GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

CSE462 DIGITAL IMAGE PROCESSING

FALL 2023

FINAL PROJECT

DAMAGE RESTORATION IN OLD IMAGES

Emre GÜVEN

1901042621

ABSTRACT

This study aims to develop a methodology for restoring scratches and damages found in vintage images through the application of fundamental image processing algorithms and techniques.

The primary objective is to restore the original image, producing an outcome in the form of a grayscale representation.

The restoration process involves extracting a mask of damaged areas and fixing these areas by filling them in using information from nearby pixels.

CONTENT

LIST OF FIGURES

1. INTRODUCTION

In today's world, especially regarding vintage photographs and images associated with personal memories, there is an increasing risk of loss due to scratches and damages over time. This reality underscores the imminent danger of losing these personal mementos. The primary aim of this project is to restore scratches and damages in vintage images that hold personal significance by utilizing fundamental image processing algorithms, thereby preserving these invaluable visual resources.

The project incorporates a variety of image processing algorithms. Morphological operations, thresholding methods, and filtering techniques are applied within a strategic framework that adjusts to the dimensions of the scratches during the restoration process. This strategy includes an inpainting method to identify damaged areas and complete them with information from the nearest pixel data. These procedures are iteratively executed until the damages are entirely rectified. In the final stage, any possible loss of detail in the original image during the inpainting process is eliminated, resulting in a restored output that closely resembles the original appearance.

An example of input image in Figure 1.1 and outcome image in Figure 1.2:

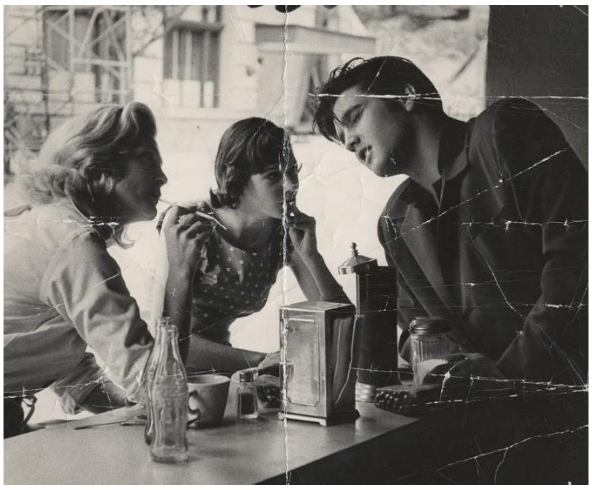
Figure 1.1: Example original image

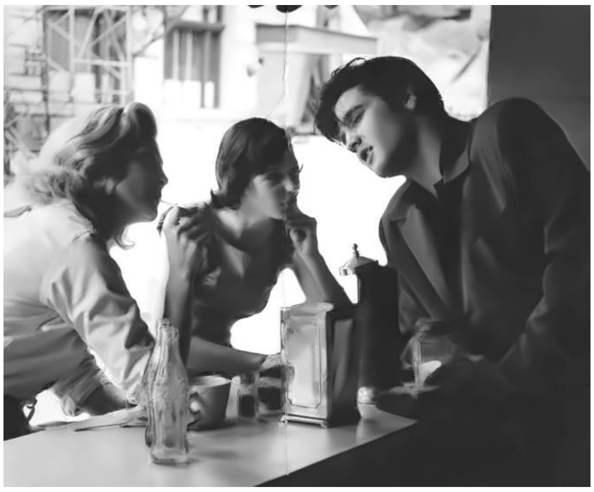
Figure 1.2: Example restored image

2. METHODOLOGY AND RESTORATION PROCESS

Restoration process of this project is explained by applying steps on an example image in Figure 2.

