1 Introduction

Fractal compression is a lossy compression method for fractal-based digital images. The method is best suited for textures and natural images, relying on the fact that parts of an image often resemble other parts of the same image. Fractal algorithms convert these parts into mathematical data called "fractal codes," which are used to recreate the encoded image.
The project is to implement the affine transformations on both grayscale and RGB images, including translation, rotation, scaling, and reflection/mirroring. In later sections, the document describe the mathematics of fractal image compression and its implementation in compressing Grayscale and RGB image.

Task	Function	Completion level (%)
1	Implement affine transformations: translation, rotation, scaling, reflection/mirroring for grayscale images.	100
2	Implement affine transformations: translation, rotation, scaling, reflection/mirroring for RGB images	100
3	Implement fractal image compression for grayscale images with affine transformations	100
4	Implement fractal image compression for grayscale images with affine transformations, and contrast + brightness	100
5	Implement fractal image compression for RGB images (bonus).	100

2 Functions description

2.1 Libraries

numpy	Utilized for numerical computations and handling powerful multidimensional arrays. It is used in your code for performing mathematical operations on image data, such as matrix manipulation, element-wise operations, and calculating error metrics like RMSE.
matplotlib.pyplot	Used for visualizing images and intermediate results. It enables rendering images (e.g., `imshow`) and displaying the plots with the results of image transformations, compression, and decompression processes. It is also used to visualize error metrics like RMSE during image comparison.
PIL (Pillow)	Used for image processing tasks. The `PIL.Image` module is used to open and manipulate images, enabling you to load image files into numpy arrays for processing and save the resulting images after transformations.
math	Provides mathematical functions. It is used for performing operations like calculating the square root, rounding numbers, and other mathematical operations related to image transformation and error calculations (e.g., `math.ceil`).

2.2 Affine transformation

Affine transformation is a linear mapping method that preserves points, straight lines, and planes. Parallel lines remain parallel after an affine transformation, but distances and angles are not preserved. This transformation can be represented as a matrix operation, where the transformation is defined by a matrix A and a

translation vector b. In a 2D space, the affine transformation can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where:

- $a_{11}, a_{12}, a_{21}, a_{22}$ are the linear coefficients that affect scaling, rotation, and shearing.
- t_x, t_y are the translation parameters that move the image along the x and y axes.
- (x, y) are the original image coordinates and (x', y') are the transformed coordinates in the output image.

In this code, an inverse affine transformation is applied because we map the pixel coordinates in the output image back to the input image. The inverse of the affine transformation matrix is computed using the following.

$$T^{-1} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}^{-1}$$

This inverse matrix allows the code to compute the corresponding coordinates in the input image for each pixel in the output image. The pixel values are then copied from the source image to the target image based on the calculated inverse coordinates.
For example, consider an affine transformation matrix T that applies a translation and scaling operation:

$$T = \begin{bmatrix} 1.2 & 0 & 50 \\ 0 & 1.2 & 50 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix scales the image by a factor of 1.2 in both the x and y directions, and then translates the image by 50 pixels in both directions.
Given an input image of size 100×100, after applying this transformation, the image will be scaled and shifted. For example, a pixel at coordinate $(10, 20)$ will be transformed as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} 10 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.2 \times 10 + 50 \\ 1.2 \times 20 + 50 \\ 1 \end{bmatrix} = \begin{bmatrix} 62 \\ 74 \\ 1 \end{bmatrix}$$

So, the pixel at $(10, 20)$ in the input image would be mapped to $(62, 74)$ in the output image.

Translation

The translation matrix is given by:

$$T(x, y) = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

where x and y are the translation distances along the x and y axes, respectively.

Scaling

The scaling matrix is given by:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where s_x and s_y are the scaling factors along the x and y axes, respectively.

Rotation

The rotation matrix is given by:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the rotation angle.

