Assignment I
Digital Image Processing Functions: Reading and Point Operation

I. Introduction

Digital image processing (DIP) has become a fundamental component in various fields, including computer vision, medical imaging, multimedia applications, and remote sensing. At its core, DIP focuses on the manipulation of pixel values in digital images to enhance visual quality, extract meaningful information, or prepare data for subsequent analysis. Unlike high-level artificial intelligence methods that rely on large-scale model training, traditional image processing emphasizes mathematical operations and algorithmic transformations that directly act on image intensity values.

This assignment introduces three essential aspects of image processing: image reading, image enhancement, and image interpolation. The first task focuses on image reading, where both RAW and BMP images are handled. Understanding image file formats and correctly reading pixel data form the basis of any image processing pipeline. The second task explores point-based image enhancement operations, including logarithmic transformation, gamma correction, and negative transformation. These classical intensity mapping functions are designed to adjust brightness, enhance contrast, and provide alternative visual perspectives of the input images. The third task addresses image resizing through downsampling and upsampling. Two commonly used interpolation methods—nearest-neighbor and bilinear interpolation—are implemented and compared to illustrate the trade-offs between computational simplicity, sharpness preservation, and smoothness.

By combining these three tasks, the assignment highlights the fundamental techniques in digital image processing and emphasizes the importance of low-level pixel operations. These experiments not only deepen understanding of image intensity manipulation but also provide practical insights into the strengths and limitations of different transformation and interpolation methods. Ultimately, the outcomes serve as a foundation for more advanced studies in image analysis and computer vision.

II. Related Work

Digital image processing has been widely studied, with foundational work by Gonzalez and Woods [1] providing a comprehensive framework for pixel-based manipulation, intensity transformations, and interpolation methods. Techniques such as logarithmic and gamma correction have long been used to enhance image visibility, particularly in applications requiring dynamic range adjustment. Negative transformations are also standard in image processing, offering simple yet effective inversion of intensity values for analysis or visualization.

Interpolation methods, including nearest-neighbor and bilinear interpolation, are well established in image resizing. Nearest-neighbor interpolation is computationally efficient and maintains sharp transitions, though it often introduces aliasing artifacts. In contrast, bilinear interpolation leverages weighted averaging to produce smoother images but may result in blurring of fine details. These trade-offs are discussed extensively in classical image scaling literature [2].

For practical implementation, lightweight libraries such as stb_image_write [3] have been adopted in modern applications due to their simplicity and portability. Such libraries allow rapid prototyping by enabling direct writing of PNG, BMP, and JPEG formats without complex dependencies. In this assignment, the use of stb_image_write.h demonstrates how open-source resources streamline the development of image processing pipelines while retaining flexibility in handling both RAW and BMP formats.

By combining classical image processing theories with modern lightweight libraries, this project situates itself within the broader landscape of applied digital imaging, bridging theoretical methods with efficient implementation strategies.

III. Methodology

A. *Image Reading and Pixel Extraction*

In the first subtask, we were required to process three grayscale RAW images (lena.raw, goldhill.raw, and peppers.raw) and three BMP images (baboon.bmp, boat.bmp, and F16.bmp). The purpose was to correctly read the image content, convert the formats, and verify the correctness by inspecting pixel intensity values in the image center region.

RAW images are characterized by the absence of a file header; they only contain raw pixel intensity values stored in row-major order, i.e., the data are written row by row from top to bottom. Each pixel is represented by one byte with values ranging from [0,255]. Consequently, a 512×512 RAW image occupies exactly $512 \times 512 = 262,144$ bytes. In our implementation, the RAW images were read using the C++ ifstream stream in binary mode (ios::binary), and the data were stored into a two-dimensional array of type unsigned char image[HEIGHT][WIDTH]. The images were subsequently converted and saved as PNG format files using the stbi_write_png() function provided by the stb_image_write library.

