Assignment 1

ZHANG YIFEI

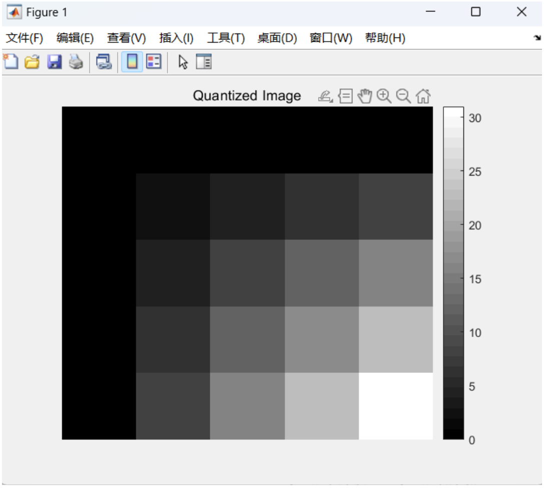
1 Image Sampling

First, create the grid and assign coordinate values to each grid.

Second, calculate the corresponding intensity values according to the formula.

Calculate the gray value based on gray level and intensity value.

```
% Define the resolution of the discrete image
num_pixels = 5;
% Define the number of gray levels
num_gray_levels = 32;

% Create a meshgrid for the x and y values
[x, y] = meshgrid(linspace(0, 1, num_pixels), linspace(1, 0, num_pixels));

% Calculate the intensity values for each pixel using the given function
intensity = x .* (1 - y);

% Quantize the intensity values to 32 gray levels
quantized_intensity = round(intensity * (num_gray_levels - 1));

% Create a colormap for the 32 gray levels
colormap(gray(num_gray_levels));

% Display the discrete image
imagesc(quantized_intensity);
axis off;
title('Quantized Image');
colorbar;
```



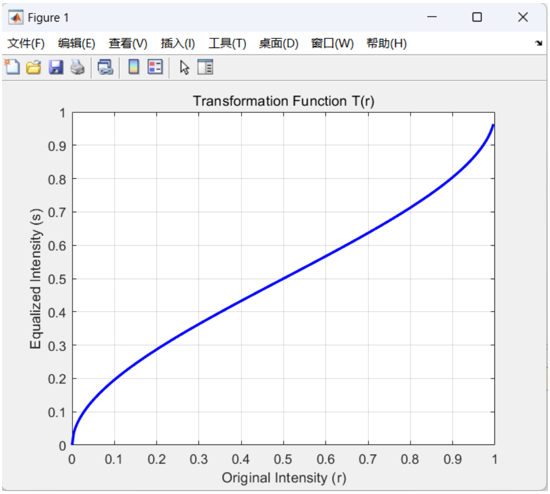
0	0	0	0	0
0	2	4	6	8
0	4	8	12	16
0	6	12	17	23
0	8	16	23	31

2 Histogram Equalization

$$S = T(r) = \int_0^r p_r(r)dr = 3r^2 - 2r^3$$

```
% Define the range
r = linspace(0, 1, 256);

% Define  pr
pr = 6 * r .* (1 - r);

% Calculate F(r)
F_r = cumtrapz(r, pr);

% Calculate the inverse of the F(r) to get the transformation s = T(r)
F_inverse = interp1(F_r, r, linspace(0, 1, numel(r)));

% Create a figure to visualize the transformation matrix T(r)
```

```
figure;
plot(r, F_inverse, 'b', 'LineWidth', 2);
xlabel('Original Intensity (r)');
ylabel('Equalized Intensity (s)');
title('Transformation Function T(r)');
grid on;