Reflection across the X-axis

The reflection matrix across the X-axis is given by:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection across the Y-axis

The reflection matrix across the Y-axis is given by:

$$R_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection across the line $y = x$

The reflection matrix across the line $y = x$ is given by:

$$R_{xy} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection across the line $y = -x$

The reflection matrix across the line $y = -x$ is given by:

$$R_{-xy} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Algorithm 1 Affine Transformation Algorithm

Initialize output_image[M, N] ▷ M = height, N = width of the output image
Setup the affine transformation matrix A as a 3x3 matrix:

$$A = \begin{bmatrix} a[1] & a[2] & a[3] \\ b[1] & b[2] & b[3] \\ c[1] & c[2] & c[3] \end{bmatrix}$$

Compute the inverse matrix of A: $inv\_matrix = \text{inverse}(A)$
Input the initial image (input_image[M, N]) for transformation
for i = 1 to M **do** ▷ Iterate over rows of output image
    for j = 1 to N **do** ▷ Iterate over columns of output image
        Convert (i, j) from the output image to homogeneous coordinates
        $src\_coords = inv\_matrix \times \begin{bmatrix} j \\ i \\ 1 \end{bmatrix}$
        $src\_x, src\_y = src\_coords[0], src\_coords[1]$
        if $0 \leq src\_x <$ width of input_image and $0 \leq src\_y <$ height of input_image **then**
            if input_image is grayscale (2D) **then**
                $output\_image[i, j] = input\_image[src\_y, src\_x]$ ▷ Grayscale pixel assignment
            else if input_image is RGB (3D) **then**
                $output\_image[i, j] = input\_image[src\_y, src\_x, :]$ ▷ RGB pixel assignment
            end if
        end if
    end for
end for
return transformed $output\_image$

2.3 Fractal Image Compression with affine transformations, and contrast + brightness

Fractal image compression is a method of compressing digital images based on self-similarity and contraction mappings in metric spaces.

2.3.1 Contraction Mapping Theorem

A function $f : X \to X$ is called a *contraction* in a metric space (X, d) if there exists a constant $c \in [0, 1)$ such that for all $x, y \in X$:

$$d(f(x), f(y)) \leq c \cdot d(x, y)$$

where $d(x, y)$ is the metric (or distance function) between points x and y in the space X. In the context of fractal image compression, the image is treated as a point in a metric space, where the distance between points is defined by pixel differences between blocks in the image.

The contraction property ensures that applying the transformation repeatedly will converge to a fixed point, which corresponds to the original image during decompression.

2.3.2 Contracting Transformations

In fractal image compression, the image is partitioned into smaller blocks, and the transformation applied to each block can be written as:

$$T_{ij}(B_i) = \alpha_{ij} B_i + \beta_{ij}$$

where:

- B_i is the i-th block of the image,
- T_{ij} is the transformation applied to the i-th block to match the j-th target block,
- α_{ij} is the scaling factor for the transformation,
- β_{ij} is the offset (brightness) applied to the block.

2.3.3 Contraction Mapping and Fixed Points

The contraction property ensures that, after applying the transformations iteratively, the sequence of images converges to a fixed point, ideally the original image. This can be formalized as:

$$I_0 = \text{initial guess}, \quad I_{k+1} = T(I_k)$$

where T is the transformation function applied to each block in the image.

2.3.4 Image Blocks and Transformations

Let O represent the original image, which is divided into smaller blocks $B_1, B_2, \ldots, B_m$, where each B_i is a block of pixels. Each block is treated as a point in the image space $I \subset \mathbb{R}^n$, where n is the number of pixels in each block (e.g., for an 8×8 block, $n = 64$).

The goal is to find a transformation T_{ij} that maps a source block B_i to a destination block D_j, minimizing the error between the transformed source block and the target block.

The error between a source block B_i and a destination block D_j is typically measured by the Mean Squared Error (MSE):

$$\text{MSE}(B_i, D_j) = \frac{1}{n} \sum_{k=1}^{n} \left(\left(\alpha_{ij} B_i(k) + \beta_{ij} \right) - D_j(k) \right)^2$$

where $B_i(k)$ and $D_j(k)$ are the pixel values of the k-th pixel in blocks B_i and D_j, respectively, and n is the number of pixels in each block.

2.3.5 Optimal Transformation Parameters

The optimal parameters α_{ij} and β_{ij} are found by solving the least squares optimization problem. This involves minimizing the sum of squared errors between the transformed source block and the destination block. The system of equations to solve for α_{ij} and β_{ij} is given by:

$$\begin{bmatrix} \sum B_i^2 & \sum B_i \\ \sum B_i & n \end{bmatrix} \begin{bmatrix} \alpha_{ij} \\ \beta_{ij} \end{bmatrix} = \begin{bmatrix} \sum B_i D_j \\ \sum D_j \end{bmatrix}$$

Solving this system yields the optimal transformation parameters.

