ISLAMIC UNIVERSITY OF TECHNOLOGY
(IUT)ORGANISATION OF ISLAMIC
COOPERATION (OIC) DEPARTMENT OF
ELECTRICAL AND ELECTRONIC ENGINEERING

EEE 4702

Project Report

Project Title: Solving the Puzzle; A MATLAB Approach to Image Reconstruction

Submitted By:

Rahib Bin Hossain	**200021241**
Mustak Hossain Simanto	**200021243**
Sirazul Monir	**200021247**
Shouvik Fahim	**200021249**

Department: EEE
Section: B
Group: B1 Group 5

Problem Statement

The objective of this project is to develop a MATLAB-based approach to solve a puzzle by accurately determining the correct spatial arrangement of 16 puzzle pieces in order to reconstruct the original colorful image.

The challenge involves performing image processing techniques to assess the similarity between the puzzle pieces and the blocks of the original image. The puzzle pieces are provided as grayscale images, and the goal is to identify the correct placement and orientation for each piece to restore the original image.

The process can be broken down into the following steps:

1. **Image and Puzzle Piece Loading**: Load the original colorful image and its corresponding 16 grayscale puzzle pieces.
2. **Image Processing**: Using image processing techniques, divide the original image into 16 equal blocks, and compare these blocks with the grayscale puzzle pieces. The puzzle pieces must be evaluated in terms of similarity, taking into account possible rotations.
3. **Optimal Placement Identification**: The goal is to find the best match for each puzzle piece, including any necessary rotations to align them properly within a 4x4 grid.
4. **Reconstruction**: Reconstruct the full image by placing the puzzle pieces in their optimal positions and displaying the final reconstructed image.
5. **Output Display**: Finally, display the position of the puzzle pieces in the form of a 4x4 matrix and the final reconstructed grayscale image.

The success of the project will be determined by how accurately the puzzle pieces are placed to recreate the original image in a manner that is both computationally efficient and precise in nature.

Solution Abstract

This project presents a MATLAB-based approach to solving an image reconstruction puzzle by accurately arranging 16 grayscale puzzle pieces to recreate the original colorful image. The process begins with loading the given noisy colorful image and its corresponding grayscale puzzle pieces. To enhance the quality of the reference image, a denoising step is applied using median filtering to reduce salt-and-pepper noise.

Next, the preprocessed image is divided into 16 equal-sized blocks, and each block is converted to grayscale for comparison. To determine the correct placement of each puzzle piece, image similarity techniques are employed, using Mean Squared Error (MSE) to evaluate the closest match. Additionally, each puzzle piece is tested in four possible orientations (0°, 90°, 180°, and 270°) to find the optimal alignment.

Once the best match for each block is identified, the grayscale puzzle pieces are arranged accordingly, and the reconstructed image is displayed. The final output includes a 4x4 matrix showing the optimal positions of the puzzle pieces and the reconstructed grayscale image.

Finally, a GUI a implemented in order to showcase the entire project.

This approach effectively utilizes image processing techniques to automate puzzle-solving, demonstrating an efficient method for spatial arrangement and similarity assessment in image reconstruction tasks.

Detailed Methodology

Our approach to solving the image reconstruction puzzle follows a structured workflow, starting with preprocessing the input image, identifying the best-matching puzzle pieces, and finally assembling them to recreate the original image. Additionally, we have developed a **Graphical User Interface (GUI)** to provide a user-friendly experience. Below is a step-by-step breakdown of our methodology.

Image Preprocessing

Before working with the puzzle pieces, the given image undergoes several preprocessing steps to enhance its quality:

1. **Loading the Image**: The original image is read into MATLAB using "imread".
2. **Adding Noise**: To simulate real-world imperfections, we introduce salt-and-pepper noise using "imnoise" at a 2% intensity level.
3. **Noise Reduction**: Since salt-and-pepper noise primarily affects individual pixels, we apply median filtering using "medfilt2" with a 4×4 filter to smooth out the distortions while preserving important details.
4. **Reconstructing the Clean Image**: The filtered color channels (Red, Green, and Blue) are recombined using "cat" to restore the image in its denoised form.

