Computer Assignment 2

Signals & systems

Kian Fotovat

SID : 810102486

PART 1

1) In this part we get the file from the user, using uigetfile and then store it in the 3_dimmensional matrix ''picture''

```
[file, path] = uigetfile({'*.jpg; *.bmp; *.png', 'choose the image'});
s = [path, file];
picture = imread(s);
```

2) In this part I resize the picture I got from the user. Note that the values for number of rows and columns are stored in 2 variables so we cared not using magic values (increased flexibility and changeability), the imresize function receives the picture and a list of two elements containing rows and cols size. Here's the code :

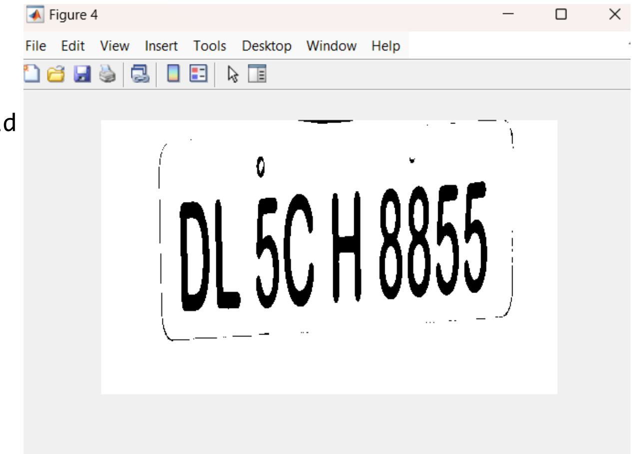
```
NUMROWS = 300;
NUMCOLS = 500;
picture = imresize(picture,[NUMROWS,NUMCOLS]);
```

3) In this part, we wanna reduce the picture we got from RGB to grayscale so our computations would be less complicated and also faster. For that I used nested for loops and iterate over pixels and combine each pixel's RGB with a specific amount and gave that result value to that pixel, so instead of storing R,G,B now we only store one element. Here's the code for my mygrayfun function that handles this :

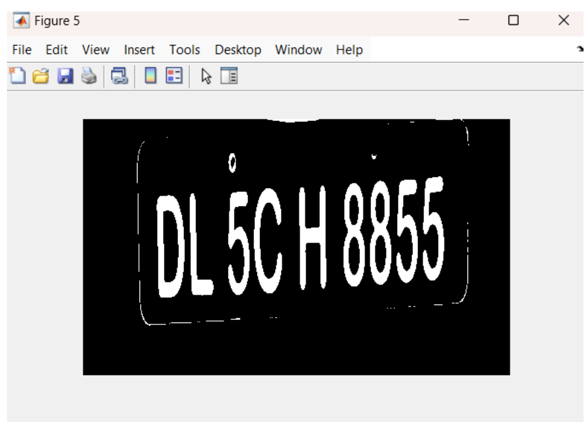
```
function gray_picture = mygrayfun (picture)

    NUMROWS = 300;
    NUMCOLS = 500;
    gray_picture = zeros(NUMROWS, NUMCOLS, 'uint8');
% I created a two dimmensional matrix to store those gray values
    for i = 1:1:NUMROWS
        for j = 1:1:NUMCOLS
            gray_picture(i,j) = 0.299*picture(i,j,1) + 0.578*picture(i,j,2) +
0.114*picture(i,j,3);
        end
    end
end
new_picture = mygrayfun(picture);
figure;
imshow(mygrayfun(picture));
```

4) In this part we go one step further, turning the grayscale picture to binary (1s for black, and 0s for white) and to do so, I wrote the function mybinaryfun that receives the image, along with a threshold that if the grayscale value was above that it would give it value 1, and vice versa. It is simple and just with a nested for loop and one if statement, we're done : (also after this, I reversed ones and zeros so the main theme would be black. Also for this purpose of English plates, threshold of 70 was fine, but I would probably change this in the Persian plates part)

```
function out = mybinaryfun(picture, threshold)

        [NUMROWS, NUMCOLS] = size(picture,1,2);
        binary = zeros(NUMROWS,NUMCOLS);
        for i = 1:1:NUMROWS
                for j = 1:1:NUMCOLS
                        if picture(i,j) > threshold
                                binary(i,j) = 1;
                        else
                                binary(i,j) = 0;
                        end
                end
        end
        out = binary;
end
new_binary = mybinaryfun(new_picture, 70);
figure;
imshow(new_binary);
```



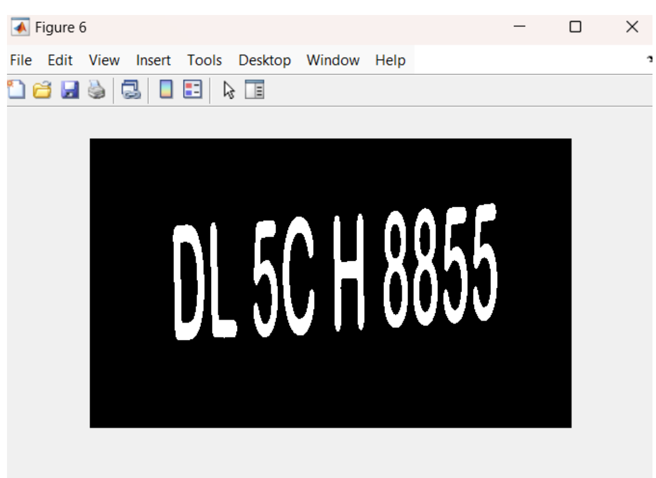

```
%% extra) part I added : change each ones with zeros and vice versa :
function out = swap_zeros_and_ones(input_matrix)
    if islogical(input_matrix)
        out = ~input_matrix;
    else
        out = input_matrix;
        out(input_matrix == 0) = 1;
        out(input_matrix == 1) = 0;
    end
end
new_binary = swap_zeros_and_ones(new_binary);
figure;
imshow(new_binary);
```



5) In this part we implement bwareaopen function of matlab, so remove those annoying noised that could cause errors and misjudging in our next parts. So the core idea is to label each bunch of connected pixels and the ones with less than n element (we get n from the user into our function) will be removed. Actually first that was my only idea, and when I implemented

that with no Technik or search algorithm, it was so heavy to run and besides that, I don't know why but it only labled the same neighbors horizontally or vertically, so I changed and enhanced my method and because I had data structure and algorithms course before, I am well familiar with bfs (breath-first-search) algorithm, so I implemented that and with idea of labeling, I made it happen. before deep diving into details, first look at the code :

```
function cleaned_image = removemycom(binary_image, n)
    [rows, cols] = size(binary_image);
    cleaned_image = binary_image;
    visited = false(rows, cols);

    for i = 1:rows
        for j = 1:cols
            if cleaned_image(i, j) == 1 && ~visited(i, j)
                queue = [i, j];
                object_pixels = [];
                visited(i, j) = true;
                neighbors = [-1, 0; 1, 0; 0, -1; 0, 1];

                while ~isempty(queue)
                    current = queue(1, :);
                    queue(1, :) = [];
                    object_pixels = [object_pixels; current];

                    for k = 1:size(neighbors, 1)
                        ni = current(1) + neighbors(k, 1);
                        nj = current(2) + neighbors(k, 2);

                        if ni >= 1 && ni <= rows && nj >= 1 && nj <= cols ...
                                && cleaned_image(ni, nj) == 1 && ~visited(ni,
nj)
                            visited(ni, nj) = true;
                            queue = [queue; ni, nj];
                        end
                    end
                end
                if numel(object_pixels) < n
                    for p = 1:size(object_pixels, 1)
                        cleaned_image(object_pixels(p, 1), object_pixels(p,
2)) = 0;
                    end
                end
            end
        end
    end
end
```



It uses **BFS** to identify all connected regions (4-connectivity: up/down/left/right). For each detected object, if its pixel count is below the threshold n, those pixels are set to 0 (background), effectively filtering out small artifacts. The algorithm iterates over all pixels, marks visited regions, and cleans the image while preserving larger structures. Now let's explain the code line by line :