```



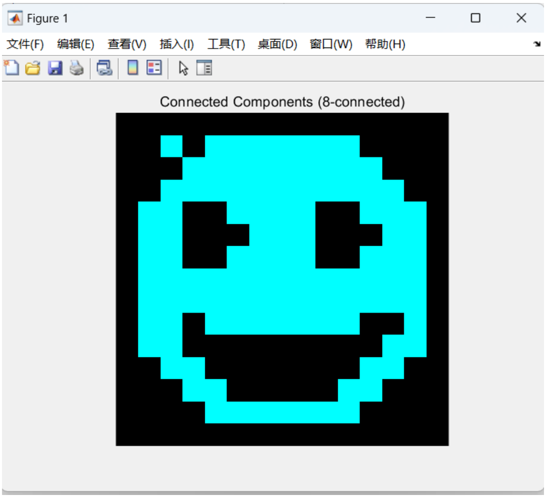
3 Neighborhood of Pixels

```
warning('off','all')
warning

% Define your binary image (replace this with your binary image data)
binary_image = [
    3 3 3 3 3 2 2 2 2 2 3 3 3 3 3;
    3 3 1 2 1 0 0 0 0 0 1 2 3 3 3;
    3 3 2 0 0 0 0 0 0 0 0 0 2 3 3;
    3 2 0 0 1 0 0 0 0 0 1 0 0 2 3;
    3 1 0 2 3 1 0 0 0 3 3 1 0 1 3;
    2 0 0 3 3 2 0 0 1 3 3 2 0 0 2;
    2 0 0 2 3 1 0 0 0 2 3 1 0 0 2;
    2 0 0 0 0 0 0 0 0 0 0 0 0 0 2;
    2 0 0 0 0 0 0 0 0 0 0 0 0 0 2;
    2 0 1 2 1 1 0 0 0 1 1 2 2 0 2;
    3 1 0 2 3 3 3 3 3 3 3 3 0 1 3;
    3 2 0 0 2 3 3 3 3 3 2 0 0 2 3;
    3 3 2 0 0 2 3 3 3 2 0 0 2 3 3;
    3 3 3 2 1 0 0 0 0 0 1 2 3 3 3;
```

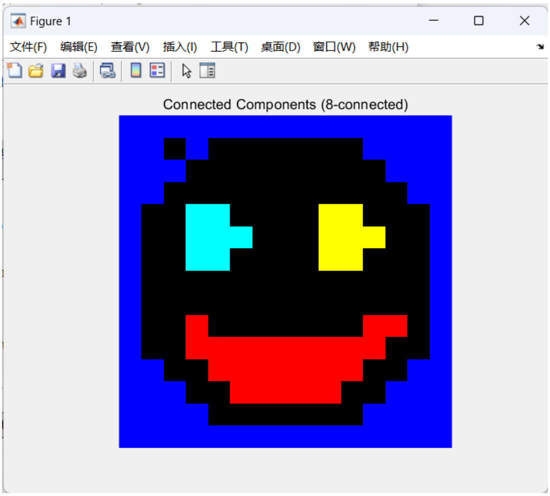
```
    3 3 3 3 3 2 2 2 2 2 3 3 3 3 3;
];

pixel = 1;
% Define 8-connected neighbors
neighbors = [-1, -1; -1, 0; -1, 1; 0, 1; 1, 1; 1, 0; 1, -1; 0, -1];

% Initialize labeled image and label counter
global labeled_visited_coordinates
global labeled_img
global connected_components
labeled_visited_coordinates = zeros(size(binary_image));
labeled_img = zeros(size(binary_image));
current_label = 0;
connected_components = cell(0);
%% Threshold the image
for i = 1:size(binary_image, 1)
    for j = 1:size(binary_image, 2)
        if binary_image(i, j) <=1
            binary_image(i, j) = 0;
        else
            binary_image(i, j) = 1;
        end
    end
end
%%

% Iterate through the binary image to find and label connected components
for i = 1:size(binary_image, 1)
    for j = 1:size(binary_image, 2)
        if labeled_visited_coordinates(i, j)==0
            if binary_image(i, j) == pixel
                current_label = current_label + 1;
                connected_components{current_label} = [];
                dfs(pixel,i, j,current_label,binary_image,labeled_img,
                 labeled_visited_coordinates,neighbors,connected_components);
            end
            labeled_visited_coordinates(i, j) = 1;
        end
    end
end

% Display the labeled image
imshow(label2rgb(labeled_img, 'jet', 'k'), 'InitialMagnification', 'fit');
title('Connected Components (8-connected)');
disp(labeled_img);
```

```
% Function to perform DFS for connected components labeling
function dfs(pixel,i, j,current_label,binary_image,labeled_img,
    labeled_visited_coordinates,neighbors,connected_components)

    global labeled_visited_coordinates
    global labeled_img
    global connected_components

    if labeled_visited_coordinates(i, j) == 0
        labeled_visited_coordinates(i, j) = 1;
        if binary_image(i,j)==pixel
            labeled_img(i, j) = current_label;
            connected_components{current_label} = [connected_components{current_label}; [i, j]];

            % Recursively call DFS on neighboring pixels
            for k = 1:size(neighbors, 1)
                ni = i + neighbors(k, 1);
                nj = j + neighbors(k, 2);
                if ni < 1 || ni > size(binary_image, 1) || nj < 1 || nj > size(binary_image, 2)
                    continue;
                else
                    dfs(pixel,ni,nj,current_label,binary_image,labeled_img,
                    labeled_visited_coordinates,neighbors,connected_components);
                end
            end
        end
    end
end
```

