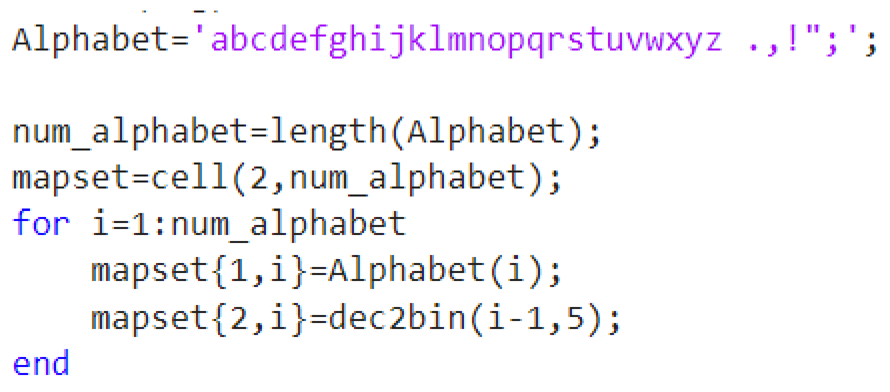
Narges Sadat Seyed Haeri – 810100165

Signals and Systems

CA3 Report

- ***Part 1:***

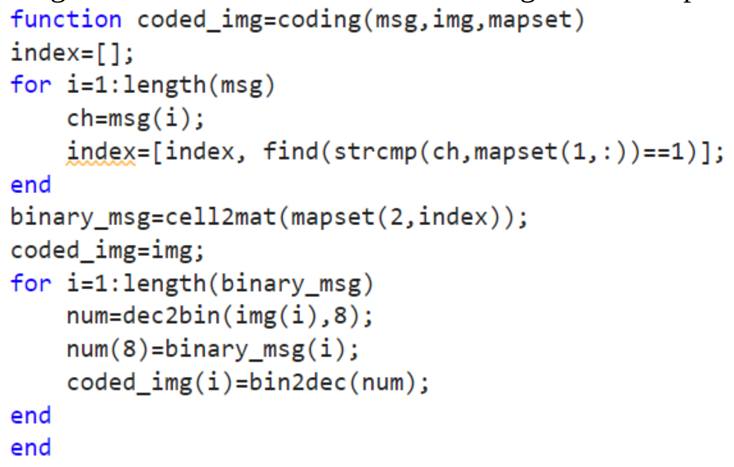
1. This code creates a mapping between characters in the Alphabet string and their 5-bit binary representations. It stores this mapping in a 2-row cell array called mapset. The first row contains the characters themselves, and the second row contains their binary representations. The loop iterates through the characters in Alphabet, assigns each character to mapset{1,i}, and calculates its 5-bit binary representation using dec2bin(i-1,5) before storing it in mapset{2,i}.



```
Alphabet='abcdefghijklmnopqrstuvwxyz .,!";';

num_alphabet=length(Alphabet);
mapset=cell(2,num_alphabet);
for i=1:num_alphabet
    mapset{1,i}=Alphabet(i);
    mapset{2,i}=dec2bin(i-1,5);
end
```