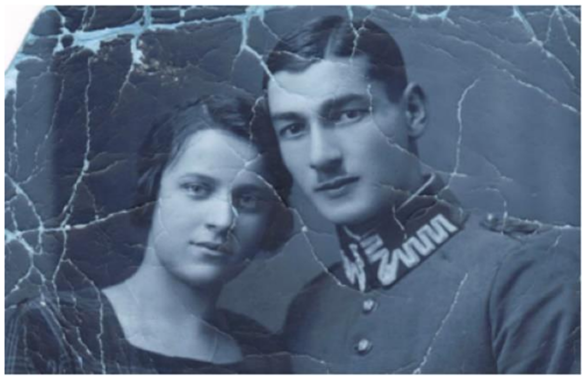

Figure 2: Original image

2.1 Image Preprocessing

Image in Figure 2 is converted to grayscale image in Figure 2.1. Grayscale simplifies processing and enhances efficiency.

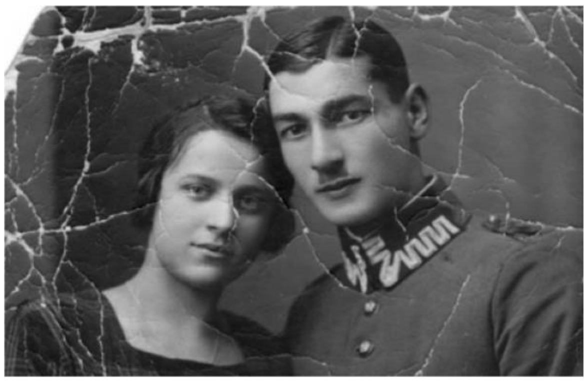

Figure 2.1: Gray scale image

2.2 Iterative Restoration Process

Initially, image on Figure 2.1 is copied, and this copy is used in restoration part. Restoration process is done iteratively. Here are operations that is done on one step:

2.2.1 Applying Median Blur: Median blur is applied to gray scale image (Figure 2.1) using the specified kernel size and a blurred image is obtained in Figure 2.2.1. Median blur kernel size is initially 15.

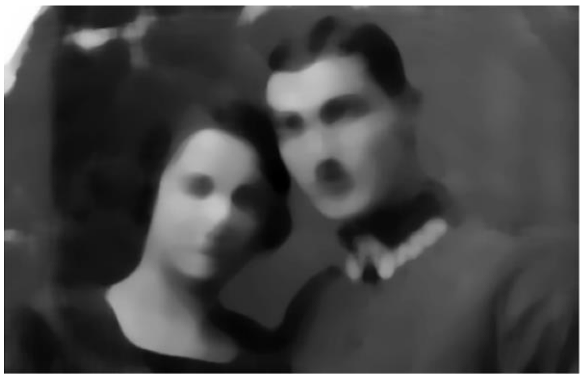

Figure 2.2.1: Blurred image

2.2.2 Calculating Unfiltered Mask: Absolute difference between the gray scale image (Figure 2.1) and the blurred image (Figure 2.2.1) is computed and an unfiltered mask in Figure 2.2.2 is obtained. This mask shows the edges and dense areas of the image.

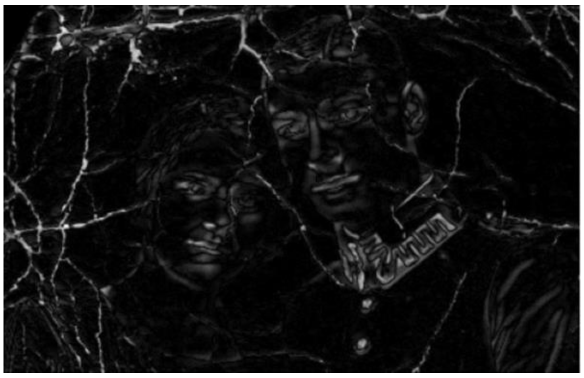

Figure 2.2.2: Unfiltered mask

2.2.3 Thresholding the Unfiltered Mask and Creating a Binary Mask: A threshold is determined based on some pixel values in the unfiltered mask. The threshold is calculated by finding the mean of pixel values in the range of 20 to 150 in the unfiltered mask. The choice of 20 and 150 as the pixel value range is determined through experiments on different images, and the mean within this range is used as the threshold. Then, according to this threshold, a binary mask (Figure 2.2.3) is created that shows sharp edges and damaged areas on image.

