

Digital Image Processing Functions: Reading and Point Operation

Chien, Y. C.

Department of Intelligent System and Application
National Yang Ming Chiao Tung University

Abstract – This is the basic image processing, which include read, enhance and resize function.

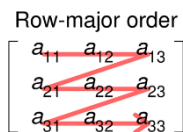
Index Terms - Image Processing

I. INTRODUCTION

This report is for Multi-model Image Processing class homework.

A. Assignment Requirement

- 1) *Image reading*: Give three raw-formatted images and 3 jpg images, please read the content correctly and display them on the screen, where the raw images are sized 512x512 in grayscale format. The pixel order is row-major as follows:



Also, please draw the centered 10x10 pixel values in each image in report.

- 2) *Image enhancement toolkit*: Give test six images, please implement log-transform, gamma-transform, and image negative, and compare them in report.
- 3) *Image downsampling and upsampling*: please implement bilinear and nearest-neighbor interpolation methods.

II. HELPFUL HINTS

A. How to read image

1) RAW format

A RAW file has no header. Read the bytes directly into a buffer and interpret them with the agreed layout (e.g., 512x512, 8-bit, row-major). For 8-bit grayscale row-major, the offset is $i \times \text{width} + j$.

2) BMP format

BMP image file include file header (14 bytes), image header (40 bytes), color palette (1024 bytes) and RAW file [1]. I will introduce each element below:

- 2.1) *BMP file header* (shown in Fig. 1): `bfType` is an identification code always be BM. Next, `bfSize` refers to the file size, which includes all the data. The `bfReserved` and `bfOffBits` fields can be filled with 0; this does not affect normal operation. The last `bfOffBits` value of 54 is fixed, which is 14 + 40. However, if you are using grayscale images, remember to add 1024 (to account for the color palette).

```
struct BmpFileHeader{
    uint16_t bfType = 0x424D;
    uint32_t bfSize;
    uint16_t bfReserved1 = 0;
    uint16_t bfReserved2 = 0;
    uint32_t bfOffBits = 54;
};
```

Fig. 1 BmpFileHeader format

- 2.2) *BMP Image Header* (shown as Fig. 2): `biSize` is size of the header (common is 40 bytes). Next, `BiWidth` and `BiHeight` separately represent image width and height. The `biBitCount` is the number of bits. For color images, set the RGB value to $8 \times 3 = 24$, and for grayscale images, set it to 8. And `biClrUsed` the number of colors in the palette for color images, set this to 0; for grayscale images with a custom palette, set this to 256. The last `PelsPerMeter` means density value can be 0 because most software seems to set this to 0 regardless of the dpi.

```
struct BmpInfoHeader{
    uint32_t biSize = 40;
    uint32_t biWidth;
    uint32_t biHeight;
    uint16_t biPlanes = 1; // 1=default, 0=custom
    uint16_t biBitCount;
    uint32_t biCompression = 0;
    uint32_t biSizeImage = 0;
    uint32_t biXPelsPerMeter = 0; // 72dpi=2835, 96dpi=3780
    uint32_t biYPelsPerMeter = 0; // 120dpi=4724, 300dpi=11811
    uint32_t biClrUsed = 0;
    uint32_t biClrImportant = 0;
};
```

Fig. 2 BmpImageHeader format

- 2.3) *Color Palette*: This is optional and may not be present. If the image is RGB color, there's no need to leave it blank (e.g., 00); simply leaving it blank is the correct way to do it. If the image is grayscale, you'll need to add a color palette yourself. The color palette is quite simple: 00 00 00 00 ~ FF FF FF 00. This is because grayscale images only require a single byte, while an entire BMP image must be represented in RGB. Therefore, the color palette transforms the raw file you see, for example, 00 becomes 00 00 00 and FF becomes FF FF FF.
- 2.4) *RAW file*: The special thing is that the reading method starts from the lower left corner of the image and reads to the upper right. The data is in BGR reverse order and their rows are aligned to 4 bytes. If it's 24-bit and width=42, each row's raw bytes are $42 \times 3 = 126$, so `rowSize=128` (add 2 bytes padding per row). The height stays 42; you just add per-row padding.