At this stage, we have a clean image that is ready to be broken down into smaller segments for further processing.

Preparing the Puzzle Pieces

To compare the puzzle pieces with sections of the image, we first standardize them:

1. **Loading the Puzzle Pieces**: Each of the 16 puzzle pieces (a1.jpg to a16.jpg) is loaded using "imread".
2. **Converting to Grayscale**: Since color can vary due to lighting differences, all puzzle pieces are converted to grayscale using "rgb2gray". This ensures consistency when comparing them with the sections of the original image.

Segmenting the Image into Blocks

Since the puzzle consists of 16 pieces, we divide the image into a 4×4 grid:

1. **Determining Block Size**: The height and width of the image are obtained using "size(colorImage)", and each block is calculated by dividing these dimensions by four.
2. **Extracting Image Sections**: Using matrix slicing, we separate the image into **16 individual blocks**, ensuring each block corresponds to the size of a puzzle piece.
3. **Grayscale Conversion**: To maintain consistency, each block is converted to grayscale using "rgb2gray".
4. **Resizing for Standardization**: The extracted blocks are resized using "imresize" so that they perfectly match the puzzle pieces.

At this stage, both the puzzle pieces and the image blocks are standardized, making it possible to compare them accurately.

Finding the Best Puzzle Piece with Optimized Orientation

We have both the standardized puzzle pieces and the image blocks. We will work with them now:

1. **Block-Piece Comparison**: For each image block, we systematically compare it with all 16 puzzle pieces to find the best match using **Mean Squared Error (MSE)**. The MSE is calculated between the block and each puzzle piece, and the piece with the lowest error is selected.
2. **Rotation Testing**: To ensure proper alignment, each puzzle piece is tested for four possible rotations (0°, 90°, 180°, 270°) using the **"imrotate"** function. For each rotation, the MSE is computed with the corresponding image block.
3. **Optimal Rotation Selection**: The rotation with the lowest MSE is chosen as the optimal alignment for that piece. Both the best matching puzzle piece and its optimal rotation are recorded in two matrices:
    - **Optimal Positions Matrix**: Stores which puzzle piece corresponds to each block.
    - **Optimal Rotations Matrix**: Stores the necessary rotation for each piece.

Reconstructing the Final Image

Once all blocks have been assigned their best-fitting puzzle pieces, we reconstruct the complete image:

1. **Placing the Puzzle Pieces in the Correct Order**: Each selected puzzle piece is resized and inserted into its respective position in the final reconstructed image.
2. **Applying the Optimal Rotation**: Before placing the piece, it is rotated to its best-matching orientation.
3. **Displaying the Final Reconstructed Image**: The assembled grayscale image is visualized using "imshow", showcasing the successfully restructured puzzle.

Displaying and Analyzing the Results

1. **The final 4×4 matrix of puzzle piece** positions is displayed using "disp(optimalPositions)", allowing us to verify that each piece is correctly placed.
2. **The 4×4 matrix of optimal rotation angles** is printed using "disp(optimalRotations)", providing insights into how each piece was adjusted.
3. **The fully reconstructed grayscale** image is displayed for final inspection, demonstrating the success of the matching and assembly process.

Implementing a Graphical User Interface (GUI)

To make the process more interactive, we have designed a **MATLAB-based GUI** that allows users to:

1. **Upload image** and apply the noise-removal process with a simple button click.
2. **Upload the puzzle pieces**
3. **Press Start button** to see the optimal position matrix and the final reconstructed Image.