Elements that has intensity 0 or 1 (non-black part):



8-connected components for $g = 1$(non-black part):



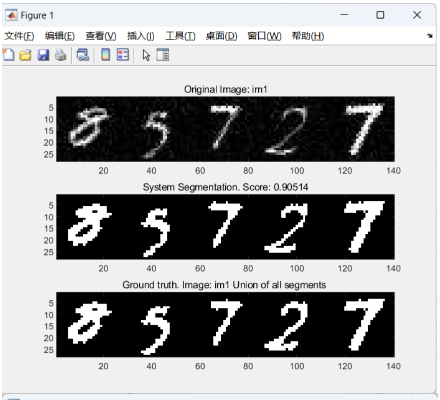
4 Segmentation Part of OCR

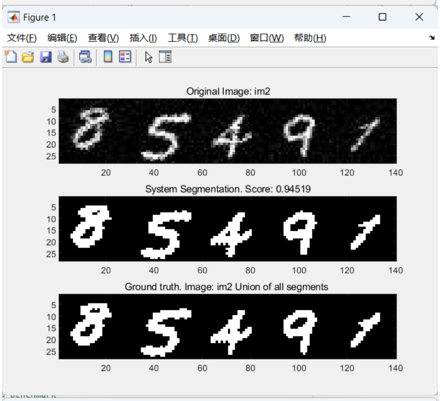
```
function S = im2segment(img)
img = uint8(img);
% figure;
% imshow(img);

% Specify the standard deviation of the Gaussian filter (to control the degree of blurring)
sigma = 0.5;

% Perform Gaussian filtering
```

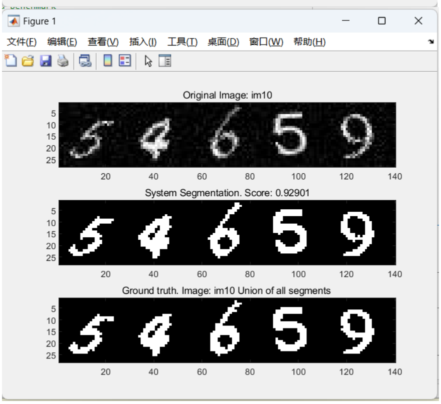
```
img = imgaussfilt(img, sigma);

% img = imbilatfilt(img);
% img = medfilt2(img, [3, 3])

threshold = 0.158 % Set the threshold of image binarize
binary_image = imbinarize(img, threshold); % binarize
% imshow(binary_image);

%% 8-connected components
labeledImage = bwlabel(binary_image, 8);
minPixels = 1 ; % set min pixel num


%% Extracting information about connected components
stats = regionprops(labeledImage, 'BoundingBox', 'PixelIdxList');

% Initialize an array of cells for storing segmented images
numStats = numel(stats);
S = cell(1, numStats);

% Create an array of flags to keep track of merged cells
merged = false(1, numStats);
small = false(1, numStats);

%% Storing coordinate indexes of different labels in cells
for i = 1:numStats
    % Get the coordinate index of the current connected component
    pixelIdxList = stats(i).PixelIdxList;

    % Create a segmented image of the same size as the original image
    segmented_image = zeros(size(labeledImage));
    segmented_image(pixelIdxList) = 1;
    numPixels = sum(segmented_image(:) == 1);
    if numPixels < minPixels
        small(i)=true;
    end
    % Storing Segmented Images into Cells
    S{i} = segmented_image;
end


%%
Dis_threshold = 20; % distance threshold

% Calculate the center coordinates of each cell and combine them
for i = 1:numStats
```

```matlab
    if ~merged(i) && ~small(i)
        for j = i+1:numStats
            if ~merged(j) && ~small(j)
                % Calculate the center coordinates
                centro_i = regionprops(S{i}, 'Centroid');
                centro_j = regionprops(S{j}, 'Centroid');

                % Extracting the center coordinates
                centro_i = centro_i.Centroid;
                centro_j = centro_j.Centroid;

                % culculate the Euclidean distance
                distance = norm(centro_i - centro_j);

                if distance < Dis_threshold
                    % Merge two cells and add elements to the first cell
                    S{i} = S{i} | S{j};
                    merged(j) = true;
                end
            end
        end
    end
end

S = S(~merged);
end
```

```
You tested 10 images in folder ../datasets/short1
The jaccard scores for all segments in all images were
    0.9512    0.8868    0.9302    0.7951    0.9444
    0.9010    0.9424    0.8692    0.9658    0.9545
    0.9419    0.9454    0.9528    0.9474    0.9333
    0.7451    0.9137    0.9438    0.9310    0.9172
    0.9262    0.9170    0.9493    0.8958    0.9448
    0.9624    0.9298    0.9732    0.9527    0.9931
    0.9461    0.8824    0.9115    0.9565    0.9328
    0.9298    0.9598    0.7843    0.9282    0.8777
    0.9268    0.9203    0.9432    0.9449    0.8901
    0.9633    0.7304    0.9231    0.9364    0.7700

The mean of the jaccard scores were 0.91628
This is great!
```