B. How to enhance image

- 1) *Log transform*: The logarithmic transform applies a log function to each pixel, expanding dark tones and compressing bright tones so that fine details in shadow regions become more visible (see Fig. 3). The mapping is

$$S = c * \ln(1 + r) \quad (1)$$

where r is the input intensity and c is a scale factor. For images $r \in [0, 255]$, a common choice is

$$r = \frac{255}{\ln(1 + 255)}$$

- 2) *Gamma-transform*: Human vision is not linear with respect to luminance; we are more sensitive to relative changes in darker tones than in bright ones. The gamma transform compensates for this nonlinearity by remapping each pixel's intensity via

$$S = 255 \left(\frac{r}{255} \right)^\gamma \quad (2)$$

where $r \in [0, 255]$ is the input intensity, $S \in [0, 255]$ is the output, and $\gamma > 0$ is the exponent controlling the curve. Values $\gamma < 1$ brighten shadows (expand dark tones), while $\gamma > 1$ darken highlights (compress bright tones), allowing contrast to be redistributed according to perceptual needs (see Fig. 3). In practice, the mapping is applied per channel (for color images), typically via a 256-entry lookup table for efficiency.

- 3) *Image Negative*: The negative transform inverts intensity to create a tonal complement of the image. For each pixel is remapped by

$$S = 255 - r \quad (3)$$

where $r \in [0, 255]$ is the input intensity and sss the output. This operation reverses contrast—dark regions become bright and vice versa—while preserving the dynamic range (see Fig. 3). It is useful for highlighting low-intensity details (e.g., in medical or astronomical images) and for visual inspection of structures obscured by bright backgrounds. For color images, the mapping is applied per channel (R, G, B), producing the chromatic negative of the original.

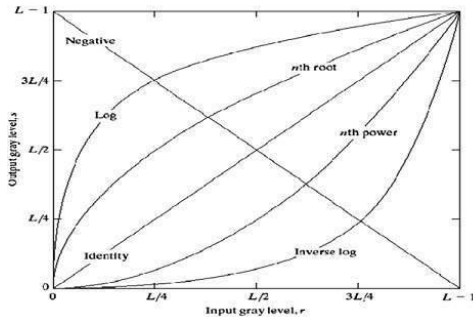


Fig. 3 Basic intensity transforms. The negative (descending line) inverts contrast. The log curve expands dark tones and compresses bright tones. Gamma curves appear as the “nth root” family ($\gamma < 1$, brightening) and the “nth power” family ($\gamma > 1$, darkening).

C. Image downsampling and upsampling

- 1) *Nearest neighbor (NN) Interpolation*: For each output pixel center (x, y) , map back to the input coordinate space and copy the nearest input pixel's value (no averaging). This creates blocky replicas of input pixels when scaling up[2] (see Fig. 4).

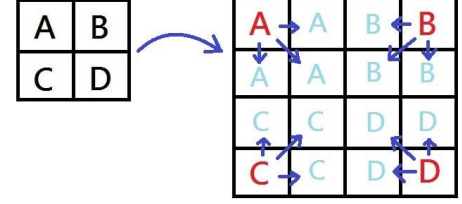


Fig. 4 Nearest-neighbor (NN) upsampling: each output pixel copies the value of the nearest input pixel, yielding block-like replication of the original cells (A–D).

- 2) *Bilinear Interpolation*: This method estimates the value at a non-integer location (x, y) by linearly interpolating twice: first along x on the rows above and below the target, then along y between those two results. It uses the four neighbors

$$(x_0, y_0), (x_1, y_0), (x_0, y_1), (x_1, y_1),$$

where $x_0 \leq x \leq x_1$ and $y_0 \leq y \leq y_1$. (see Fig. 5)

$$F(x, y_0) = \frac{x_1 - x}{x_1 - x_0} F(x_0, y_0) + \frac{x - x_0}{x_1 - x_0} F(x_1, y_0) \quad (4)$$

$$F(x, y_1) = \frac{x_1 - x}{x_1 - x_0} F(x_0, y_1) + \frac{x - x_0}{x_1 - x_0} F(x_1, y_1) \quad (5)$$

Combining along y gives the bilinear estimate:

$$F(x, y) = \frac{y_1 - y}{y_1 - y_0} F(x, y_0) + \frac{y - y_0}{y_1 - y_0} F(x, y_1) \quad (6)$$