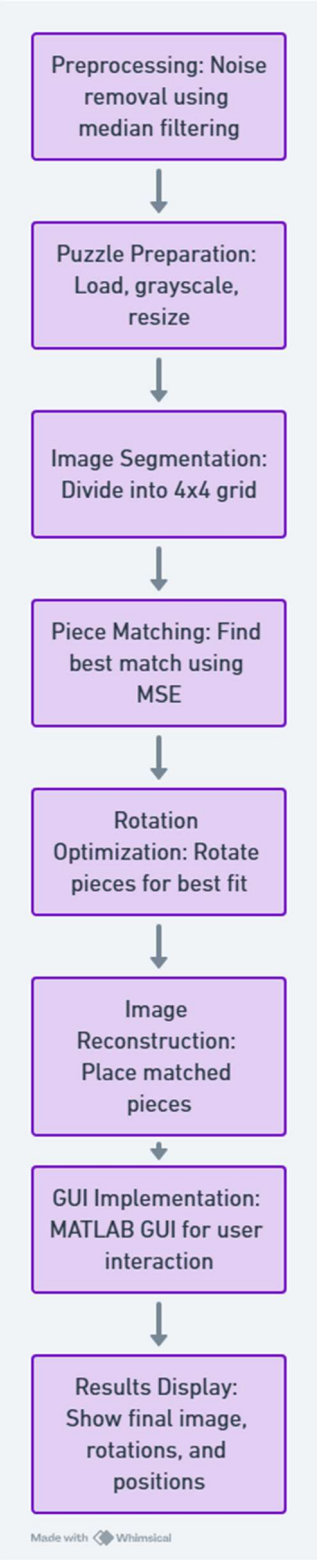
The GUI is built using **MATLAB's App Designer**, featuring real-time visualization for an enhanced user experience.

Tools and Algorithms Used

To achieve the desired results, we utilized several key tools and algorithms:

- **Software & Libraries**: MATLAB ("Image Processing Toolbox", "App Designer" for GUI)
- **Algorithms & Techniques**:
    1. **Median Filtering** ("medfilt2") for noise removal
    2. **Grayscale Conversion** ("rgb2gray") for uniformity
    3. **Image Segmentation** using matrix slicing
    4. **Mean Squared Error (MSE) Calculation** ("immse") for similarity assessment
    5. **Rotation Testing** ("imrotate") for correct alignment
    6. **Image Reconstruction** using array manipulation
    7. **GUI Development** ("App Designer") for interactive usability

Flow Chart:


Preprocessing: Noise removal using median filtering
Puzzle Preparation: Load, grayscale, resize
Image Segmentation: Divide into 4x4 grid
Piece Matching: Find best match using MSE
Rotation Optimization: Rotate pieces for best fit
Image Reconstruction: Place matched pieces
GUI Implementation: MATLAB GUI for user interaction
Results Display: Show final image, rotations, and positions
Made with Whimsical