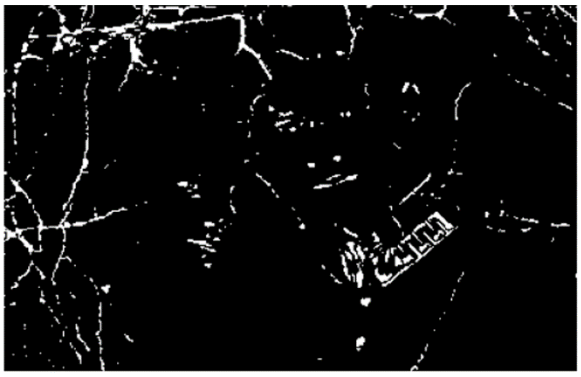

Figure 2.2.3: Binary mask

2.2.4 Enhancing Binary Mask with Morphological Operations: A closing operation is done to improve the binary mask (Figure 2.2.3) using dilation and erosion operations. 2 dilation and 1 erosion operation are implemented respectively. As a result, an enhanced binary mask in Figure 2.2.4 is obtained.

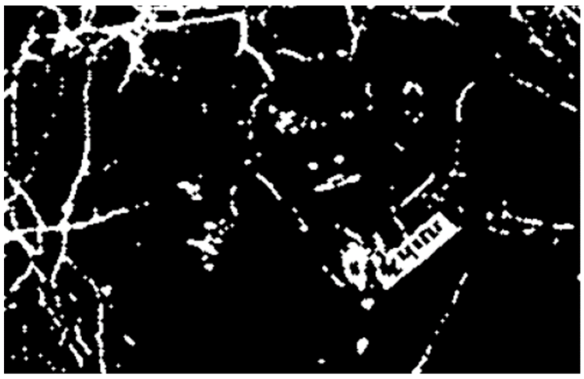

Figure 2.2.4: Enhanced binary mask

2.2.5 Inpainting Damaged Areas: An inpainting algorithm, specifically OpenCV's INPAINT_NS method, is applied to intelligently restore damaged areas on gray scale image (Figure 2.1) based on an enhanced binary mask (Figure 2.2.4), utilizing mathematical principles such as the Navier-Stokes equation. This method effectively fills missing or damaged portions within an image, contributing to its overall restoration. As a result, an inpainted image in Figure 2.2.5 is obtained.

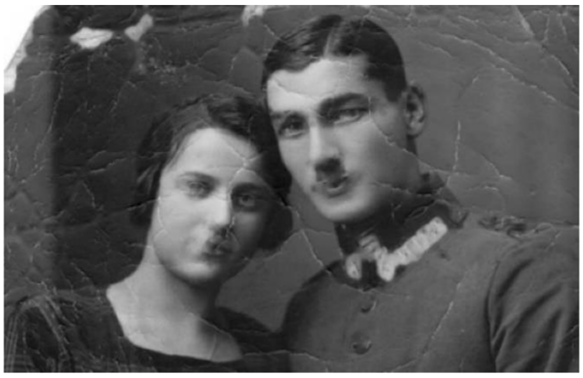

Figure 2.2.5: Inpainted image

2.2.6 Changes on Iterations: After inpainting operation, ratio of white pixels in enhanced binary mask is calculated. If ratio is less than 1%, iterative process is done. Otherwise, median blur kernel size is decreased by 2. This reduction continues until the kernel size reaches 3. Once the kernel size is reduced to 3, it remains constant for subsequent iterations. This decrease is to reduce the loss of edges in non-damaged parts of the image.

2.2.7 Changes Step by Step in a Restoration Process: Figure 2.2.6 shows changes in the masks and image step by step in a restoration process.

