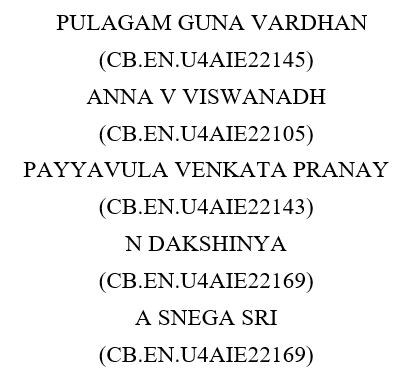
**THREAD ART PORTRAIT USING SVD**

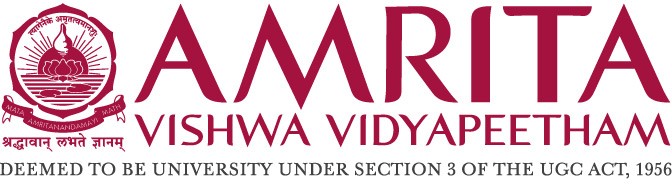
**Project report**

Submitted by

BATCH-B GROUP-17

AS A PART OF SUBJECT

Mathematics for Computing-2



**Centre for Computational Engineering and Networking**

**AMRITA SCHOOL OF ENGINEERING**

## AMRITA VISHWA VIDYAPEETHAM

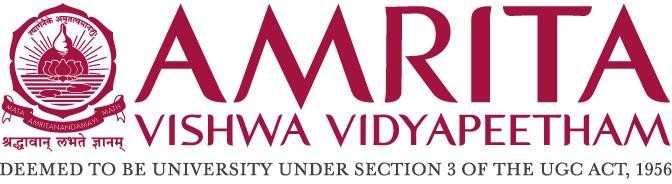
COIMBATORE - 641112 (INDIA)

**JULY - 2023**

**ARTIFICIAL INTELLIGENCE**

**AMRITA VISHWA VIDYAPEETHAM**

**COIMBATORE - 641 112**



**BONAFIDE CERTIFICATE**

This is to certify that the thesis entitled “**Mathematics for computing-2”,** “Submitted by PULAGAM GUNA VARDHAN (CB.EN.U4AIE22145), ANNA V VISWANADH (CB.EN.U4AIE22105), PAYYAVULA VENKATA PRANAY (CB.EN.U4AIE22143), N DAKSHINYA (CB.EN.U4AIE22169), A SNEGA SRI (CB.EN.U4AIE22163) for the award of the Degree of Bachelor of Technology in the “CSE(AI)” is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

**Dr. Neethu Mohan**

Project Guide

**Dr. K.P.Soman**

Professor and Head CEN

Submitted for the university examination held on 14/07/23

## 

## DECLARATION

We hereby declare that this project submitted to the Center for Computational Engineering and Networking is a record of the original work done under the guidance of Dr.Neethu Mohan, Professor At CEN,

Amrita Vishwa Vidyapeetham.

Signature of the faculty –

**Place: Coimbatore**

**Date: 12-07-2023**

**Name Roll number Signature of Students**

Pulagam Guna Vardhan CB.EN.U4AE22145

Anna V Viswanadh CB.EN.U4AE22105

Payyavula Venkata Pranay CB.EN.U4AE22143

N Dakshinya CB.EN.U4AE22169

A Snega Sri CB.EN.U4AE22163

**CHAPTER 1**

**DENOISING THE INPUT IMAGE USING SVD**

Denoising an input image using Singular Value Decomposition (SVD) refers to the process of reducing the noise present in the image while preserving important image features. SVD is a matrix factorization technique that decomposes a matrix into three components: U, Σ, and V^T. Denoising with SVD involves modifying the singular values of the image's SVD decomposition to suppress noise.