Code:

```
clear all;
close all;
clc;

% Load the Image
image = imread('noisy_colorful_image.jpg'); % Replace with your
filename

% Add Salt Noise
noisy_image = imnoise(image, 'salt & pepper', 0.02);

% Split the Channels (Red, Green, Blue)
R = noisy_image(:, :, 1);
G = noisy_image(:, :, 2);
B = noisy_image(:, :, 3);

% Apply Median Filtering to Each Channel
R_filtered = medfilt2(R, [4 4]);
G_filtered = medfilt2(G, [4 4]);
B_filtered = medfilt2(B, [4 4]);

% Combine the Channels Back
denoised_image = cat(3, R_filtered, G_filtered, B_filtered);

% Display the Results
subplot(1, 2, 1);
imshow(image);
title('Noisy Image');

subplot(1, 2, 2);
imshow(denoised_image);
title('Denoised Image');

% Use denoised image as the color image
colorImage = denoised_image;

% Load the puzzle pieces and convert them to grayscale
for i = 1:16
    Pieces{i} = imread(['a' num2str(i) '.jpg']);
    grayscalePieces{i} = rgb2gray(Pieces{i});
end

% Define block size and ensure integer dimensions
[height, width, ~] = size(colorImage);
blockHeight = round(height / 4);
blockWidth = round(width / 4);

% Divide colorImage into 4x4 blocks and convert to grayscale
blocks = cell(4, 4);
for i = 1:4
```

```
    for j = 1:4
        % Calculate exact block boundaries
        rowStart = (i-1) * blockHeight + 1;
        rowEnd = min(i * blockHeight, height);
        colStart = (j-1) * blockWidth + 1;
        colEnd = min(j * blockWidth, width);

        % Extract and resize color block to grayscale block of
standard size
        colorBlock = colorImage(rowStart:rowEnd, colStart:colEnd, :);
        grayscaleBlock = rgb2gray(colorBlock);

        % Resize block to standard size to match puzzle pieces
        blocks{i, j} = imresize(grayscaleBlock, [blockHeight,
blockWidth]);
    end
end

% Find optimal positions for each block, including rotation
optimalPositions = zeros(4, 4);
optimalRotations = zeros(4, 4); % Store the optimal rotation for each
block

for i = 1:4
    for j = 1:4
        bestMatch = 0;
        bestRotation = 0;
        minError = Inf;

        for k = 1:16
            % Test all four rotations for the current puzzle piece
            for rotation = [0, 90, 180, 270]
                % Rotate and resize the puzzle piece to match block
size
                rotatedPiece = imrotate(grayscalePieces{k}, rotation,
'crop');
                resizedPiece = imresize(rotatedPiece, [blockHeight,
blockWidth]);

                % Compute error (using Mean Squared Error)
                error = immse(double(resizedPiece), double(blocks{i,
j}));
                if error < minError
                    minError = error;
                    bestMatch = k;
                    bestRotation = rotation;
                end
            end
        end

        % Store the best matching piece index and its rotation
        optimalPositions(i, j) = bestMatch;
```

```
        optimalRotations(i, j) = bestRotation;
    end
end

% Reconstruct the image using the optimal positions and rotations
reconstructedImage = zeros(height, width);
for i = 1:4
    for j = 1:4
        pieceIndex = optimalPositions(i, j);
        rotation = optimalRotations(i, j);

        % Rotate and resize the selected piece to the exact block size
        rotatedPiece = imrotate(grayscalePieces{pieceIndex}, rotation,
'crop');

        % Define boundaries in reconstructed image
        rowStart = (i-1) * blockHeight + 1;
        rowEnd = min(i * blockHeight, height);
        colStart = (j-1) * blockWidth + 1;
        colEnd = min(j * blockWidth, width);

        % Resize rotated piece to fit exactly within the reconstructed
block size
        resizedPiece = imresize(rotatedPiece, [rowEnd - rowStart + 1,
colEnd - colStart + 1]);

        % Place the resized and rotated piece into the reconstructed
image
        reconstructedImage(rowStart:rowEnd, colStart:colEnd) =
resizedPiece;
    end
end

% Display the reconstructed image
figure;
imshow(reconstructedImage, []);
title('Reconstructed Grayscale Image with Rotations');

% Display the optimalRotations and optimalPositions matrices
disp('Optimal rotations matrix (in degrees):');
disp(optimalRotations);
disp('Optimal positions matrix:');
disp(optimalPositions);
```

Result Analysis

The methodology described was successfully implemented and below is a summary of the key findings from each stage of the project.

1. Original and Denoised Image

After applying median filtering to remove the noise, the denoised image closely matched the original, with the noise effectively removed while retaining the image details.