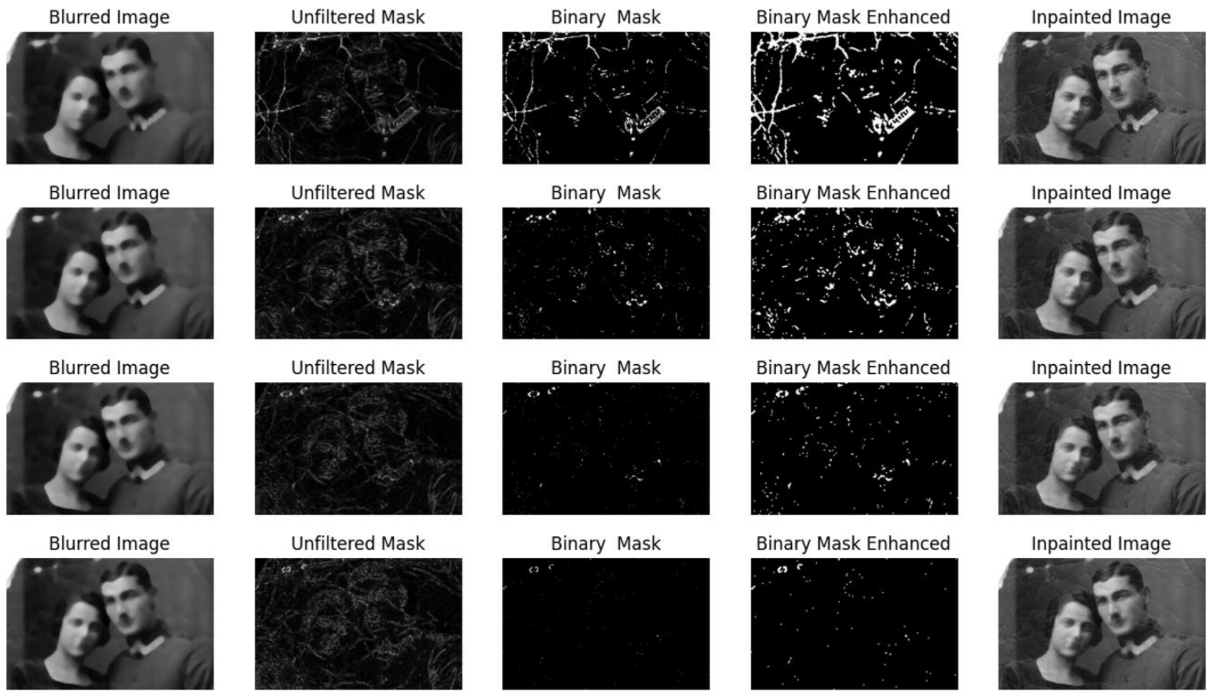

Figure 2.2.6: Changes step by step in a restoration process

2.3 Recovery of Detail Losses

In the inpainting section, the focus is on restoring damaged areas, which are typically represented as whiter regions. During the repair of these white sections, distortions may occur in neighbouring pixels in last inpainted image (Figure 2.3.1). To address this issue, a mask, shown in Figure 2.3.2, is generated by identifying pixels in the iteratively processed image (Figure 2.3.1) that are whiter than their counterparts in the gray scale image (Figure 2.1). These pixels, while not initially damaged, have undergone inpainting. To mitigate potential detail losses, values from the unprocessed grayscale image are utilized for the pixels identified in this mask. Corrected image is shown in the Figure 2.3.3. This strategy aims to preserve image details during the restoration process.