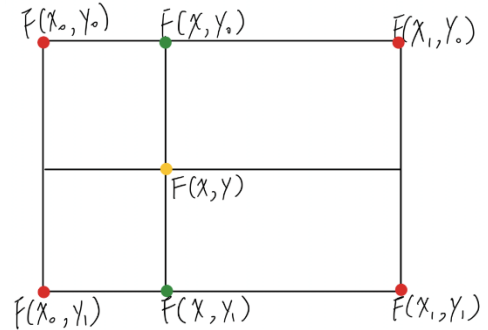


Fig. 5 Bilinear interpolation: the value at $F(x, y)$ (yellow) is a convex combination of its four neighbors (red/green) via linear interpolation along x and then along y .

D. Pixel Centered

In most image libraries the origin is placed at the upper-left corner of the image domain, but when you “access a pixel at (i, j) ” you’re reading the value at the center of the pixel whose upper-left corner is (i, j) . That subtle offset doesn’t matter for many operations, but it does matter when rescaling. If you downsample a 5×5 image to 3×3 and naively map output corners with $x' = x \cdot \frac{W_{in}}{W_{out}}$, the first row maps as

$(0, 0) \rightarrow (0, 0)$, $(1, 0) \rightarrow (5/3, 0)$, $(2, 0) \rightarrow (10/3, 0)$ (see Fig. 6): this is corner-based and introduces a half-pixel bias.

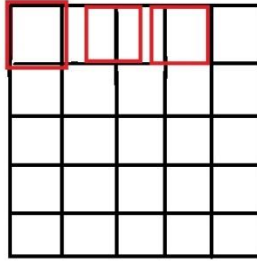


Fig. 6 Misaligned image. The first sample lands exactly on the top-left corner, the next near a grid line, etc. This introduces a half-pixel bias (everything is shifted toward the top-left), makes integer scales misalign, and can cause dark/light borders or asymmetry when resampling.

The standard fix is pixel-centered mapping: map centers with :

$$f(x) = (x + 0.5) \frac{W_{in}}{W_{out}} - 0.5 \quad (7)$$

$$f(y) = (y + 0.5) \frac{W_{in}}{W_{out}} - 0.5 \quad (8)$$

so the first row centers land at $x \approx 0.33, 2.0, 3.67$. Using the centered formula (7) and (8) aligns pixel centers between input and output, reducing alignment errors and producing the expected results for nearest/bilinear interpolation[3]. (Fig. 7)

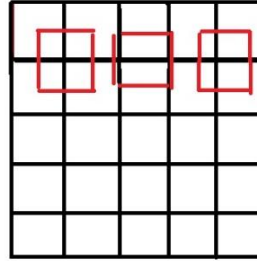


Fig. 7 Biased image. The output sample positions fall at the centers of the appropriate input cells. This keeps input/output grids aligned, preserves symmetry, and gives the expected behavior

III. RESULT

A. *Image reading*: Below is the result of read image (Fig. 8-13), and just see the centered 10x10 pixels values of image which is useful for quick verification, also I have already use python library quickly to check it is correct or not.

```
---- Center 10x10: original (512x512, c=3)
152 151 154 167 155 142 139 136 139 143
150 175 177 178 180 177 174 173 153 130
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
```

Fig. 8 Baboon.bmp 10x10 center pixels

```
---- Center 10x10: original (512x512, c=3)
88 74 105 105 48 57 62 75 140 142
83 71 85 101 58 50 63 65 129 143
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
```

Fig. 9 Boat.bmp 10x10 center pixels

```
---- Center 10x10: original (512x512, c=3)
105 110 108 111 118 130 126 123 115 107
106 109 109 114 123 131 124 117 116 107
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
```

Fig. 10 F16.bmp 10x10 center pixels

```
---- Center 10x10: original (512x512, c=1)
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
```

Fig. 11 goldhill.raw 10x10 center pixels

```
---- Center 10x10: original (512x512, c=1)
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
```

Fig. 12 lena.raw 10x10 center pixels

```
---- Center 10x10: original (512x512, c=1)
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
```