1. **function cleaned_image = removemycom(binary_image, n)**
   - **Defines a function that takes a binary image (binary_image) and a threshold n (minimum object size to retain). Returns cleaned_image after noise removal.**
2. **[rows, cols] = size(binary_image);**
   - **Extracts the dimensions of the input image (rows = height, cols = width).**
3. **cleaned_image = binary_image;**
   - **Creates a copy of the input image to modify, preserving the original.**
4. **visited = false(rows, cols);**
   - **Initializes a visited matrix (same size as the image) to track processed pixels. All pixels start as false (unvisited).**
5. **for i = 1:rows and for j = 1:cols**
   - **Nested loops to iterate over every pixel in the image (row-by-row, column-by-column).**
6. **if cleaned_image(i, j) == 1 && ~visited(i, j)**
   - **Checks if the current pixel is part of a foreground object (1) and has not been visited yet.**
7. **queue = [i, j];**
   - **Initializes a queue (BFS starting point) with the coordinates of the current pixel.**
8. **object_pixels = [];**
   - **Initializes an empty array to store all pixels belonging to the current connected object.**
9. **visited(i, j) = true;**

- **Marks the current pixel as visited to avoid reprocessing.**

10. neighbors = [-1, 0; 1, 0; 0, -1; 0, 1];

- **Defines 4-connectivity offsets (up, down, left, right) for neighbor pixel exploration.**

11. while ~isempty(queue)

- **Runs until the queue is empty, ensuring all connected pixels are processed.**

12. current = queue(1, :); and queue(1, :) = [];

- **Extracts the first pixel from the queue (FIFO order) and removes it from the queue.**

13. object_pixels = [object_pixels; current];

- **Adds the current pixel to the object_pixels list for later deletion if needed.**

14. for k = 1:size(neighbors, 1)

- **Loops over the 4 possible neighbor directions.**

15. ni = current(1) + neighbors(k, 1); and nj = current(2) + neighbors(k, 2);

- **Computes the row (ni) and column (nj) indices of the neighbor pixel.**

16. if ni >= 1 && ni <= rows && nj >= 1 && nj <= cols ...
&& cleaned_image(ni, nj) == 1 && ~visited(ni, nj)

- **Checks if the neighbor pixel is within image bounds, is foreground (1), and is unvisited.**

17. visited(ni, nj) = true; and queue = [queue; ni, nj];

- **Marks the neighbor as visited and adds it to the queue for further exploration.**

18. if numel(object_pixels) < n

- **After BFS completes, checks if the detected object has fewer pixels than the threshold n.**

19. cleaned_image(object_pixels(p, 1), object_pixels(p, 2)) = 0;

- **Sets all pixels of small objects (size < n) to 0 (background), effectively removing them.**

For now, I have cleaned the image with n set to 350, and things were fine, but I may change this later or not.

6) For this part, we're going to implement segmenting the image but we're forced to implement it without bwlabel matlab function. This function receives the picture to be segmented, and returns first, a new picture which is labeled and each bunch of connected pixels have their own unique integer value as labels.

the output's second element would be the number of those connected elements found in the picture. So assume we have letters 'A' and 'B' with black background in our picture, it will return a new picture with labels 1 and 2 and zero (for nor labeled ones) and also a second output, in this case 2, which represents that number of elements found in the picture.

The code is simple, the function's input/output acts as mentioned earlier, and before explaining that equivalence handling part, we have a nested for loop to iterate over our pixels, and for each pixel, first we check if the pixel's value is 1, because we're not going to label zeros. Then we create a vector neighbors to store neighbors' labels (if exist). Then we start checking neighbors from above pixel (again if i-1 exists) and it had value 1, we store its label. We do the same for the left pixel. We don't check right and down labels because they can't possibly have labels, because they're gonna be processed later after this current pixel.

Then we check if the neighbors matrix is empty or not, if not, we find the minimum label in that vector an assign that to our current pixel. We do this till all the pixels are processed.
But as you possibly noticed, this ain't the final result we want, and it doesn't handle equivalences flawlessly, and this is the purpose of this equivalence handling part:

1. **First Pass: Pixels are labeled based on their neighbors (e.g., left and top pixels in a 4/8-connected scan).**
2. **Label Conflicts: If two neighboring pixels belong to the same region but were assigned different labels, these labels are marked as equivalent (they represent the same region).**
3. **Second Pass: Equivalences are resolved, and all pixels in a connected region receive the same final label.**

Line-by-Line Explanation

1. equivalence = containers.Map('KeyType', 'double', 'ValueType', 'double');

- **Creates a containers.Map (dictionary) to track label equivalences.**
- **Key: A label that is equivalent to another label.**
- **Value: The "root" label it maps to (used to resolve conflicts).**

2. function root = find_root(l)

- **Implements the Union-Find (Disjoint Set Union) algorithm's find operation.**
- **Goal: Find the root label of a given label l by traversing the equivalence map recursively.**
- **Example: If equivalence(2) = 1 and equivalence(1) = 3, find_root(2) returns 3.**

3. while isKey(equivalence, l)

- **Traverses the equivalence map until the root label (not present in the map) is found.**

4. Second Pass Loop (for i = 1:rows ...)

- **Iterates over every pixel in the image.**
- **Goal: Replace temporary labels with their root labels (resolving conflicts).**

5. if labeling_matrix(i, j) > 0

- **Processes only foreground pixels (labeled > 0).**

6. labeling_matrix(i, j) = find_root(labeling_matrix(i, j));

- **Updates the pixel’s label to its root label (resolving equivalences).**

Here 's the code :

```
function [L, Ne] = mysegmentation(picture)
    [rows, cols] = size(picture);

    labeling_matrix = zeros(rows, cols);
    label = 1; % Start labels from 1

    % To handle label equivalences
    equivalence = containers.Map('KeyType', 'double', 'ValueType', 'double');

    for i = 1:rows
        for j = 1:cols
            if picture(i, j) == 1
                % Collect labels of connected neighbors
                neighbors = [];

                % Check top neighbor
                if i > 1 && picture(i-1, j) == 1
                    neighbors = [neighbors, labeling_matrix(i-1, j)];
                end

                % Check left neighbor
                if j > 1 && picture(i, j-1) == 1
                    neighbors = [neighbors, labeling_matrix(i, j-1)];
                end

                % Assign the smallest label or a new label
                if isempty(neighbors)
                    labeling_matrix(i, j) = label;
                    label = label + 1;
                else
                    min_label = min(neighbors);
                    labeling_matrix(i, j) = min_label;

                    % Record equivalences
                    for neighbor_label = neighbors
                        if neighbor_label ~= min_label
                            equivalence(neighbor_label) = min_label;
                        end
                    end
                end
            end
        end
    end

    % Resolve label equivalences (Union-Find)
    function root = find_root(l)
        while isKey(equivalence, l)
            l = equivalence(l);
        end
```

```
            root = l;
        end

        % Second pass: Resolve equivalences and assign final labels
        for i = 1:rows
            for j = 1:cols
                if labeling_matrix(i, j) > 0
                    labeling_matrix(i, j) = find_root(labeling_matrix(i, j));
                end
            end
        end

        % Return the labeled image and the number of connected components
        L = labeling_matrix;
        Ne = max(L(:)); % Number of connected components
    end
```

our segmenting doesn't end with bwlabel function, we now load our reference images from the Map_Set directory, containing the English plates.

Firs we set our map set directory address, then we get a list of all .png and .bmp files in the directory, and do this separately for each format. Next we combine the file lists of different formats into one single list. Then we initialize the TRAIN cell array with row 1 for images and row 2 for lables, this way : TRAIN `= cell(2, numel(file_list));`

This is the code to initialize TRAIN :

```
map_set_dir = 'Map_Set'; % Directory containing reference images (letters/numbers)

% Get a list of all .png and .bmp files in the directory
file_list_png = dir(fullfile(map_set_dir, '*.png')); % Get .png files
file_list_bmp = dir(fullfile(map_set_dir, '*.bmp')); % Get .bmp files

% Combine the file lists
file_list = [file_list_png; file_list_bmp];

% Initialize the TRAIN cell array: Row 1 = Images, Row 2 = Labels
TRAIN = cell(2, numel(file_list));
```

Next , we load images in binary form into our TRAIN, this piece of code does that and after reading each image, checks if it is RGB and if yes, turns it binary, using im2bw (I know we have already wrote a function to do that, but let it go) and then resizes that and after that stores it in the TRAIN. We do this for each pixel and then display the number of images loaded (first I added this for debugging purpose, cause it was ignoring .bmb files, but I let it stay there).