5 Dimensionality

A:

Dimension k for A:

The set of gray-scale images with 3 × 2 pixels forms a vector space. To determine the dimension, we need to consider the number of independent basis images that can span this space. In this case, each pixel in a 3 × 2 image contributes to the dimension, so the total number of pixels is 3 * 2 = 6. Therefore, the dimension k for A is 6.

Basis for A:

To define a basis for this vector space, we can choose 6 linearly independent 3 × 2 images. Here's an example of such a basis:

$$e_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, e_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, e_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, e_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, e_5 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}, e_6 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Each basis element is a 3 × 2 image with a single pixel set to 1, and all other pixels set to 0. These basis images are linearly independent and can span the vector space of all 3 × 2 images.

B:

Dimension k for B:

The set of gray-scale images with 1500 × 2000 pixels forms a vector space. In this case, the dimension k is equal to the total number of pixels in each image, which is 1500 * 2000 = 3,000,000.

Choosing Basis Elements for B:

In the case of such high-dimensional vector spaces, it's impractical to explicitly list individual basis elements. However, you can choose a basis for this space by considering pixel patterns. For example, you can select basis images that represent certain features or patterns commonly found in images. These basis images should be linearly independent and span the entire space.

For instance, you might choose basis images that represent horizontal lines, vertical lines, diagonal lines, gradients, textures, and so on. The choice of basis elements can depend on the specific application or problem you are working on. There are various techniques for automatically extracting or learning basis elements from a set of images, such as Principal Component Analysis (PCA) or Independent Component Analysis (ICA).

The key is to ensure that the chosen basis elements are diverse enough to capture a wide range of image variations and are capable of representing any image in the vector space through linear combinations.

6 Scalar products and norm on images

Scalar Product for Images: The scalar product (or dot product) for images is defined as the sum of the element-wise products of corresponding pixels in two images. If we have two images, u and v, both of the same size, the scalar product u · v is computed as follows:

$$u \cdot v = \sum_{i=1}^{M} \sum_{j=1}^{N} u(i,j) \cdot v(i,j)$$

Where M and N are the dimensions (rows and columns) of the images u and v, respectively.

Norm of an Image: The norm of an image represents the "size" or magnitude of the image as if it were a vector. There are various ways to define the norm of an image, but one common approach is to use the Frobenius Norm. For an image u of size M x N, Frobenius Norm, denoted as ||u||, is defined as:

$$|| u ||= \sqrt{\sum_{i=1}^{M} \sum_{j=1}^{N} u|(i,j)|^2}$$

```
clc
clear
close all
% Given images
u = [3 -7; -1 4];
v = 1/2 * [1 -1; -1 1];
w = 1/2 * [-1 1; -1 1];

% Calculate norms
norm_u = norm(u, 'fro');  % Frobenius norm for the image
norm_v = norm(v, 'fro');
norm_w = norm(w, 'fro');

% Calculate scalar products
u_dot_v = sum(sum(u .* v));
u_dot_w = sum(sum(u .* w));
v_dot_w = dot(v, w);

% Check if matrices u and v_dot_w are orthonormal
is_orthonormal = isequal(norm_v , 1) && isequal(norm_w , 1)&& isequal(dot(v(:), w(:)), 0);

% Calculate the orthogonal projection of u onto the subspace spanned by {v, w}
```

```
projection = (u_dot_v / (norm_v^2)) * v + (u_dot_w / (norm_w^2)) * w;

%%
approximation_error = sum(abs(u(:) - projection(:)).^2);
u_norm = (norm(u, 'fro'))^2;
abs_diff=abs(u(:) - projection(:));
diff_norm=(norm(abs_diff(:), 'fro'))^2;
diff = diff_norm/u_norm;

%%
% Display results
fprintf('Norm of u: %.2f\n', norm_u);
fprintf('Norm of v: %.2f\n', norm_v);
fprintf('Norm of w: %.2f\n', norm_w);
fprintf('Scalar Product u · v: %.2f\n', u_dot_v);
fprintf('Scalar Product u · w: %.2f\n', u_dot_w);
fprintf('Scalar Product v · w: %.2f\n', v_dot_w);
fprintf('Are matrices {v, w} orthonormal? %d\n', is_orthonormal);
disp('Orthogonal Projection of u onto {v, w}:');
disp(projection)
disp(['my diff: ',num2str(diff)]);
```

```
Norm of u: 8.66
Norm of v: 1.00
Norm of w: 1.00
Scalar Product u · v: 7.50
Scalar Product u · w: -2.50
Scalar Product v · w: 0.00
Are matrices {v, w} orthonormal? 1
Orthogonal Projection of u onto {v, w}:
    5.0000   -5.0000
   -2.5000    2.5000

my diff: 0.16667
>>
```

Now, let's calculate the norms and scalar products for the given images u, v, and w:

1. ||u|| : 8.660
2. ||v||: 1
3. ||w||: 1
4. u · v = 7.50
5. u · w = -2.50
6. v · w = 0

7. Matrices {v, w} are orthonormal because their scalar product (v · w) is zero, and the norm of v and w are both one.

8. $$projection = \begin{bmatrix} 5 & -5 \\ -2.5 & 2.5 \end{bmatrix}$$

9. The projection is the best approximation of u within the subspace

7 Image Compression

1. **Background**:
    - A is a known matrix containing a set of basis vectors as columns.
    - x is the parameter vector for which we require a solution, denoting the coefficients of the basis vectors.
    - f(:) is the vector form of the observations.
2. **Problem Description**: We wish to find the value of the parameter vector x that best matches the linear model A * x with the observed data f(:).
3. **Objective of least squares**:

    Minimize the norm of the residual vector, i.e., minimize the following equation:

    $$minimize||A * x - f(:)||_{2^2}$$

    This is equivalent to finding the parameter vector x such that the linear model A * x is as close as possible to the observed data f(:).
4. **Solution process**:
    - By computing A * x, we can obtain an estimate of the linear model.
    - Calculate the residual vector: residual = A * x - f(:).
    - The goal of least squares is to find the parameter vector x that minimizes the norm of the residual vector residual.
    - This is accomplished by solving the following regular equation:

      A' * A * x = A' * f(:)

      where A' denotes the transpose matrix of A.
    - Ultimately, the x-value will be solved such that A * x is closest to f(:) and the residual vector residual minimizes the norm.
5. **Solve for the value of x**:
    - It is convenient to solve regular equations to find the value of x using the left division operator () in MATLAB. Specifically, x = A \ f(:) will automatically compute the regular equation and solve for the value of x.

```
clc
clear
```

```
close all

% Define the basis images
phi1 = 1/2 * [1 0 -1; 1 0 -1; 0 0 0; 0 0 0];
phi2 = 1/3 * [1 1 1; 1 0 1; -1 -1 -1; 0 -1 0];
phi3 = 1/3 * [0 1 0; 1 1 1; 1 0 1; 1 1 1];
phi4 = 1/2 * [0 0 0; 0 0 0; 1 0 -1; 1 0 -1];


% Define the original image f
f = [-2 6 3; 13 7 5; 7 1 8; -3 4 4];

% Verify orthonormality of basis images
orthonormality = isequal(norm(phi1,1),1) &&
isequal(norm(phi2,1),1) && ...
    isequal(norm(phi3,1),1) && isequal(norm(phi4,1),1) &&...
    isequal(dot(phi3(:), phi4(:)), 0) && ...
    isequal(dot(phi1(:), phi2(:)), zeros(size(3))) && ...
    isequal(dot(phi1(:), phi3(:)), zeros(size(3))) && ...
    isequal(dot(phi1(:), phi4(:)), zeros(size(3))) && ...
    isequal(dot(phi2(:), phi3(:)), zeros(size(3))) && ...
    isequal(dot(phi2(:), phi4(:)), zeros(size(3)));


%% pseudo-inverse
% % Stack the basis images into a matrix
% A = [phi1(:), phi2(:), phi3(:), phi4(:)];
%
% % Calculate the coefficients using the pseudo-inverse
% x = pinv(A) * f(:);
%
% % Reconstruct the approximate image
% fa = A * x;
%
%
% fa_matrix = reshape(fa, 4, 3);

%%
% Stack the basis images into a matrix
A = [phi1(:), phi2(:), phi3(:), phi4(:)];

% Calculate the coefficients using the
x = A \ f(:);

% Reconstruct the approximate image
fa = A * x;

% Calculate the approximation error
approximation_error = sum(abs(f(:) - fa).^2);
% approximation_error = norm(f(:) - fa,'fro');