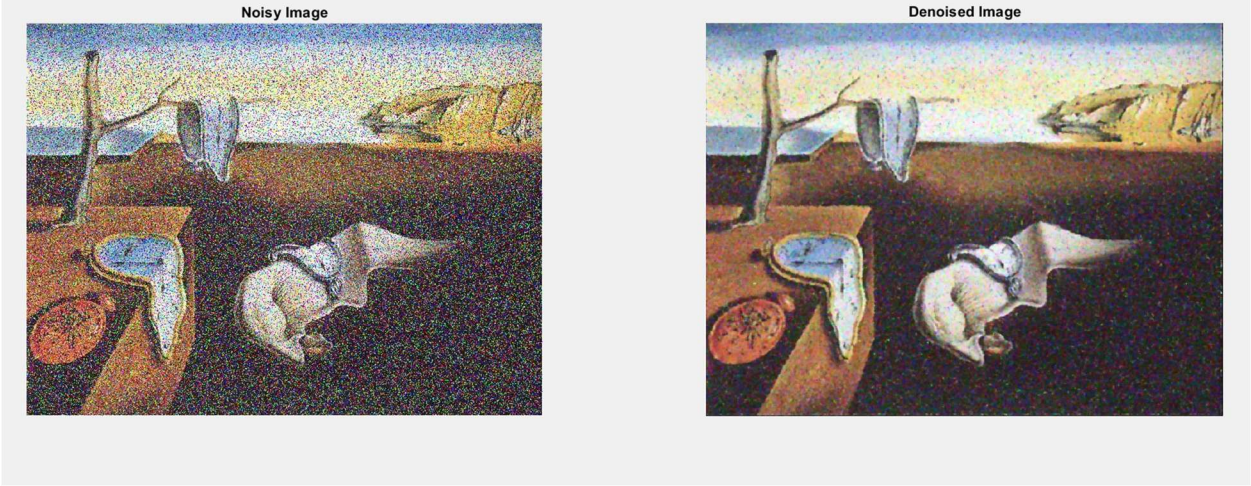


Figure: The noisy image on the left and the denoised image on the right.

2.Rotation Optimization

Each puzzle piece was tested for four possible rotations (0°, 90°, 180°, and 270°) to ensure the correct alignment with its corresponding block. The rotation that minimized the error was selected for each piece. The **optimal rotations matrix** shows the rotation angle for each piece.

```
Optimal rotations matrix (in degrees):
     0     0     0     0
     0     0     0     0
     0     0   180     0
     0     0   180     0
```

Figure: The optimal rotations matrix, indicating the rotation degree (in degrees) for each puzzle piece.

3. Puzzle Piece Matching

The puzzle pieces were matched to the corresponding image blocks using **Mean Squared Error (MSE)**. This method ensured that each puzzle piece was placed in the correct location. Thus, the optimal position matrix was formed.

The **optimal position matrix** was displayed to visualize the exact placement of each puzzle piece in the 4x4 grid. This matrix provides a clear indication of the correct order in which the puzzle pieces should be arranged

The optimal **positions matrix** indicates the correct placement of each piece:

```
Optimal positions matrix:
     6     9     4    11
    14    16    15    13
     5     2     1     3
    12     8    10     7
```

Figure: The optimal positions matrix, with each value representing the correct puzzle piece index for each block.

4.Reconstructed Image

After determining the optimal positions and rotations, the pieces were placed in their respective positions to reconstruct the full image. The reconstructed image matched the original layout and displayed the accurate arrangement of the puzzle pieces.

