

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
new_picture = mygrayfun(picture);
figure;
imshow(mygrayfun(picture));
```



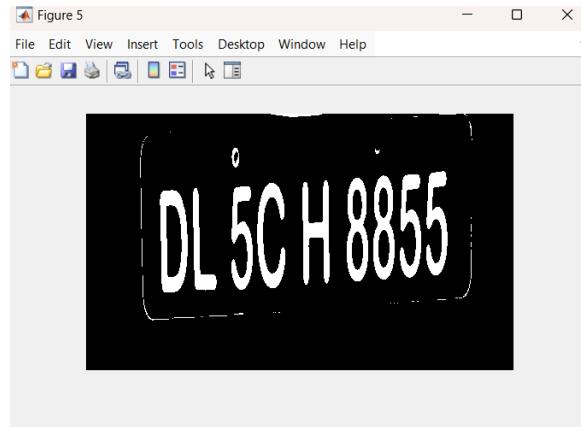
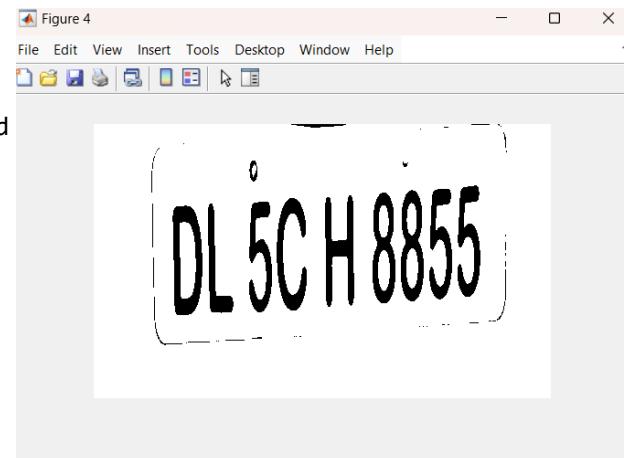
- 4) In this part we go one step further, turning the grayscale picture to binary (1s for black, and 0s for white) and to do so, I wrote the function mybinaryfun that receives the image, along with a threshold that if the grayscale value was above that it would give it value 1, and vice versa. It is 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, NUMCOLUMNS] = size(picture,1,2);
    binary = zeros(NUMROWS,NUMCOLUMNS);
    for i = 1:1:NUMROWS
        for j = 1:1:NUMCOLUMNS
            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 labeled 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);

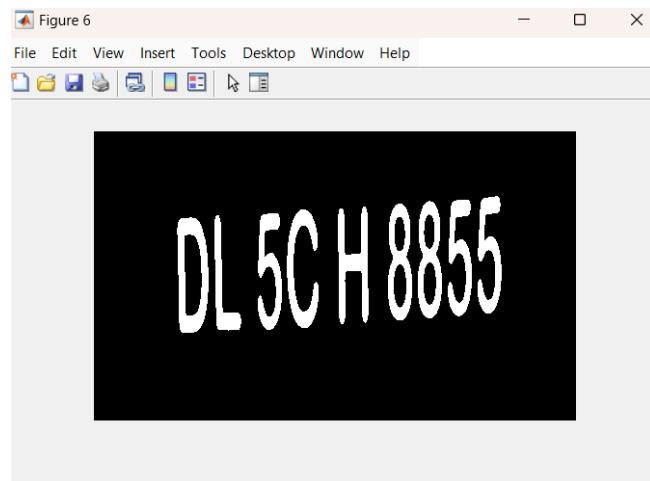
for i = 1:rows
    for j = 1:cols
        if cleaned_image(i, j) == 1 && ~visited(i, j)
            queue = [i, j];
            object_pixels = [];
            visited(i, j) = true;
            neighbors = [-1, 0; 1, 0; 0, -1; 0, 1];

            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 if 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 and 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 $\text{equivalence}(2) = 1$ and $\text{equivalence}(1) = 3$, $\text{find_root}(2)$ 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
    end
```

```

        root = 1;
    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

```

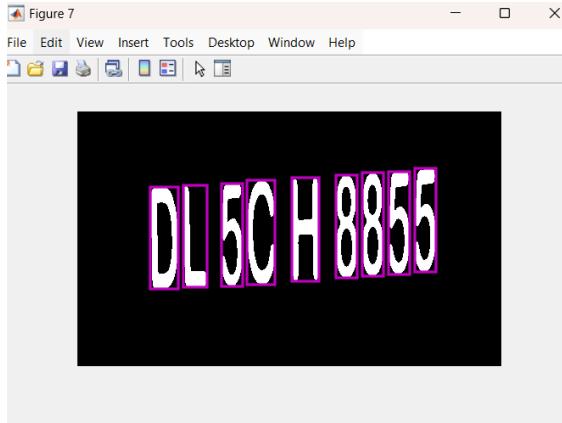
in the next part we draw those boxes we already projected, but later filtered. I used RGB values to draw them with purple/magenta color :

```

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`**

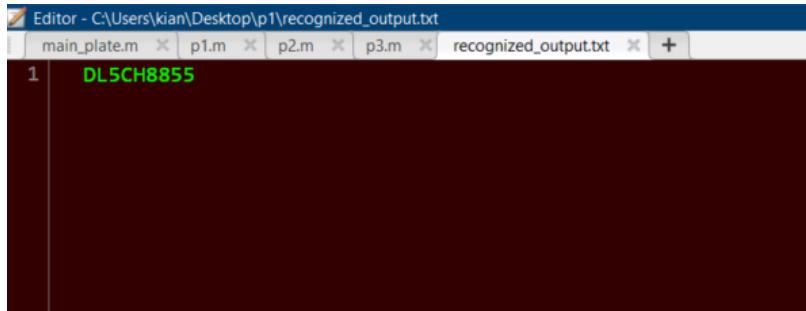
- Checks if the maximum correlation exceeds a threshold (0.45).
 - This ensures only sufficiently similar matches are accepted.

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 :

```

10 // This is the main function for the program
11 void main() {
12     // Initialize the matrix
13     int N = 10;
14     int** matrix = new int*[N];
15     for (int i = 0; i < N; i++) {
16         matrix[i] = new int[N];
17         for (int j = 0; j < N; j++) {
18             matrix[i][j] = 0;
19         }
20     }
21     // Set some values in the matrix
22     matrix[0][1] = 1;
23     matrix[1][0] = 1;
24     matrix[2][3] = 1;
25     matrix[3][2] = 1;
26     matrix[4][5] = 1;
27     matrix[5][4] = 1;
28     matrix[6][7] = 1;
29     matrix[7][6] = 1;
30     matrix[8][9] = 1;
31     matrix[9][8] = 1;
32     // Print the matrix
33     printMatrix(matrix);
34     // Call the transpose function
35     transposeMatrix(matrix);
36     // Print the transposed matrix
37     printMatrix(matrix);
38 }
39
40 // Function to print the matrix
41 void printMatrix(int** matrix) {
42     for (int i = 0; i < 10; i++) {
43         for (int j = 0; j < 10; j++) {
44             cout << matrix[i][j] << " ";
45         }
46         cout << endl;
47     }
48 }
49
50 // Function to transpose the matrix
51 void transposeMatrix(int** matrix) {
52     for (int i = 0; i < 10; i++) {
53         for (int j = 0; j < 10; j++) {
54             if (matrix[i][j] == 1) {
55                 matrix[j][i] = 1;
56             }
57         }
58     }
59 }
60
61 // Function to calculate the determinant of a matrix
62 double calculateDeterminant(int** matrix, int N) {
63     if (N == 1) {
64         return matrix[0][0];
65     }
66     if (N == 2) {
67         return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
68     }
69     double result = 0;
70     for (int i = 0; i < N; i++) {
71         result += matrix[0][i] * calculateDeterminant(minorMatrix(matrix, 0, i), N - 1);
72         if (i % 2 == 1) {
73             result -= matrix[0][i] * calculateDeterminant(minorMatrix(matrix, 0, i), N - 1);
74         }
75     }
76     return result;
77 }
78
79 // Function to calculate the minor matrix of a given row and column
80 int** minorMatrix(int** matrix, int row, int col, int N) {
81     int** minorMatrix = new int*[N - 1];
82     for (int i = 0; i < N - 1; i++) {
83         minorMatrix[i] = new int[N - 1];
84         for (int j = 0; j < N - 1; j++) {
85             if (i == row || j == col) {
86                 continue;
87             }
88             minorMatrix[i][j] = matrix[i + 1][j + 1];
89         }
90     }
91     return minorMatrix;
92 }
93
94 // Function to calculate the cofactor matrix
95 int** cofactorMatrix(int** matrix, int N) {
96     int** cofactorMatrix = new int*[N];
97     for (int i = 0; i < N; i++) {
98         cofactorMatrix[i] = new int[N];
99         for (int j = 0; j < N; j++) {
100            if ((i + j) % 2 == 1) {
101                cofactorMatrix[i][j] = -matrix[i][j];
102            } else {
103                cofactorMatrix[i][j] = matrix[i][j];
104            }
105        }
106    }
107    return cofactorMatrix;
108 }
109
110 // Function to calculate the adjoint matrix
111 int** adjointMatrix(int** matrix, int N) {
112     int** adjointMatrix = new int*[N];
113     for (int i = 0; i < N; i++) {
114         adjointMatrix[i] = new int[N];
115         for (int j = 0; j < N; j++) {
116             adjointMatrix[i][j] = cofactorMatrix(matrix, N)[i][j];
117         }
118     }
119     return adjointMatrix;
120 }
121
122 // Function to calculate the inverse matrix
123 int** inverseMatrix(int** matrix, int N) {
124     double determinant = calculateDeterminant(matrix, N);
125     if (determinant == 0) {
126         cout << "The matrix is singular, no inverse exists." << endl;
127         return NULL;
128     }