```

```
fa_matrix = reshape(fa, 4, 3);
%%
% Display results
disp('Orthonormality of Basis Images:');
disp(['Are basis images orthonormal ? orthonormality = ', num2str(orthonormality)]);
disp('Coordinates (x1, x2, x3, x4):');
disp(x);
disp('Approximate Image fa:');
disp(fa_matrix);
disp([Norm Approximation Error: ', num2str(approximation_error)]);
```

```
Orthonormality of Basis Images:
Are basis images orthonormal ? orthonormality = 1
Coordinates (x1, x2, x3, x4):
    1.5000
    1.6667
   17.0000
   -4.0000

Approximate Image fa:
    1.3056    6.2222   -0.1944
    6.9722    5.6667    5.4722
    3.1111   -0.5556    7.1111
    3.6667    5.1111    7.6667

Norm Approximation Error: 136.9722
my diff: 0.30643
```

I think the result of task 2 is better, considering that there is a difference in the dimension of the matrix, I first calculate the norm of the error between the elements of the matrix, and then calculate the ratio of diff_norm and Norm Approximation Error as a basis for judgment, the smaller the ratio the more approximate it is.

8 Image Bases

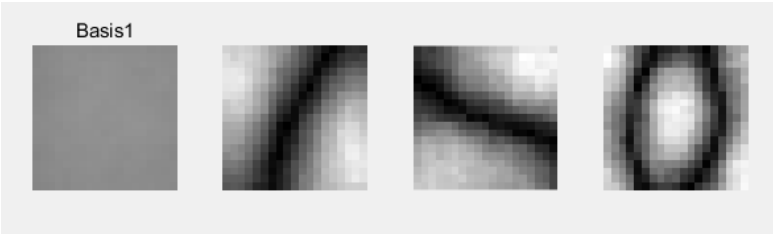
```
clc
clear
close all
% Load the dataset
load('assignment1bases.mat');

% Initialize variables to store the mean error norms
mean_error_norms = zeros(2, 3); % Rows: test sets, Columns: bases

%% Iterate through the test sets (general and face)
```

```
for test_set = 1:2
    % Select the test images from the corresponding stack
    test_images = stacks{test_set};

    % Iterate through the three bases
    for basis_idx = 1:3
        % Select the basis for this iteration
        basis = bases{basis_idx};

        % Initialize an array to store error norms for individual images
        error_norms = zeros(size(test_images, 3), 1);
        % Iterate through all test images
        for img_idx = 1:400
            % Get the current test image
            img = test_images(:, :, img_idx);

            % Project the image onto the basis and calculate the error norm
            [up, r] = projectAndCalculateError(img, basis);

            % Store the error norm
            error_norms(img_idx) = r;
        end

        % Calculate the mean error norm for this basis and test set
        mean_error_norm = mean(error_norms);

        % Store the result in the mean_error_norms matrix
        mean_error_norms(test_set, basis_idx) = mean_error_norm;

    end
end

%% print result
% choose the test image
plotset_idx=2;
plotimg_idx=375;
test_images = stacks{plotset_idx};
plot_img = zeros(19,19,3);
    for basis_idx = 1:3
        basis = bases{basis_idx};
        [plot_up, r] = projectAndCalculateError(test_images(:, :, plotimg_idx), basis);
        plot_img(:,:,basis_idx)=plot_up;
    end

% plot results images
figure;
```

```
subplot(1, 4, 1);
imshow(uint8(test_images(:, :, plotimg_idx)));
title('Test Image');

subplot(1, 4, 2);
imshow(uint8(plot_img(:,:,1)));
title('Projection1');

subplot(1, 4, 3);
imshow(uint8(plot_img(:,:,2)));
title('Projection2');

subplot(1, 4, 4);
imshow(uint8(plot_img(:,:,3)));
title('Projection3');

basis1 = abs(bases{1});
min_value = min(basis1(:));
max_value = max(basis1(:));
basis1 = 255 * (basis1 - min_value) / (max_value - min_value);
basis1 = uint8(basis1);

figure;
subplot(1, 4, 1);
imshow(basis1(:, :, 1));
title('Basis1');
subplot(1, 4, 2);
imshow(basis1(:, :, 2));
subplot(1, 4, 3);
imshow(basis1(:, :, 3));
subplot(1, 4, 4);
imshow(basis1(:, :, 4));

% Display or use the mean_error_norms matrix as needed
disp('Mean Error Norms:');
disp(mean_error_norms);
```

```
function [up, r] = projectAndCalculateError(u, basis)
    % Flatten the image into a column vector
    reshape_u = u(:) ;
    % Create a matrix containing the basis vectors as columns
    reshape_basis = reshape(basis, [], 4);
    x = reshape_basis \ u(:);
    up = reshape_basis * x;
    % Calculate the error norm
     r = norm(u(:) - up,"fro");
%    r = sum(abs(reshape_u - up).^2);
    up = reshape(up, 19, 19);
end