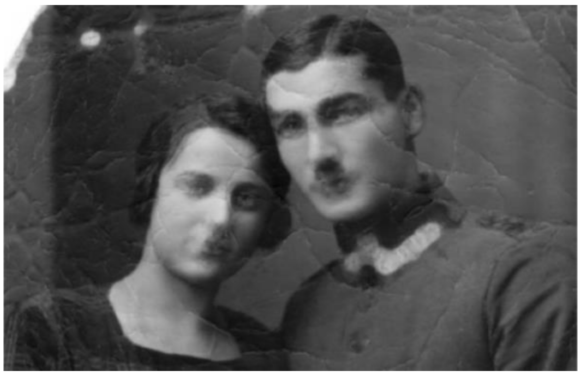
Figure 2.3.1: Last inpainted image

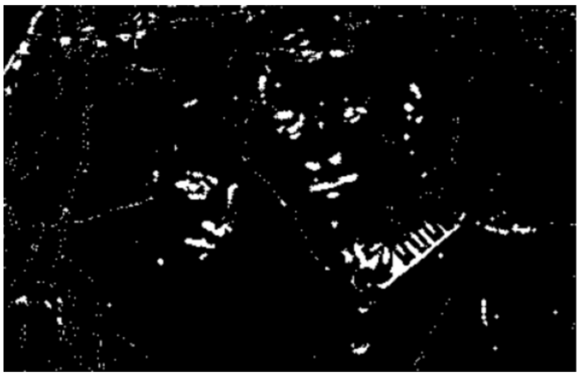
Figure 2.3.2: Detail pixel mask

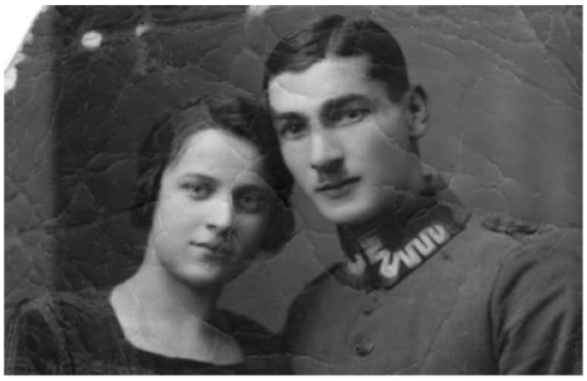
Figure 2.3.3: Image with loss of detail corrected

2.4 Reducing Noise While Preserving Details

In the final stage of the entire process, the focus is on reducing unimportant noises and enhancing contrast. To reduce unimportant noises in the restored image (Figure 2.3.3), non-local means denoising is implemented. For image denoising, the fastNlMeansDenoising function from the OpenCV library is utilized. This function implements a non-local means denoising algorithm, known for its effectiveness in noise reduction while preserving essential details in the image. After denoising, the contrast of the resulting image is increased by 20%. This adjustment aims to enhance the visual quality of the image, making important features more prominent. Result is shown in Figure 2.4.