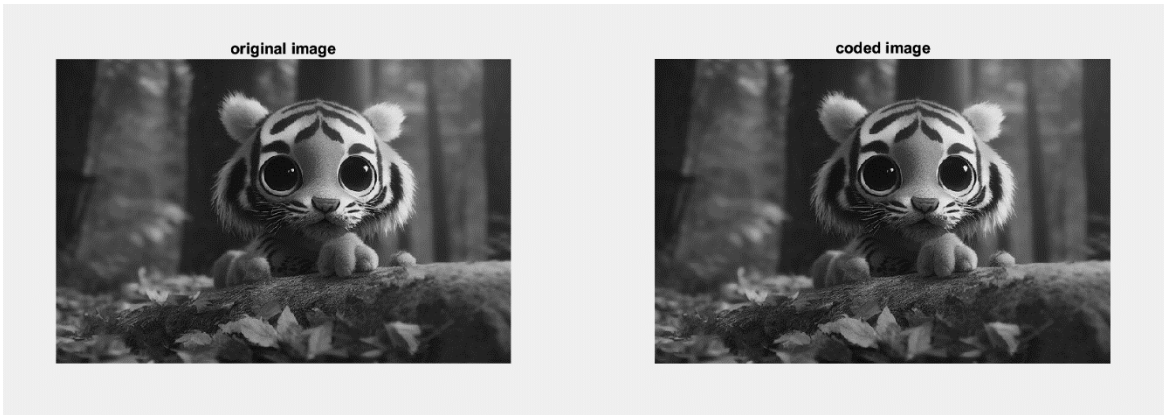
2. In this function, we encode a message into an image using a binary mapping. First, we initialize an empty array index to store character-to-binary code mappings. Then, we iterate through the message and determine the corresponding index in the binary mapping for each character, appending these indices to the index array. After processing the entire message, we create a binary message (binary_msg) by combining the binary codes from the mapping using the indices in the index array. Next, we make a copy of the original image (img) into a new variable called coded_img. We then iterate through each binary digit in binary_msg and use it to modify the corresponding pixel values in coded_img. Finally, we return the modified image, which now contains the encoded message. This process conceals the message within the image's pixel data while maintaining its visual appearance, and the

message can be later extracted using a corresponding decoding function.

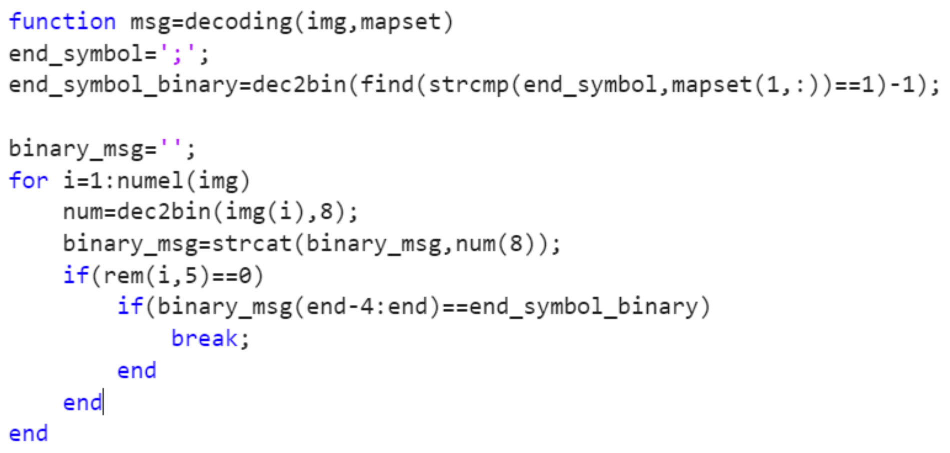


```
function coded_img=coding(msg,img,mapset)
index=[];
for i=1:length(msg)
    ch=msg(i);
    index=[index, find(strcmp(ch,mapset(1,:))==1)];
end
binary_msg=cell2mat(mapset(2,index));
coded_img=img;
for i=1:length(binary_msg)
    num=dec2bin(img(i),8);
    num(8)=binary_msg(i);
    coded_img(i)=bin2dec(num);
end
end
```

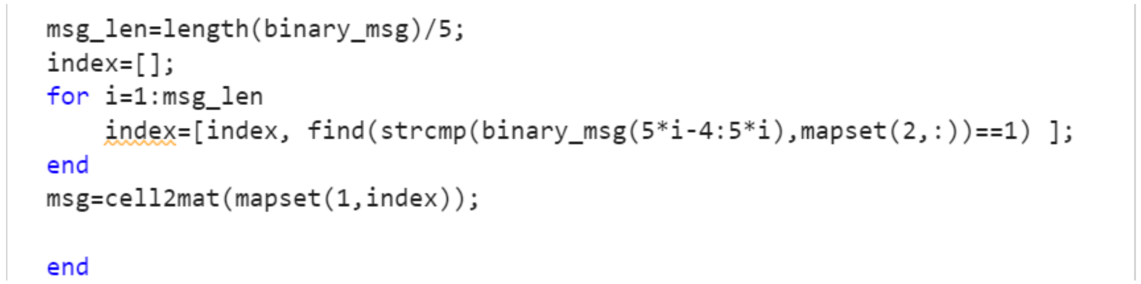
3. the changes in the image may not be readily apparent. The alterations to the image are made in a way that is visually imperceptible, based on the binary mapping. This is typically done to hide information within the image without being easily noticeable. As long as you have the corresponding inverse function (for message retrieval), the hidden information in the image will not be visibly distinguishable.