Next we wanna project that magical rectangular out of our elements using regionprops , it is this way :

```
[L, Ne] = mysegmentation(new_cleaned);
propied = regionprops(L, 'BoundingBox', 'Area');
```

next I sort these boxes from left to right according to their x axis value, need to mention that I added this part because I witnessed some cases that it read elements reversed lol. Here's the code which uses sort and cat to make that happen :

```
boundingBoxes = cat(1, propied.BoundingBox);
[~, sortOrder] = sort(boundingBoxes(:, 1));
sorted_propied = propied(sortOrder);
```

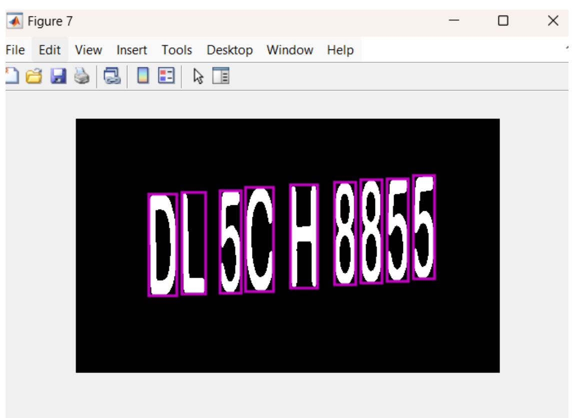
Next part is I assume to be the most important part, all the maintainings and controlling happens here, from controlling the size of elements to pick (ignore too big and too small ones) to setting a boundary for the aspect ratio. There is a for loop there that iterates over all segmented elements, and with defining a max/min_area_threshold which for English plate I set 20000 for max and 1000 for min and [0.1,4] to be the min and max aspect ratio. Then calculated the element's aspect ratio by aspect_ratio = width / height; and also its area by area = width * height; and if they were out of boundary, I wouldn't pick them as segmented . Here's the code :

```
valid_indices = [];
for n = 1:Ne
    bbox = sorted_propied(n).BoundingBox;
    width = bbox(3);
    height = bbox(4);
    area = width * height;

    % Criteria (adjust these thresholds)
    max_area_threshold = 20000;
    min_area_threshold = 1000;
    aspect_ratio_range = [0.1, 4]; % [min_aspect_ratio, max_aspect_ratio]
```

```
    % Calculate aspect ratio
    aspect_ratio = width / height;

    % Check if the region meets all criteria
    if area > min_area_threshold && area < max_area_threshold && ...
       aspect_ratio >= aspect_ratio_range(1) && aspect_ratio <= aspect_ratio_range(2)
        valid_indices = [valid_indices, n]; % Add index to valid regions
    end
end
```

in the next part we draw those boxes we already projected, but later filtered. I used RGB values to draw them with purple/magenda color :

```
filtered_propied = sorted_propied(valid_indices);
filtered_Ne = numel(valid_indices);

% Draw bounding boxes
figure;
imshow(new_cleaned);
hold on;
for n = 1:filtered_Ne
    rectangle('Position', filtered_propied(n).BoundingBox, 'EdgeColor', 'g',
'LineWidth', 2);
end
hold off;
```



now in this part we recognize the characters and print them on a .txt file. This is the line by line explanation of this element recognizing part :

Line by Line Explanation

1. **final_output = [];**
    - Initializes an empty array to store the recognized characters.
2. **for n = 1:filtered_Ne**
    - Loops over the number of valid regions (filtered_Ne) to process each character.
3. **region_idx = valid_indices(n);**

- Retrieves the index of the current valid region from valid_indices.

4. **[r, c] = find(L == sortOrder(region_idx));**
   - Finds the row (r) and column (c) coordinates of the current region in the labeled image L.
   - sortOrder(region_idx) ensures regions are processed in the correct order.
5. **if isempty(r) || isempty(c)**
   - Skips the iteration if no valid coordinates are found for the region.
6. **Y = new_cleaned(min(r):max(r), min(c):max(c));**
   - Extracts the region of interest (Y) from the cleaned image (new_cleaned) using the bounding box defined by min(r), max(r), min(c), and max(c).
7. **if isempty(Y)**
   - Skips the iteration if the extracted region is empty.
8. **Y = imresize(Y, [42, 24]);**
   - Resizes the extracted region to a fixed size (42x24) to match the size of the reference images in TRAIN.
9. **ro = zeros(1, size(TRAIN, 2));**
   - Initializes an array (ro) to store correlation values between the extracted region (Y) and each reference image in TRAIN.
10. **for k = 1:size(TRAIN, 2)**
    - Loops over all reference images in TRAIN.
11. **ro(k) = corr2(TRAIN{1, k}, Y);**
    - Computes the 2D correlation coefficient between the current reference image (TRAIN{1, k}) and the extracted region (Y).
    - Stores the result in ro(k).
12. **[MAXRO, pos] = max(ro);**
    - Finds the maximum correlation value (MAXRO) and its position (pos) in the ro array.
13. **if MAXRO > 0.45**

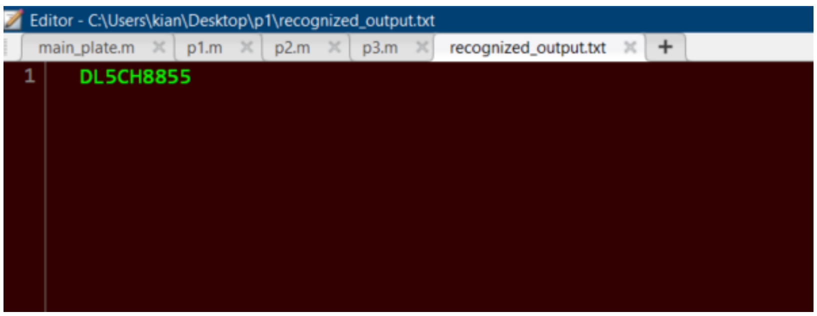
- Checks if the maximum correlation exceeds a threshold (0.45).
- This ensures only sufficiently similar matches are accepted.

14. **out = cell2mat(TRAIN(2, pos));**

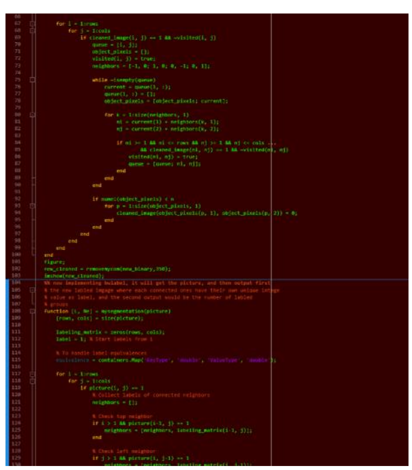
- Retrieves the label (character/number) corresponding to the best-matching reference image.

15. **final_output = [final_output out];**

- Appends the recognized character to the final_output string.