1. Obtain the noisy input image that you want to denoise.
2. Convert the image to grayscale if it is in RGB format (optional, if applicable).
3. Perform Singular Value Decomposition (SVD) on the matrix representation of the image. This results in three matrices: U, Σ, and V^T.
4. Analyze the singular values in the diagonal matrix Σ. The singular values indicate the amount of energy or importance associated with each singular vector.
5. Determine a threshold or rule to decide which singular values to keep and which to discard. Typically, the threshold is based on the noise level and the desired denoising effect. Higher threshold values retain more singular values, preserving more image details but potentially retaining more noise.
6. Modify the diagonal matrix Σ by setting the singular values beyond the threshold to zero. This operation removes the noise components associated with those singular values.
7. Reconstruct the denoised image by multiplying the modified U, Σ, and V^T matrices. The denoised image is obtained by multiplying U, the modified Σ, and the transpose of V.
8. Convert the denoised image to the desired data type (e.g., uint8) and rescale the pixel values if necessary. This step ensures that the denoised image is in the appropriate format for further processing or display.

By following these steps, you can apply SVD-based denoising to portrait images, reducing noise while preserving important image features and enhancing the overall quality of the portrait.

**MATLAB CODE:**

% Read the noisy image

noisyImage = orgimage;

% Convert the image to grayscale if it is in RGB format

if size(noisyImage, 3) == 3

noisyImage = rgb2gray(noisyImage);

end

% Perform Singular Value Decomposition (SVD) on the noisy image matrix

[U, S, V] = svd(double(noisyImage));

% Specify the number of singular values to keep for denoising

numSingularValues = 100; % Adjust this parameter as per your requirement

% Set the singular values beyond the specified threshold to zero

S(numSingularValues+1:end, numSingularValues+1:end) = 0;

% Reconstruct the denoised image using the modified singular values

denoisedImage = U \* S \* V';

% Convert the denoised image to uint8 format (8-bit) and rescale the values

denoisedImage = uint8(255 \* mat2gray(denoisedImage));

% Display the denoised image

imshow(denoisedImage);

title('Denoised Image');

**CODE EXPLAINATION:**

1. In this first line, the variable noisyImage is assigned the value of orgimage, which represents the noisy image that you want to denoise. orgimage is assumed to be the original image data.
2. The code checks if the noisyImage is in RGB format by checking the number of dimensions (channels) of the image. If the image has three channels, it means it is an RGB image, so rgb2gray function is used to convert it to grayscale. This step is necessary because the subsequent SVD operation is typically performed on grayscale images.
3. The third line applies the Singular Value Decomposition (SVD) to the noisyImage matrix. svd is a built-in MATLAB function that decomposes the matrix into three components: U, S, and V. U and V are orthogonal matrices, and S is a diagonal matrix containing the singular values in descending order.
4. Here, you specify the number of singular values you want to keep for denoising. This value determines the level of denoising and can be adjusted based on your requirements.
5. The next line sets the singular values beyond the specified threshold to zero. It modifies the diagonal matrix S such that all singular values beyond numSingularValues are set to zero. By setting these values to zero, you discard the high-frequency noise components associated with them.
6. The denoised image is reconstructed by multiplying the modified U, S, and V matrices. This operation restores the denoised image based on the modified singular values. The result is assigned to the variable denoisedImage.
7. Next it converts the denoised image to the uint8 format, which represents an 8-bit image. Additionally, mat2gray function is used to rescale the pixel values of the denoised image between 0 and 1. Then, the rescaled image is multiplied by 255 to map the pixel values back to the range of 0 to 255, suitable for display.
8. Finally, this code displays the denoised image using imshow function and sets the title of the displayed image as 'Denoised Image'.
9. By running this code, we will be able to denoise the noisy image using Singular Value Decomposition (SVD) and visualize the resulting denoised image.

**CHAPTER 2**

**SHARPENING THE DE-NOISED IMAGE USING SVD**

Sharpening a denoised image using Singular Value Decomposition (SVD) is a technique commonly employed in image processing to enhance the details and improve the overall quality of an image. SVD is a mathematical matrix decomposition method that can be used to analyze and manipulate image data.

Here's a general approach to sharpening a denoised image using SVD:

1. Denoising: Begin by applying a denoising technique to the original image. This step aims to remove or reduce the noise present in the image while preserving the essential details as much as possible.
2. SVD Decomposition: Once the denoising step is completed, perform an SVD decomposition of the denoised image. SVD breaks down the image matrix into three components: U, Σ, and VT. U represents the left singular vectors, Σ contains the singular values, and VT denotes the right singular vectors.
3. Reconstruct the Image: Reconstruct the modified SVD representation by multiplying the modified U, Σ, and V^T matrices. This step combines the modified singular values with the original singular vectors to create a sharpened version of the image.
4. Overall, sharpening the denoised image using SVD can be an effective technique to enhance the visual quality of an image by emphasizing the important details and reducing the blurring effects caused by noise.