2.3.6 Decompression Steps

The compression process involves the following steps:

1. **Block Division**: The image O is divided into smaller blocks.
2. **Block Matching**: Each source block B_i is matched with a similar destination block D_j by minimizing the error function.

3. **Storing Transformations**: The transformation parameters α_{ij} and β_{ij} for each source block are stored.

4. **Compression Representation**: The image is represented by the set of transformations without explicitly storing the pixel values.

2.3.7 Decompression Steps

The decompression process reconstructs the image by iteratively applying the stored transformations:

1. **Initialization**: Start with an initial guess for the image (typically a blank image or simple approximation).

2. **Iterative Transformation**: Apply the stored transformations iteratively to reconstruct the image.

3. **Convergence**: Due to the contraction property, the sequence of images generated by the transformations converges to the original image.

2.3.8 Reconstruction Error

The reconstruction error between the decompressed image I_{real} and the original image O is given by:

$$\text{Error}(I_{\text{real}}, O) = \frac{1}{m} \sum_{i=1}^{m} \| I_{\text{real}}(B_i) - O(B_i) \|^2$$

where $\| \cdot \|$ represents the norm (typically Euclidean distance) and m is the number of blocks in the image.

Algorithm 2 Fractal Image Compression Algorithm

Input: Image I of size $M \times N$ with pixel values
Output: Compressed representation consisting of transformations
Step 1: Initialization
Divide the image into non-overlapping blocks of size $B \times B$ (e.g., 8×8 or 16×16)
Initialize an empty list of transformations $T = []$
Step 2: Block Matching and Transformation Search
for each block B_d in the image (destination block) **do**
    Set the minimum error $E_{\text{min}} = \infty$
    Set the best transformation $T_{\text{best}} = \text{None}$
    for each block B_s in the image (source block) **do**
        for each possible transformation T (translation, scaling, rotation, flipping) **do**
            Apply transformation T to B_s to get transformed block B'_s
            Calculate the error between B_d and B'_s:

$$E_T = \sum_{i,j} (B_d[i,j] - B'_s[i,j])^2$$

            if $E_T < E_{\text{min}}$ **then**
                Set $E_{\text{min}} = E_T$
                Set $T_{\text{best}} = T$
            end if
        end for
    end for
    Store the transformation T_{best} for block B_d in the list of transformations T
end for
Step 3: Compression
for each transformation T_i in T **do**
    Store T_i as part of the compressed image representation
end for
Step 4: Decompression
Input: Compressed list of transformations T
Initialize an empty output image I_{out} of size $M \times N$
Initialize $I_{\text{out}} = 0$ (initialize the output image with zeros)
for each block B_d in the output image **do**
    Retrieve the transformation T_{best} for block B_d from T
    Apply transformation T_{best} to the corresponding source block B_s to generate transformed block B'_s
    Place B'_s at the corresponding position in I_{out}
end for
Output: Reconstructed image I_{out}

During the transformation, the brightness and contrast is added to the result of the affine transformation function in the previous section following the formula:

$$f(s) = brightness + s \times contrast$$

where s is the pixel at the current time.

2.4 Fractal Image Compression for RGB Images

Fractal compression for RGB images can simply be done by compressing and decompressing each channel of the RGB image, then resemble them.