So as mentioned, we use corr2 to do a 2D correlation and also set a threshold for the match so we only print if we are almost sure (this helps me to avoid recognizing and printing some artifact out of nowhere) . then at the end we open a txt file for writing using fopen, then print in that file id and then close it and writing an accomplished message to confirm that we actually wrote in the .txt file . I guess this was the end of this part and these are the images for my p1.m scripts as asked to bring here :



```matlab
% =========================================================
% STEP 1: LOAD REFERENCE IMAGES FROM THE Map_Set DIRECTORY
% =========================================================
map_set_dir = 'Map_Set'; % Directory containing reference images (letters/numbers)

% Get a list of all .png and .bmp files in the directory
file_list_png = dir(fullfile(map_set_dir, '*.png')); % Get .png files
file_list_bmp = dir(fullfile(map_set_dir, '*.bmp')); % Get .bmp files

% Combine the file lists
file_list = [file_list_png; file_list_bmp];

% Initialize the TRAIN cell array: Row 1 = Images, Row 2 = Labels
TRAIN = cell(2, numel(file_list));

for k = 1:numel(file_list)
    % Load image and preprocess
    img = imread(fullfile(map_set_dir, file_list(k).name));
    if size(img, 3) == 3
        img = im2bw(img); % Convert RGB to binary
    end
    img = imresize(img, [42, 24]); % Resize to match character size

    % Store in TRAIN
    TRAIN{1, k} = img; % Image
    [~, label, ~] = fileparts(file_list(k).name); % Extract label (filename without extension)
    TRAIN{2, k} = label;
end

% Display the number of images loaded
fprintf('Loaded %d images from the Map_Set directory.\n', numel(file_list));

% =========================================================
% STEP 2: LICENSE PLATE SEGMENTATION AND RECOGNITION
% =========================================================
[L, Ne] = mysegmentation(new_cleaned);
propied = regionprops(L, 'BoundingBox', 'Area');

% Sort bounding boxes by x-coordinate (left-to-right)
boundingBoxes = cat(1, propied.BoundingBox);
[~, sortOrder] = sort(boundingBoxes(:, 1));
sorted_propied = propied(sortOrder);

% Filter out unwanted regions (e.g., outer border)
valid_indices = [];
for n = 1:Ne
    bbox = sorted_propied(n).BoundingBox;
    width = bbox(3);
    height = bbox(4);
    area = width * height;

    % Criteria (adjust these thresholds)
    max_area_threshold = 20000;
    min_area_threshold = 1000;
    aspect_ratio_range = [0.1, 4]; % [min_aspect_ratio, max_aspect_ratio]

    % Calculate aspect ratio
    aspect_ratio = width / height;

    % Check if the region meets all criteria
    if area > min_area_threshold && area < max_area_threshold && ...
       aspect_ratio >= aspect_ratio_range(1) && aspect_ratio <= aspect_ratio_range(2)
        valid_indices = [valid_indices, n]; % Add index to valid regions
    end
end
```

```matlab

filtered_propied = sorted_propied(valid_indices);
filtered_Ne = numel(valid_indices);

% Draw bounding boxes
figure;
imshow(new_cleaned);
hold on;
for n = 1:filtered_Ne
    rectangle('Position', filtered_propied(n).BoundingBox, 'EdgeColor', [0.75,0,0.75], 'LineWidth', 2);
end
hold off;

% Recognize characters
final_output = [];
for n = 1:filtered_Ne
    region_idx = valid_indices(n);
    [r, c] = find(L == sortOrder(region_idx));

    if isempty(r) || isempty(c)
        continue;
    end

    Y = new_cleaned(min(r):max(r), min(c):max(c));

    if isempty(Y)
        continue;
    end

    Y = imresize(Y, [42, 24]);

    ro = zeros(1, size(TRAIN, 2));
    for k = 1:size(TRAIN, 2)
        ro(k) = corr2(TRAIN{1, k}, Y);
    end

    [MAXRO, pos] = max(ro);
    if MAXRO > 0.45
        out = cell2mat(TRAIN(2, pos));
        final_output = [final_output out];
    end
end

% Display recognized output
disp('Recognized Output:');
disp(final_output);

% Save final_output to a text file
fileID = fopen('recognized_output.txt', 'w'); % Open file for writing
fprintf(fileID, '%s', final_output); % Write the recognized characters
fclose(fileID); % Close the file
disp('Recognized output saved to recognized_output.txt');
```

PART 2

In this part I just implement the first part code, but specialized for Persian map sets and car plates. And I used my own functions not the built in ones.

So without messing around, let's highlight the differences this code has from the previous one.

- `%new_picture = histeq(new_picture);`
- `new_picture = imgaussfilt(new_picture, 2);  % Adjust sigma as needed`
- `%new_picture = medfilt2(new_picture);`

  - The codes above are added at the end of 'to grayscale' part and as you can see this section only the middle one helped me and the rest are commented. These help the image to become sharper, increase contrast and reduce noise, I probably use all of them in next part but for this part the middle is enough
  - I set the binary threshold to 90, so it wouldn't absorb noise and artifacts in those Persian plates.
  - The map set directory changes to 'persian_matset'
  - Max_area_threshold would now be 15000 and the min_area_threshold would be 1450
  - The aspect ratio range is the same , and The rest is the same !!!
  - Also it's important to notice that the result printed on the console and the recognized_output2 would be in Fenglish, meaning I used M for example to represent 'م' and B to be 'ب' and etc, I hope you note that.