4. In this function, "decoding," we reverse the process performed by the "coding" function to extract a hidden message from an image encoded using a binary mapping. First, we define an "end_symbol" character, and its binary representation, "end_symbol_binary," is calculated based on the provided mapping. We initialize an empty string, "binary_msg," to store the binary message as we iterate through the image's pixel values, converting each to an 8-digit binary representation. We extract the 8th binary digit from each pixel and search for a specified end symbol within the binary message. Once found, we break out of the loop. We then divide the binary message into 5-digit segments and map each segment to its corresponding character using the binary mapping. The function returns the extracted message, "msg," which was originally hidden in the image.

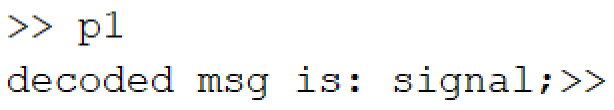


```
function msg=decoding(img,mapset)
end_symbol=';';
end_symbol_binary=dec2bin(find(strcmp(end_symbol,mapset(1,:))==1)-1);

binary_msg='';
for i=1:numel(img)
    num=dec2bin(img(i),8);
    binary_msg=strcat(binary_msg,num(8));
    if(rem(i,5)==0)
        if(binary_msg(end-4:end)==end_symbol_binary)
            break;
        end
    end
end
```



```
msg_len=length(binary_msg)/5;
index=[];
for i=1:msg_len
    index=[index, find(strcmp(binary_msg(5*i-4:5*i),mapset(2,:))==1) ];
end
msg=cell2mat(mapset(1,index));

end
```

Here is the result on the picture that we have coded before:



```
>> p1
decoded msg is: signal;>>
```

5. In general, if noise is added to an image after LSB-based steganography, because noise introduces unpredictable changes to pixel values, including the least significant bit (LSB) where the hidden message is encoded. Since the LSB is typically used for hiding information, noise can overwrite or distort the hidden data, making it difficult to recover the message accurately. it can make it more challenging or even impossible to accurately recover the hidden message. Therefore, the answer is typically "No."

6. We suggest two ways:

   - We can use Natural Language Processing (NLP) to investigate LSB steganography for patterns related to frequent English letters. Start by analyzing a significant English text corpus to identify high-frequency letters. Then, encode messages in images using LSB steganography, assigning specific bits to these common letters. After extracting the LSBs, analyze the text for patterns that match the frequently occurring letters, indicating their presence in the hidden message. This unique blend of NLP and steganography offers a captivating way to uncover hidden text patterns within images.

- Steganalysis involves various tests to detect the presence of hidden messages, distinct from the extraction process. Basic approaches include visual, structural, and statistical attacks aimed at identifying the steganographic algorithms used. The difficulty of detection depends on factors such as message size and cover object size, which can distort statistics. Targeted steganalysis focusing on specific algorithms is more successful in detecting anomalies. For instance, the least significant bits in images often exhibit non-random patterns due to factors like camera sensor quality and compression. Secret messages hidden in these bits may introduce conspicuous perfect randomization, detectable by comparing LSBs to the next-to-least significant bits in uncompressed images.