In contrast, BMP images contain a file header that stores metadata such as the image width, height, bit depth, and the offset to the actual pixel data. To parse this information, we defined a custom BMPHeader structure in C++. To ensure that the structure's layout matches the BMP file format specification, the directive #pragma pack(push, 1) was used to disable compiler padding between structure members. Depending on the bit depth, two scenarios were handled. For 24-bit BMP images, each pixel consists of three components (blue, green, red). These were averaged to obtain a grayscale intensity value, computed as:

$$I(x,y) = \frac{R(x,y) + G(x,y) + B(x,y)}{3}. \quad (1)$$

For 8-bit BMP images, the pixel values directly represent grayscale intensities. In this case, the program skipped over the color palette (1024 bytes) before reading the pixel data. Once loaded, the BMP images were converted and stored in JPEG format using the stbi_write_jpg() function from the stb_image_write library.

Finally, to validate the correctness of the reading process, we extracted and displayed the central 10×10 region of each image. For a 512×512 image, the center is approximately located at coordinate (256, 256). Thus, rows and columns in the range [251,261) were selected for output. Printing this numerical matrix of pixel intensities provides an objective verification method in addition to visual inspection, confirming that the image data were successfully parsed and interpreted.

B. Image Enhancement and Transformations

In this subtask, we focused on pixel-level image enhancement by applying three classical intensity transformations: logarithmic, gamma correction, and negative transformation. These techniques are widely used in digital image processing for contrast adjustment, brightness manipula-tion, and detail enhancement. Since the image loading and preprocessing procedures for both RAW and BMP images were already described in Section A, here we concentrate on the transformation methods and their effects.

1) *Logarithmic Transformation*: The logarithmic transformation enhances visibility in darker regions by expanding low-intensity pixel values while compressing higher intensities. The mapping function is:

$$s = c \cdot log(1 + r). \quad (2)$$

where r is the input intensity, s is the transformed output, and the scaling constant

$$c = \frac{255}{log(1+255)} \ . \quad (3)$$

ensures that the maximum output intensity is mapped to 255. In practice, this method is particularly effective for images where critical details are hidden in shadowed or dark regions.

2) *Gamma Transformation:* Gamma correction is a nonlinear intensity mapping that controls overall brightness and contrast. The transformation is expressed as:

$$s = c \cdot r^r, \ c = 255^{1-r} \ . \quad (4)$$

Two cases were implemented: Gamma = 0.5: Brightens the image by stretching lower intensity values. Gamma = 2.0: Darkens the image by compressing higher intensity values. This dual implementation highlights the sensitivity of images to different gamma values, which is crucial in display systems where gamma correction is necessary to match the nonlinear response of monitors and human vision.

3) *Negative Transformation:* The negative transformation is a simple yet effective point-wise operation that inverts the grayscale values of an image. It is mathematically defined as:

$$s = 255 - r. \quad (5)$$

where r is the original pixel intensity and s is the transformed output. For an 8-bit image, where intensity values range from 0 (black) to 255 (white), this transformation flips the intensity scale such that bright pixels become dark and dark pixels become bright. In terms of implementation, the transformation is computationally efficient, as it only requires a single subtraction per pixel. The operation was applied iteratively over the entire 2D pixel array using nested loops. The resulting inverted image was then saved into the output directory in PNG format using the stb_image_write library.

C. Image Downsampling and Upsampling

This part of the assignment addresses the task of image resizing using two classical interpolation techniques: nearest-neighbor interpolation and bilinear interpolation. The implementation was conducted with custom functions designed to downsample and upsample grayscale images across multiple scenarios, as specified in the problem statement. The resizing process was encapsulated in two functions: resizeNearest and resizeBilinear. Both functions accepted the original grayscale pixel array, the source dimensions, and the desired target dimensions as input parameters, and generated a new array containing the resized image.

In **nearest-neighbor interpolation**, the target pixel coordinates (x_t, y_t) were scaled back to the source image coordinates using the ratio of dimensions. The closest integer coordinates were computed using the rounding operation, and the corresponding source pixel intensity was directly assigned to the target pixel. This method was efficient, as it only required integer arithmetic, but it often resulted in visible blockiness during downsampling and jagged edges during upsampling. In the case of **bilinear interpolation,** the computation can be understood as performing two consecutive linear interpolations.

First, interpolation is carried out along the horizontal axis. Given a target pixel with fractional coordinates (x, y), the integer neighbors on the left and right are denoted as (x_1, y_1) and (x_2, y_1) for the top row, and (x_1, y_2) and (x_2, y_2) for the bottom row. The horizontal interpolation results are expressed as:

$$I_{top} = (1 - a)I(x_1, y_1) + aI(x_2, y_1), \quad (6)$$

$$I_{bottom} = (1 - a)I(x_1, y_2) + aI(x_2, y_2), \quad (7)$$

where $\boldsymbol{a}$ represents the normalized horizontal distance between the new point and the left neighbor. These intermediate values approximate the intensity at the top and bottom edges of the local square region.