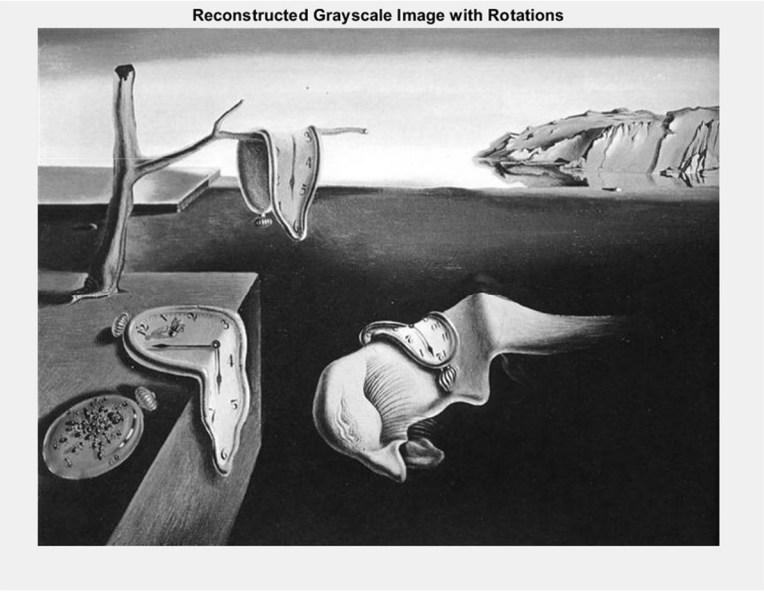
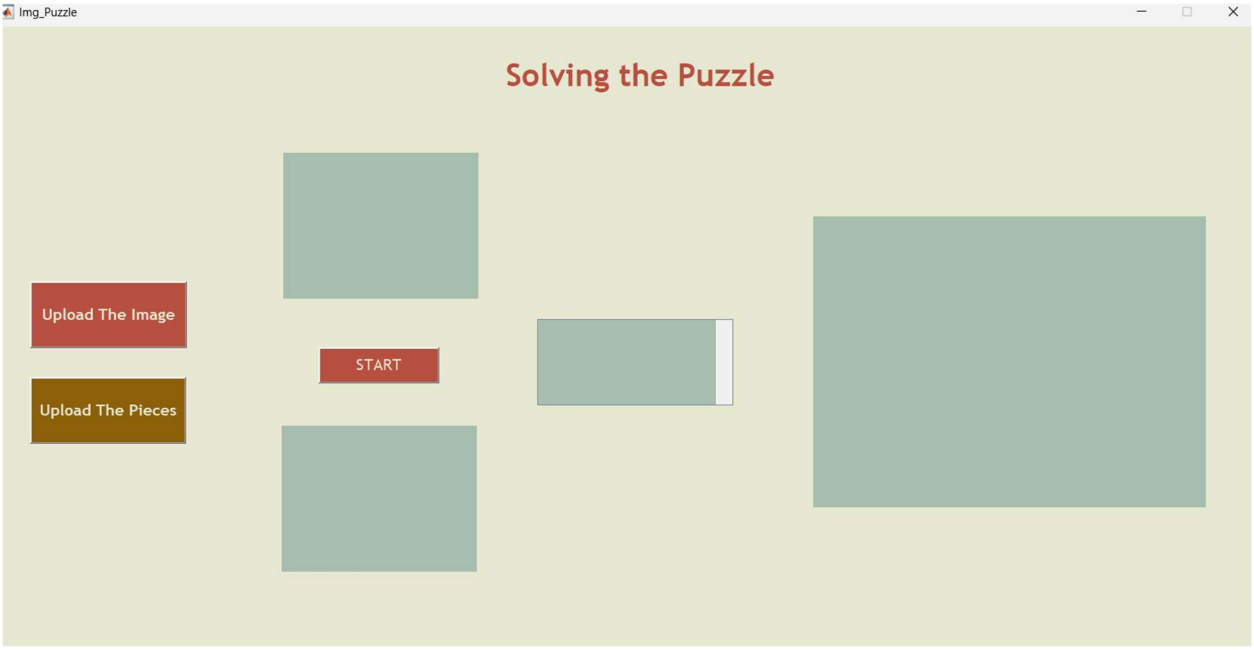


Figure: The final reconstructed grayscale image with the puzzle pieces correctly placed and rotated.