- ***Part 2:***

1. we create a Dual-Tone Multi-Frequency (DTMF) signal by encoding a sequence of DTMF digits. Here's what we do:

   We define the DTMF frequencies and the sampling rate. Then initialize a signal and set parameters for tone and silence durations. For each digit in the sequence:

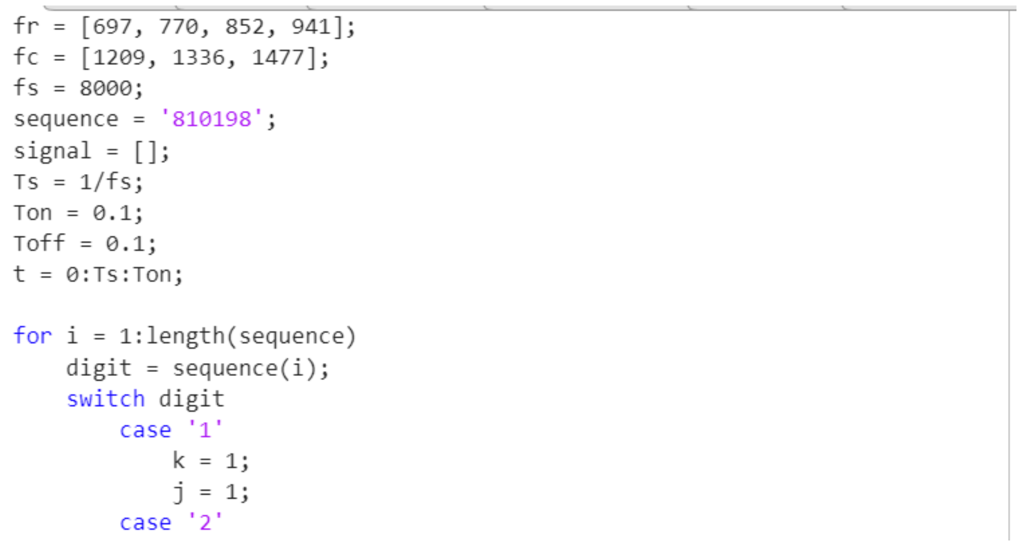
   We determine the corresponding row and column frequencies.

   Generate DTMF tones by combining these frequencies.

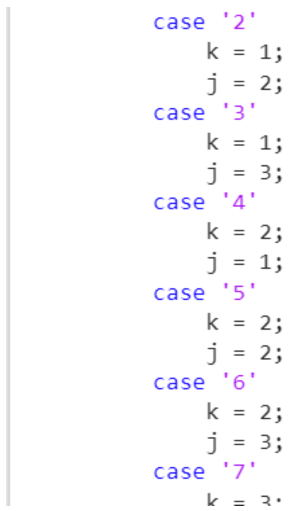
   We append these tones to the signal.

   We insert silence to separate consecutive DTMF tones if needed.

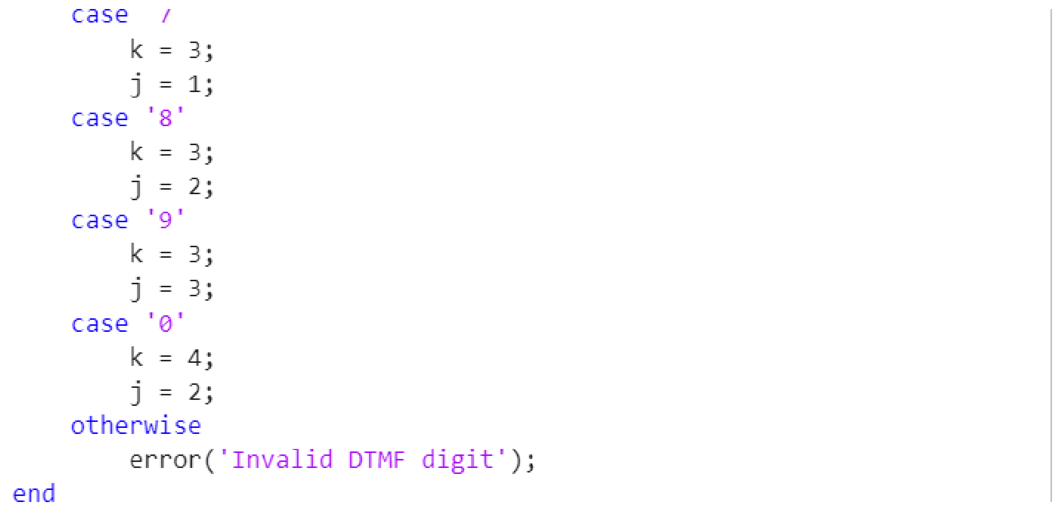
   Finally, we save the generated DTMF signal as an audio file.