**MATLAB CODE:**

%% sharpening the de-noised image using SVD

% Read the image

blurImage = denoisedImage;

% Perform Singular Value Decomposition (SVD) on the image matrix

[U, S, V] = svd(double(blurImage));

% Specify the percentage of singular values to retain

sharpnessFactor = 1.2;

% Compute the number of singular values to keep

numSingularValues = round(sharpnessFactor \* min(size(blurImage)));

% Set the singular values beyond the specified threshold to zero

S(numSingularValues+1:end, numSingularValues+1:end) = 0;

% Reconstruct the image using the modified singular values

sharpImage = U \* S \* V';

% Convert the sharp image to uint8 format (8-bit) and rescale the values

sharpImage = uint8(255 \* mat2gray(sharpImage));

% Display the sharp image

imshow(sharpImage);

title('Sharp Image');

**CODE EXPLAINATION:**

1. In this code first line assigns the denoised image to the variable blurImage.
2. Here, the SVD decomposition is performed on the image matrix blurImage. The resulting matrices U, S, and V represent the left singular vectors, singular values, and right singular vectors, respectively.
3. Next step sets the sharpnessFactor variable, which determines the percentage of singular values to keep. Adjusting this factor allows you to control the level of sharpening.
4. Next it calculates the number of singular values to retain based on the sharpnessFactor and the smaller dimension (width or height) of the image. The round function is used to ensure an integer value.
5. Here, the singular values beyond the specified threshold (determined by numSingularValues) are set to zero. This step effectively removes high-frequency components, reducing the amount of blurring in the image.
6. Next it reconstructs the image using the modified singular values. The matrices U, S, and V are multiplied together, and the result is assigned to sharpImage.
7. Here, the sharpImage is converted to uint8 format, which represents values in the range of 0 to 255 (8-bit). The mat2gray function is used to rescale the values between 0 and 1, and then multiplied by 255 to scale them to the appropriate range.
8. Finally, the sharpened image is displayed using the imshow function, and a title is assigned to the figure.

**CHAPTER 3**

**EDGE DETECTION USING SVD FUNCTION**

**EDGE DETECTION:**

Edge detection is a fundamental technique in image processing and computer vision that aims to identify and highlight the boundaries or edges of objects within an image. The edges represent significant changes in intensity, color, or texture in the image and often correspond to object boundaries or regions of interest.

The goal of edge detection is to distinguish between the foreground and background of an image based on the spatial variations in pixel intensity. By detecting edges, we can extract important features, such as contours, shapes, and textures, which can be used for further analysis or understanding of the image content.

Edge detection algorithms typically operate on grayscale or single-channel images, although they can be applied to color images by converting them to grayscale or by considering individual color channels separately.

**USING SVD:**

% Apply edge detection on the reconstructed image

edges = edge(reconstructedImage, 'log');

* The edge function returns a binary image where the detected edges are marked as white pixels and the rest of the image is black. This binary image is assigned to the variable edges.

**MATLAB CODE:**

% Load the image

image = imread('sample.jpeg');

% Convert the image to grayscale

grayImage = rgb2gray(image);

% Perform SVD on the grayscale image

[U, S, V] = svd(double(grayImage));

% Set the number of singular values to keep (adjust this as needed)

numSingularValues = 200;

% Reconstruct the image using the selected singular values

reconstructedImage = U(:, 1:numSingularValues) \* S(1:numSingularValues, 1:numSingularValues) \* V(:, 1:numSingularValues)';

% Apply edge detection on the reconstructed image

edges = edge(reconstructedImage, 'log');

% Display the original image and the detected edges

figure;

subplot(1, 2, 1);

imshow(grayImage);

title('Original Image');

subplot(1, 2, 2);

imshow(edges);

**CODE EXPLANATION:**

% Load the image

image = imread('sample.jpeg');

* This line loads an image named 'sample.jpeg' and stores it in the variable image. Make sure to replace 'sample.jpeg' with the actual filename and path of your image.