```

Basis1:



Basis2:

Basis3:

- TestSet1 :

image100:

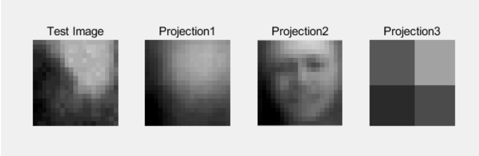


image150:

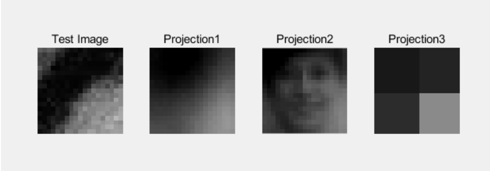


image250:

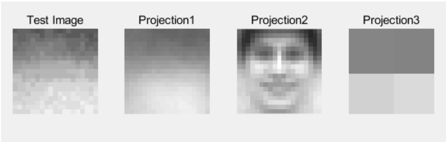
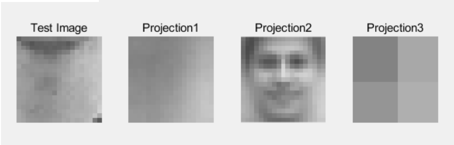


image375:



- TestSet2 :

image75:

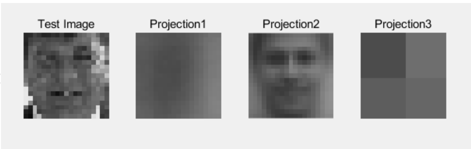


image175:

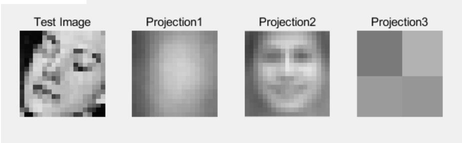


image275:

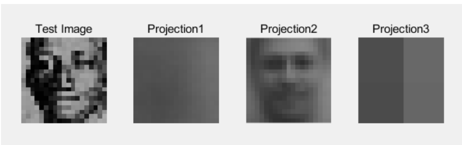
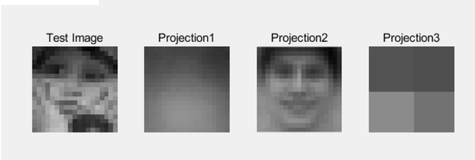


image375:



- Mean Error Norms :

```
Mean Error Norms:
  649.2013  795.1902  697.3214
  860.4754  821.0271  944.9009
```

Basis 1 works best on test set 1 ,because the mean of the error norm on test set 1 is the smallest.

And basis 2 works best on test set 2 ,because the mean of the error norm on test set 1 is the smallest. Moreover, test set 2 is all face images, and the images processed by basis2 are also face images.