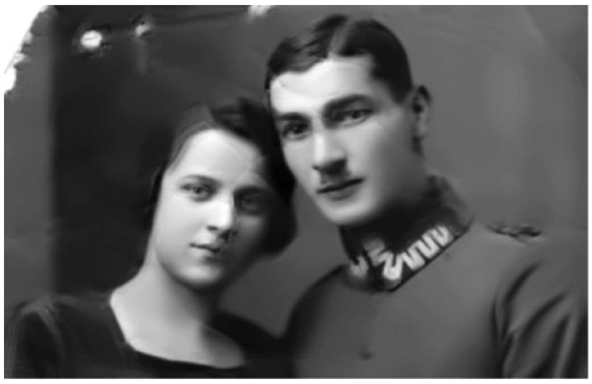

Figure 2.4: Restored image after enhancements

3. EXAMPLES

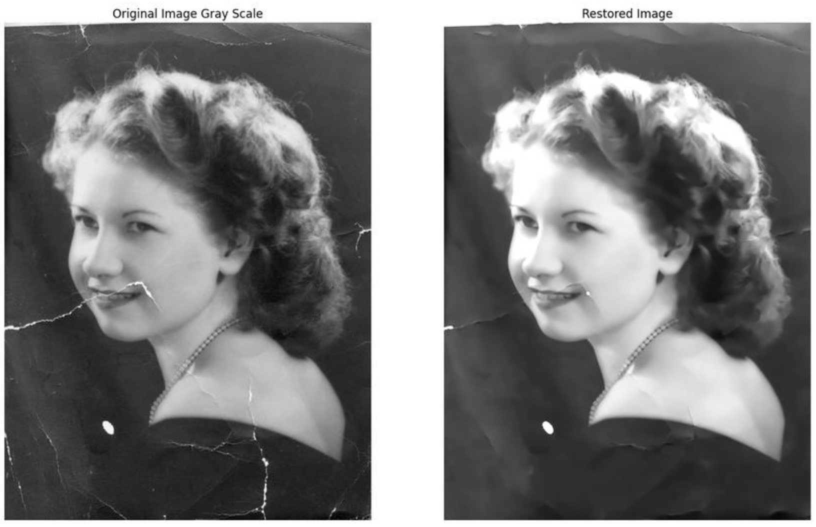

Figure 3.1: Example restoration 1

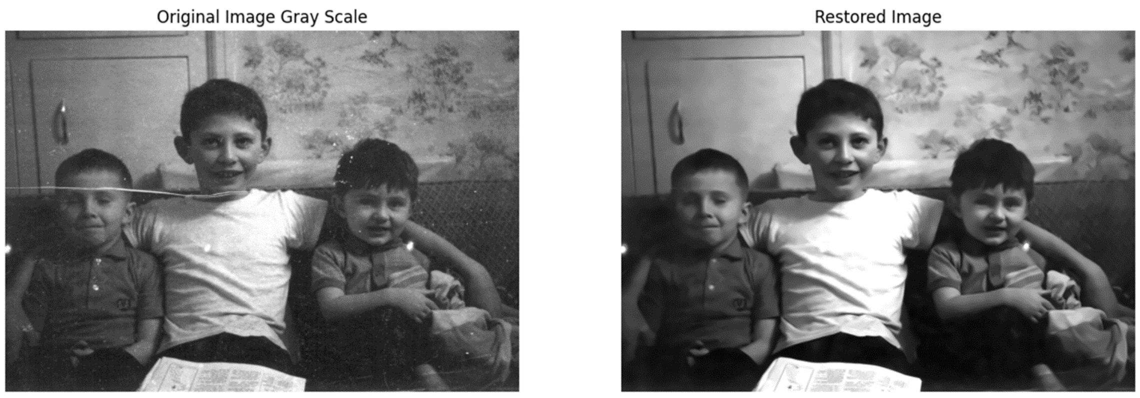

Figure 3.2: Example restoration 2

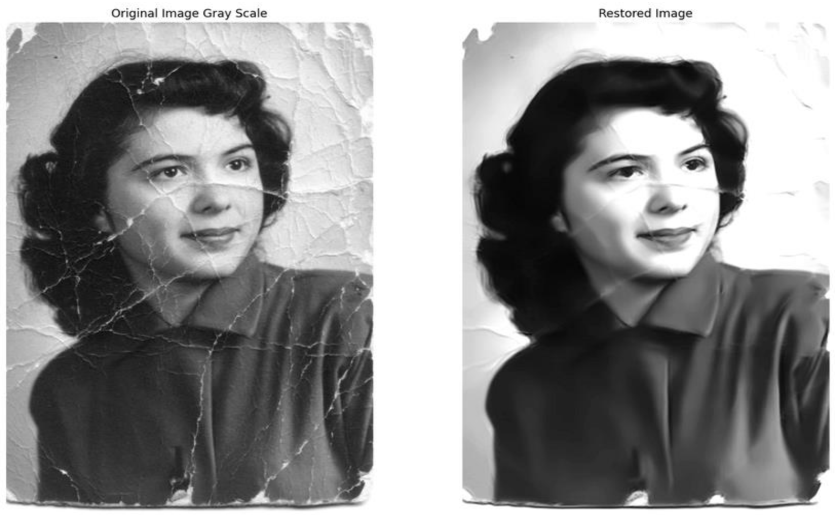

Figure 3.3: Example restoration 3

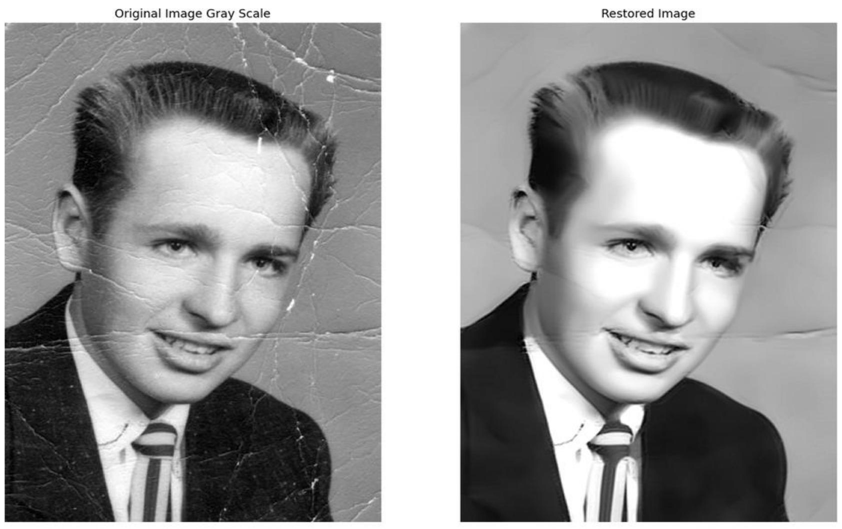

Figure 3.4: Example restoration 4

4. CONCLUSION

This study ensures the effective restoration of damaged areas in the input image, particularly those in white tones and of significant magnitude. However, detail loss can be observed in the undamaged white borders. When determining the threshold in the iterative process, choosing a threshold that is sensitive to the color values of the damaged area can produce more effective results in the repair process. Additionally, if the inpainting intensity is better adjusted according to the structure of the image and damaged areas, error repair can be more accurate, and loss of detail can be reduced.

In conclusion, while this method proves successful in efficiently repairing substantial damage, further enhancements may be required to better preserve details in undamaged regions.

APPENDIX

PYTHON IMPLEMENTATION IN GOOGLE COLAB

```
# Import necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image_path = 'image.png'
original_image = cv2.imread(image_path)
image_gray_scale = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)

# Define morphological kernels for dilation and erosion
kernel_dilation = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
kernel_eroding = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))

# Create a copy of the grayscale image
image = image_gray_scale.copy()
median_blur_kernel_size = 15

# Iterative process for image restoration
while True:
    # Apply median blur to the current image
    blurred_image = cv2.medianBlur(image, median_blur_kernel_size)

    # Create an unfiltered mask
    unfiltered_mask = cv2.absdiff(image, blurred_image)

    # Calculate a dynamic threshold based on non-zero pixel values in the unfiltered