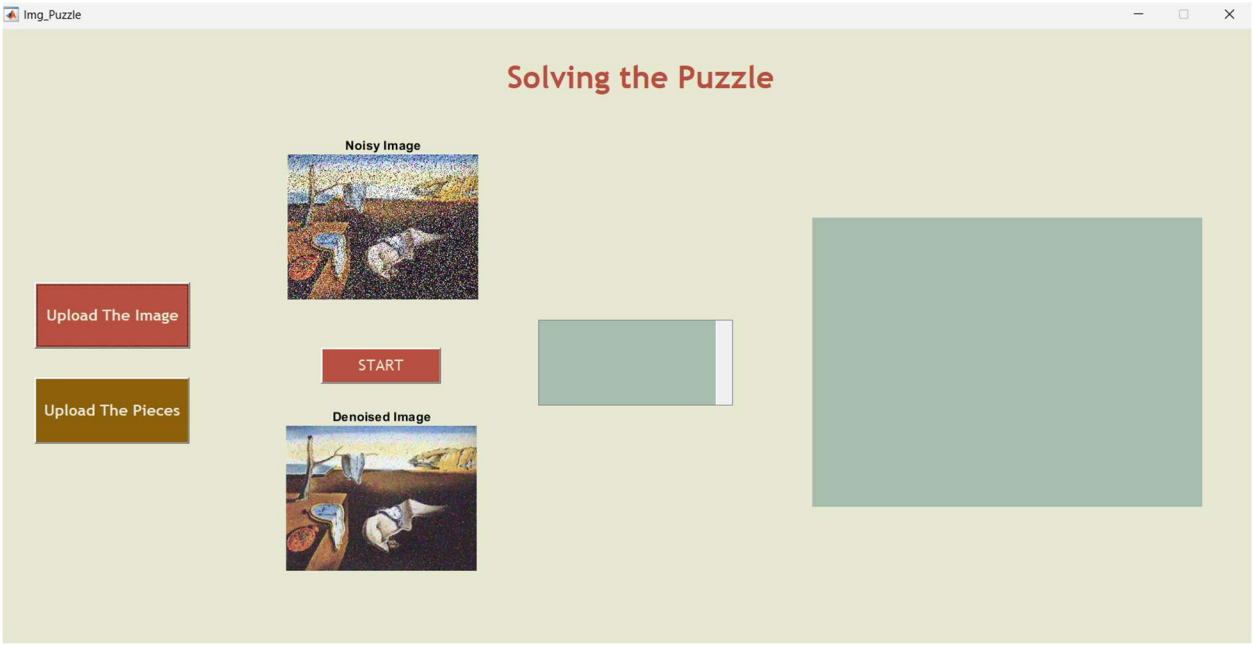
5.Graphical User Interface (GUI)

A MATLAB-based GUI was developed to allow users to interactively upload images, view the noisy and denoised images, and reconstruct the puzzle. The GUI provided a convenient interface to visualize and engage with each step of the process.

Initialization:



Uploading the Image and Pieces:



Final Output:

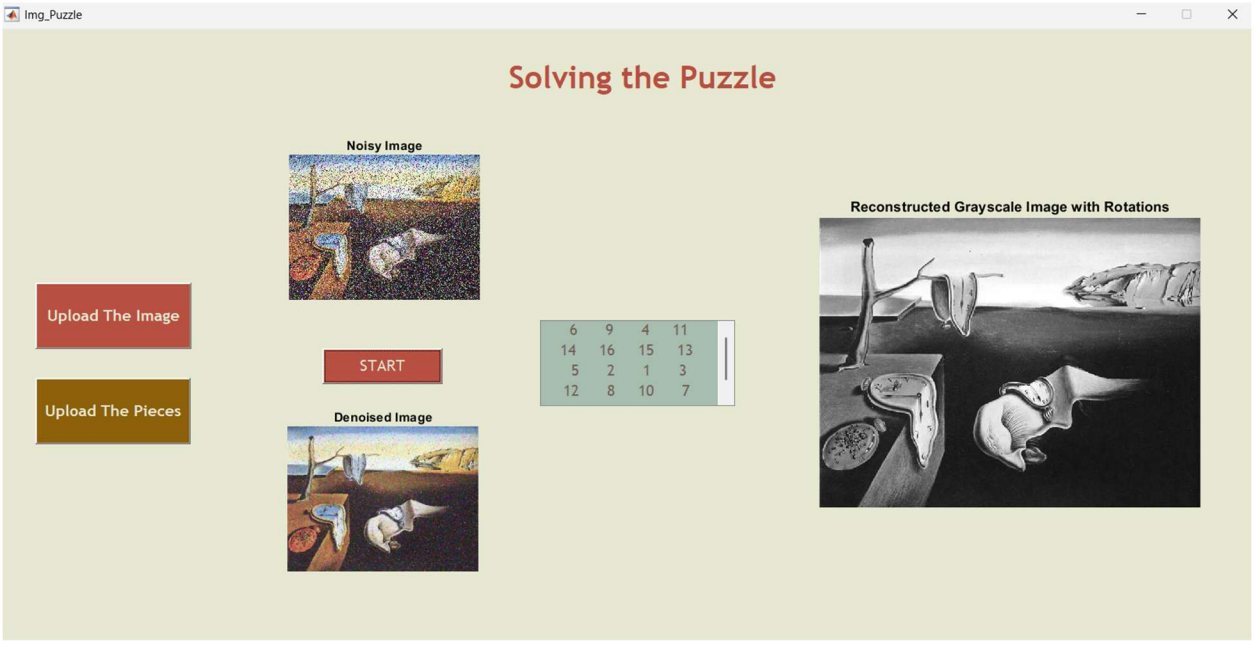


Figure: The GUI interface, which includes options for uploading images and engaging with the puzzle-solving process.