Second, interpolation is performed vertically between I_{top} and I_{bottom}, using the vertical fractional distance $\boldsymbol{b}$:

$$I(x,y) = (1-b)I_{top} + bI_{bottom}. \tag{8}$$

Expanding this expression yields the final bilinear interpolation formula:

$$I(x,y) = (1-a)(1-b)\,I(x_1,y_1) + a(1-b)(x_2,y_1) + (1-a)bI(x_1,y_2) + abI(x_2,y_2). \tag{9}$$

This derivation shows that the interpolated intensity is a weighted average of the four nearest neighbors, with weights directly determined by the fractional distances a and b. In practice, this method produces smoother and more visually consistent results compared to nearest-neighbor interpolation, at the cost of increased computational complexity due to floating-point multiplications.

IV. Implementation and Output

The implementation of this assignment was carried out in C++17, and all three subtasks were developed as independent programs (hw1a.cpp, hw1b.cpp, hw1c.cpp). For file operations, the ifstream stream in binary mode was used to handle raw byte data, and a custom-defined BMPHeader structure was employed to parse metadata for BMP files. To ensure output portability, the stb_image_write library was utilized for saving results in standard formats (PNG/JPG). In addition, the std::filesystem library was used to automatically create dedicated output directories for each subtask.

A. Image Reading

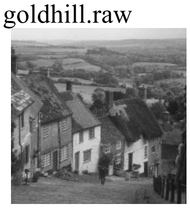
The program successfully read three RAW images (lena.raw, goldhill.raw, peppers.raw) of size 512×512, and three BMP images (baboon.bmp, boat.bmp, F16.bmp). The RAW images were directly interpreted as grayscale arrays, whereas BMP images required header parsing. For 24-bit BMP files, RGB channels were averaged to grayscale; for 8-bit BMP files, the color palette was skipped before pixel reading. The outputs were saved as PNG or JPG images in the output/ directory. In addition, the central 10×10 pixel region was printed to the console for numerical verification.

Output Images and the central 10×10 pixel:

1) lena.raw

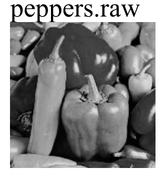
195	195	195	192	169	135	133	137	138	148
196	196	189	157	124	128	135	144	145	145
196	187	149	123	127	131	135	142	142	137
180	135	116	117	124	130	131	132	137	133
124	102	115	116	120	126	124	120	114	111
100	102	114	114	114	118	120	117	112	92
101	100	113	106	98	89	90	104	101	84
105	96	99	94	92	78	75	79	83	76
92	90	90	84	79	75	71	73	72	71
85	79	76	77	76	71	73	73	69	77

2) goldhill.raw



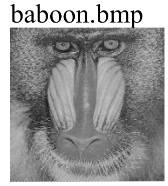
56	60	69	64	54	74	80	78	78	121
47	59	62	69	54	62	48	50	52	70
70	74	73	69	51	58	44	46	46	57
85	85	87	72	45	59	45	45	55	68
65	81	80	78	49	64	52	54	65	69
60	63	78	80	53	83	59	63	78	62
59	66	69	66	57	72	66	71	74	51
53	64	63	75	54	67	68	59	57	53
57	64	63	69	66	74	75	68	89	101
77	68	74	76	94	96	99	88	106	103

3) peppers.raw



63	43	45	60	71	61	58	25	4	20
66	50	46	42	55	40	52	34	4	26
60	57	35	36	47	32	48	25	15	56
86	57	33	37	23	34	25	35	66	59
112	91	57	27	27	14	27	49	57	46
108	69	69	39	20	9	42	73	52	46
113	92	45	32	13	44	55	65	55	32
131	90	59	27	33	62	72	62	59	47
144	96	40	30	57	92	73	66	60	45
133	72	34	61	99	112	100	97	80	48