mask
    non_zero_pixels = unfiltered_mask[(unfiltered_mask > 20) & (unfiltered_mask <
150)]
    min_threshold = np.mean(non_zero_pixels)

    # Threshold the unfiltered mask to create a binary mask
    _, binary_mask = cv2.threshold(unfiltered_mask, min_threshold, 255,
cv2.THRESH_BINARY)

    # Enhance the binary mask using dilation and erosion
    binary_mask_enhanced = cv2.dilate(binary_mask, kernel_dilation, iterations=2)
    binary_mask_enhanced = cv2.erode(binary_mask_enhanced, kernel_eroding,
iterations=1)
    binary_mask_enhanced = binary_mask_enhanced.astype(np.uint8)

    # Inpainting to repair damaged areas based on the enhanced binary mask
    image = cv2.inpaint(image, binary_mask_enhanced, 5, cv2.INPAINT_NS)

    # Calculate the ratio of white pixels in the enhanced binary mask
    white_pixel_count = np.sum(binary_mask_enhanced == 255)
    ratio_white_pixels_enhanced = white_pixel_count / (binary_mask_enhanced.shape[0]
* binary_mask_enhanced.shape[1])

    # Break the loop if the ratio of white pixels is below a threshold
    if ratio_white_pixels_enhanced < 0.01:
        break

    # Decrease the median blur kernel size in each iteration
    if(median_blur_kernel_size <= 3):
      median_blur_kernel_size = 3
    else:
      median_blur_kernel_size -= 2

# Create a pixel mask and blend the restored image with the original grayscale image
pixel_mask = image > image_gray_scale
image = np.where(pixel_mask, image_gray_scale, image)
```

```
# Apply additional denoising and contrast adjustment using fastNlMeansDenoising
restored_image = cv2.fastNlMeansDenoising(image, None, h=7, templateWindowSize=15,
searchWindowSize=25)
restored_image = cv2.convertScaleAbs(restored_image, alpha=1.2, beta=-1)

# Display the original and restored images side by side
plt.figure(figsize=(15, 10))
plt.subplot(1,2,1), plt.imshow(image_gray_scale, cmap='gray'), plt.title('Original
Image Gray Scale'), plt.axis('off')
plt.subplot(1,2,2), plt.imshow(restored_image, cmap='gray'), plt.title('Restored
Image'), plt.axis('off')
plt.show()
```