129     int** adjoint = adjointMatrix(matrix, N);
130     int** inverse = new int*[N];
131     for (int i = 0; i < N; i++) {
132         inverse[i] = new int[N];
133         for (int j = 0; j < N; j++) {
134             inverse[i][j] = adjoint[i][j] / determinant;
135         }
136     }
137     return inverse;
138 }
139
140 // Function to calculate the trace of a matrix
141 double traceMatrix(int** matrix, int N) {
142     double trace = 0;
143     for (int i = 0; i < N; i++) {
144         trace += matrix[i][i];
145     }
146     return trace;
147 }
148
149 // Function to calculate the determinant of a sparse matrix
150 double calculateSparseDeterminant(int** matrix, int N) {
151     if (N == 1) {
152         return matrix[0][0];
153     }
154     if (N == 2) {
155         return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
156     }
157     double result = 0;
158     for (int i = 0; i < N; i++) {
159         if (matrix[0][i] != 0) {
160             result += matrix[0][i] * calculateSparseDeterminant(minorMatrix(matrix, 0, i), N - 1);
161             if (i % 2 == 1) {
162                 result -= matrix[0][i] * calculateSparseDeterminant(minorMatrix(matrix, 0, i), N - 1);
163             }
164         }
165     }
166     return result;
167 }
168
169 // Function to calculate the minor matrix of a given row and column for a sparse matrix
170 int** minorMatrixSparse(int** matrix, int row, int col, int N) {
171     int** minorMatrix = new int*[N - 1];
172     for (int i = 0; i < N - 1; i++) {
173         minorMatrix[i] = new int[N - 1];
174         for (int j = 0; j < N - 1; j++) {
175             if (i == row || j == col) {
176                 continue;
177             }
178             minorMatrix[i][j] = matrix[i + 1][j + 1];
179         }
180     }
181     return minorMatrix;
182 }
183
184 // Function to calculate the cofactor matrix for a sparse matrix
185 int** cofactorMatrixSparse(int** matrix, int N) {
186     int** cofactorMatrix = new int*[N];
187     for (int i = 0; i < N; i++) {
188         cofactorMatrix[i] = new int[N];
189         for (int j = 0; j < N; j++) {
190             if ((i + j) % 2 == 1) {
191                 cofactorMatrix[i][j] = -matrix[i][j];
192             } else {
193                 cofactorMatrix[i][j] = matrix[i][j];
194             }
195         }
196     }
197     return cofactorMatrix;
198 }
199
200 // Function to calculate the adjoint matrix for a sparse matrix
201 int** adjointMatrixSparse(int** matrix, int N) {
202     int** adjointMatrix = new int*[N];
203     for (int i = 0; i < N; i++) {
204         adjointMatrix[i] = new int[N];
205         for (int j = 0; j < N; j++) {
206             adjointMatrix[i][j] = cofactorMatrixSparse(matrix, N)[i][j];
207         }
208     }
209     return adjointMatrix;
210 }
211
212 // Function to calculate the inverse matrix for a sparse matrix
213 int** inverseMatrixSparse(int** matrix, int N) {
214     double determinant = calculateSparseDeterminant(matrix, N);
215     if (determinant == 0) {
216         cout << "The matrix is singular, no inverse exists." << endl;
217         return NULL;
218     }
219     int** adjoint = adjointMatrixSparse(matrix, N);
220     int** inverse = new int*[N];
221     for (int i = 0; i < N; i++) {
222         inverse[i] = new int[N];
223         for (int j = 0; j < N; j++) {
224             inverse[i][j] = adjoint[i][j] / determinant;
225         }
226     }
227     return inverse;
228 }
229
230 // Function to calculate the trace of a sparse matrix
231 double traceMatrixSparse(int** matrix, int N) {
232     double trace = 0;
233     for (int i = 0; i < N; i++) {
234         if (matrix[0][i] != 0) {
235             trace += matrix[0][i];
236         }
237     }
238     return trace;
239 }
240
241 // Function to calculate the determinant of a symmetric matrix
242 double calculateSymmetricDeterminant(int** matrix, int N) {
243     if (N == 1) {
244         return matrix[0][0];
245     }
246     if (N == 2) {
247         return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
248     }
249     double result = 0;
250     for (int i = 0; i < N; i++) {
251         if (matrix[0][i] != 0) {
252             result += matrix[0][i] * calculateSymmetricDeterminant(minorMatrix(matrix, 0, i), N - 1);
253             if (i % 2 == 1) {
254                 result -= matrix[0][i] * calculateSymmetricDeterminant(minorMatrix(matrix, 0, i), N - 1);
255             }
256         }
257     }
258     return result;
259 }
260
261 // Function to calculate the minor matrix of a given row and column for a symmetric matrix
262 int** minorMatrixSymmetric(int** matrix, int row, int col, int N) {
263     int** minorMatrix = new int*[N - 1];
264     for (int i = 0; i < N - 1; i++) {
265         minorMatrix[i] = new int[N - 1];
266         for (int j = 0; j < N - 1; j++) {
267             if (i == row || j == col) {
268                 continue;
269             }
270             minorMatrix[i][j] = matrix[i + 1][j + 1];
271         }
272     }
273     return minorMatrix;
274 }
275
276 // Function to calculate the cofactor matrix for a symmetric matrix
277 int** cofactorMatrixSymmetric(int** matrix, int N) {
278     int** cofactorMatrix = new int*[N];
279     for (int i = 0; i < N; i++) {
280         cofactorMatrix[i] = new int[N];
281         for (int j = 0; j < N; j++) {
282             if ((i + j) % 2 == 1) {
283                 cofactorMatrix[i][j] = -matrix[i][j];
284             } else {
285                 cofactorMatrix[i][j] = matrix[i][j];
286             }
287         }
288     }
289     return cofactorMatrix;
290 }
291
292 // Function to calculate the adjoint matrix for a symmetric matrix