3 Result

Figure 1: Test image

3.1 Affine Transformations for Grayscale Images

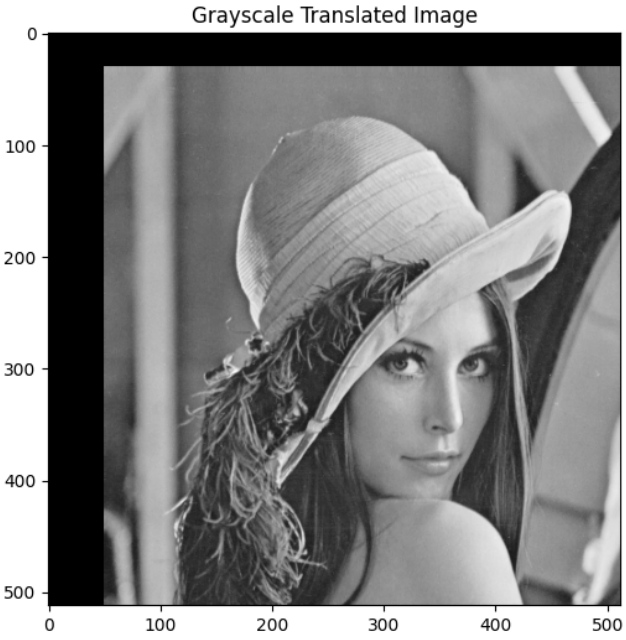

Figure 2: Grayscale transformation

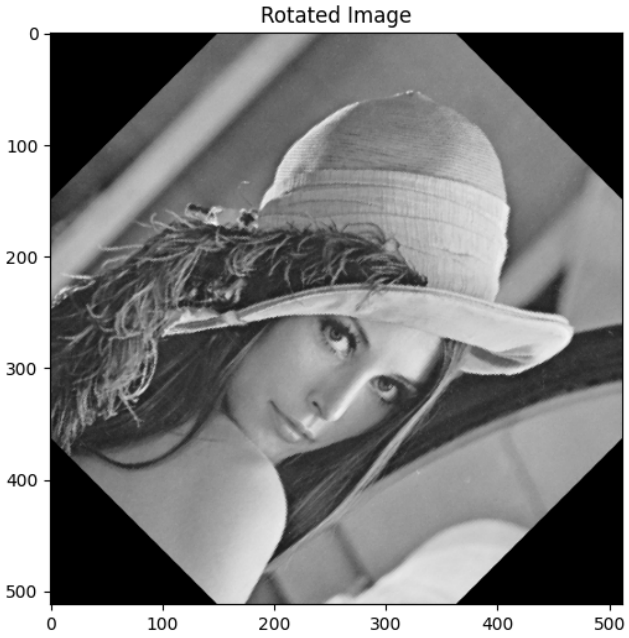

Figure 3: Grayscale rotation

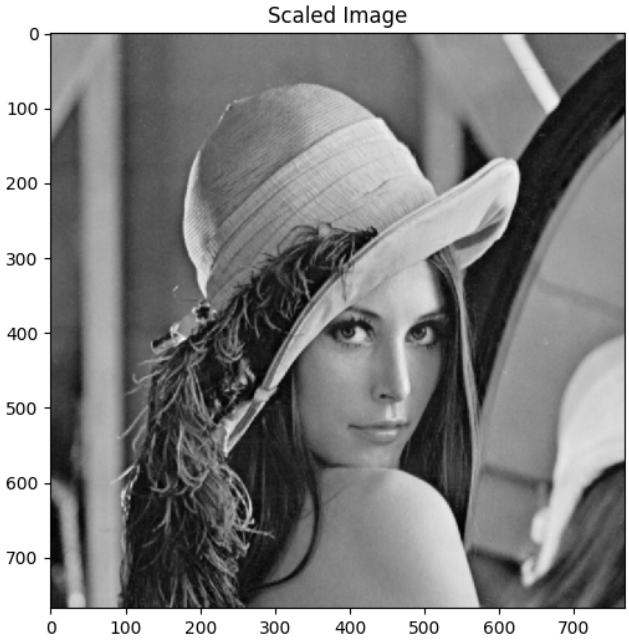

Figure 4: Grayscale scaling