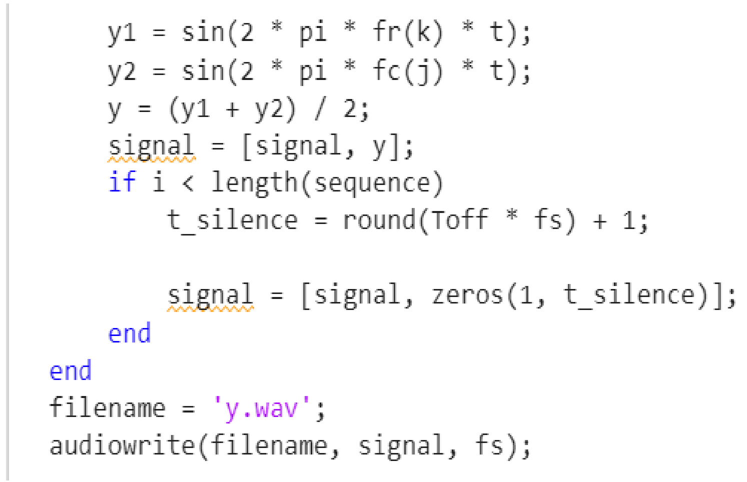
```
fr = [697, 770, 852, 941];
fc = [1209, 1336, 1477];
fs = 8000;
sequence = '810198';
signal = [];
Ts = 1/fs;
Ton = 0.1;
Toff = 0.1;
t = 0:Ts:Ton;

for i = 1:length(sequence)
    digit = sequence(i);
    switch digit
        case '1'
            k = 1;
            j = 1;
        case '2'
```



```
        case '2'
            k = 1;
            j = 2;
        case '3'
            k = 1;
            j = 3;
        case '4'
            k = 2;
            j = 1;
        case '5'
            k = 2;
            j = 2;
        case '6'
            k = 2;
            j = 3;
        case '7'
            k = 3;
```



```
        case '7'
            k = 3;
            j = 1;
        case '8'
            k = 3;
            j = 2;
        case '9'
            k = 3;
            j = 3;
        case '0'
            k = 4;
            j = 2;
        otherwise
            error('Invalid DTMF digit');
    end
```



```
        y1 = sin(2 * pi * fr(k) * t);
        y2 = sin(2 * pi * fc(j) * t);
        y = (y1 + y2) / 2;
        signal = [signal, y];
        if i < length(sequence)
            t_silence = round(Toff * fs) + 1;

            signal = [signal, zeros(1, t_silence)];
        end
    end
    filename = 'y.wav';
    audiowrite(filename, signal, fs);
```

2. In the below code, we did the following:

We loaded an audio file containing DTMF (Dual-Tone Multi-Frequency) tones.defined the frequencies associated with the four rows and four columns of the

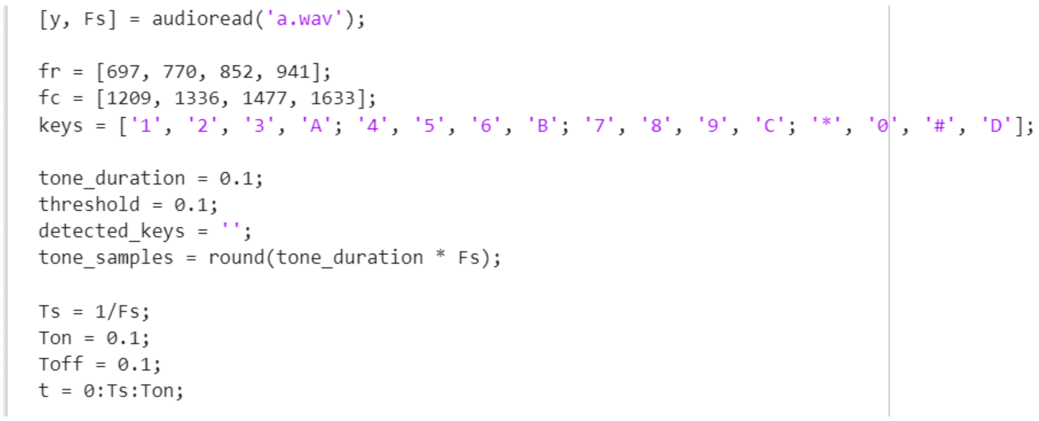
DTMF keypad.We created a matrix representing the DTMF keypad layout, allowing us to map detected frequencies to their corresponding digits. Then set parameters, such as the duration of each DTMF tone (tone_duration) and the correlation threshold for detection (threshold).We initialized an empty string called detected_keys to store the DTMF keys that we would identify.To facilitate the analysis, we calculated the number of samples within a DTMF tone (tone_samples) and established time-related variables.processed the audio file in chunks and, for each chunk:

computing correlations between the audio data and both the row and column DTMF tones.

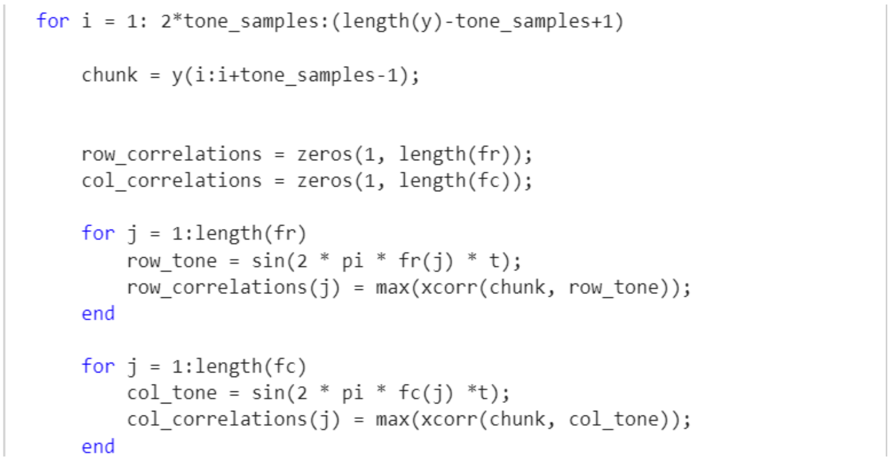
identified the index of the maximum correlation for both rows and columns.

If the correlation for both the row and column exceeded the specified threshold, we appended the corresponding DTMF key to the detected_keys string.

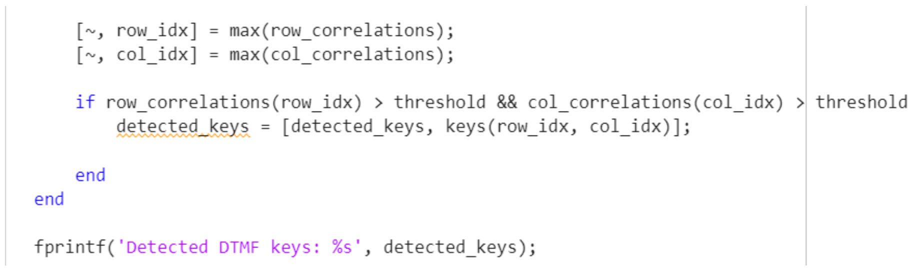
Finally, we printed the detected DTMF keys to the console, providing a way to recognize the keypad inputs encoded in the audio file.



```
[y, Fs] = audioread('a.wav');

fr = [697, 770, 852, 941];
fc = [1209, 1336, 1477, 1633];
keys = ['1', '2', '3', 'A'; '4', '5', '6', 'B'; '7', '8', '9', 'C'; '*', '0', '#', 'D'];

tone_duration = 0.1;
threshold = 0.1;
detected_keys = '';
tone_samples = round(tone_duration * Fs);

Ts = 1/Fs;
Ton = 0.1;
Toff = 0.1;
t = 0:Ts:Ton;
```