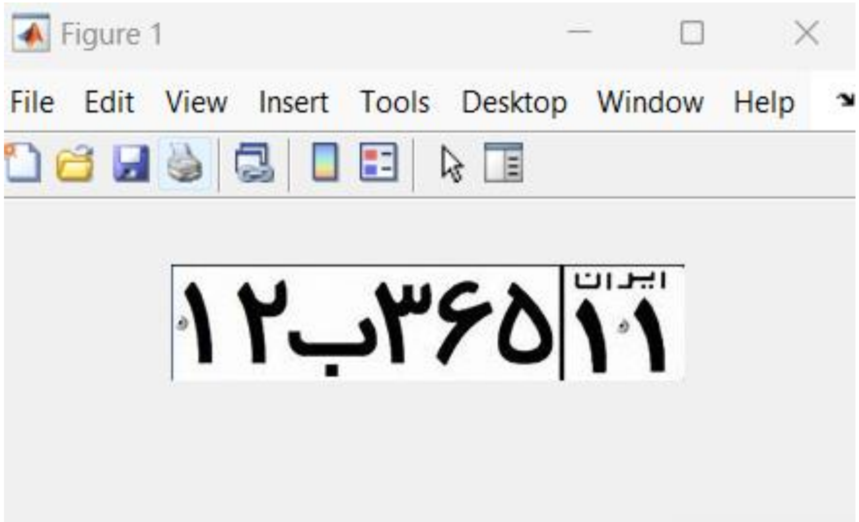


Figure 1
File Edit View Insert Tools Desktop Window Help
۱۲ب۳۶۵
ایران
۱۱

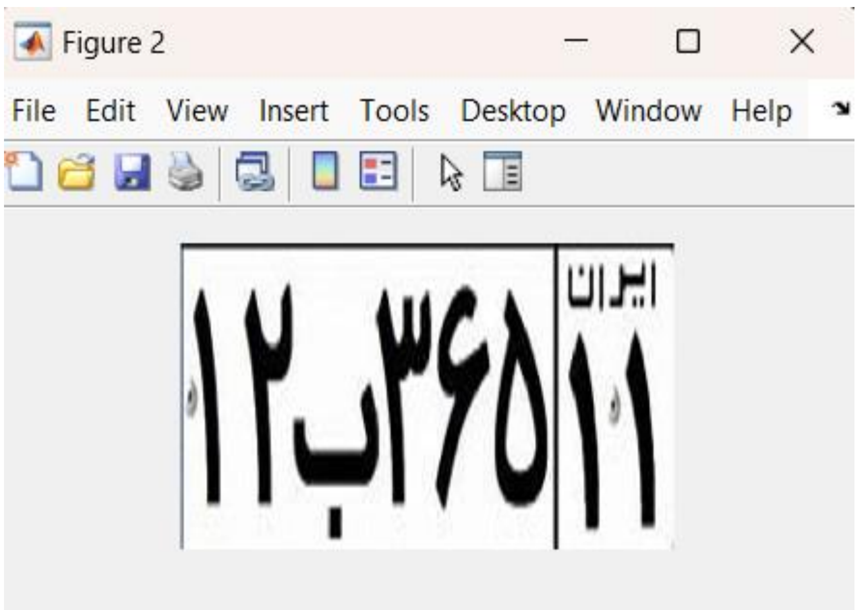


Figure 2
File Edit View Insert Tools Desktop Window Help
۱۲ب۳۶۵
ایران
۱۱

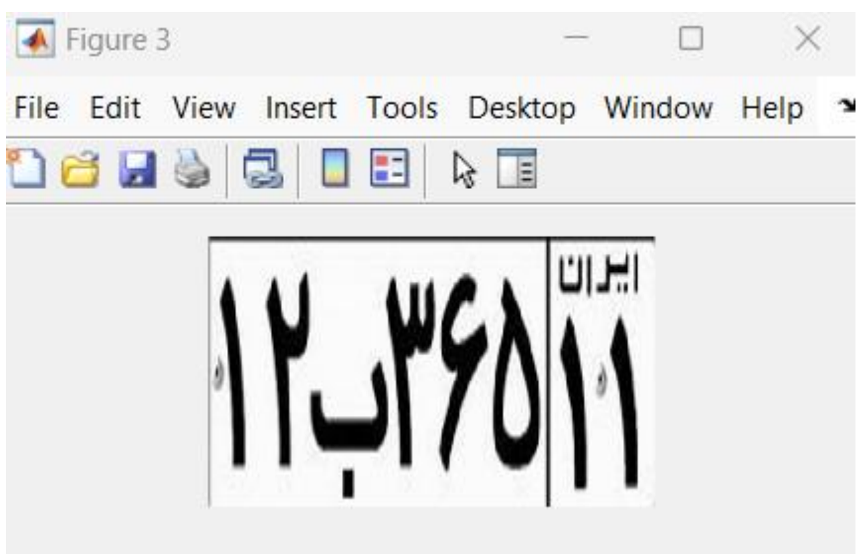


Figure 3
File Edit View Insert Tools Desktop Window Help
۱۲ب۳۶۵
ایران
۱۱

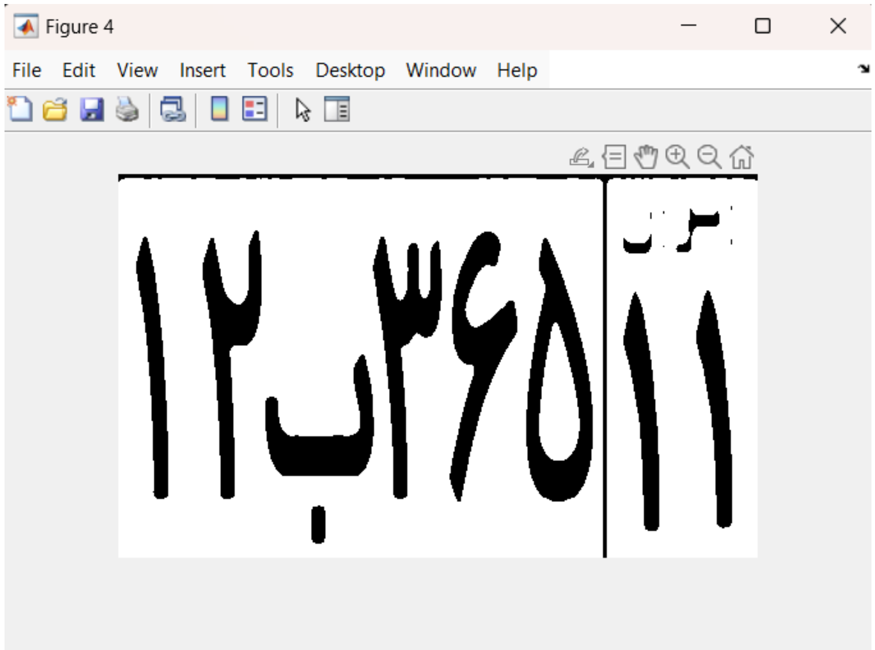


Figure 4
File Edit View Insert Tools Desktop Window Help
۱۲ ب ۳۶۵
۱۱

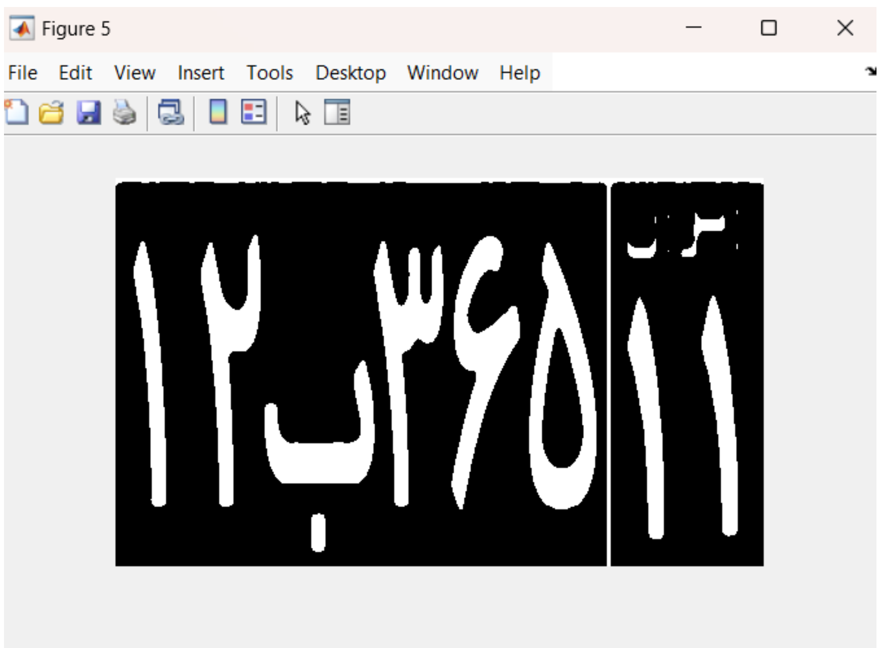


Figure 5
File Edit View Insert Tools Desktop Window Help
۱۲ ب ۳۶۵
۱۱

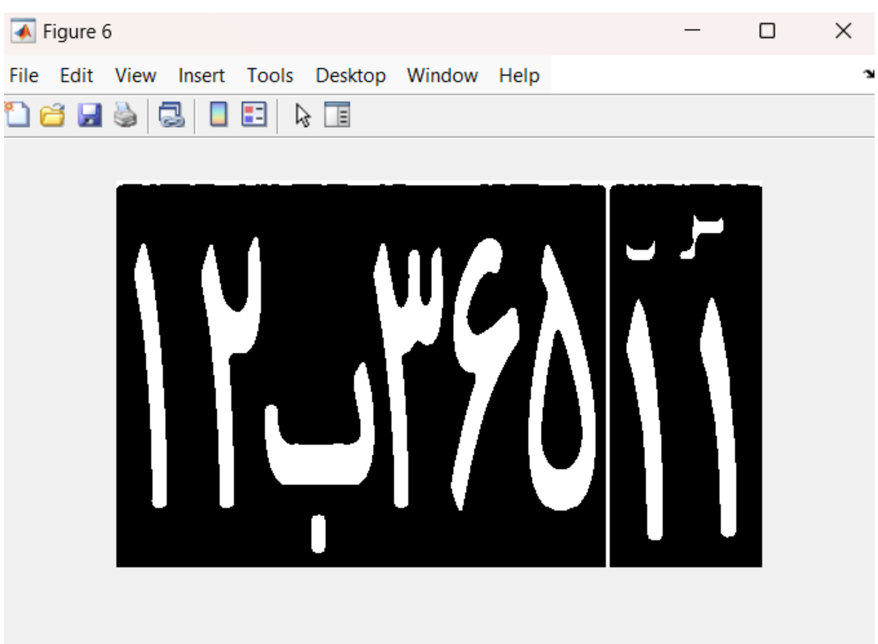
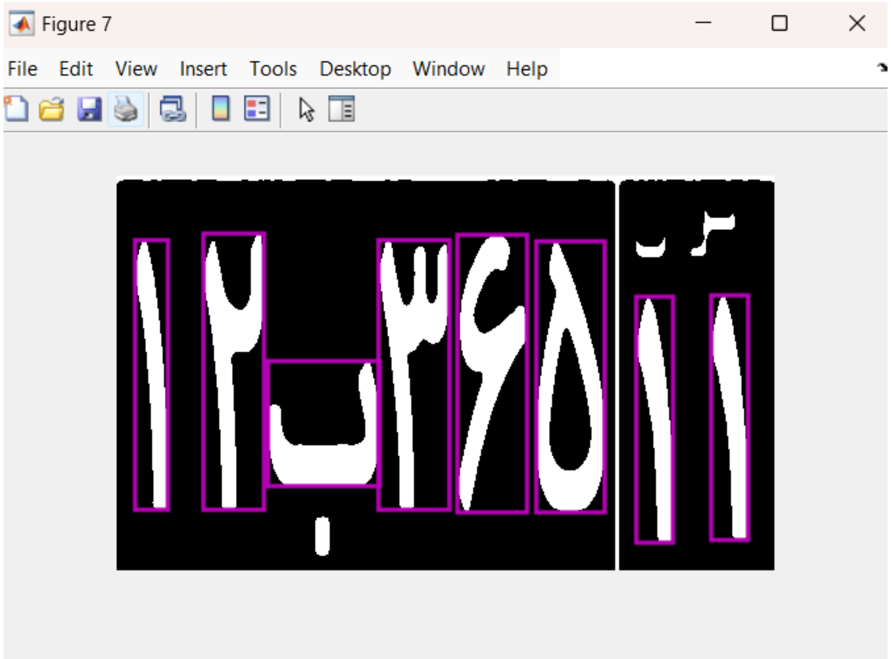
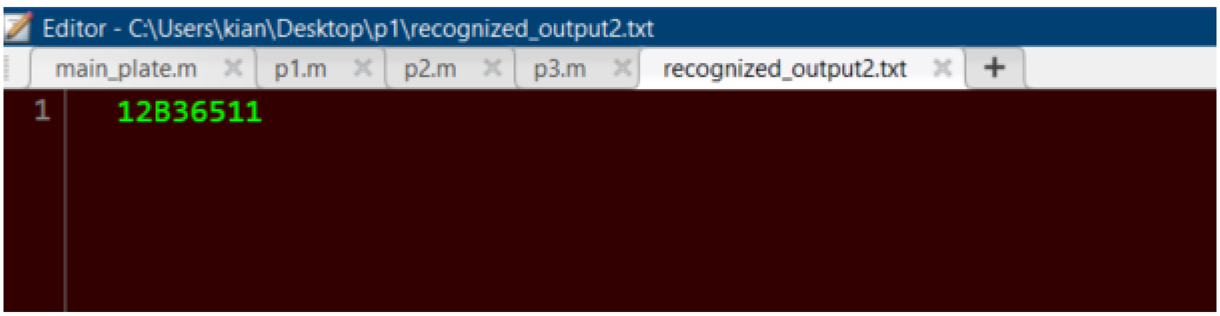


Figure 6
File Edit View Insert Tools Desktop Window Help
۱۲ ب ۳۶۵
۱۱





PART 3

In this section we are going to crop and pop out the plate out a car picture, and then do the rest to read its elements, so I also attached part 2 to its end so it would actually do something fun. The first part would be reading the template I chose for it and you can see it down here :

And this is going to be my template. I am gonna loop over my car picture And each time with a certain scale, I am gonna calculate the correlation And then increase the scale and do this again. Finally find the maximum Of this correlations and by some pre assumed constants I assumed (which I will talk more about it later) I will get the full image of the plate. Now

Without messing around let’s check my code and algorithm

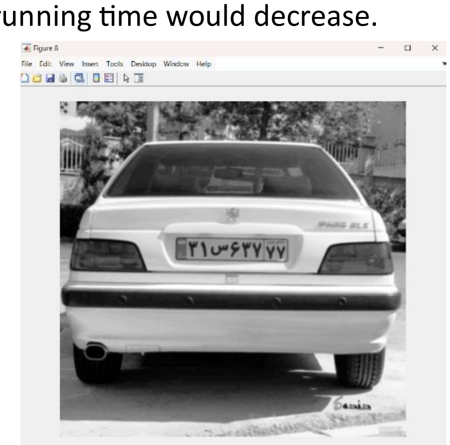
```
% Load the template
template_file = 'new.jpg';
plate_template = imread(template_file);
template_filee = 'OIP.jpg';
plate_templatee = imread(template_filee);
[rows1,cols1] = size(plate_template);
[rows2,cols2] = size(plate_templatee);
kian_constant = (cols2/cols1)*0.66;
disp(kian_constant)

[file, path] = uigetfile({'*.jpg; *.bmp; *.png', 'Choose the image'});
picture = imread(fullfile(path, file));
```

in this part I load that template and a full car plate, alongside the car image that user would choose, and reading them and then calculate that constant I was talking about, I divided the cols of the full one with cols of the template we had, and multiplied by 0.9 (by experimenting) and this would likely create my full plate later

```
% Convert images to grayscale (if they are RGB)
if size(picture, 3) == 3
    gray_image = rgb2gray(picture);
else
    gray_image = picture;
end
gray_image = histeq(gray_image);
gray_image = imgaussfilt(gray_image, 2);  % Adjust sigma as needed
gray_image = medfilt2(gray_image);

figure;

if size(plate_template, 3) == 3
    gray_template = rgb2gray(plate_template);
else
    gray_template = plate_template;
end
imshow(gray_image);
```

then I convert the car image to grayscale so the calculation and running time would decrease.

I also used those filtering stuff I commented two of them , here.




```
% Define the range of scaling factors)
scales = 0.5:0.05:2;
best_corr = -inf;   % initialize best correlation score
best_scale = 1;
best_ypk = 0;
```

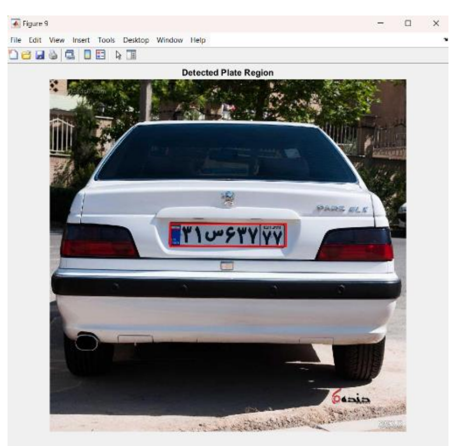
```
best_xpk = 0;
best_template = gray_template; % best matched (resized) template
```

these are our initializations and the interval for the scales . note that I found those scales by experimenting on different plates, too.

```
% Loop over the scales and compute correlation at each scale
for s = scales
    % Resize the template keeping the aspect ratio constant
    resized_template = imresize(gray_template, s);

    % Compute normalized cross-correlation
    corr = normxcorr2(resized_template, gray_image);

    % Find the peak correlation value and its location
    [ypeak, xpeak] = find(corr == max(corr(:)));
    current_corr = max(corr(:));

    % Update the best match if current correlation is higher
    if current_corr > best_corr
        best_corr = current_corr;
        best_scale = s;
        best_ypk = ypeak;
        best_xpk = xpeak;
        best_template = resized_template;
    end
end
```

my comments are clear, but I mention the core again, we loop over those scales, and for each scale , resize the template according to it, and calculate the correlation using normxcorr2. Then find where it peaked with that scale and if it is the new highest record, replace the older record.

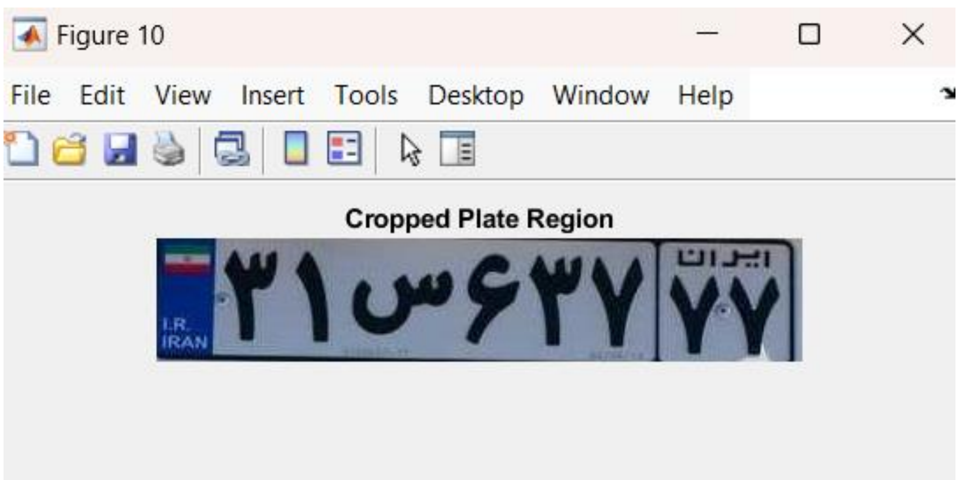
```
% Note: normxcorr2 returns coordinates with an offset equal to the template size.
plate_x = best_xpk - size(best_template, 2) + 1;
plate_y = best_ypk - size(best_template, 1) + 1;
plate_width = size(best_template, 2);
plate_height = size(best_template, 1);

figure;
imshow(picture);
hold on;
rectangle('Position', [plate_x, plate_y, plate_width*kian_constant, plate_height],
'EdgeColor', 'r', 'LineWidth', 2);
title('Detected Plate Region');
```