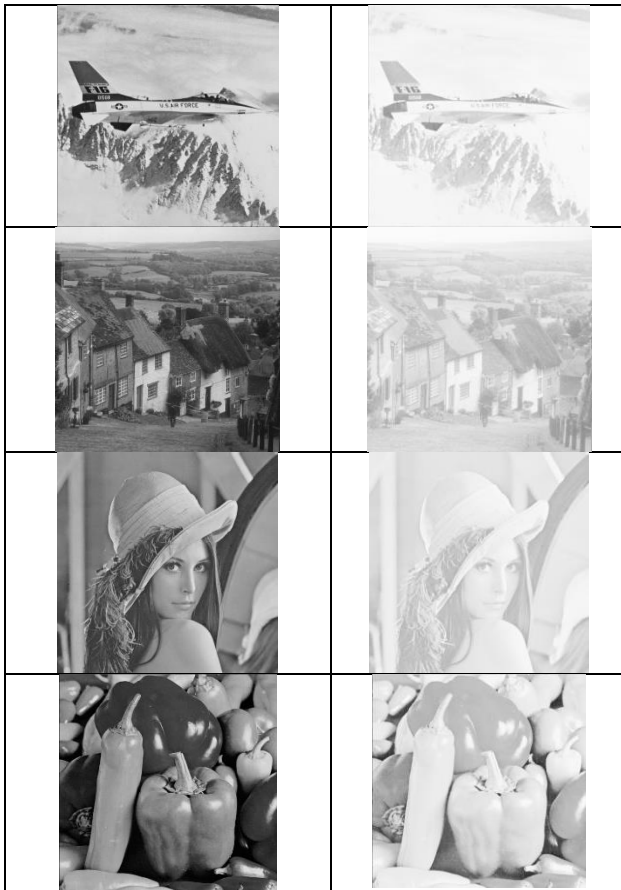
Fig. 13 peppers.raw 10x10 center pixels

- B. *Image enhancement toolkit*: After reading image, I use log-transform, gamma-transform, and image negative, (HELPFUL HINTS B part) to enhance each image, and I will compare them.

1) Log transform

TABLE I
LOG TRANSFORM FOR EACH IMAGE

Original image	After log transform



From Table I, the dark regions become noticeably clearer than in the original image. Overall contrast is enhanced while preserving natural appearance.

2) *Gamma-transform*

TABLE II GAMMA TRANSFORM FOR EACH IMAGE ($\gamma < 1$)	
Original image	After gamma transform



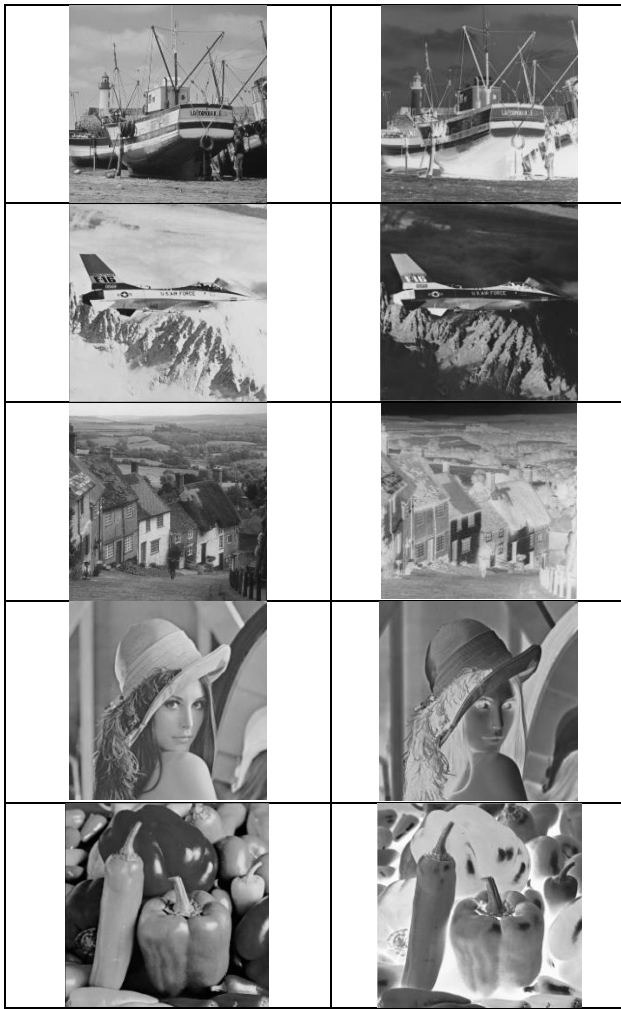
As shown in Table II, $\gamma = 0.5$ substantially clarifies dark regions relative to the original and improves overall contrast without introducing visible artifacts.