4) baboon.bmp



149 165 155 155 170 174 173 171 163 159
149 158 169 171 172 166 151 165 156 138
164 170 177 177 180 177 172 178 170 156
172 186 185 175 177 181 184 180 176 165
184 183 166 168 173 170 167 176 171 171
181 170 175 190 190 188 178 176 175 171
183 187 186 190 189 188 176 162 158 153
190 192 184 176 188 186 184 174 164 159
191 186 165 143 166 178 185 187 183 171
160 168 146 143 156 164 177 174 170 169

5) boat.bmp

84 70 65 67 53 49 54 66 123 141
82 66 56 53 45 53 55 69 127 136
86 65 60 49 47 52 55 67 118 138
79 71 59 50 51 50 55 72 122 133
86 76 61 53 54 54 58 68 117 132
89 77 61 59 52 56 55 65 116 132
84 78 61 57 55 57 58 65 117 137
78 77 55 58 52 56 57 68 127 136
75 67 54 57 48 59 58 67 131 136
76 69 54 61 46 53 61 61 126 138

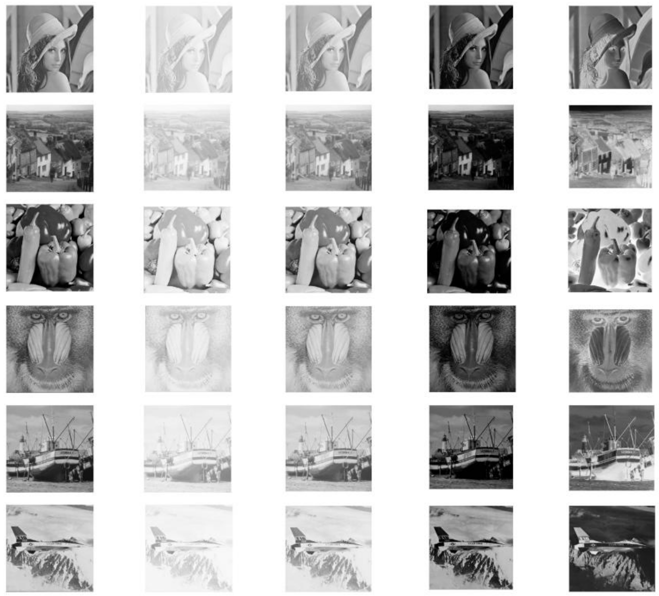
6) f16.bmp

111 107 111 120 129 127 124 115 115 101
110 107 116 118 123 126 119 111 113 108
106 109 109 114 112 113 108 109 114 108
99 101 102 103 104 108 109 108 109 109
101 102 95 100 104 106 106 106 107 108
94 96 102 105 108 101 102 104 108 107
94 99 104 104 103 101 103 98 106 108
101 101 101 99 104 99 104 101 104 104
98 99 103 101 101 100 105 100 105 101
96 103 99 101 100 98 102 106 105 105

B. *Image Enhancement and Transformations*

For each input image, four enhanced versions were generated: logarithmic, gamma correction (γ =0.5 and γ =2.0), and negative transformation. Each transformation was applied pixel by pixel using nested loops, ensuring full control over the intensity mapping process. The results demonstrated the expected effects: logarithmic enhancement emphasized dark regions, gamma correction provided brightness adjustment, and negative transformation inverted image tones. The final outputs were saved as PNG files using the lightweight stb_image_write library. To maintain organization, all processed images were written into a dedicated output/transform/ directory, preventing confusion with original datasets. From the visual comparison, distinct characteristics can be observed for each transformation:

- **Logarithmic Transform**: The log operation significantly increased the visibility of darker details in the images, such as textures in shadowed areas and low-contrast regions. However, bright areas tended to saturate, resulting in a washed-out appearance in highlights. For example, in the Lena and Boat images, fine details in the darker regions of the background became more pronounced, while brighter surfaces lost subtle contrast.
- **Gamma Correction** (γ =0.5): This setting performed a power-law transformation that brightened the entire image, especially emphasizing mid-tone and shadow regions. The effect was similar to applying a global lightening filter, making darker areas clearer at the cost of reduced contrast in already bright regions. This was evident in the Peppers image, where the vegetable contours in shadow became visible, though some highlight regions appeared flat.
- **Gamma Correction** (γ =2.0): In contrast, a higher gamma compressed the dynamic range of bright areas and darkened the overall image. As a result, the transformation emphasized highlights while strongly suppressing low-intensity details. For example, in the F16 aircraft image, the sky and mountain background became more contrasted, but fine details in shadowed parts of the aircraft were suppressed.
- **Negative Transformation**: By inverting pixel intensities, the images produced a photographic negative effect. This transformation reversed dark and bright regions, creating a visually striking but less natural output. For instance, in the Baboon image, the facial details appeared inverted, making bright fur darker and the background appear unusually bright. While not enhancing details in the conventional sense, this method can be useful for applications requiring feature extraction in complementary intensity ranges.



