Signals and Systems

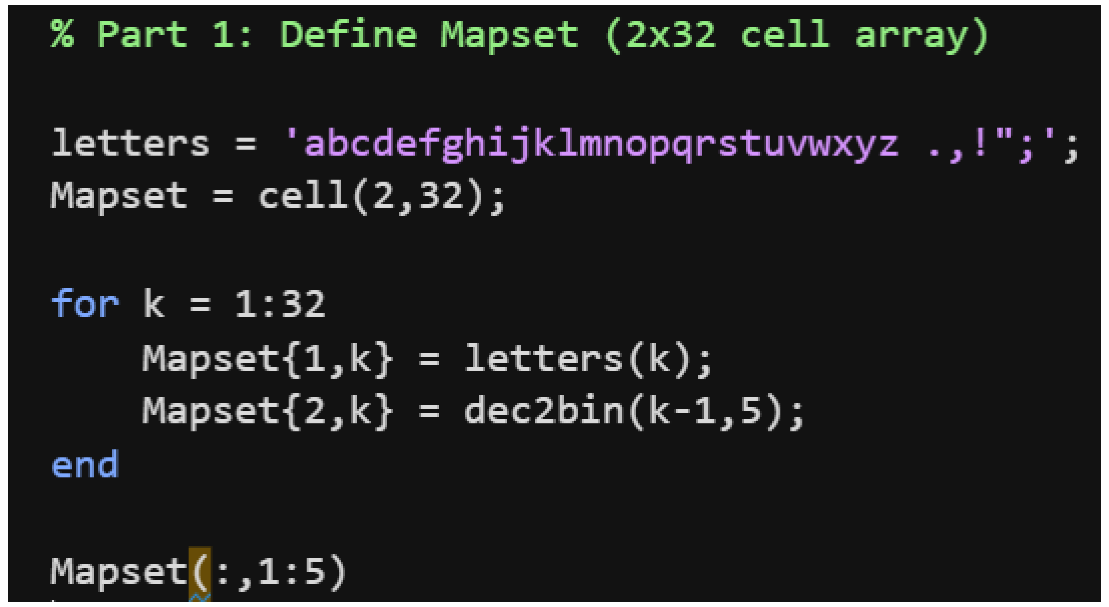
Project Report

Parsa KafshduziBukani 810102501

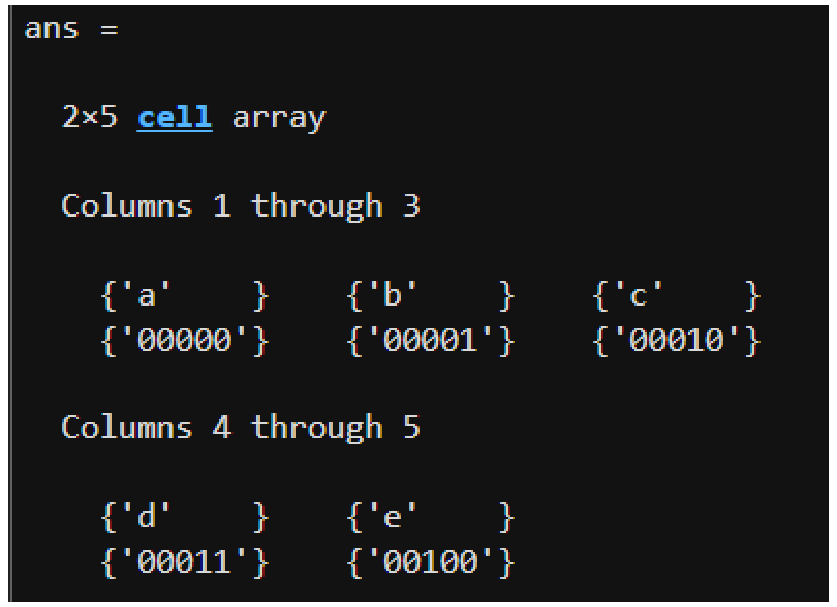
CA3

Section 1

Part 1:



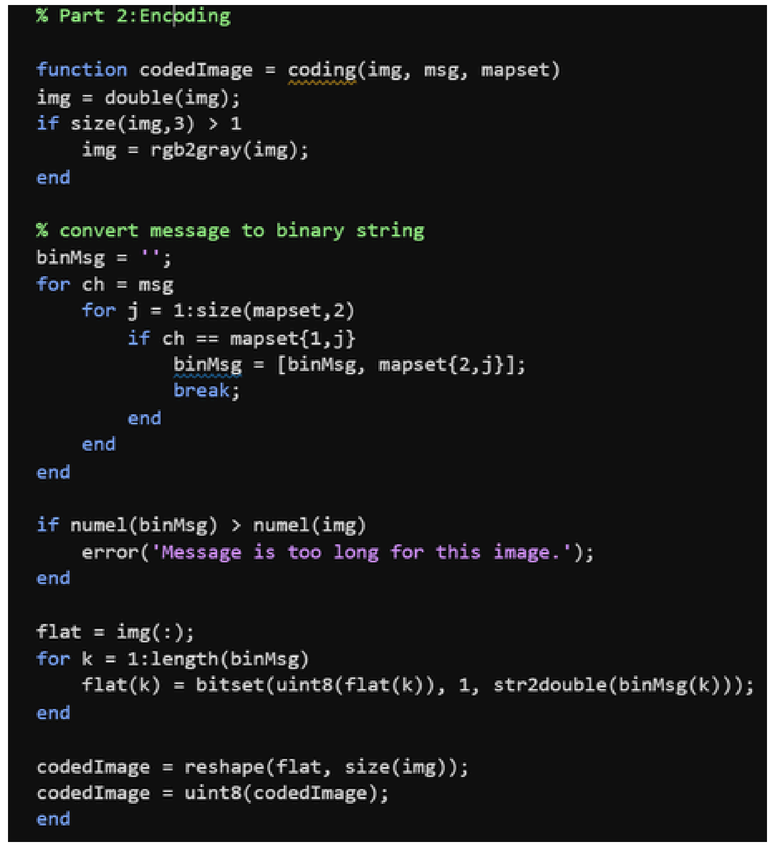
```
% Part 1: Define Mapset (2x32 cell array)

letters = 'abcdefghijklmnopqrstuvwxyz .,!";';
Mapset = cell(2,32);

for k = 1:32
    Mapset{1,k} = letters(k);
    Mapset{2,k} = dec2bin(k-1,5);
end

Mapset(:,1:5)
```



```
ans =

  2×5 cell array

  Columns 1 through 3

    {'a'    }    {'b'    }    {'c'    }
    {'00000'}    {'00001'}    {'00010'}

  Columns 4 through 5

    {'d'    }    {'e'    }
    {'00011'}    {'00100'}
```

Part 2:

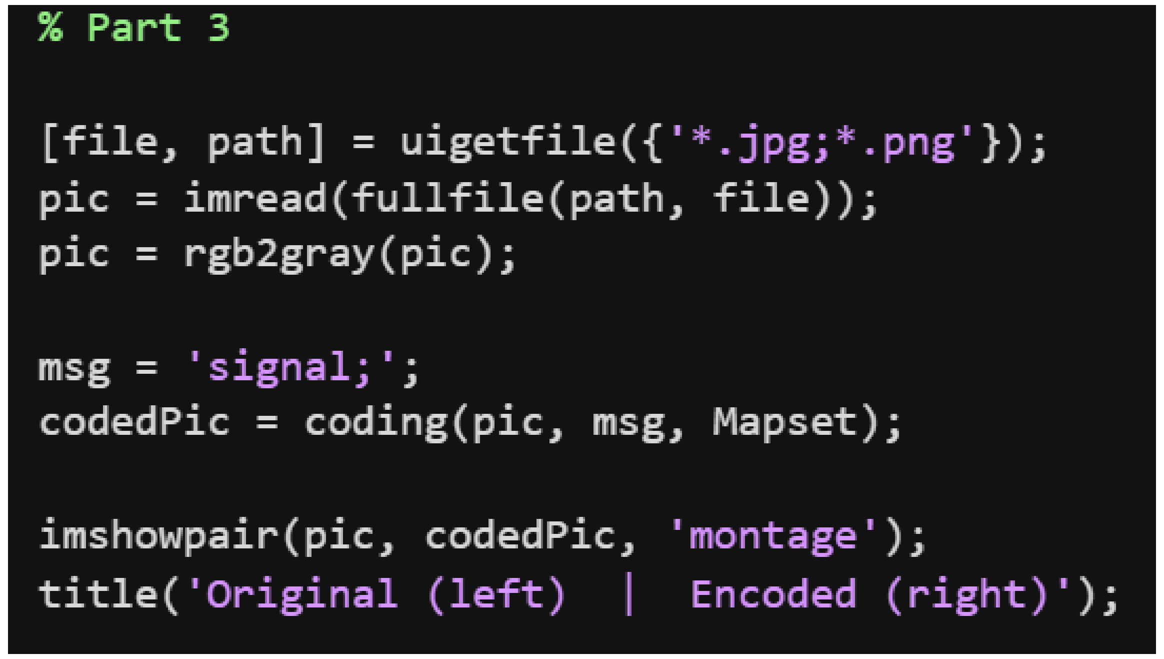
In this part, a text message is hidden inside a grayscale image using the Least Significant Bit (LSB) method. Each character of the message is first converted to its 5-bit binary code based on the Mapset table. Then, the image pixels are slightly modified so that the least significant bit of each pixel stores one bit of the binary message. This change is extremely small and does not noticeably alter the image, so the hidden message remains invisible. Before embedding, the program checks that the image has enough pixels to hold the message; otherwise, it displays an error. The result is a new image that looks identical to the original but securely contains the encoded text.



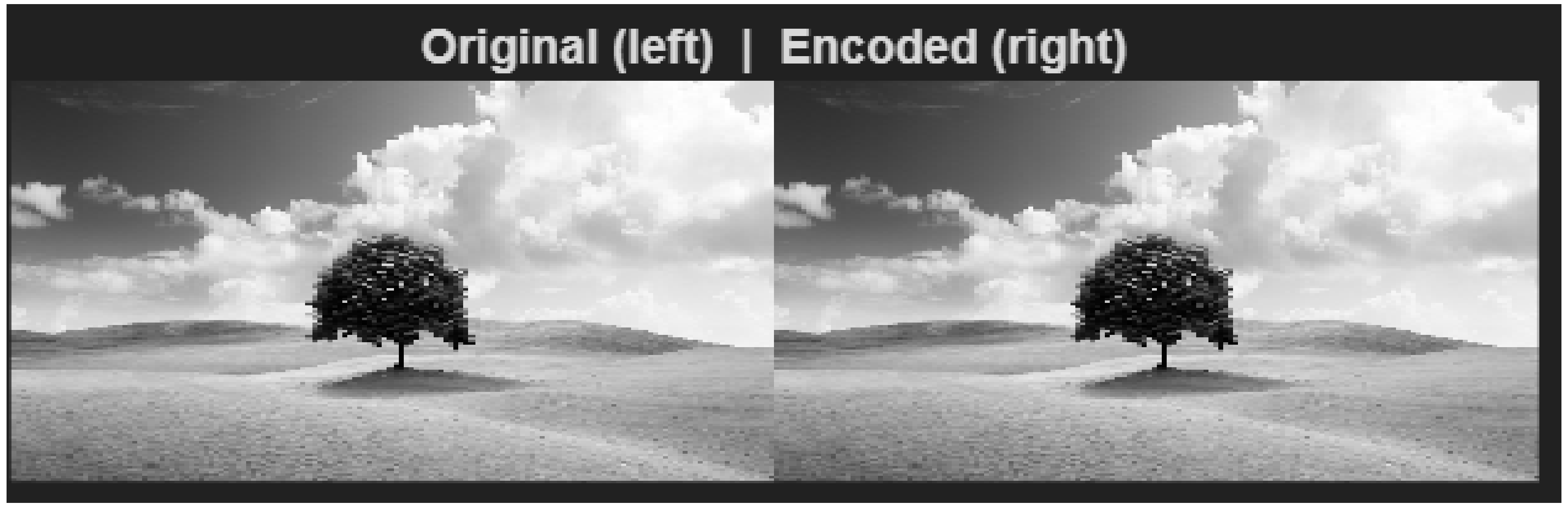
```
% Part 2:Encoding

function codedImage = coding(img, msg, mapset)
img = double(img);
if size(img,3) > 1
    img = rgb2gray(img);
end

% convert message to binary string
binMsg = '';
for ch = msg
    for j = 1:size(mapset,2)
        if ch == mapset{1,j}
            binMsg = [binMsg, mapset{2,j}];
            break;
        end
    end
end

if numel(binMsg) > numel(img)
    error('Message is too long for this image.');
end

flat = img(:);
for k = 1:length(binMsg)
    flat(k) = bitset(uint8(flat(k)), 1, str2double(binMsg(k)));
end

codedImage = reshape(flat, size(img));
codedImage = uint8(codedImage);
end
```

Part 3:

After encoding the message “**signal;**” into the grayscale image using the coding function, both the original and the coded images were displayed side-by-side with the subplot command.
By visual comparison, no visible difference can be observed between the two images.
This is because only the least significant bit of each pixel was modified to store the message bits.
Changes in these bits cause extremely small intensity variations—too small for the human eye to detect—so the image appears completely unchanged while still containing the hidden data.



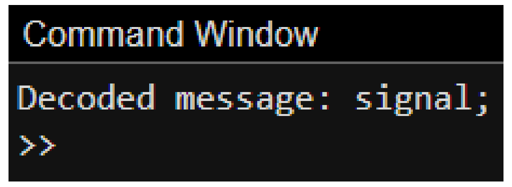
```
% Part 3

[file, path] = uigetfile({'*.jpg;*.png'});
pic = imread(fullfile(path, file));
pic = rgb2gray(pic);

msg = 'signal;';
codedPic = coding(pic, msg, Mapset);

imshowpair(pic, codedPic, 'montage');
title('Original (left)  |  Encoded (right)');
```



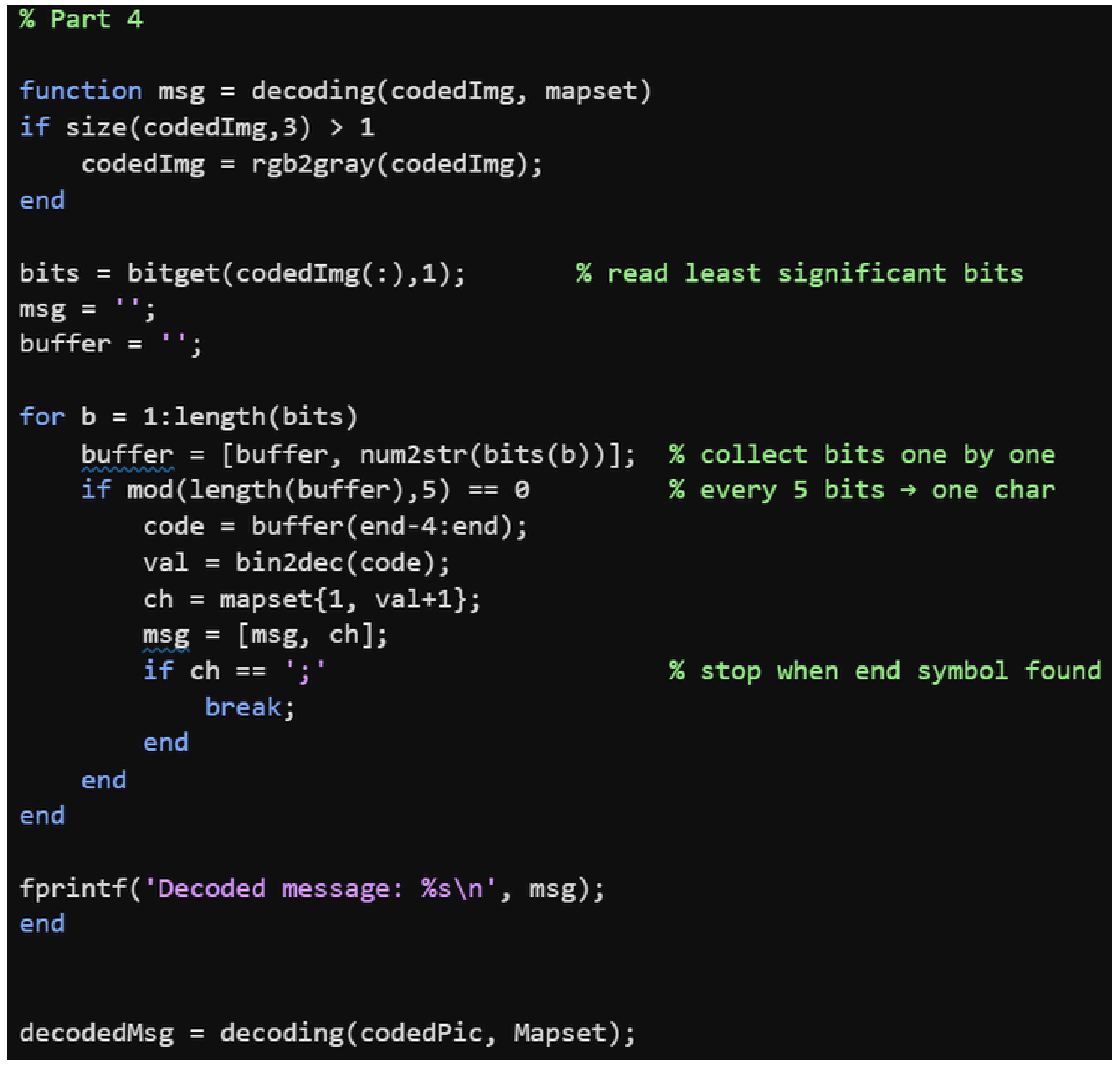
Part 4:

In this part, a function named decoding was written to extract the hidden message from the encoded grayscale image.
The program reads the least significant bits (LSB) of the pixels sequentially and groups them into 5-bit segments. Each 5-bit code is then matched with the corresponding character in the Mapset table to rebuild the original text. Decoding continues automatically until the **;** symbol is reached, which marks the end of the message.

When tested on the same image containing the hidden text “**signal;**”, the function successfully printed the exact original message, confirming that the encoding and decoding processes work correctly and without visible distortion.



```
Command Window
Decoded message: signal;
>>
```



```
% Part 4

function msg = decoding(codedImg, mapset)
if size(codedImg,3) > 1
    codedImg = rgb2gray(codedImg);
end

bits = bitget(codedImg(:),1);       % read least significant bits
msg = '';
buffer = '';

for b = 1:length(bits)
    buffer = [buffer, num2str(bits(b))];  % collect bits one by one
    if mod(length(buffer),5) == 0         % every 5 bits → one char
        code = buffer(end-4:end);
        val = bin2dec(code);
        ch = mapset{1, val+1};
        msg = [msg, ch];
        if ch == ';'                      % stop when end symbol found
            break;
        end
    end
end

fprintf('Decoded message: %s\n', msg);
end


decodedMsg = decoding(codedPic, Mapset);
```

Part 5: **If noise is unintentionally added to the image, can we still decode it?**

Whether the hidden message can still be decoded after noise is added depends on where the noise occurs. If the noise affects image areas that were not used for message embedding, the extracted message will remain correct.
However, if the noise changes pixels inside the selected blocks (where the LSBs contain the message bits), those bits may flip and corrupt the hidden data, making part or all of the message unreadable. The end **symbol ;** helps stop decoding properly, but if that symbol itself is damaged, message recovery fails.

Section 2

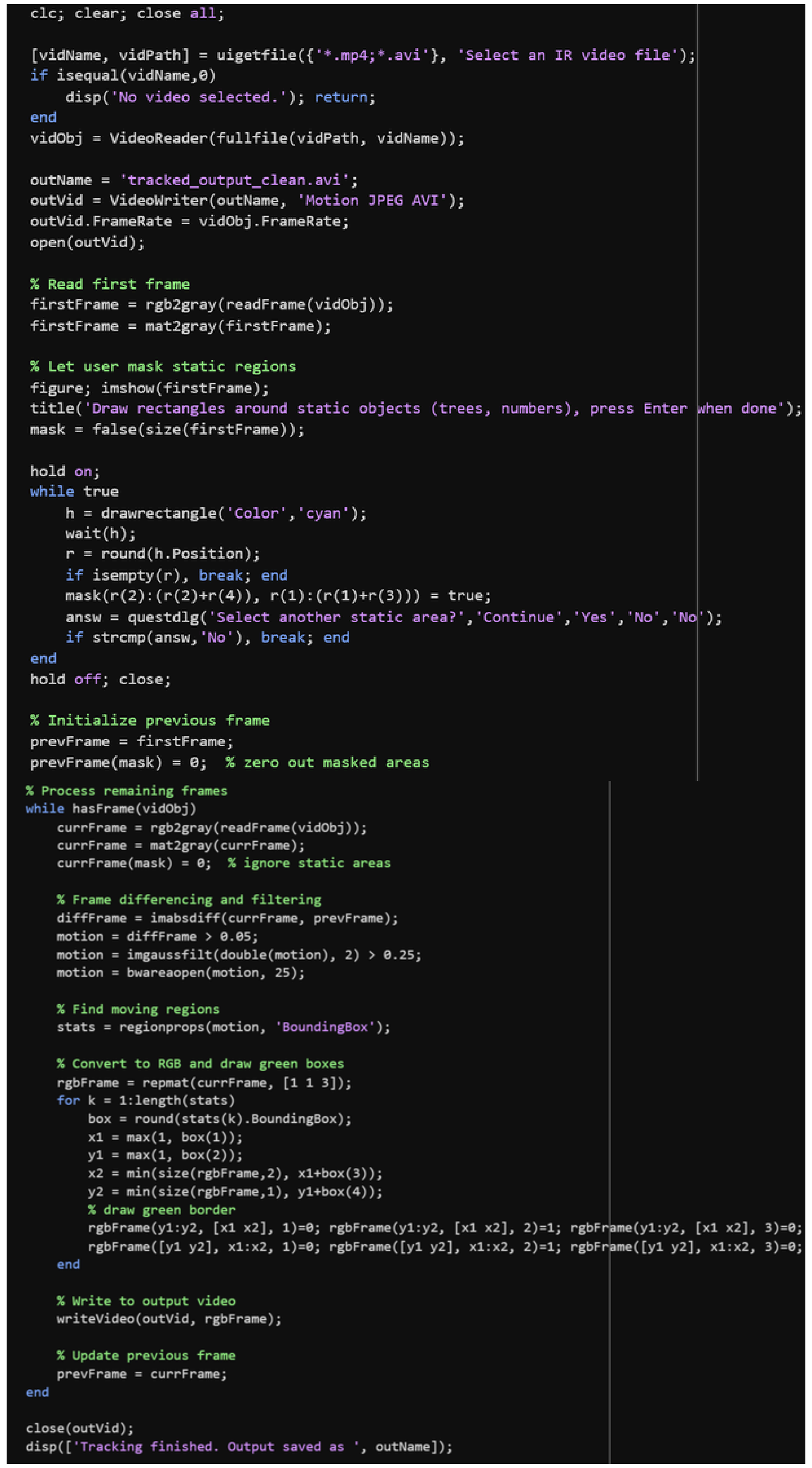
In this part, a MATLAB script named **myTracker.m** was developed to automatically detect and track airplanes in infrared (IR) videos.

The program first allows the user to select a video and manually mark static regions such as trees, poles, or on-screen text. These masked areas are then excluded from further analysis to prevent false detections.

After masking, each frame is compared with the previous one using frame differencing, which highlights areas that have changed between consecutive frames. By applying thresholding, Gaussian filtering, and morphological cleaning, the program isolates the true moving object—the airplane.

For every detected region, a green rectangular box is drawn directly on the frame, and its size automatically adjusts as the airplane moves closer or farther from the camera.

The resulting processed frames are saved into a new output video that clearly shows the airplane's motion over time.



```matlab
clc; clear; close all;

[vidName, vidPath] = uigetfile({'*.mp4;*.avi'}, 'Select an IR video file');
if isequal(vidName,0)
    disp('No video selected.'); return;
end
vidObj = VideoReader(fullfile(vidPath, vidName));

outName = 'tracked_output_clean.avi';
outVid = VideoWriter(outName, 'Motion JPEG AVI');
outVid.FrameRate = vidObj.FrameRate;
open(outVid);

% Read first frame
firstFrame = rgb2gray(readFrame(vidObj));
firstFrame = mat2gray(firstFrame);

% Let user mask static regions
figure; imshow(firstFrame);
title('Draw rectangles around static objects (trees, numbers), press Enter when done');
mask = false(size(firstFrame));

hold on;
while true
    h = drawrectangle('Color','cyan');
    wait(h);
    r = round(h.Position);
    if isempty(r), break; end
    mask(r(2):(r(2)+r(4)), r(1):(r(1)+r(3))) = true;
    answ = questdlg('Select another static area?','Continue','Yes','No','No');
    if strcmp(answ,'No'), break; end
end
hold off; close;

% Initialize previous frame
prevFrame = firstFrame;
prevFrame(mask) = 0;  % zero out masked areas
% Process remaining frames
while hasFrame(vidObj)
    currFrame = rgb2gray(readFrame(vidObj));
    currFrame = mat2gray(currFrame);
    currFrame(mask) = 0;  % ignore static areas

    % Frame differencing and filtering
    diffFrame = imabsdiff(currFrame, prevFrame);
    motion = diffFrame > 0.05;
    motion = imgaussfilt(double(motion), 2) > 0.25;
    motion = bwareaopen(motion, 25);

    % Find moving regions
    stats = regionprops(motion, 'BoundingBox');

    % Convert to RGB and draw green boxes
    rgbFrame = repmat(currFrame, [1 1 3]);
    for k = 1:length(stats)
        box = round(stats(k).BoundingBox);
        x1 = max(1, box(1));
        y1 = max(1, box(2));
        x2 = min(size(rgbFrame,2), x1+box(3));
        y2 = min(size(rgbFrame,1), y1+box(4));
        % draw green border
        rgbFrame(y1:y2, [x1 x2], 1)=0; rgbFrame(y1:y2, [x1 x2], 2)=1; rgbFrame(y1:y2, [x1 x2], 3)=0;
        rgbFrame([y1 y2], x1:x2, 1)=0; rgbFrame([y1 y2], x1:x2, 2)=1; rgbFrame([y1 y2], x1:x2, 3)=0;
    end

    % Write to output video
    writeVideo(outVid, rgbFrame);

    % Update previous frame
    prevFrame = currFrame;
end

close(outVid);
disp(['Tracking finished. Output saved as ', outName]);
```