293 int** adjointMatrixSymmetric(int** matrix, int N) {
294     int** adjointMatrix = new int*[N];
295     for (int i = 0; i < N; i++) {
296         adjointMatrix[i] = new int[N];
297         for (int j = 0; j < N; j++) {
298             adjointMatrix[i][j] = cofactorMatrixSymmetric(matrix, N)[i][j];
299         }
300     }
301     return adjointMatrix;
302 }
303
304 // Function to calculate the inverse matrix for a symmetric matrix
305 int** inverseMatrixSymmetric(int** matrix, int N) {
306     double determinant = calculateSymmetricDeterminant(matrix, N);
307     if (determinant == 0) {
308         cout << "The matrix is singular, no inverse exists." << endl;
309         return NULL;
310     }
311     int** adjoint = adjointMatrixSymmetric(matrix, N);
312     int** inverse = new int*[N];
313     for (int i = 0; i < N; i++) {
314         inverse[i] = new int[N];
315         for (int j = 0; j < N; j++) {
316             inverse[i][j] = adjoint[i][j] / determinant;
317         }
318     }
319     return inverse;
320 }
321
322 // Function to calculate the trace of a symmetric matrix
323 double traceMatrixSymmetric(int** matrix, int N) {
324     double trace = 0;
325     for (int i = 0; i < N; i++) {
326         if (matrix[0][i] != 0) {
327             trace += matrix[0][i];
328         }
329     }
330     return trace;
331 }
332
333 // Function to calculate the determinant of a triangular matrix
334 double calculateTriangularDeterminant(int** matrix, int N) {
335     if (N == 1) {
336         return matrix[0][0];
337     }
338     if (N == 2) {
339         return matrix[0][0] * matrix[1][1];
340     }
341     double result = 0;
342     for (int i = 0; i < N; i++) {
343         if (matrix[0][i] != 0) {
344             result += matrix[0][i] * calculateTriangularDeterminant(minorMatrix(matrix, 0, i), N - 1);
345         }
346     }
347     return result;
348 }
349
350 // Function to calculate the minor matrix of a given row and column for a triangular matrix
351 int** minorMatrixTriangular(int** matrix, int row, int col, int N) {
352     int** minorMatrix = new int*[N - 1];
353     for (int i = 0; i < N - 1; i++) {
354         minorMatrix[i] = new int[N - 1];
355         for (int j = 0; j < N - 1; j++) {
356             if (i == row || j == col) {
357                 continue;
358             }
359             minorMatrix[i][j] = matrix[i + 1][j + 1];
360         }
361     }
362     return minorMatrix;
363 }
364
365 // Function to calculate the cofactor matrix for a triangular matrix
366 int** cofactorMatrixTriangular(int** matrix, int N) {
367     int** cofactorMatrix = new int*[N];
368     for (int i = 0; i < N; i++) {
369         cofactorMatrix[i] = new int[N];
370         for (int j = 0; j < N; j++) {
371             if ((i + j) % 2 == 1) {
372                 cofactorMatrix[i][j] = -matrix[i][j];
373             } else {
374                 cofactorMatrix[i][j] = matrix[i][j];
375             }
376         }
377     }
378     return cofactorMatrix;
379 }
380
381 // Function to calculate the adjoint matrix for a triangular matrix
382 int** adjointMatrixTriangular(int** matrix, int N) {
383     int** adjointMatrix = new int*[N];
384     for (int i = 0; i < N; i++) {
385         adjointMatrix[i] = new int[N];
386         for (int j = 0; j < N; j++) {
387             adjointMatrix[i][j] = cofactorMatrixTriangular(matrix, N)[i][j];
388         }
389     }
390     return adjointMatrix;
391 }
392
393 // Function to calculate the inverse matrix for a triangular matrix
394 int** inverseMatrixTriangular(int** matrix, int N) {
395     double determinant = calculateTriangularDeterminant(matrix, N);
396     if (determinant == 0) {
397         cout << "The matrix is singular, no inverse exists." << endl;
398         return NULL;
399     }
400     int** adjoint = adjointMatrixTriangular(matrix, N);
401     int** inverse = new int*[N];
402     for (int i = 0; i < N; i++) {
403         inverse[i] = new int[N];
404         for (int j = 0; j < N; j++) {
405             inverse[i][j] = adjoint[i][j] / determinant;
406         }
407     }
408     return inverse;
409 }
410
411 // Function to calculate the trace of a triangular matrix
412 double traceMatrixTriangular(int** matrix, int N) {
413     double trace = 0;
414     for (int i = 0; i < N; i++) {
415         if (matrix[0][i] != 0) {
416             trace += matrix[0][i];
417         }
418     }
419     return trace;
420 }
421
422 // Function to calculate the determinant of a sparse triangular matrix
423 double calculateSparseTriangularDeterminant(int** matrix, int N) {
424     if (N == 1) {
425         return matrix[0][0];
426     }
427     if (N == 2) {
428         return matrix[0][0] * matrix[1][1];
429     }
430     double result = 0;
431     for (int i = 0; i < N; i++) {
432         if (matrix[0][i] != 0) {
433             result += matrix[0][i] * calculateSparseTriangularDeterminant(minorMatrixSparse(matrix, 0, i), N - 1);
434         }
435     }
436     return result;
437 }
438
439 // Function to calculate the minor matrix of a given row and column for a sparse triangular matrix
440 int** minorMatrixSparseTriangular(int** matrix, int row, int col, int N) {
441     int** minorMatrix = new int*[N - 1];
442     for (int i = 0; i < N - 1; i++) {
443         minorMatrix[i] = new int[N - 1];
444         for (int j = 0; j < N - 1; j++) {
445             if (i == row || j == col) {
446                 continue;
447             }
448             minorMatrix[i][j] = matrix[i + 1][j + 1];
449         }
450     }
451     return minorMatrix;
452 }
453
454 // Function to calculate the cofactor matrix for a sparse triangular matrix