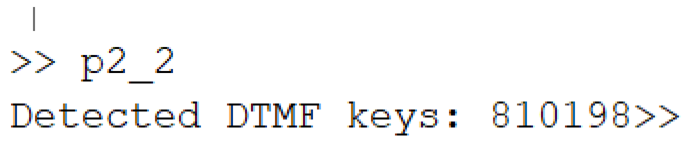


```
for i = 1: 2*tone_samples:(length(y)-tone_samples+1)

    chunk = y(i:i+tone_samples-1);


    row_correlations = zeros(1, length(fr));
    col_correlations = zeros(1, length(fc));

    for j = 1:length(fr)
        row_tone = sin(2 * pi * fr(j) * t);
        row_correlations(j) = max(xcorr(chunk, row_tone));
    end

    for j = 1:length(fc)
        col_tone = sin(2 * pi * fc(j) *t);
        col_correlations(j) = max(xcorr(chunk, col_tone));
    end
```



```
        [~, row_idx] = max(row_correlations);
        [~, col_idx] = max(col_correlations);

        if row_correlations(row_idx) > threshold && col_correlations(col_idx) > threshold
            detected_keys = [detected_keys, keys(row_idx, col_idx)];

        end
    end

    fprintf('Detected DTMF keys: %s', detected_keys);
```

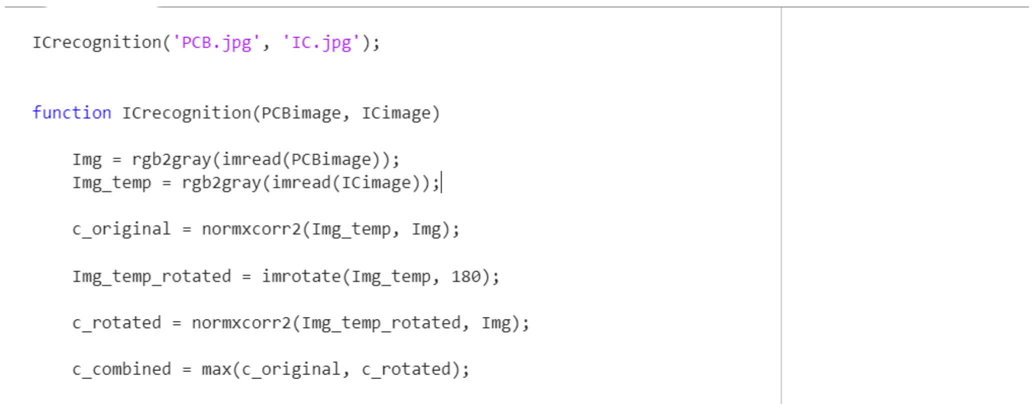
Here is the result for the sample “a.wav” :



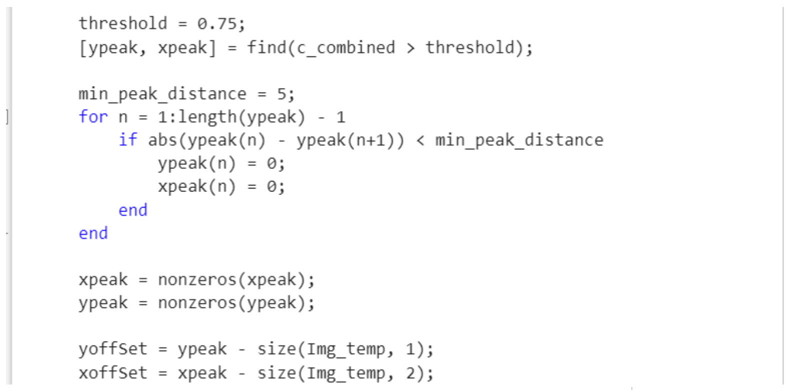
```
>> p2_2
Detected DTMF keys: 810198>>
```