original | log-transform | gamma-transform(0.5) | gamma-transform(2.0) | negative-transform

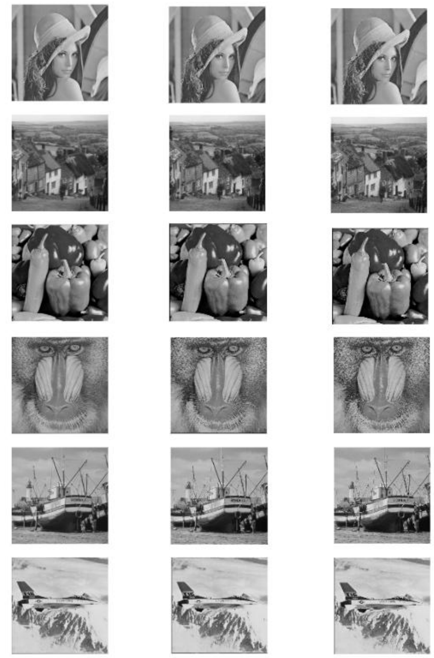
Overall, these transformations highlighted the trade-offs between enhancing specific regions of an image and maintaining natural contrast. Logarithmic and low-gamma corrections improved visibility in dark regions, high-gamma correction emphasized bright regions, and negative transformation provided an alternative intensity perspective. The comparative results allowed us to better understand the effect of different pixel-wise intensity mappings in digital image processing. The final outputs were saved as PNG files using the lightweight stb_image_write library. To maintain organization, all processed images were written into a dedicated output/transform directory, preventing confusion with original datasets.

C. Image Downsampling and Upsampling

The third program implemented both nearest-neighbor and bilinear interpolation for image resizing. Five resizing experiments were conducted for each input image, including downsampling from 512×512 to 128×128 and 32×32, as well as upsampling from 32×32 and 128×128 back to higher resolutions, and an enlargement from 512×512 to 1024×512 and 256x512.

Experimental results confirmed clear differences between the two interpolation strategies. Nearest-neighbor interpolation preserved sharp edges more faithfully in some cases, but it introduced visible aliasing artifacts and pixelation, especially during severe downsampling or subsequent upsampling. This produced blocky and coarse textures, which were particularly noticeable in fine-detailed regions such as the Baboon’s fur or the boat’s ropes. On the other hand, bilinear interpolation produced smoother transitions by averaging neighboring pixel intensities, leading to visually more natural results. However, this smoothing also caused slight blurring and loss of edge sharpness, which was apparent in structured patterns such as Lena’s hat and the building edges in the Goldhill image. The following is the image output result:

- (512x512) -> (128x128):

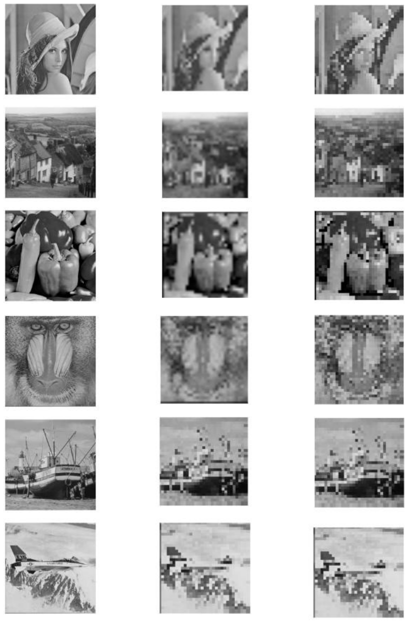


original 512x512 | Bilinear interpolation | Nearest-neighbor interpolation

- (512x512) -> (32x32):