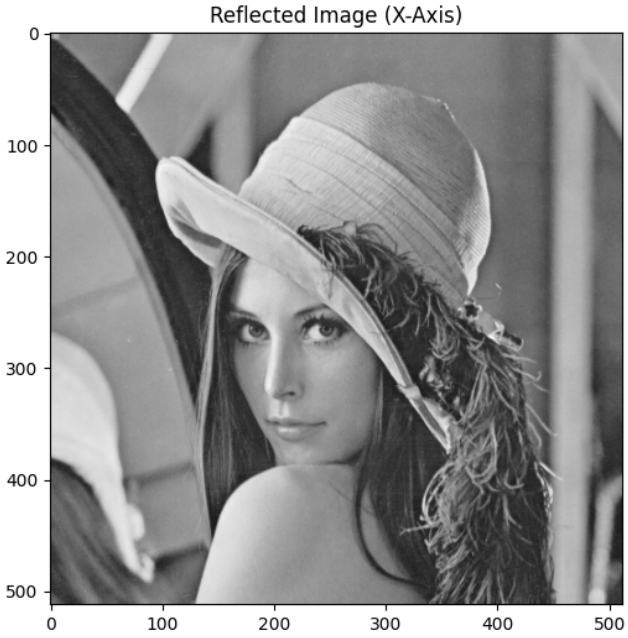

Figure 5: Grayscale reflection over x-axis

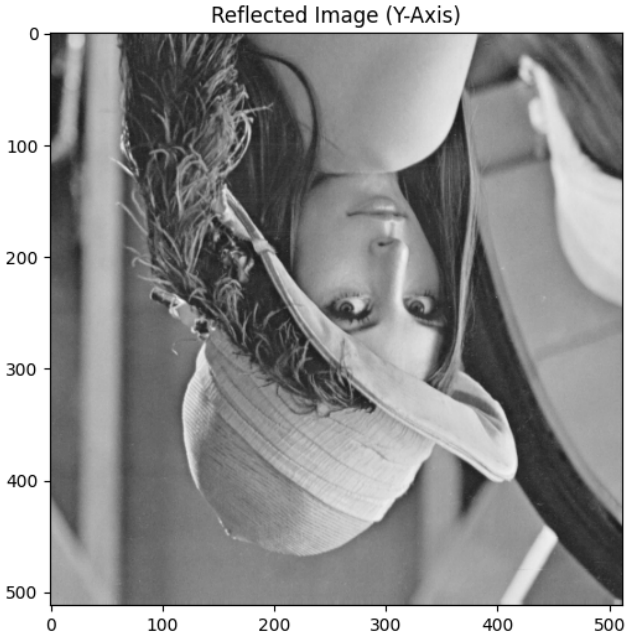

Figure 6: Grayscale reflection over y-axis

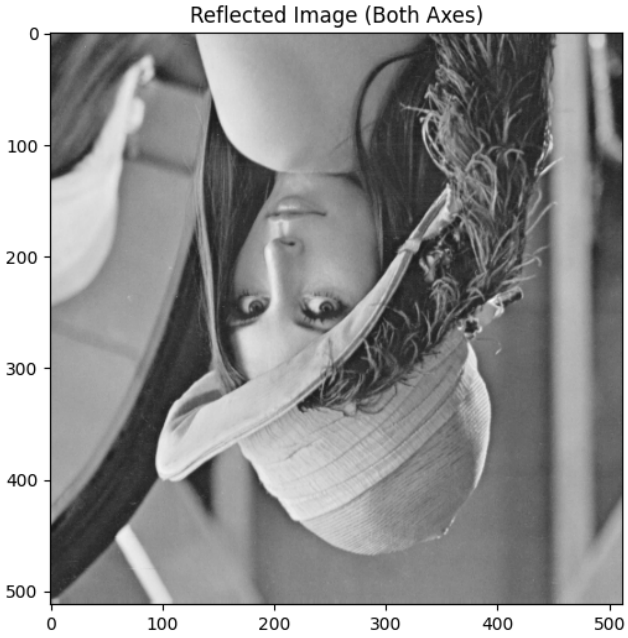

Figure 7: Grayscale reflection over both axis

3.2 Affine Transformation for RGB Images