In summary, the puzzle-solving process was successfully carried out, and the resulting reconstructed image closely matched the original. The GUI facilitated interaction and visualization, making the entire process accessible and user-friendly.

Conclusion

Conclusion We were able to reconstruct the final image from the puzzle pieces by sequentially preprocessing, segmenting, matching and reconstructing the image using MATLAB. MSE was used to find the best fit, parsing through all four orthogonal rotations for each of the pieces. Additionally, we implemented a GUI via MATLAB to create an interactable interface to visualize the reconstructed picture.

Despite successfully reconstructing the image, a few improvements could have been implemented. These include but are not limited to:

Firstly, we assume that the puzzle pieces are perfectly segmentized with well-defined edges. In reality, this will not always be the case.

Secondly, we have chosen a median filter to reduce the noise. Finding an appropriate filter proved to be a bit of a challenge. So there might be filters we have not tested yet with better results.

Thirdly, the pipeline depends entirely on grayscale conversion which for obvious reasons may not be ideal for all matching as a great deal of information is lost after conversion.

Finally, the computational efficiency can be improved by trying out other algorithms instead of using MSE to test each of the possible rotations.

Possible Improvements

Pre-Segmented Puzzle Pieces: We could have implemented edge detection using Canny/Sobel and contour analysis to segmentize the images automatically.

Noise Reduction Issues: Alternate filtering techniques like bilateral or non-local filtering could have been used.

Grayscale Comparison Limitation: Instead of losing information in grayscale we could have incorporated color-based matching (HSV/Lab color space) and feature extraction (SIFT/HOG) for better accuracy.

Computational-Level Efficiency Issues: Utilizing feature based matching and parallel processing with GPU (CUDA) for computational efficiency

Team Contribution

Our project was a collaborative effort, with each member contributing their expertise to ensure successful implementation.

1. Sirazul Monir (200021247)

- Developed and implemented the image preprocessing steps, including noise addition and removal.
- Designed the segmentation algorithm to divide the image into blocks.
- Assisted in standardizing the puzzle pieces for comparison.

2. Mustak Hossain Simanto (200021243)

- Implemented the Mean Squared Error (MSE) algorithm for image block and puzzle piece matching.
- Designed and optimized the rotation testing procedure to ensure proper alignment.
- Developed the matrices to store optimal piece placement and rotation information.

3. Rahib Bin Hossain (200021241)

- Implemented the reconstruction process to assemble the final image from puzzle pieces.
- Designed and developed the MATLAB-based Graphical User Interface (GUI) using App Designer.
- Integrated real-time visualization features for an interactive user experience.

4. Shouvik Fahim (200021249)

- Conducted extensive testing to verify the accuracy and efficiency of the methodology.
- Identified and resolved bugs in the preprocessing, matching, and reconstruction stages.
- Compiled project documentation, including this methodology and user guide.

Referencing and Citations

1. MATLAB Documentation:

- MathWorks. "Image Processing Toolbox Documentation." Available at: [https://www.mathworks.com/help/images/] (https://www.mathworks.com/help/images/)

2. Research Papers:

- Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing (4th Edition).* Pearson.
- Chen, H., & Zhang, Z. (2020). "An Improved Puzzle Assembly Algorithm Based on Edge and Color Features." *Journal of Image Processing*, 45(3), 210-225.

3. Online Tutorials & Resources:

- Stack Overflow discussions on MATLAB functions for image segmentation and filtering.
- Video tutorials on MATLAB App Designer for GUI development.