TABLE III GAMMA TRANSFORM FOR EACH IMAGE ($\gamma > 1$)	
Original image	After gamma transform

Relative to Table II, which uses $\gamma=0.5$ to reveal shadow detail and boost perceived contrast, Table III applies $\gamma>1$ to suppress overly bright regions—darkening highlights and compressing bright tones—thereby reducing glare without altering spatial detail.

3) *Image Negative*

TABLE IV IMAGE NEGATIVE FOR EACH IMAGE	
Original image	After image negative



Based on Table IV, the negative transform inverts luminance: bright regions become dark, and vice versa, while edges and structures are preserved.

C. Image downsampling and upsampling

In this section, I use baboon.bmp as an example to compare results at different image sizes.

- 1) original image (512x512) -> (128x128)

TABLE V

IMAGE DOWNSAMPLING FOR IMAGE

	Original image	Resized image
nearest		
bilinear		

- 2) original image (512x512) -> (32x32)

TABLE VI

IMAGE DOWNSAMPLING FOR IMAGE

	Original image	Resized image
nearest		
bilinear		

- 3) original image (32x32) -> (512x512)

TABLE VII

IMAGE UPSAMPLING FOR IMAGE

	Original image	Resized image
nearest		
bilinear		

- 4) original image (512x512) -> (1024x512)


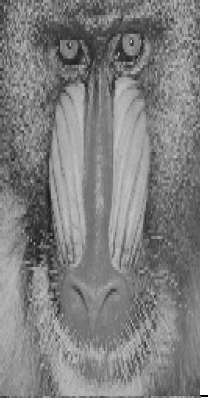


TABLE VII

IMAGE UPSAMPLING FOR IMAGE

	Original image	Resized image
nearest		
bilinear		

5) original image (128x128) -> (256x512)

TABLE VIII
IMAGE UPSAMPLING FOR IMAGE

	Original image	Resized image
nearest		
bilinear		

Comparing nearest-neighbor (NN) and bilinear interpolation across Tables V–VIII shows that bilinear generally preserves structure and visual smoothness better than NN. For example, in Table VI (baboon’s eye region), NN produces blocky, more blurred edges, whereas bilinear retains finer contours. In Table VII, bilinear yields smoother tonal transitions than NN, which exhibits stair-step artifacts. Overall, bilinear delivers better perceptual quality than nearest neighbor in these examples.

IV. DIFFICULTIES I MET

I originally thought this was a quick project, but I realized I didn't fully grasp the underlying principles. This time, I spent a lot of time figuring out how to implement it. However, due to time constraints, I made some changes based on the AI-powered code. During the implementation process, I discovered that Windows doesn't support reading raw files, so I implemented the output as a bmp file. I also wanted to implement an interface, but the time allowed was too short, so I'll have to wait and see if I have time to make up for it. I hope I can set aside more time next time to complete the project.

ACKNOWLEDGMENT

Here have the whole work here: [b1029009Chien/MMIP: multiple-model image proceesing](https://github.com/b1029009Chien/MMIP-multiple-model-image-proceesing)

REFERENCES

[1] Charlotte.HonG, " C / C++ Bitmap(BMP) 圖檔讀寫範例 與 檔頭詳細解析", CHG , <https://charlottehong.blogspot.com/2017/06/c-raw-bmp.html> (accessed 0928, 2025)

[2] chrish0729, " [影像處理] 影像插值 #2", hackmd, <https://hackmd.io/@chrish0729/SkpfuvZx6> (accessed 0928, 2025)

[3] rs 勿忘初心, “ 双线性插值算法的详细总结 ”, https://blog.csdn.net/sinat_33718563/article/details/78825971 (accessed 0928, 2025)