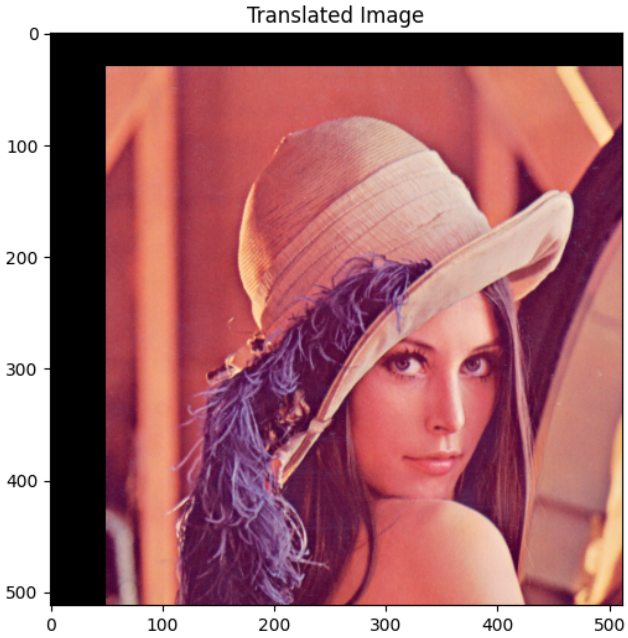

Figure 8: RGB translation

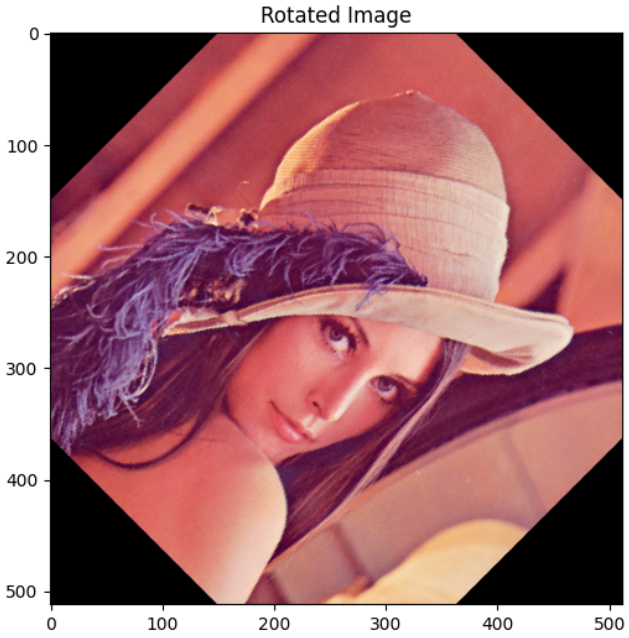

Figure 9: RGB rotation

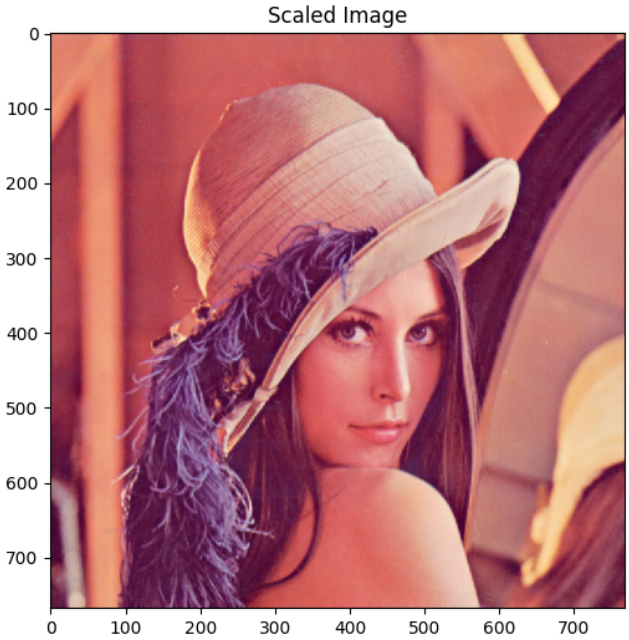

Figure 10: RGB scaling

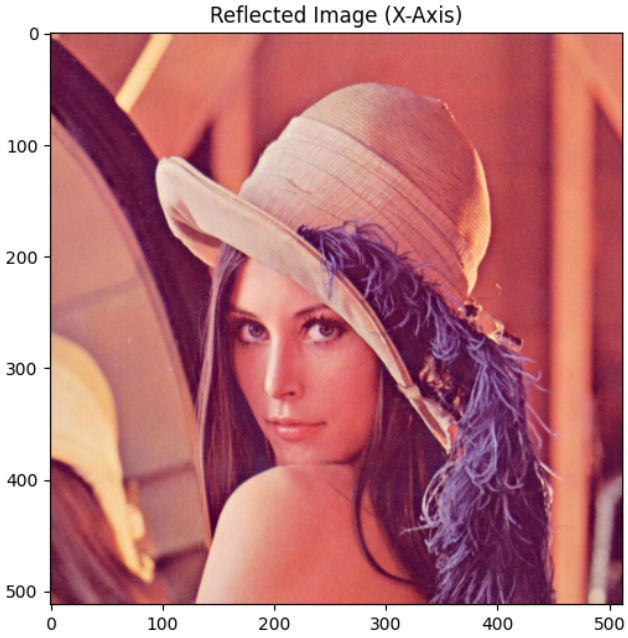

Figure 11: RGB reflection on x-axis