455 int** cofactorMatrixSparseTriangular(int** matrix, int N) {
456     int** cofactorMatrix = new int*[N];
457     for (int i = 0; i < N; i++) {
458         cofactorMatrix[i] = new int[N];
459         for (int j = 0; j < N; j++) {
460             if ((i + j) % 2 == 1) {
461                 cofactorMatrix[i][j] = -matrix[i][j];
462             } else {
463                 cofactorMatrix[i][j] = matrix[i][j];
464             }
465         }
466     }
467     return cofactorMatrix;
468 }
469
470 // Function to calculate the adjoint matrix for a sparse triangular matrix
471 int** adjointMatrixSparseTriangular(int** matrix, int N) {
472     int** adjointMatrix = new int*[N];
473     for (int i = 0; i < N; i++) {
474         adjointMatrix[i] = new int[N];
475         for (int j = 0; j < N; j++) {
476             adjointMatrix[i][j] = cofactorMatrixSparseTriangular(matrix, N)[i][j];
477         }
478     }
479     return adjointMatrix;
480 }
481
482 // Function to calculate the inverse matrix for a sparse triangular matrix
483 int** inverseMatrixSparseTriangular(int** matrix, int N) {
484     double determinant = calculateSparseTriangularDeterminant(matrix, N);
485     if (determinant == 0) {
486         cout << "The matrix is singular, no inverse exists." << endl;
487         return NULL;
488     }
489     int** adjoint = adjointMatrixSparseTriangular(matrix, N);
490     int** inverse = new int*[N];
491     for (int i = 0; i < N; i++) {
492         inverse[i] = new int[N];
493         for (int j = 0; j < N; j++) {
494             inverse[i][j] = adjoint[i][j] / determinant;
495         }
496     }
497     return inverse;
498 }
499
500 // Function to calculate the trace of a sparse triangular matrix
501 double traceMatrixSparseTriangular(int** matrix, int N) {
502     double trace = 0;
503     for (int i = 0; i < N; i++) {
504         if (matrix[0][i] != 0) {
505             trace += matrix[0][i];
506         }
507     }
508     return trace;
509 }
510
511 // Function to calculate the determinant of a symmetric triangular matrix
512 double calculateSymmetricTriangularDeterminant(int** matrix, int N) {
513     if (N == 1) {
514         return matrix[0][0];
515     }
516     if (N == 2) {
517         return matrix[0][0] * matrix[1][1];
518     }
519     double result = 0;
520     for (int i = 0; i < N; i++) {
521         if (matrix[0][i] != 0) {
522             result += matrix[0][i] * calculateSymmetricTriangularDeterminant(minorMatrixSymmetric(matrix, 0, i), N - 1);
523         }
524     }
525     return result;
526 }
527
528 // Function to calculate the minor matrix of a given row and column for a symmetric triangular matrix
529 int** minorMatrixSymmetricTriangular(int** matrix, int row, int col, int N) {
530     int** minorMatrix = new int*[N - 1];
531     for (int i = 0; i < N - 1; i++) {
532         minorMatrix[i] = new int[N - 1];
533         for (int j = 0; j < N - 1; j++) {
534             if (i == row || j == col) {
535                 continue;
536             }
537             minorMatrix[i][j] = matrix[i + 1][j + 1];
538         }
539     }
540     return minorMatrix;
541 }
542
543 // Function to calculate the cofactor matrix for a symmetric triangular matrix
544 int** cofactorMatrixSymmetricTriangular(int** matrix, int N) {
545     int** cofactorMatrix = new int*[N];
546     for (int i = 0; i < N; i++) {
547         cofactorMatrix[i] = new int[N];
548         for (int j = 0; j < N; j++) {
549             if ((i + j) % 2 == 1) {
550                 cofactorMatrix[i][j] = -matrix[i][j];
551             } else {
552                 cofactorMatrix[i][j] = matrix[i][j];
553             }
554         }
555     }
556     return cofactorMatrix;
557 }
558
559 // Function to calculate the adjoint matrix for a symmetric triangular matrix
560 int** adjointMatrixSymmetricTriangular(int** matrix, int N) {
561     int** adjointMatrix = new int*[N];
562     for (int i = 0; i < N; i++) {
563         adjointMatrix[i] = new int[N];
564         for (int j = 0; j < N; j++) {
565             adjointMatrix[i][j] = cofactorMatrixSymmetricTriangular(matrix, N)[i][j];
566         }
567     }
568     return adjointMatrix;
569 }
570
571 // Function to calculate the inverse matrix for a symmetric triangular matrix
572 int** inverseMatrixSymmetricTriangular(int** matrix, int N) {
573     double determinant = calculateSymmetricTriangularDeterminant(matrix, N);
574     if (determinant == 0) {
575         cout << "The matrix is singular, no inverse exists." << endl;
576         return NULL;
577     }
578     int** adjoint = adjointMatrixSymmetricTriangular(matrix, N);
579     int** inverse = new int*[N];
580     for (int i = 0; i < N; i++) {
581         inverse[i] = new int[N];
582         for (int j = 0; j < N; j++) {
583             inverse[i][j] = adjoint[i][j] / determinant;
584         }
585     }
586     return inverse;
587 }
588
589 // Function to calculate the trace of a symmetric triangular matrix
590 double traceMatrixSymmetricTriangular(int** matrix, int N) {
591     double trace = 0;
592     for (int i = 0; i < N; i++) {
593         if (matrix[0][i] != 0) {
594             trace += matrix[0][i];
595         }
596     }
597     return trace;
598 }
599
600 // Function to calculate the determinant of a triangular matrix
601 double calculateTriangularDeterminant(int** matrix, int N) {
602     if (N == 1) {
603         return matrix[0][0];
604     }
605     if (N == 2) {
606         return matrix[0][0] * matrix[1][1];
607     }
608     double result = 0;
609     for (int i = 0; i < N; i++) {
610         if (matrix[0][i] != 0) {
611             result += matrix[0][i] * calculateTriangularDeterminant(minorMatrixTriangular(matrix, 0, i), N - 1);
612         }
613     }
614     return result;
615 }
616
617 // Function to calculate the minor matrix of a given row and column for a triangular matrix
618 int** minorMatrixTriangular(int** matrix, int row, int col, int N) {
619     int** minorMatrix = new int*[N - 1];
620     for (int i = 0; i < N - 1; i++) {
621         minorMatrix[i] = new int[N - 1];
622         for (int j = 0; j < N - 1; j++) {
623             if (i == row || j == col) {
624                 continue;
625             }
626             minorMatrix[i][j] = matrix[i + 1][j + 1];
627         }
628     }
629     return minorMatrix;
630 }
631
632 // Function to calculate the cofactor matrix for a triangular matrix
633 int** cofactorMatrixTriangular(int** matrix, int N) {
634     int** cofactorMatrix = new int*[N];
635     for (int i = 0; i < N; i++) {
636         cofactorMatrix[i] = new int[N];
637         for (int j = 0; j < N; j++) {
638             if ((i + j) % 2 == 1) {
639                 cofactorMatrix[i][j] = -matrix[i][j];
640             } else {
641                 cofactorMatrix[i][j] = matrix[i][j];
642             }
643         }
644     }
645     return cofactorMatrix;
646 }
647
648 // Function to calculate the adjoint matrix for a triangular matrix
649 int** adjointMatrixTriangular(int** matrix, int N) {
650     int** adjointMatrix = new int*[N];
651     for (int i = 0; i < N; i++) {
652         adjointMatrix[i] = new int[N];
653         for (int j = 0; j < N; j++) {
654             adjointMatrix[i][j] = cofactorMatrixTriangular(matrix, N)[i][j];
655         }
656     }
657     return adjointMatrix;
658 }
659
660 // Function to calculate the inverse matrix for a triangular matrix
661 int** inverseMatrixTriangular(int** matrix, int N) {
662     double determinant = calculateTriangularDeterminant(matrix, N);
663     if (determinant == 0) {
664         cout << "The matrix is singular, no inverse exists." << endl;
665         return NULL;
666     }
667     int** adjoint = adjointMatrixTriangular(matrix, N);
668     int** inverse = new int*[N];
669     for (int i = 0; i < N; i++) {
670         inverse[i] = new int[N];
671         for (int j = 0; j < N; j++) {
672             inverse[i][j] = adjoint[i][j] / determinant;
673         }
674     }
675     return inverse;
676 }
677
678 // Function to calculate the trace of a triangular matrix
679 double traceMatrixTriangular(int** matrix, int N) {
680     double trace = 0;
681     for (int i = 0; i < N; i++) {
682         if (matrix[0][i] != 0) {
683             trace += matrix[0][i];
684         }
685     }
686     return trace;
687 }
688
689 // Function to calculate the determinant of a sparse triangular matrix
690 double calculateSparseTriangularDeterminant(int** matrix, int N) {
691     if (N == 1) {
692         return matrix[0][0];
693     }
694     if (N == 2) {
695         return matrix[0][0] * matrix[1][1];
696     }
697     double result = 0;
698     for (int i = 0; i < N; i++) {
699         if (matrix[0][i] != 0) {
700             result += matrix[0][i] * calculateSparseTriangularDeterminant(minorMatrixSparseTriangular(matrix, 0, i), N - 1);
701         }
702     }
703     return result;
704 }
705
706 // Function to calculate the minor matrix of a given row and column for a sparse triangular matrix
707 int** minorMatrixSparseTriangular(int** matrix, int row, int col, int N) {
708     int** minorMatrix = new int*[N - 1];
709     for (int i = 0; i < N - 1; i++) {
710         minorMatrix[i] = new int[N - 1];
711         for (int j = 0; j < N - 1; j++) {
712             if (i == row || j == col) {
713                 continue;
714             }
715             minorMatrix[i][j] = matrix[i + 1][j + 1];
716         }
717     }
718     return minorMatrix;
719 }
720
721 // Function to calculate the cofactor matrix for a sparse triangular matrix
722 int** cofactorMatrixSparseTriangular(int** matrix, int N) {
723     int** cofactorMatrix = new int*[N];
724     for (int i = 0; i < N; i++) {
725         cofactorMatrix[i] = new int[N];
726         for (int j = 0; j < N; j++) {
727             if ((i + j) % 2 == 1) {
728                 cofactorMatrix[i][j] = -matrix[i][j];
729             } else {
730                 cofactorMatrix[i][j] = matrix[i][j];
731             }
732         }
733     }
734     return cofactorMatrix;
735 }
736
737 // Function to calculate the adjoint matrix for a sparse triangular matrix
738 int** adjointMatrixSparseTriangular(int** matrix, int N) {
739     int** adjointMatrix = new int*[N];
740     for (int i = 0; i < N; i++) {
741         adjointMatrix[i] = new int[N];
742         for (int j = 0; j < N; j++) {
743             adjointMatrix[i][j] = cofactorMatrixSparseTriangular(matrix, N)[i][j];
744         }
745     }
746     return adjointMatrix;
747 }
748
749 // Function to calculate the inverse matrix for a sparse triangular matrix
750 int** inverseMatrixSparseTriangular(int** matrix, int N) {
751     double determinant = calculateSparseTriangularDeterminant(matrix, N);
752     if (determinant == 0) {
753         cout << "The matrix is singular, no inverse exists." << endl;
754         return NULL;
755     }
756     int** adjoint = adjointMatrixSparseTriangular(matrix, N);
757     int** inverse = new int*[N];
758     for (int i = 0; i < N; i++) {
759         inverse[i] = new int[N];
760         for (int j = 0; j < N; j++) {
761             inverse[i][j] = adjoint[i][j] / determinant;
762         }
763     }
764     return inverse;
765 }
766
767 // Function to calculate the trace of a sparse triangular matrix
768 double traceMatrixSparseTriangular(int** matrix, int N) {
769     double trace = 0;
770     for (int i = 0; i < N; i++) {
771         if (matrix[0][i] != 0) {
772             trace += matrix[0][i];
773         }
774     }
775     return trace;
776 }
777
778 // Function to calculate the determinant of a symmetric sparse triangular matrix
779 double calculateSymmetricSparseTriangularDeterminant(int** matrix, int N) {
780     if (N == 1) {
781         return matrix[0][0];
782     }
783     if (N == 2) {
784         return matrix[0][0] * matrix[1][1];
785     }
786     double result = 0;
787     for (int i = 0; i < N; i++) {
788         if (matrix[0][i] != 0) {
789             result += matrix[0][i] * calculateSymmetricSparseTriangularDeterminant(minorMatrixSymmetricTriangular(matrix, 0, i), N - 1);
790         }
791     }
792     return result;
793 }
794
795 // Function to calculate the minor matrix of a given row and column for a symmetric sparse triangular matrix
796 int** minorMatrixSymmetricTriangular(int** matrix, int row, int col, int N) {
797     int** minorMatrix = new int*[N - 1];
798     for (int i = 0; i < N - 1; i++) {
799         minorMatrix[i] = new int[N - 1];
800         for (int j = 0; j < N - 1; j++) {
801             if (i == row || j == col) {
802                 continue;
803             }
804             minorMatrix[i][j] = matrix[i + 1][j + 1];
805         }
806     }
807     return minorMatrix;
808 }
809
810 // Function to calculate the cofactor matrix for a symmetric sparse triangular matrix
811 int** cofactorMatrixSymmetricTriangular(int** matrix, int N) {
812     int** cofactorMatrix = new int*[N];
813     for (int i = 0; i < N; i++) {
814         cofactorMatrix[i] = new int[N];
815         for (int j = 0; j < N; j++) {
816             if ((i + j) % 2 == 1) {
817                 cofactorMatrix[i][j] = -matrix[i][j];
818             } else {
819                 cofactorMatrix[i][j] = matrix[i][j];
820             }
821         }
822     }
823     return cofactorMatrix;
824 }
825
826 // Function to calculate the adjoint matrix for a symmetric sparse triangular matrix
827 int** adjointMatrixSymmetricTriangular(int** matrix, int N) {
828     int** adjointMatrix = new int*[N];
829     for (int i = 0; i < N; i++) {
830         adjointMatrix[i] = new int[N];
831         for (int j = 0; j < N; j++) {
832             adjointMatrix[i][j] = cofactorMatrixSymmetricTriangular(matrix, N)[i][j];
833         }
834     }
835     return adjointMatrix;
836 }
837
838 // Function to calculate the inverse matrix for a symmetric sparse triangular matrix
839 int** inverseMatrixSymmetricTriangular(int** matrix, int N) {
840     double determinant = calculateSymmetricSparseTriangularDeterminant(matrix, N);
841     if (determinant == 0) {
842         cout << "The matrix is singular, no inverse exists." << endl;
843         return NULL;
844     }
845     int** adjoint = adjointMatrixSymmetricTriangular(matrix, N);
846     int** inverse = new int*[N];
847     for (int i = 0; i < N; i++) {
848         inverse[i] = new int[N];
849         for (int j = 0; j < N; j++) {
850             inverse[i][j] = adjoint[i][j] / determinant;
851         }
852     }
853     return inverse;
854 }
855
856 // Function to calculate the trace of a symmetric sparse triangular matrix
857 double traceMatrixSymmetricTriangular(int** matrix, int N) {
858     double trace = 0;
859     for (int i = 0; i < N; i++) {
860         if (matrix[0][i] != 0) {
861             trace += matrix[0][i];
862         }
863     }
864     return trace;
865 }
866
867 // Function to calculate the determinant of a triangular sparse matrix
868 double calculateTriangularSparseDeterminant(int** matrix, int N) {
869     if (N == 1) {
870         return matrix[0][0];
871     }
872     if (N == 2) {
873         return matrix[0][0] * matrix[1][1];
874     }
875     double result = 0;
876     for (int i = 0; i < N; i++) {
877         if (matrix[0][i] != 0) {
878             result += matrix[0][i] * calculateTriangularSparseDeterminant(minorMatrixSparseTriangular(matrix, 0, i), N - 1);
879         }
880     }
881     return result;
882 }
883
884 // Function to calculate the minor matrix of a given row and column for a triangular sparse matrix
885 int** minorMatrixSparseTriangular(int** matrix, int row, int col, int N) {
886     int** minorMatrix = new int*[N - 1];
887     for (int i = 0; i < N - 1; i++) {
888         minorMatrix[i] = new int[N - 1];
889         for (int j = 0; j < N - 1; j++) {
890             if (i == row || j == col) {
891                 continue;
892             }
893             minorMatrix[i][j] = matrix[i + 1][j + 1];
894         }
895     }
896     return minorMatrix;
897 }
898
899 // Function to calculate the cofactor matrix for a triangular sparse matrix
900 int** cofactorMatrixSparseTriangular(int** matrix, int N) {
901     int** cofactorMatrix = new int*[N];
902     for (int i = 0; i < N; i++) {
903         cofactorMatrix[i] = new int[N];
904         for (int j = 0; j < N; j++) {
905             if ((i + j) % 2 == 1) {
906                 cofactorMatrix[i][j] = -matrix[i][j];
907             } else {
908                 cofactorMatrix[i][j] = matrix[i][j];
909             }
910         }
911     }
912     return cofactorMatrix;
913 }
914
915 // Function to calculate the adjoint matrix for a triangular sparse matrix
916 int** adjointMatrixSparseTriangular(int** matrix, int N) {
917     int** adjointMatrix = new int*[N];
918     for (int i = 0; i < N; i++) {
919         adjointMatrix[i] = new int[N];
920         for (int j = 0; j < N; j++) {
921             adjointMatrix[i][j] = cofactorMatrixSparseTriangular(matrix, N)[i][j];