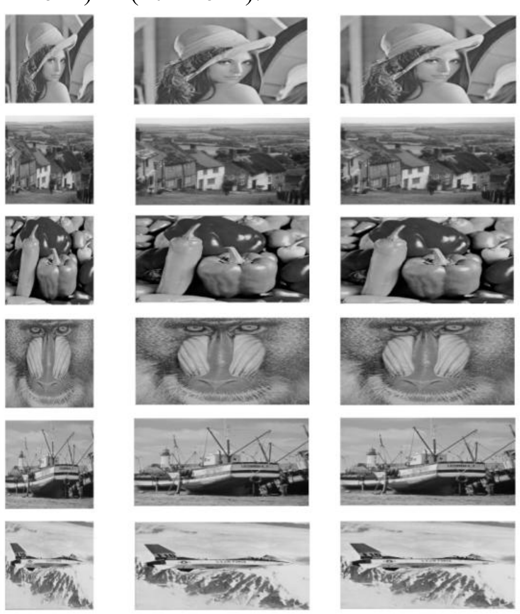
original 512x512 | Bilinear interpolation | Nearest-neighbor interpolation

- (32x32) -> (512x512):



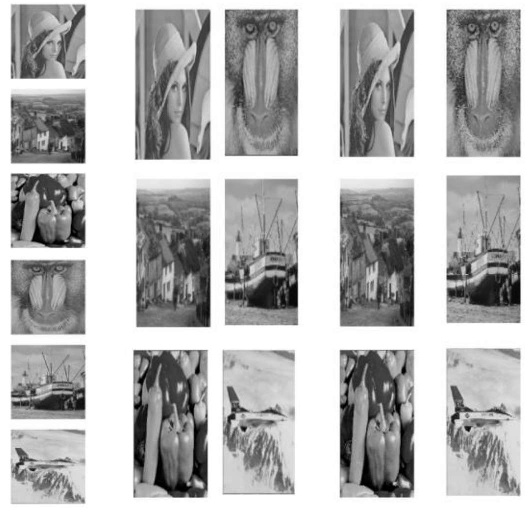
original 512x512 | Bilinear interpolation | Nearest-neighbor interpolation

- (512x512) -> (1024x512):



original 512x512 | Bilinear interpolation | Nearest-neighbor interpolation

- (128x128) -> (256x512):



original 512x512 | Bilinear interpolation | Nearest-neighbor interpolation

When downsampling to 128×128, both methods maintained recognizable structures, but nearest-neighbor outputs exhibited higher granularity and jagged edges, while bilinear results appeared smoother. At extreme downsampling (32×32), nearest-neighbor interpolation caused severe blockiness, with individual pixels becoming visible, giving the image a mosaic-like effect. Bilinear interpolation softened these artifacts, but at the cost of making the image appear blurry and lacking fine texture.

In the upsampling experiments, the differences became even more pronounced. Upscaling from 32×32 back to 512×512 highlighted the trade-offs: nearest-neighbor magnified pixel blocks directly, resulting in a coarse, grid-like pattern; bilinear interpolation produced smoother gradients but blurred edges. In the F16 aircraft image, nearest-neighbor interpolation made the jet contours appear jagged, whereas bilinear interpolation gave a smoother outline but reduced the sharpness of structural details. Similarly, in the Peppers image, nearest-neighbor produced block artifacts within the pepper textures, while bilinear maintained smoother shading but diminished fine granularity.

Finally, enlargement from 512×512 to 1024×512 showed that bilinear interpolation scaled naturally with less visual distortion, while nearest-neighbor produced harsher stair-step effects along edges. Overall, nearest-neighbor interpolation is computationally simple but suffers from coarse artifacts in both downsampling and upsampling, while bilinear interpolation achieves visually more natural results at the cost of blurring and reduced sharpness.

References

[1] R. C. Gonzalez and R. E. Woods, Digital Image Processing, 4th ed., Pearson, 2018.
[2] A. K. Jain, Fundamentals of Digital Image Processing, Prentice-Hall, 1989.

[3] S. Barrett, "stb single-file public domain libraries," [Online]. Available: https://github.com/nothings/stb

Bilinear interpolation:
https://en.wikipedia.org/wiki/Bilinear_interpolation.
Nearest neighbor interpolation:
https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation