...");

A = greyscale(img); % Convert image to grayscale

%top_left_row, top_left_col, bottom_right_row, bottom_right_col
disp("Step 3: cropping the image...");

%A = simon_crop(A, top_row, top_col, bottom_row, bottom_col);
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)];

    matrix_total = matrix_1;
    matrix_total(:,:,2) = matrix_2;
    matrix_total(:,:,3) = matrix_3;

    created_matrix = matrix_total;

end
```

2.3.6 all surrounding positions

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

    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
function placing = top_right(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position top right of the given position
    x = position(1) - 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
function placing = right(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position to the right 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
function placing = bottom_right(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position bottom right of the given position
    x = position(1) +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
function placing = bottom(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position below the given position
    x = position(1) +1;
    y = position(2) ;

    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
function placing = bottom_left(position, img, MAX_ROW, MAX_COLUMB)
    % returns the position bottom left of the given position
    x = position(1) +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
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
    [nb_rows_depth,~]=size(depth);

    nb_pixels_depth_per_degree_height = nb_rows_depth / height_color_angle;

    nb_height_pixels_removed_depth = (height_depth_angle-height_color_angle)*nb_pixels_depth_per_degree_height;
    reformed_depth = depth(round(nb_height_pixels_removed_depth/2,0):round(nb_rows_depth -(nb_height_pixels_removed_depth/2),0),:);
end
```

3.1.2 Get the pixels per mm

```
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)
    % this function returns the pixels per millimeter for the given depth
    % and color matrices

    depth_size = size(reformed_depth);

    MAX_ROW_DEPTH = depth_size(1);

    MAX_COLUMN_DEPTH = depth_size(2);

    color_size = size(reformed_color);

    MAX_ROW_COLOR = color_size(1);

    MAX_COLUMN_COLOR = color_size(2);

    tot_width = 2*h*tan(((res_width_angle)/2)*(pi/180));

    tot_height = 2*h*tan(((res_height_angle)/2)*(pi/180));

    pipemm_depth_H = MAX_ROW_DEPTH/tot_height;

    pipemm_depth_W = MAX_COLUMN_DEPTH/tot_width;

    pipemm_color_H = MAX_ROW_COLOR/tot_height;

    pipemm_color_W = MAX_COLUMN_COLOR/tot_width;

end
```

3.1.3 Get the proportion between depth and RGB pixels

```
function [prop,nb_rows_color , nb_columns_color,nb_rows_depth, nb_columns_depth] = proportion(reformed_depth , reformed_color)
    % this function returns the size of the given color and depth matrices,
    % and the proportion between the depth and color pixels

    [nb_rows_color , nb_columns_color,~]=size(reformed_color);
    [nb_rows_depth, nb_columns_depth]= size(reformed_depth);

    nb_pixels_color=nb_rows_color * nb_columns_color;
    nb_pixels_depth=nb_rows_depth * nb_columns_depth;

    x= max(nb_pixels_color,nb_pixels_depth);
    y= min(nb_pixels_color,nb_pixels_depth);

    prop = x/y;

end

function the_size=size_matching(prop)
    the_size= round(sqrt(prop));
end
```

3.1.4 Get the exact positions from depth to RGB

```
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)

    mm_width_from_left = col/pipemm_depth_W;
    mm_height_from_top = row/pipemm_depth_H;

    corr_pixel_col_color = round(mm_width_from_left * pipemm_color_W);
    corr_pixel_row_color = round(mm_height_from_top * pipemm_color_H);

    steps = floor(the_size/2);
    %steps=5;

    row_start=corr_pixel_row_color-steps;
    row_stop=corr_pixel_row_color+steps;

    col_start=corr_pixel_col_color-steps;
    col_stop=corr_pixel_col_color+steps;

    if row_start<1
        row_start = 1;
    end

    if row_stop > nb_rows_color
        row_stop=nb_rows_color;
    end

    if col_start<1
        col_start = 1;
    end

    if col_stop > nb_columns_color
        col_stop = nb_columns_color;
    end




end
```

3.2 Overlap from depth to RGB

```
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;

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

        else
            img(row, col, 1) = 255;
            img(row, col, 2) = 255;
            img(row, col, 3) = 255;
            col = col + 1;
        end
    end
    %go from right to left
    row = MAX_ROW;
    col = MAX_COLUMN;
    while (row ~= 1)
        if col == 1
            col = MAX_COLUMN;
            row = row - 1;

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

        else
            img(row, col, 1) = 255;
            img(row, col, 2) = 255;
            img(row, col, 3) = 255;
            col = col - 1;
        end
    end
    %go from top to bottom
    row = 1;
    col = 1;
    while (col ~= MAX_COLUMN)
        if row == MAX_ROW
            row = 1;
            col = col + 1;

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

        else
            img(row, col, 1) = 255;
            img(row, col, 2) = 255;
            img(row, col, 3) = 255;
            row = row + 1;
        end
    end
    %go from bottom to top
    row = MAX_ROW;
    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

                if GROUP_MAX_COL(1,group_nb) == 0 || GROUP_MAX_COL(1,group_nb) < col
                    GROUP_MAX_COL(1,group_nb) = col;
                end

                if GROUP_MIN_ROW(1,group_nb) == 0 || GROUP_MIN_ROW(1,group_nb) > row
                    GROUP_MIN_ROW(1,group_nb) = row;
                end
                if GROUP_MIN_COL(1,group_nb) == 0 || GROUP_MIN_COL(1,group_nb) > col
                    GROUP_MIN_COL(1,group_nb) = col;
                end
            end

        end
        result = [GROUP_MIN_ROW; GROUP_MIN_COL; GROUP_MAX_ROW; GROUP_MAX_COL];
    end

end
```

4.8 Remove objects within objects

4.8.1 Remove box edge

```
function [result, new_nb_of_groups] = remove_box_edge(corner_points, nb_of_groups)
    mat_size = size(corner_points);
    groups = mat_size(2);
    surfaces = zeros(groups); % Every column is a group, the value is the distance

    for i=1:groups

        min_row = corner_points(1,i);
        min_col = corner_points(2,i) ;
        max_row = corner_points(3,i);
        max_col = corner_points(4,i);

        surfaces(i) = (max_row - min_row) * (max_col - min_col);
    end

    %Now find biggest surface
    [max_value, max_col] = max(surfaces);
    for i=1:4
        % Set the coordinates of the outer points to 0
        corner_points(i, max_col) = 0;
    end

    result = corner_points;
    new_nb_of_groups = nb_of_groups-1;
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;
        max_row = corner_points(3,i) + top_row;
        max_col = corner_points(4,i) + top_col;
        % First draw horizontal lines
        for col=min_col:max_col
            for e=0:THICKNESS
                if is_valid_position(MAX_ROW, MAX_COLUMN, min_row+e, col) == 1
                    img(min_row+e, col, 1) = 255;
                    img(min_row+e, col, 2) = 1;
                    img(min_row+e, col, 3) = 1;
                end
                if is_valid_position(MAX_ROW, MAX_COLUMN, max_row-e, col) == 1
                    img(max_row-e, col,1) = 255;
                    img(max_row-e, col,2) = 1;
                    img(max_row-e, col,3) = 1;
                end
            end
        end

        % Vertical lines
        for row=min_row:max_row
            for e=0:THICKNESS
                if is_valid_position(MAX_ROW, MAX_COLUMN, row, min_col + e) == 1
                    img(row, min_col+e, 1) = 255;
                    img(row, min_col+e, 2) = 1;
                    img(row, min_col+e, 3) = 1;
                end
                if is_valid_position(MAX_ROW, MAX_COLUMN, row, max_col - e) == 1
                    img(row, max_col-e, 1) = 255;
                    img(row, max_col-e, 2) = 1;
                    img(row, max_col-e, 3) = 1;
                end
            end

        end
    end
end
```

5 Implementation: packaging code

5.1 Gathering objects

5.1.1 Get objects

```
function objects = get_objects(updated_corner_points, original_img, regrouped)
    % Returns a list of all objects in the given image counted by the
    % counting system.
    % Each object is placed in the smallest possible package.
    % The list is sorted from largest to smallest object.

    % Initializing variables
    mat_size = size(updated_corner_points);
    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
```

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

end
```

5.2.2 Packaged object

```
function boxedDim = packaged_objec(boxedObjec, angle, flag)
    % Function which will return either the dimensions of the new boundary
    % box after rotating the image (if flag == 1) or returns the whole
    % image after it has been cropped to the new boundary box of the
    % rotated object (if flag == 0)

    % Initializing constants
    THRESHOLD_VALUE = 2;
    MIN_ROW_LINES_BETWEEN_GROUPS = 10;
    SAME_PIXELS_SEARCH_GRID_SIZE = 10;
    GROUP_SEARCH_GRID_SIZE = 15;
    SURROUDING_PERCENTAGE = 10;
    MIN_NB_SURROUNDING_PIXELS = floor((SAME_PIXELS_SEARCH_GRID_SIZE * 2)^2 * SURROUDING_PERCENTAGE/100);

    % Rotate the image
    rotatedObj = rotator(boxedObjec, angle);

    % Finding the object in the rotated image.
    rotatedObj = gaussian_blur(mean_blur(rotatedObj));
    first_edge_detect = edge_detect(rotatedObj);
    without_noise_removal = threshold_edge(remove_boundary(first_edge_detect, 15), THRESHOLD_VALUE);
    [grouped, nb_of_groups] = group(~without_noise_removal, SAME_PIXELS_SEARCH_GRID_SIZE, GROUP_SEARCH_GRID_SIZE, MIN_NB_SURROUNDING_PIXELS);
    [regrouped, nb_of_groups2] = regroup(grouped, nb_of_groups, MIN_ROW_LINES_BETWEEN_GROUPS);
    corner_points = find_corner_points(regrouped, nb_of_groups);
    [updated_corner_points, nb_of_groups3] = remove_corner_points_within_corner_points(corner_points, nb_of_groups2);

    % Checking whether one object is found
    if nb_of_groups3 == 1

        cropped_rotated_boxedObjec = generic_crop(rotatedObj, updated_corner_points);
        % imshow(cropped_rotated_boxedObjec,[]);


        if flag == 1
            % Return the new boundary box dimensions.
            matSize = size(cropped_rotated_boxedObjec);
            boxedDim = matSize(1)*matSize(2);
        elseif flag == 0
            % return the whole image.
            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;

    % Iterate over every pixel
    for row=1:mat_size(1)+1
        for col=1:mat_size(2)+1

           % Checking wether there is an object at this location.
           if row ~= mat_size(1)+1 && col ~= mat_size(2)+1
             % If there is an object, continue with the next pixel.
             if replacedObjects(row,col) ~= -1
               continue
             end
           end

           % Measuring the size of the package if it were appended with
           % object at on this position.
           vert_diff = object_size(1)  + row - 1;
           hor_diff = object_size(2) + col - 1;

           if vert_diff < mat_size(1)
               vert_diff = mat_size(1);
           end
           if hor_diff < mat_size(2)
               hor_diff = mat_size(2);
           end

           extra_size = vert_diff * hor_diff;

           % Measuring the same size as above, but the object is rotated
           % 90 .
           vert_diff_rot = object_size(2)  + row - 1;
           hor_diff_rot = object_size(1) + col - 1;

           if vert_diff_rot < mat_size(1)
               vert_diff_rot = mat_size(1);
           end
           if hor_diff_rot < mat_size(2)
               hor_diff_rot = mat_size(2);
           end

           extra_size_rot = vert_diff_rot * hor_diff_rot;


           % If the this position results in a smaller package than the
           % previous one use this one as the best position.
           if extra_size < extra_size_smallest
               % Appending the object at this position may not result
               % in overlap of objects.
               bool = position_tester(replacedObjects, objects{i}, row, col);
               if bool == 1
                 extra_size_smallest = extra_size;
                 smallest_row = row;
                 smallest_col = col;
                 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);
    object_size = size(object);


    % The old package can fit the object.
    if mat_size(1) >= object_size(1) + smallest_row && mat_size(2) >= object_size(2) + smallest_col
        % The package is appended with the object at the given position.
        for row = smallest_row:(object_size(1)+smallest_row-1)
            for col = smallest_col:(object_size(2)+smallest_col-1)
                package(row,col) = object(row - smallest_row + 1, col - smallest_col + 1);
            end
        end
        new_package = package;


    % The old package can't fit the object horizontally.
    elseif mat_size(1) >= object_size(1) + smallest_row && mat_size(2) < object_size(2) + smallest_col
        % Create a new package which can fit the object.
        new_package =  ones(mat_size(1), object_size(2) + smallest_col)*-1;

        % Fill the new package with the old one.
        for row = 1:mat_size(1)
            for col = 1:mat_size(2)
                new_package(row,col) = package(row,col);
            end
        end

        % The package is appended with the object at the given position.
        for row = smallest_row:(object_size(1)+smallest_row-1)
            for col = smallest_col:(object_size(2) + smallest_col-1)
                new_package(row,col) = object(row - smallest_row+1, col - smallest_col+1);
            end
        end


    % The old package can't fit the object vertically.
    elseif mat_size(1) < object_size(1) + smallest_row && mat_size(2) >= object_size(2) + smallest_col
        % Create a new package which can fit the object.
        new_package =  ones(object_size(1) + smallest_row, mat_size(2))*-1;

        % Fill the new package with the old one.
        for row = 1:mat_size(1)
            for col = 1:mat_size(2)
                new_package(row,col) = package(row,col);
            end
        end

        % The package is appended with the object at the given position.
        for row = smallest_row:(object_size(1) + smallest_row-1)
            for col = smallest_col:(object_size(2)+smallest_col-1)
```

```
                new_package(row,col) = object(row - smallest_row+1, col - smallest_col+1);
            end
        end


    % The old package can't fit the object both vertically and horizontally.
    else
        % Create a new package which can fit the object.
        new_package =  ones(object_size(1) + smallest_row, object_size(2) + smallest_col)*-1;

        % Fill the new package with the old one.
        for row = 1:mat_size(1)
            for col = 1:mat_size(2)
                new_package(row,col) = package(row,col);
            end
        end

        % The package is appended with the object at the given position.
        for row = smallest_row:(object_size(1) + smallest_row-1)
            for col = smallest_col:(object_size(2) + smallest_col-1)
                new_package(row,col) = object(row - smallest_row + 1, col - smallest_col + 1);
            end
        end
    end


end
```