% Convert the image to grayscale

grayImage = rgb2gray(image);

* This line converts the loaded RGB image (image) to grayscale using the rgb2gray function. The resulting grayscale image is stored in the variable grayImage.

% Perform SVD on the grayscale image

[U, S, V] = svd(double(grayImage));

* This line applies Singular Value Decomposition (SVD) on the grayscale image (grayImage). SVD decomposes the image matrix into three separate matrices: U, S, and V, which represent the singular value decomposition of the image.

% Set the number of singular values to keep (adjust this as needed)

numSingularValues = 200;

* This line sets the number of singular values to retain during the reconstruction process. The variable numSingularValues is assigned a value of 200, indicating that the top 200 singular values will be used for reconstruction. You can modify this value to control the level of image reconstruction.

% Reconstruct the image using the selected singular values

reconstructedImage = U(:, 1:numSingularValues) \* S(1:numSingularValues, 1:numSingularValues) \* V(:, 1:numSingularValues)';

* This line reconstructs the image using the selected number of singular values. It multiplies the corresponding portions of matrices U, S, and V to obtain the reconstructed image matrix. The reconstructed image is stored in the variable reconstructedImage.

% Apply edge detection on the reconstructed image

edges = edge(reconstructedImage, 'log');

* This line performs edge detection on the reconstructed image (reconstructedImage) using the 'log' method. The edge function detects edges in the image and produces a binary image (edges) where edges are marked as white pixels.

% Display the original image and the detected edges

figure;

subplot(1, 2, 1);

imshow(grayImage);

title('Original Image');

subplot(1, 2, 2);

imshow(edges);

* These lines create a figure and display the original grayscale image (grayImage) and the detected edges (edges) side by side in a subplot arrangement. The imshow function is used to display the images, and the title function adds a title to the original image subplot.

**CHAPTER 4**

**IMAGE BLENDING USING FOURIER**

Image blending using Fourier transforms is a technique that combines two or more images by manipulating their frequency domain representations. The Fourier transform is a mathematical tool that decomposes a signal or image into its constituent frequencies.

Here's a general overview of the steps involved in blending images using Fourier transforms:

Load the source images: Begin by loading the images you want to blend together. Let's assume you have two images, Image A and Image B.

Convert images to grayscale: Convert both images to grayscale if they are not already in that format. This step ensures that the blending process focuses on luminance values rather than color.

Perform Fourier transform: Apply the Fourier transform to both grayscale images. The Fourier transform converts the images from the spatial domain to the frequency domain. This transformation represents the images as a sum of sine and cosine waves of different frequencies.

Combine frequency components: Take the frequency components (amplitudes and phases) from both Fourier transforms and blend them according to the desired blending ratio. The blending ratio determines how much of each image's frequency components should contribute to the final result. For example, if you want Image A to contribute more, you would use a higher blending ratio for its frequency components.

Inverse Fourier transform: Apply the inverse Fourier transform to the blended frequency components. This step converts the frequency domain representation back to the spatial domain, resulting in the blended image.

Adjust the pixel values: Depending on the range of pixel values in the resulting image, you may need to adjust them to fit within the valid range (e.g., 0-255 for an 8-bit image). This step ensures that the blended image appears visually appealing and avoids any artifacts caused by pixel value overflow.

Display or save the blended image: Finally, you can display the blended image or save it to a file for further use.

**MATLAB CODE**

% Read the source and destination images

sourceImage = imread('sample.jpeg');

destinationImage = imread('sample.jpeg');

% Convert the images to double precision for calculations

sourceImage = im2double(sourceImage);

destinationImage = im2double(destinationImage);

% Compute the size of the images

[height, width, ~] = size(sourceImage);

% Perform Fourier Transform on the images

sourceImageF = fft2(sourceImage);

destinationImageF = fft2(destinationImage);

% Define the blending mask

maskWidth = 50; % Width of the blending region

mask = zeros(height, width);

mask(:, 1:maskWidth) = 1;

% Perform image blending in the frequency domain

blendedImageF = sourceImageF .\* mask + destinationImageF .\* (1 - mask);

% Compute the inverse Fourier Transform to obtain the blended image

blendedImage = real(ifft2(blendedImageF));

% Display the blended image

imshow(blendedImage);

**CODE EXPLANATION:**

1. Reading the source and destination images:

The code reads two images: 'sample.jpeg' for both the source and destination images. These images are in jpeg format and iamread function is used to read image.