922         }
923     }
924     return adjointMatrix;
925 }
926
927 // Function to calculate the inverse matrix for a triangular sparse matrix
928 int** inverseMatrixSparseTriangular(int** matrix, int N) {
929     double determinant = calculateTriangularSparseDeterminant(matrix, N);
930     if (determinant == 0) {
931         cout << "The matrix is singular, no inverse exists." << endl;
932         return NULL;
933     }
934     int** adjoint = adjointMatrixSparseTriangular(matrix, N);
935     int** inverse = new int*[N];
936     for (int i = 0; i < N; i++) {
937         inverse[i] = new int[N];
938         for (int j = 0; j < N; j++) {
939             inverse[i][j] = adjoint[i][j] / determinant;
940         }
941     }
942     return inverse;
943 }
944
945 // Function to calculate the trace of a triangular sparse matrix
946 double traceMatrixSparseTriangular(int** matrix, int N) {
947     double trace = 0;
948     for (int i = 0; i < N; i++) {
949         if (matrix[0][i] != 0) {
950             trace += matrix[0][i];
951         }
952     }
953     return trace;
954 }
955
956 // Function to calculate the determinant of a symmetric triangular sparse matrix
957 double calculateSymmetricTriangularSparseDeterminant(int** matrix, int N) {
958     if (N == 1) {
959         return matrix[0][0];
960     }
961     if (N == 2) {
962         return matrix[0][0] * matrix[1][1];
963     }
964     double result = 0;
965     for (int i = 0; i < N; i++) {
966         if (matrix[0][i] != 0) {
967             result += matrix[0][i] * calculateSymmetricTriangularSparseDeterminant(minorMatrixSymmetricTriangular(matrix, 0, i), N - 1);
968         }
969     }
970     return result;
971 }
972
973 // Function to calculate the minor matrix of a given row and column for a symmetric triangular sparse matrix
974 int** minorMatrixSymmetricTriangular(int** matrix, int row, int col, int N) {
975     int** minorMatrix = new int*[N - 1];
976     for (int i = 0; i < N - 1; i++) {
977         minorMatrix[i] = new int[N - 1];
978         for (int j = 0; j < N - 1; j++) {
979             if (i == row || j == col) {
980                 continue;
981             }
982             minorMatrix[i][j] = matrix[i + 1][j + 1];
983         }
984     }
985     return minorMatrix;
986 }
987
988 // Function to calculate the cofactor matrix for a symmetric triangular sparse matrix
989 int** cofactorMatrixSymmetricTriangular(int** matrix, int N) {
990     int** cofactorMatrix = new int*[N];
991     for (int i = 0; i < N; i++) {
992         cofactorMatrix[i] = new int[N];
993         for (int j = 0; j < N; j++) {
994             if ((i + j) % 2 == 1) {
995                 cofactorMatrix[i][j] = -matrix[i][j];
996             } else {
997                 cofactorMatrix[i][j] = matrix[i][j];
998             }
999         }
1000    }
1001   return cofactorMatrix;
1002 }
1003
1004 // Function to calculate the adjoint matrix for a symmetric triangular sparse matrix
1005 int** adjointMatrixSymmetricTriangular(int** matrix
```

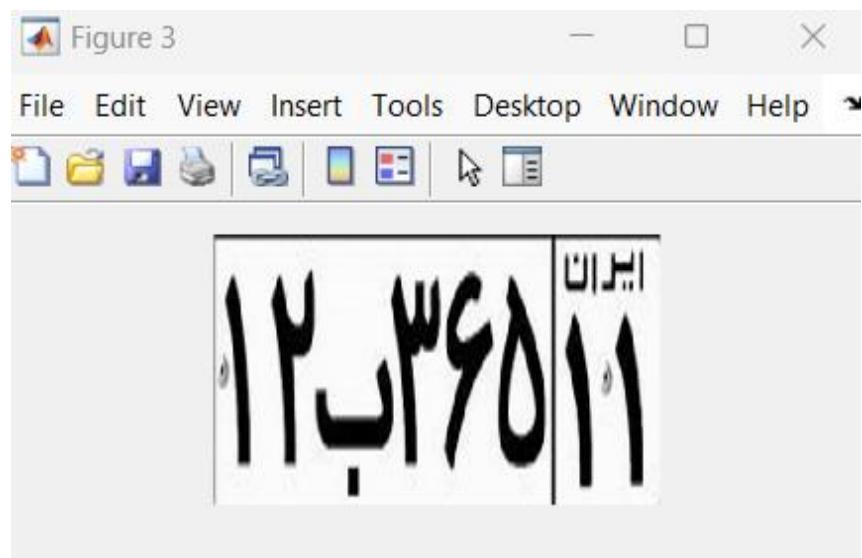
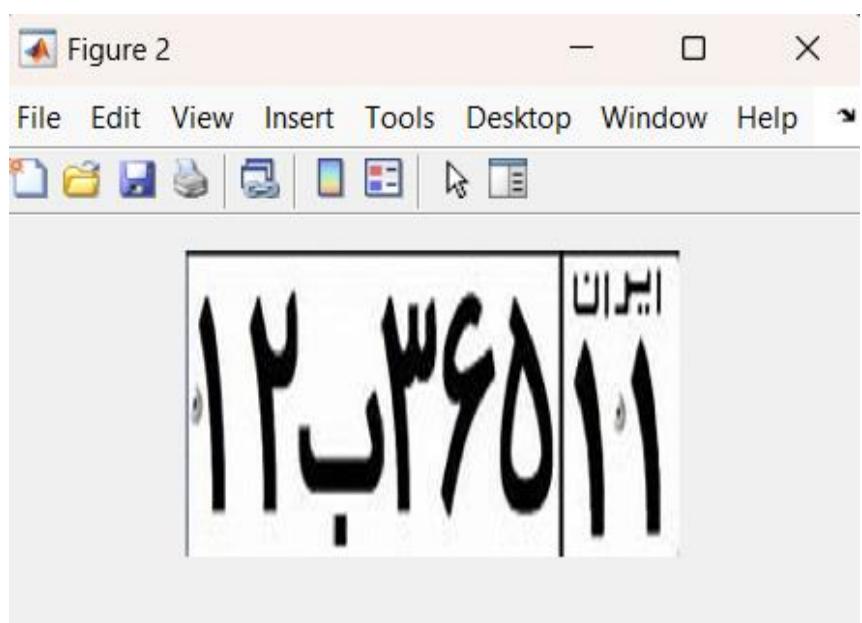
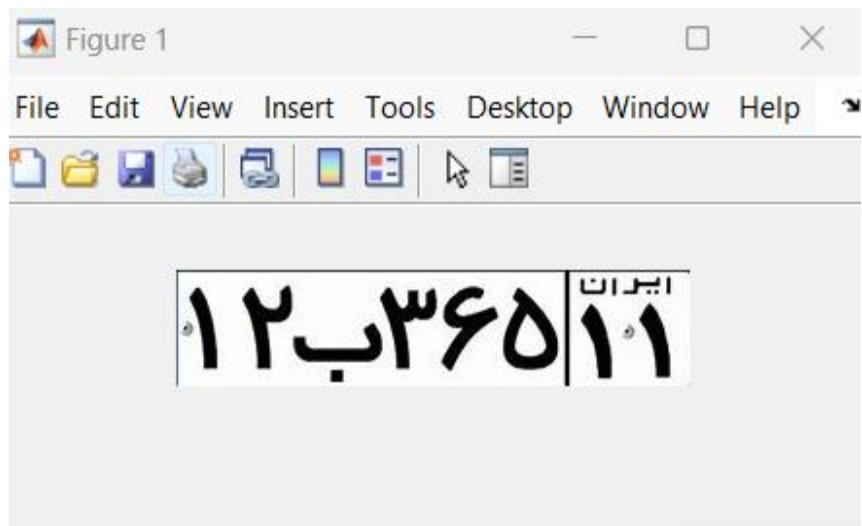
<pre> 173 % STEP 1: LOAD REFERENCE IMAGES FROM THE Map_Set DIRECTORY 174 % ----- 175 % Map_Set_Dir = 'Map_Set'; % Directory containing reference images (letters/numbers) 176 177 % Get a list of all .png and .bmp files in the directory 178 file_list_png = dir(fullfile(Map_Set_Dir, '*.png')); % Get .png files 179 file_list_bmp = dir(fullfile(Map_Set_Dir, '*.bmp')); % Get .bmp files 180 181 % Combine the file lists 182 file_list = [file_list_png, file_list_bmp]; 183 184 % Initialize the TRAIN cell array: Row 1 = Images, Row 2 = Labels 185 TRAIN = cell(2, numel(file_list)); 186 187 for k = 1:size(file_list) 188 189 % Load image and preprocess 190 img = imread(fullfile(Map_Set_Dir, file_list(k).name)); 191 if size(img, 3) == 3 192 img = im2bw(img); % Convert RGB to binary 193 end 194 img = imresize(img, [42, 24]); % Resize to match character size 195 196 % Store in TRAIN 197 TRAIN{1, k} = img; % Image 198 [~, label, ~] = fileparts(file_list(k).name); % Extract label (filename without extension) 199 TRAIN{2, k} = label; 200 201 end 202 203 % Display the number of images loaded 204 fprintf('Loaded %d images from the Map_Set directory.\n', numel(file_list)); 205 206 % ----- 207 % STEP 2: LICENSE PLATE SEGMENTATION AND RECOGNITION 208 % ----- 209 % (L, Nc) = wsegmentation(mej_lease); 210 % propred = regionprops(L, 'BoundingBox', 'Area'); 211 212 % Sort Bounding Boxes by x-coordinate (left-to-right) 213 % BoundingBoxes = cell2mat(propred.BoundingBox); 214 % [~, sortOrder] = sort(BoundingBoxes(:, 1)); 215 % sorted_propred = propred(sortOrder); 216 217 % Filter out unwanted regions (e.g., outer border) 218 valid_indices = {}; 219 for n = 1:mej_lease 220 boxes = sorted_propred(n).BoundingBox; 221 width = boxes(1); 222 height = boxes(4); 223 area = width * height; 224 225 % Criteria (adjust these thresholds) 226 max_area_threshold = 20000; 227 min_area_threshold = 1000; 228 aspect_ratio_range = [0.1, 4]; % [min_aspect_ratio, max_aspect_ratio] 229 230 % Calculate aspect ratio 231 aspect_ratio = width / height; 232 233 % Check if the region meets all criteria 234 if area > min_area_threshold && area < max_area_threshold && ... 235 aspect_ratio >= aspect_ratio_range(1) && aspect_ratio <= aspect_ratio_range(2) 236 valid_indices = [valid_indices, n]; % Add index to valid regions 237 end 238 end </pre>	<pre> 239 filtered_propred = sorted_propred(valid_indices); 240 filtered_Nc = numel(valid_indices); 241 242 % Draw bounding boxes 243 figure; 244 imshow(new_cleaned); 245 hold on; 246 for n = 1:filtered_Nc 247 rectangle('Position', filtered_propred(n).BoundingBox, 'EdgeColor', [0.75, 0, 0.75], 'LineWidth', 2); 248 end 249 hold off; 250 251 % Recognize characters 252 final_output = {}; 253 for n = 1:filtered_Nc 254 region_idx = valid_indices(n); 255 [r, c] = find(l == sortOrder(region_idx)); 256 257 if isempty(r) isempty(c) 258 continue; 259 end 260 261 Y = imresize(Y, [42, 24]); 262 263 ro = zeros(3, size(TRAIN, 2)); 264 for k = 1:size(TRAIN, 2) 265 ro(k) = core2(TRAIN{1, k}, Y); 266 end 267 268 [MAXRO, pos] = max(ro); 269 if MAXRO > 0.45 270 out = cell2mat([TRAIN{1, pos}]); 271 final_output = [final_output out]; 272 end 273 274 end 275 276 % Display recognized output 277 disp('Recognized Output:'); 278 disp(final_output); 279 280 % Save final_output to a text file 281 fileID = fopen('recognized_output.txt', 'w'); % Open file for writing 282 fprintf(fileID, '%s', final_output); % Write the recognized characters 283 fclose(fileID); % Close the file 284 disp('Recognized output saved to recognized_output.txt'); 285 286 </pre>
---	--

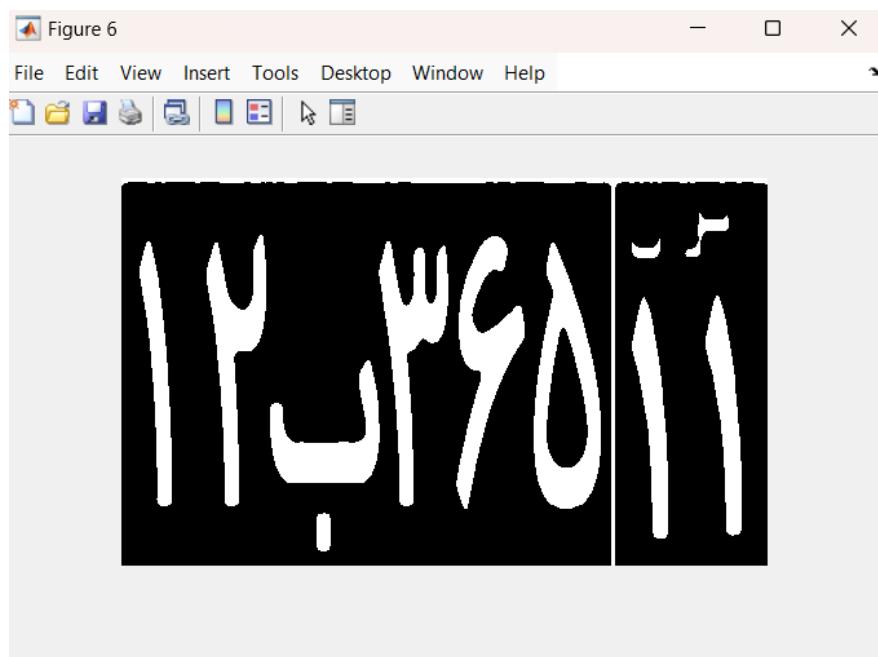
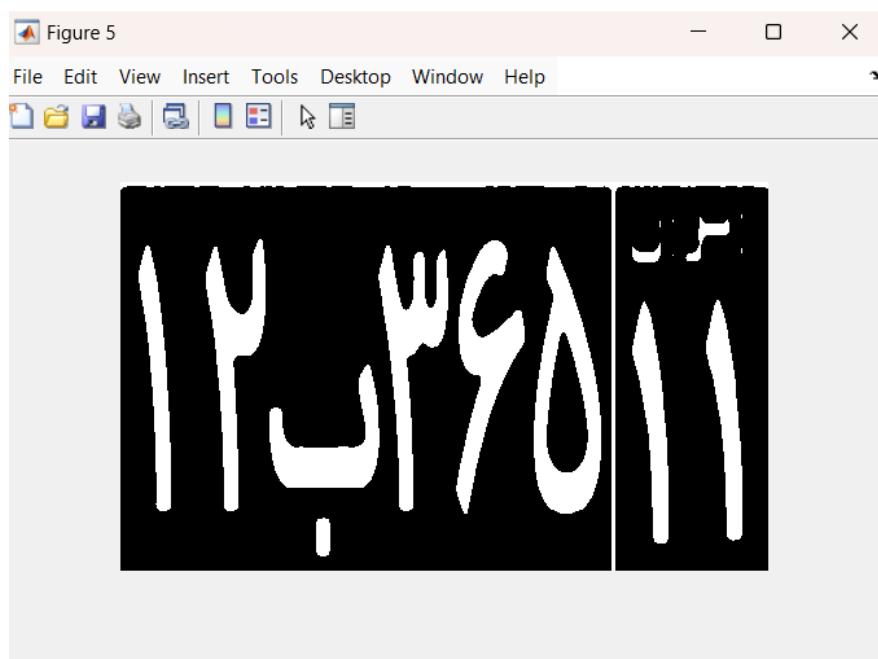
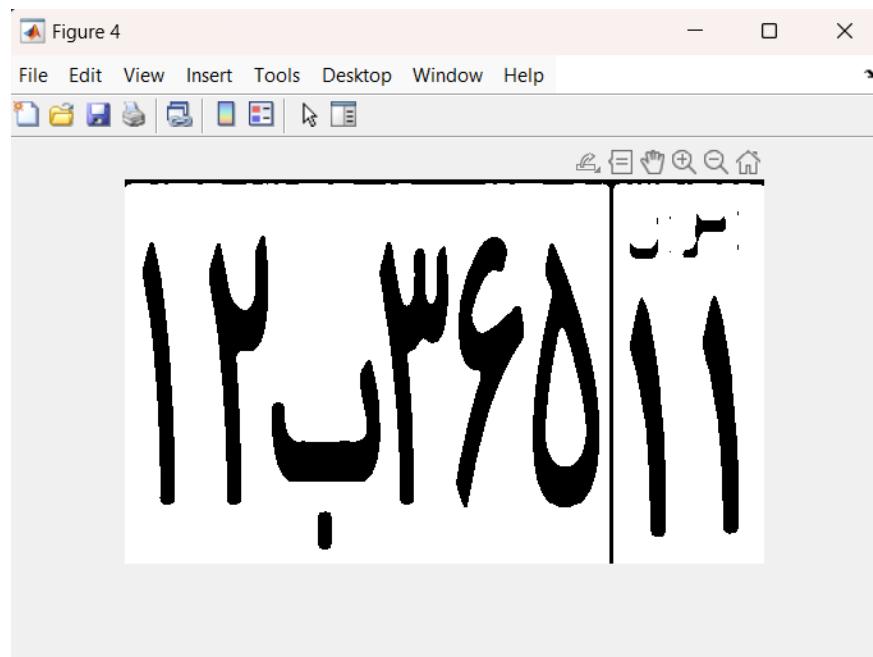
PART 2

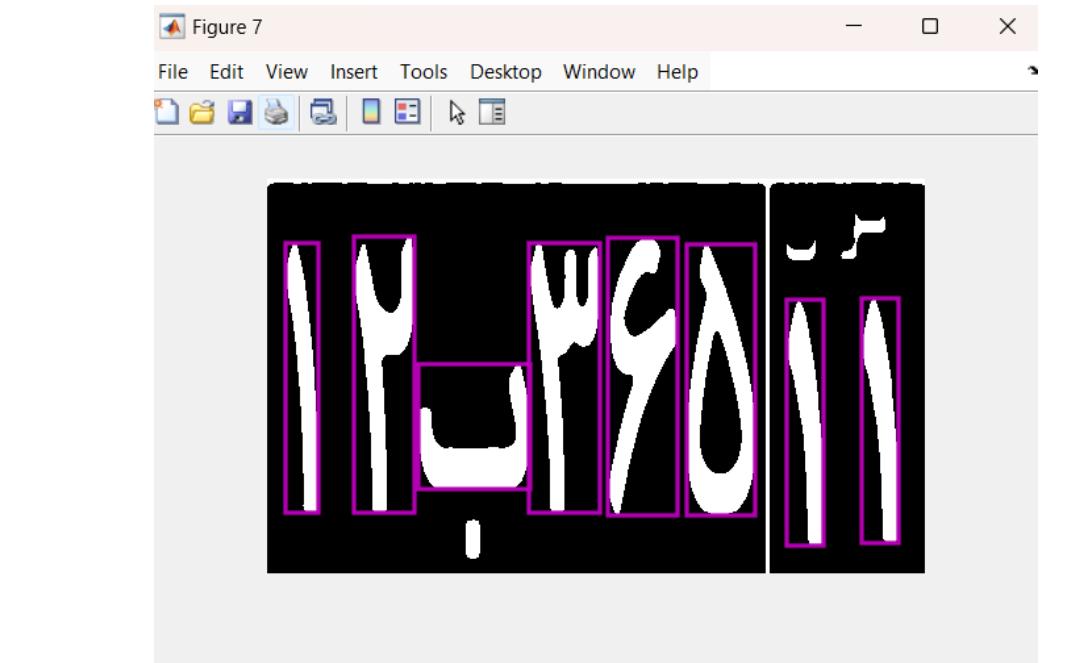
In this part I just implement the first part code, but specialized for Persian map sets and car plates. And I used my own functions not the built in ones.

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 '؟' and B to be 'ـ' and etc, I hope you note that.**





A screenshot of a MATLAB editor window titled "Editor - C:\Users\kian\Desktop\p1\recognized_output2.txt". The window shows a single line of text: "1 12B36511". The line number "1" is in blue, and the license plate number "12B36511" is in green. The background of the editor is dark brown.

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);

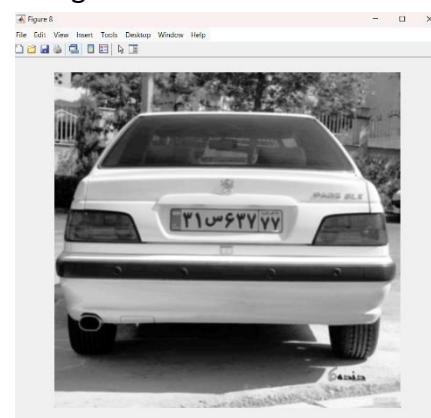
figure;

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);

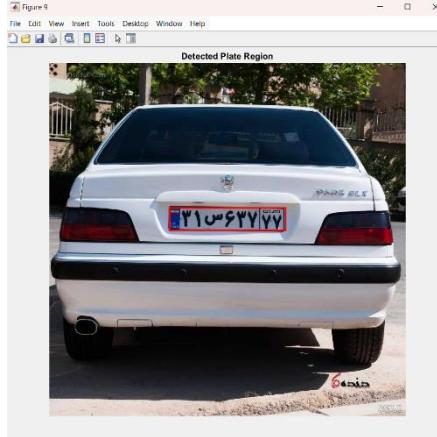
    % Find the peak correlation value and its location
    [ypeak, xpeak] = find(corr == max(corr(:)));
    current_corr = max(corr(:));

    % 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');
```

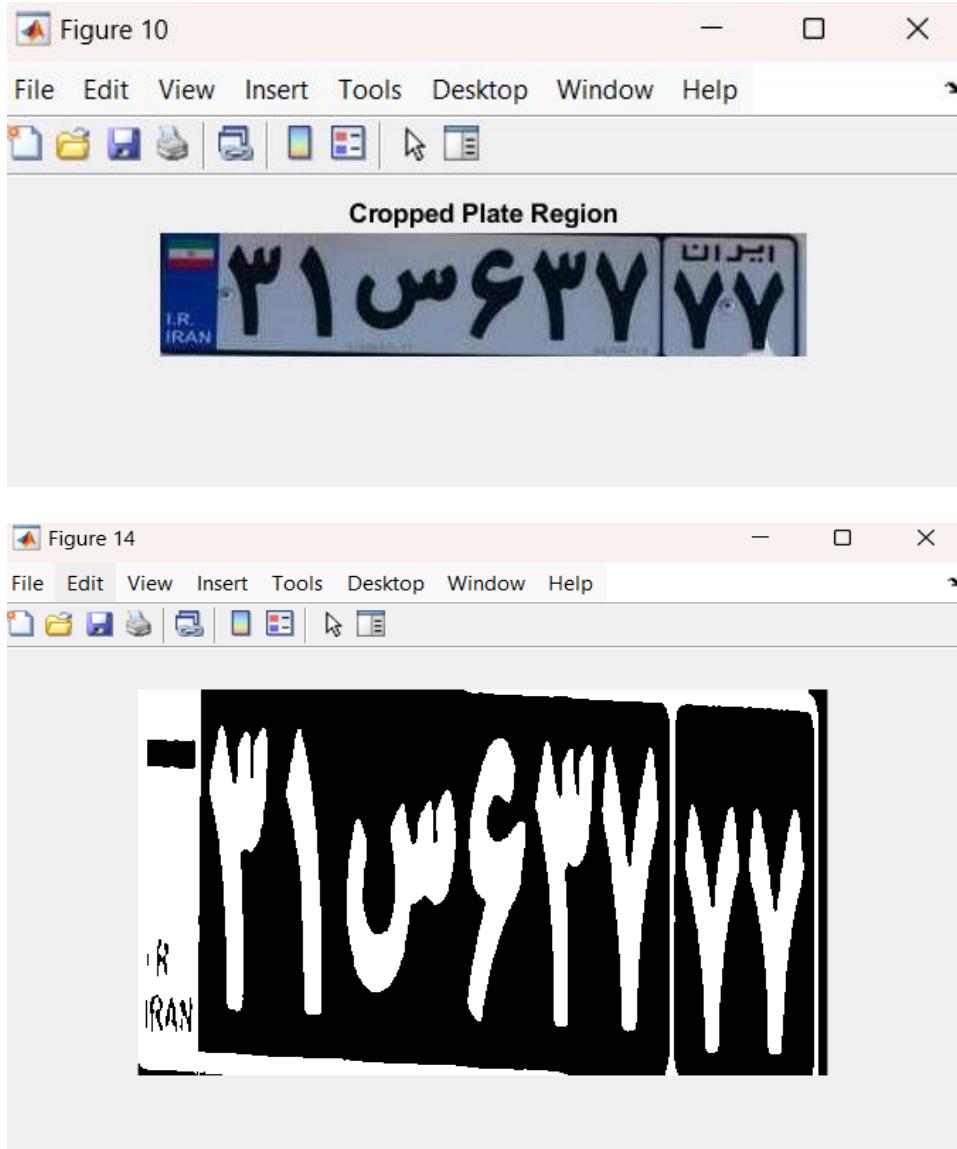


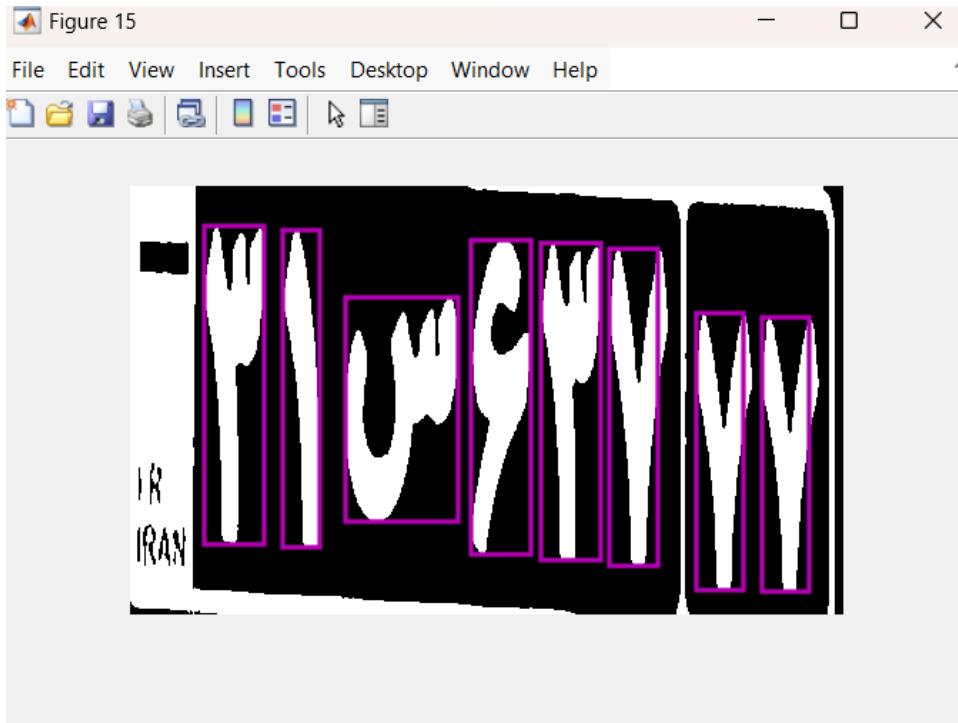
in the first part of this code, I calculate the plate's width and height, but it would be only that blue rectangular with Iran's flag on it, I want the whole plate so I use kian_constant and multiply it by that width already found, and now when I print that red rectangular on my car image and later crop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

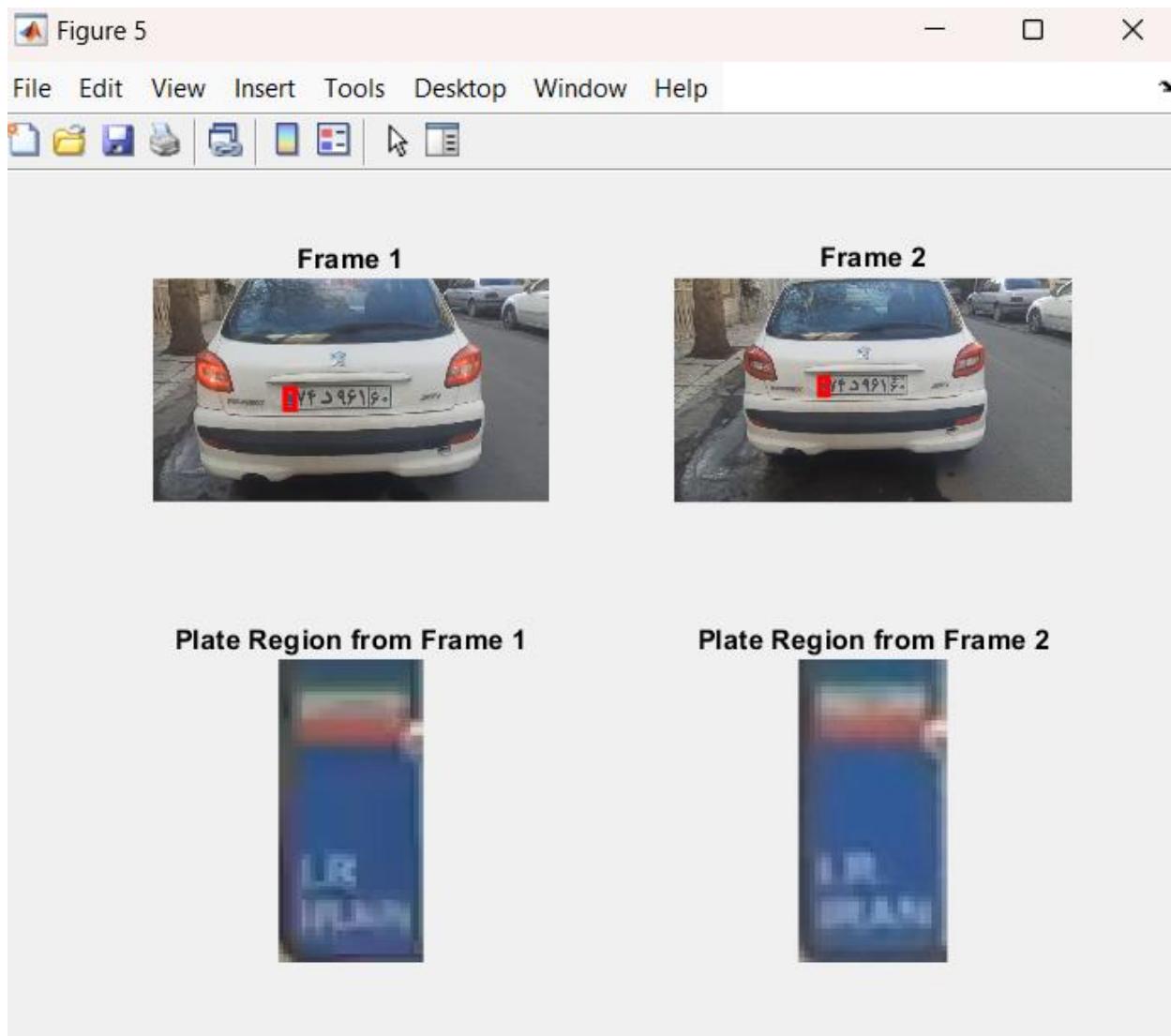
```

1 % Load the template
2 template_file = 'new.jpg';
3 plate_template = imread(template_file);
4 template_file = 'OIP.jpg';
5 plate_template = imread(template_file);
6 [rows1,col1] = size(plate_template);
7 [rows2,col2] = size(plate_template);
8 kian_constant = (col2/col1)*0.66;
9 disp(kian_constant)
10
11 [file, path] = uigetfile('*.*jpg; *.bmp; *.png', 'Choose the image');
12 picture = imread(fullfile(path, file));
13
14 % Convert images to grayscale (if they are RGB)
15 if size(picture, 3) == 3
16     gray_image = rgb2gray(picture);
17 else
18     gray_image = picture;
19 end
20 gray_image = histeq(gray_image);
21 gray_image = imgaussfilt(gray_image, 2); % Adjust sigma as needed
22 gray_image = medfilt2(gray_image);
23
24 figure;
25
26 if size(plate_template, 3) == 3
27     gray_template = rgb2gray(plate_template);
28 else
29     gray_template = plate_template;
30 end
31 imshow(gray_image);
32 % Define the range of scaling factors
33 scales = 0.5:0.05:2;
34 best_corr = -Inf; % Initialize best correlation score
35 best_scale = 1;
36 best_xpk = 0;
37 best_ypk = 0;
38 best_template = gray_template; % best matched (resized) template
39
40 % Loop over the scales and compute correlation at each scale
41 for s = scales
42     % Resize the template keeping the aspect ratio constant
43
44     resized_template = imresize(gray_template, s);
45     % Compute normalized cross-correlation
46     corr = normxcorr2(resized_template, gray_image);
47
48     % Find the peak correlation value and its location
49     [ypeak, xpeak] = find(corr == max(corr(:)));
50     current_corr = max(corr(:));
51
52     % Update the best match if current correlation is higher
53     if current_corr > best_corr
54         best_corr = current_corr;
55         best_scale = s;
56         best_xpk = xpeak;
57         best_ypk = ypeak;
58         best_template = resized_template;
59     end
60
61     % Note: normxcorr2 returns coordinates with an offset equal to the template size.
62     plate_x = best_xpk - size(best_template, 2) + 1;
63     plate_y = best_ypk - size(best_template, 1) + 1;
64     plate_width = size(best_template, 2);
65     plate_height = size(best_template, 1);
66
67     figure;
68     imshow(picture);
69     hold on;
70     rectangle('Position', [plate_x, plate_y, plate_width*kian_constant, plate_height], 'EdgeColor', 'r', 'LineWidth', 2);
71     title('Detected Plate Region');
72
73     % Crop the plate region from the original image
74     plate_region = imcrop(picture, [plate_x, plate_y, plate_width*kian_constant, plate_height]);
75
76     % Display the cropped plate region
77     figure;
78     imshow(plate_region);
79     title('Cropped Plate Region');
80
81 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% from now on it's the same as earlier parts
82 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

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)

```

```

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.