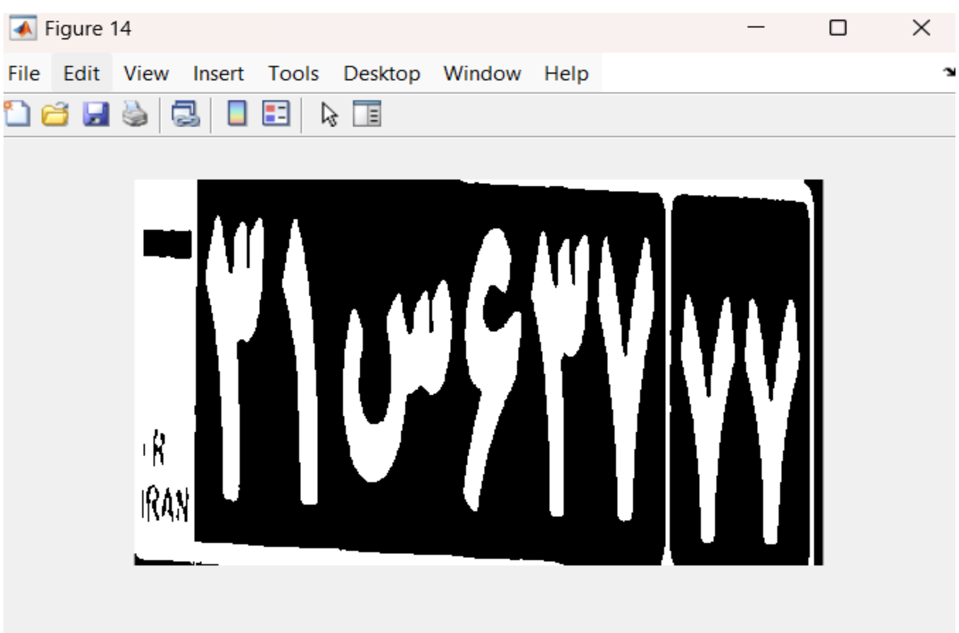


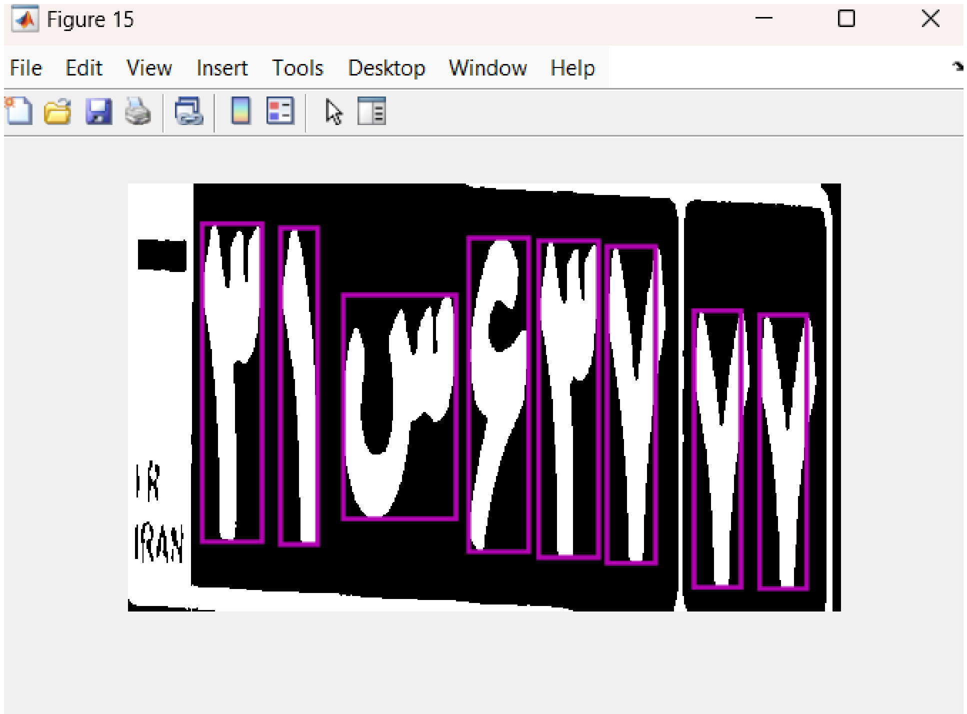
in the first part of this code, I calculate the plate’s width and height, but it would be only that blue rectangular with Iran’s flag on it, I want the whole plate so I use kian_constant and multiply it by that width already found, and now when I print that red rectangular on my car image and later crop that, the width would be extended to plate’s width

```
% Crop the plate region from the original image
plate_region = imcrop(picture, [plate_x, plate_y, plate_width*kian_constant,
plate_height]);

% Display the cropped plate region
figure;
imshow(plate_region);
title('Cropped Plate Region');
```

these are two last steps, cropping that plate using imcrop, with input values influenced by kian_constant, and then printing that new plate on a new figure.








```
Loaded 24 images from the Map_Set directory.
Recognized Output:
31S63777
>>
```