- ***Part 3:***

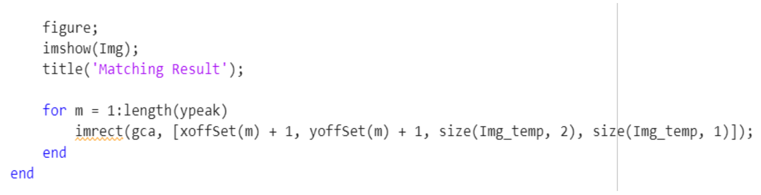
In this code, we first preprocess a printed circuit board (PCB) image and an integrated circuit (IC) template by converting them to grayscale. We then employ normalized cross-correlation to detect potential IC matches, considering both the original and a rotated IC template. The results from these cross-correlations are combined to account for potential rotations and multiple IC instances. By setting a correlation threshold at 0.75, we identify strong matches, and peak detection is used to locate these matches in the combined correlation image. To eliminate closely spaced duplicate detections, a minimum peak distance check is performed. The code calculates the offsets of recognized ICs in the PCB image and finally displays the PCB image with rectangles drawn around the recognized ICs. This code is valuable for automated IC recognition in PCB images, applicable in quality control and manufacturing for efficient component localization.



```
ICrecognition('PCB.jpg', 'IC.jpg');


function ICrecognition(PCBimage, ICimage)

    Img = rgb2gray(imread(PCBimage));
    Img_temp = rgb2gray(imread(ICimage));

    c_original = normxcorr2(Img_temp, Img);

    Img_temp_rotated = imrotate(Img_temp, 180);

    c_rotated = normxcorr2(Img_temp_rotated, Img);

    c_combined = max(c_original, c_rotated);
```

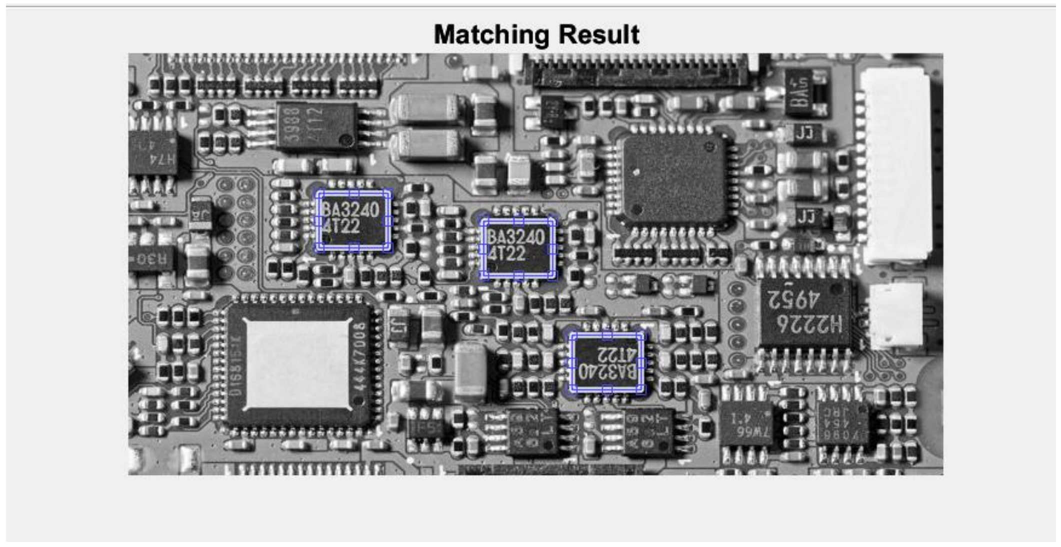


```
    threshold = 0.75;
    [ypeak, xpeak] = find(c_combined > threshold);

    min_peak_distance = 5;
    for n = 1:length(ypeak) - 1
        if abs(ypeak(n) - ypeak(n+1)) < min_peak_distance
            ypeak(n) = 0;
            xpeak(n) = 0;
        end
    end

    xpeak = nonzeros(xpeak);
    ypeak = nonzeros(ypeak);

    yoffSet = ypeak - size(Img_temp, 1);
    xoffSet = xpeak - size(Img_temp, 2);
```



```
    figure;
    imshow(Img);
    title('Matching Result');

    for m = 1:length(ypeak)
        imrect(gca, [xoffSet(m) + 1, yoffSet(m) + 1, size(Img_temp, 2), size(Img_temp, 1)]);
    end
end
```

Here is the results for the sample that we were given:


Matching Result