1. Converting the images to double precision:

To perform calculations accurately, the images are converted from their original data type to double precision using the im2double function.This ensures that the pixel values are represented as floating-point numbers ranging from 0.0 to 1.0.

1. Computing the size of the images:

The size of the source image is determined by extracting the height and width using the size function.

1. Performing Fourier Transform on the images:

The fft2 function is used to compute the two-dimensional Fast Fourier Transform (FFT) of the source and destination images. The result is stored in sourceImageF and destinationImageF, respectively. The FFT represents the images in the frequency domain, allowing for manipulation of their frequency components.

1. Defining the blending mask:

A blending mask is created using a rectangular shape. The maskWidth variable determines the width of the blending region.

1. Performing image blending in the frequency domain:

The blending is achieved by combining the frequency components of the source and destination images.

1. Computing the inverse Fourier Transform:

The ifft2 function is applied to the blendedImageF to obtain the inverse Fourier Transform, which brings the image back to the spatial domain. The result is stored in the blendedImage variable.

1. Displaying the blended image:

The imshow function is used to display the blended image on the screen.

**CHAPTER 5**

**Logic Implementation**

The concept of "Darkest line removal using watermarking in SVD" is not a well-known technique or standard procedure. It seems to be a specific approach that combines watermarking and Singular Value Decomposition (SVD) to remove the darkest line from an image. However, without further context or details about the specific algorithm or implementation you're referring to, it is difficult to provide a detailed explanation.

SVD, as mentioned earlier, is a mathematical technique for decomposing a matrix into three separate matrices: U, S, and V. It is often used for dimensionality reduction, noise reduction, image compression, and other applications. Watermarking, on the other hand, is a technique used to embed or extract information (such as a digital watermark) within digital media, including images.

Combining watermarking with SVD might involve modifying the singular values or vectors to embed or extract watermark information while preserving the quality of the image. However, the specific details and methods of this approach can vary depending on the particular watermarking algorithm and the desired goal of removing the darkest line.

**MATLAB CODE:**

%% Main code

im = blendedImage; % Input image

num\_hooks = 270; % Number of hooks around the circle

num\_chords = 2500; % Number of chords used in the weave

fade = 25/255; % Reduce the image pixels by this much along each chord

min\_distance = 20; % Minimum distance between each pair of hooks (pixels)

s = size(im);

if s(1) ~= s(2) % Make image square if not

im = im(1:min(s),1:min(s));

s = size(im);

end

imagesc(im), colormap gray, axis equal;

hold on

% Invert image so that dark pixels are scored highest

im = ones(s) - im;

imshow(im)

% Compute hook positions around a circle

angle = linspace(0, 2\*pi, num\_hooks);

rad = (s(1) - 1) / 2;

hook\_pos = rad \* [cos(angle); sin(angle)] + repmat(s'/2, 1, num\_hooks);

plot(hook\_pos(1,:), hook\_pos(2,:), '.r')

min\_dist\_sq = min\_distance \* min\_distance;

hook = ones(1,num\_chords);

% Pre-compute pixels between each pair of hooks

disp('Pre-computing chord pixels');

chord\_pixels = cell(num\_hooks);

for h1 = 1 : num\_hooks

for h2 = h1+1 : num\_hooks

dif = hook\_pos(:,h1) - hook\_pos(:,h2);

% Ignore hook pairs that are close together

if dif'\*dif > min\_dist\_sq

% Compute pixels that lie along chord between hook pair

max\_dim = round(max(abs(dif)));

x = round(linspace(hook\_pos(1,h1), hook\_pos(1,h2), max\_dim));

y = round(linspace(hook\_pos(2,h1), hook\_pos(2,h2), max\_dim));

chord\_pixels{h1,h2} = uint32(sub2ind(s,y,x));

end

end

end

i\_prev = 1;

% Loop over chords

for i = 1 : num\_chords - 1

best\_score = -1000000;

best\_hook = 1;