Again notice that these are fenglish

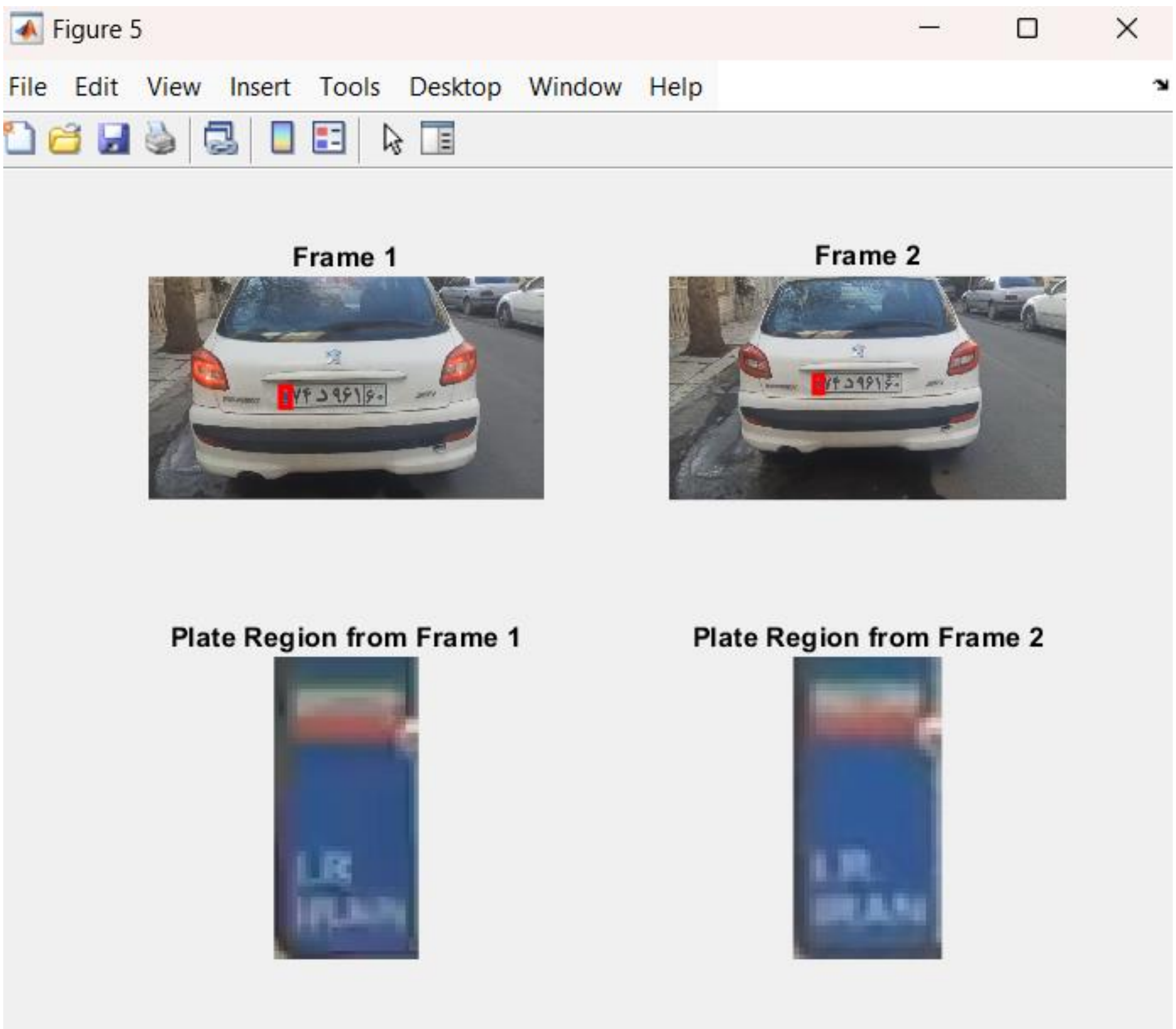
```matlab
% Load the template
template_file = 'new.jpg';
plate_template = imread(template_file);
template_filee = 'OIP.jpg';
plate_templatee = imread(template_filee);
[rows1,cols1] = size(plate_template);
[rows2,cols2] = size(plate_templatee);
kian_constant = (cols2/cols1)*0.66;
disp(kian_constant)

[file, path] = uigetfile({'*.jpg; *.bmp; *.png', 'Choose the image'});
picture = imread(fullfile(path, file));

% Convert images to grayscale (if they are RGB)
if size(picture, 3) == 3
    gray_image = rgb2gray(picture);
else
    gray_image = picture;
end
gray_image = histeq(gray_image);
gray_image = imgaussfilt(gray_image, 2);  % Adjust sigma as needed
gray_image = medfilt2(gray_image);

figure;

if size(plate_template, 3) == 3
    gray_template = rgb2gray(plate_template);
else
    gray_template = plate_template;
end
imshow(gray_image);
% Define the range of scaling factors)
scales = 0.5:0.05:2;
best_corr = -inf;   % initialize best correlation score
best_scale = 1;
best_ypk = 0;
best_xpk = 0;
best_template = gray_template; % best matched (resized) template

% Loop over the scales and compute correlation at each scale
for s = scales
    % Resize the template keeping the aspect ratio constant
```

```matlab
        resized_template = imresize(gray_template, s);
        % Compute normalized cross-correlation
        corr = normxcorr2(resized_template, gray_image);

        % Find the peak correlation value and its location
        [ypeak, xpeak] = find(corr == max(corr(:)));
        current_corr = max(corr(:));

        % Update the best match if current correlation is higher
        if current_corr > best_corr
            best_corr = current_corr;
            best_scale = s;
            best_ypk = ypeak;
            best_xpk = xpeak;
            best_template = resized_template;
        end
    end
    % Note: normxcorr2 returns coordinates with an offset equal to the template size.
    plate_x = best_xpk - size(best_template, 2) + 1;
    plate_y = best_ypk - size(best_template, 1) + 1;
    plate_width = size(best_template, 2);
    plate_height = size(best_template, 1);

    figure;
    imshow(picture);
    hold on;
    rectangle('Position', [plate_x, plate_y, plate_width*kian_constant, plate_height], 'EdgeColor', 'r', 'LineWidth', 2);
    title('Detected Plate Region');

    % Crop the plate region from the original image
    plate_region = imcrop(picture, [plate_x, plate_y, plate_width*kian_constant, plate_height]);

    % Display the cropped plate region
    figure;
    imshow(plate_region);
    title('Cropped Plate Region');
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% from now on it's the
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% same as earlier parts
```

Part 4 is on the next page

PART 4

In this part I wanna find the speed of the car in my video, my idea and approach is that I get those plate again in the two frames we get from the video, and then check how much the size and scale is changed, and by some pre assuming and contracting, I would relate a distance taken to each change in scale, and by dividing that to the time between the frames, I will find the average speed of the car approximately. So I did some contracting and assumed some constants to a given change in scale, and now I can find any speed related to a change in scale,



```
template_file = 'new.jpg';
plate_template = imread(template_file);

[file, path] = uigetfile({'*.mp4;*.avi'}, 'Select Video File');
video_path = fullfile(path, file);
video = VideoReader(video_path);

known_distance = 0.40; % 40 cm in meters
t1 = 0.0;
t2 = 1.5;

frame1 = read(video, round(t1 * video.FrameRate) + 1);
frame2 = read(video, round(t2 * video.FrameRate) + 1);

[scale1, proc_frame1, rect1] = process_frame(frame1, plate_template);
[scale2, proc_frame2, rect2] = process_frame(frame2, plate_template);

% Calibrate constant using known distance and measured scales
kian_constant = known_distance / (1/scale2 - 1/scale1);

time_interval = t2 - t1;
speed = calculate_speed(scale1, scale2, kian_constant, time_interval);

disp(['Estimated Speed: ', num2str(speed), ' km/h']);

figure;
subplot(2,2,1);
imshow(frame1);
if ~isempty(rect1)
    rectangle('Position', rect1, 'EdgeColor', 'r', 'LineWidth', 2);
end
title('Frame 1');

subplot(2,2,2);
imshow(frame2);
if ~isempty(rect2)
    rectangle('Position', rect2, 'EdgeColor', 'r', 'LineWidth', 2);
end
title('Frame 2');

if ~isempty(rect1)
    plate_region1 = imcrop(frame1, rect1);
    subplot(2,2,3);
    imshow(plate_region1);
    title('Plate Region from Frame 1');
end
if ~isempty(rect2)
    plate_region2 = imcrop(frame2, rect2);
    subplot(2,2,4);
    imshow(plate_region2);
    title('Plate Region from Frame 2');
end

function [best_scale, gray_image, best_match_rect] = process_frame(picture,
plate_template)
```

```matlab
    if size(picture, 3) == 3
        gray_image = rgb2gray(picture);
    else
        gray_image = picture;
    end
    gray_image = histeq(gray_image);
    gray_image = imgaussfilt(gray_image, 2);
    gray_image = medfilt2(gray_image);

    if size(plate_template, 3) == 3
        gray_template = rgb2gray(plate_template);
    else
        gray_template = plate_template;
    end

    scales = 0.5:0.05:2;
    best_corr = -inf;
    best_scale = 1;
    best_match_rect = [];

    for s = scales
        resized_template = imresize(gray_template, s);
        corr = normxcorr2(resized_template, gray_image);
        current_corr = max(corr(:));

        if current_corr > best_corr
            best_corr = current_corr;
            best_scale = s;
            [ypeak, xpeak] = find(corr == current_corr, 1);
            yoff = ypeak - size(resized_template,1);
            xoff = xpeak - size(resized_template,2);
            best_match_rect = [xoff+1, yoff+1, size(resized_template,2),
size(resized_template,1)];
        end
    end
end

function speed = calculate_speed(scale1, scale2, k_const, dt)
    distance1 = k_const / scale1;
    distance2 = k_const / scale2;
    speed = abs(distance2 - distance1) / dt * 3.6; % Convert m/s to km/h
end
```

- **Load Data** – Reads plate template and prompts the user to select a video.
- **Extract Frames** – Captures frames at **t = 0s** and **t = 1.5s**.
- **Detect Plate** – Identifies the plate in each frame using **template matching** and finds its scale.
- **Compute Speed** – Uses the known **40 cm distance** to calibrate scale changes and calculate speed.
- **Display Results** – Shows the original frames and detected plate regions for verification.