% Loop over possible hooks

for hook\_new = 1 : num\_hooks

% Sort hook pair into ascending order for pixels lookup

h = sort([hook(i) hook\_new]);

if ~isempty(chord\_pixels{h(1),h(2)})

% Compute score based on mean pixel values along chord

score = mean(im(chord\_pixels{h(1),h(2)}));

if score > best\_score

best\_score = score;

best\_hook = hook\_new;

end

end

end

% Reduce pixel values along best chord

h = sort([hook(i) best\_hook]);

ind = chord\_pixels{h(1),h(2)};

im(ind) = im(ind) - fade;

% Clear list for this pair to prevent re-use

chord\_pixels{h(1),h(2)} = [];

% Store as next hook

hook(i+1) = best\_hook;

end

disp(hook)

**CODE EXPLANATION:**

This line assigns the variable im to the input image blendedImage. The input image is the image on which the weaving effect will be applied.

These lines define the parameters for the weaving process. num\_hooks specifies the number of hooks that will be placed around the circle, num\_chords determines the number of chords used in the weave, fade represents the amount by which the image pixels will be reduced along each chord, and min\_distance sets the minimum distance between each pair of hooks.

Here, the size of the input image im is determined using the size function, and the result is assigned to s. The subsequent if condition checks if the image is not square (i.e., the number of rows is not equal to the number of columns). If the image is not square, it is cropped to a square shape by taking the minimum dimension, and the updated size is stored in s.

This line displays the image im using imagesc, which creates a scaled image plot. The colormap gray sets the color scheme to grayscale, and axis equal ensures that the aspect ratio of the plot is equal. The hold on command allows subsequent plotting commands to be overlaid on the existing plot.

In this section, the image im is inverted by subtracting it from a matrix of ones (ones(s)). This inversion is done to change the interpretation of pixel values, so that dark pixels become light and vice versa. The inverted image is then displayed using imshow.

These lines compute the positions of the hooks around a circle. The angle variable defines the angles at which the hooks will be placed around the circle. The rad variable calculates the radius of the circle based on the image size. The hook\_pos matrix is then computed by multiplying the radius by the [cos(angle); sin(angle)] matrix and adding the center of the image to each coordinate. Finally, the positions of the hooks are plotted using plot with red dots.

These lines initialize the min\_dist\_sq variable as the square of the minimum distance between each pair of hooks. The hook array is initialized with ones and will store the indices of the chosen hooks for each chord in the weaving process.

This line displays a message indicating that the code is about to pre-compute chord pixels. It serves as a notification to the user about the progress of the computation.

This line initializes the chord\_pixels cell array, which will store the pixel indices that lie along the chords between different pairs of hooks.

This block of code pre-computes the pixels that lie along the chord between each pair of hooks. It iterates over all possible combinations of h1 and h2 (the indices of the hooks) and calculates the difference (dif) between their positions. If the square of the Euclidean distance between the hooks exceeds min\_dist\_sq, indicating that they are not too close, the code computes the x and y pixel coordinates using linspace to generate a sequence of values along the chord. These coordinates are then converted to linear indices using sub2ind and stored in the chord\_pixels cell array.

This loop iterates over each chord in the weaving process. The variable i represents the current chord number.

These lines initialize the best\_score variable with a very low value and best\_hook with the index of the first hook. These variables will be used to track the highest score and the best hook for each chord.

This loop iterates over possible hooks (hook\_new) and computes the score for each hook pair. The code sorts the hook pair (hook(i) and hook\_new) in ascending order to ensure consistent indexing for the chord\_pixels cell array. If the chord\_pixels array is not empty for the current hook pair, the code computes the score based on the mean pixel values along the chord using mean(im(chord\_pixels{h(1),h(2)})). If the score is higher than the previous best score, the best\_score and best\_hook variables are updated.

After finding the best hook for the current chord, this code reduces the pixel values along the best chord by subtracting the fade value from the corresponding indices in the im image. The chord\_pixels cell array for the current hook pair is cleared to prevent its re-use. Finally, the index of the best hook for the current chord is stored in the hook array.

This line displays the hook array, which contains the indices of the chosen hooks for each chord in the weaving process.

**INPUT**



**OUTPUT**

**OUTPUT**