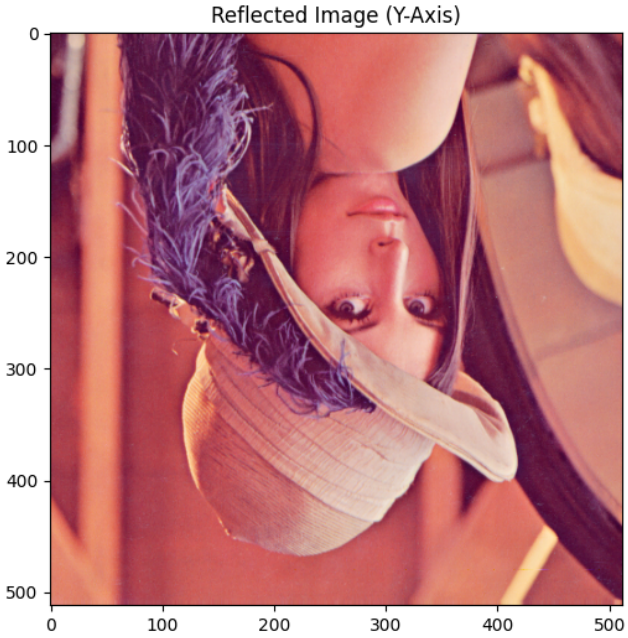

Figure 12: RGB reflection on y-axis

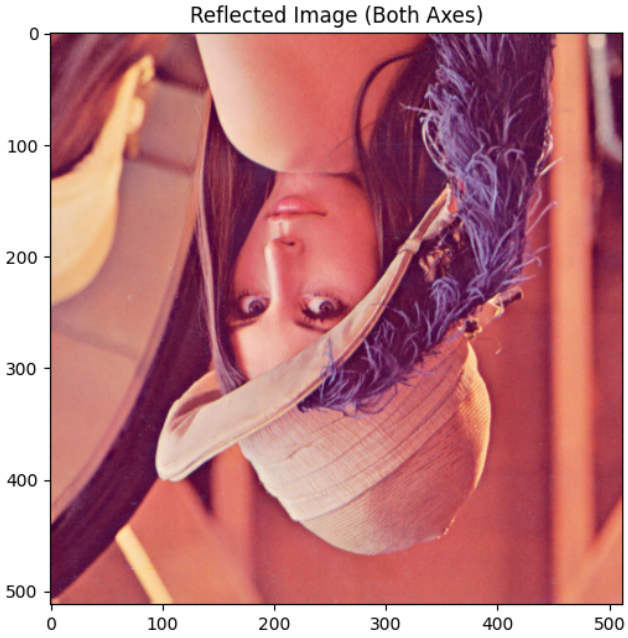

Figure 13: RGB reflection on both axis

3.3 Image Fractal Compression

3.3.1 Grayscale Image

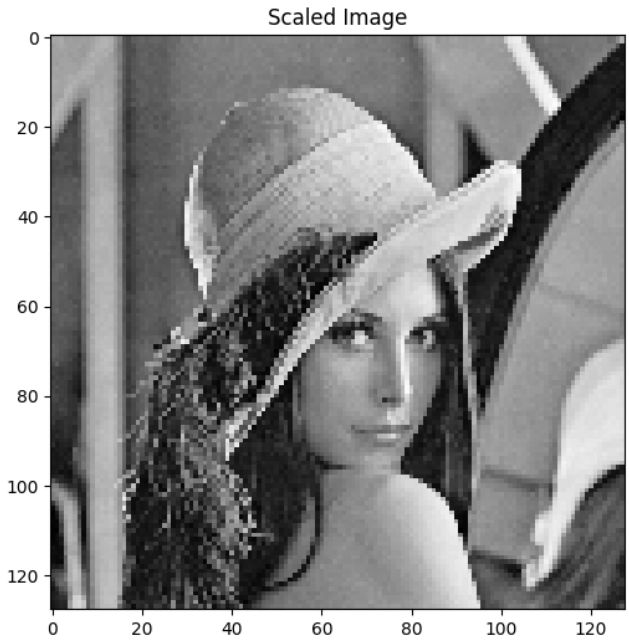

Figure 14: Resized input image

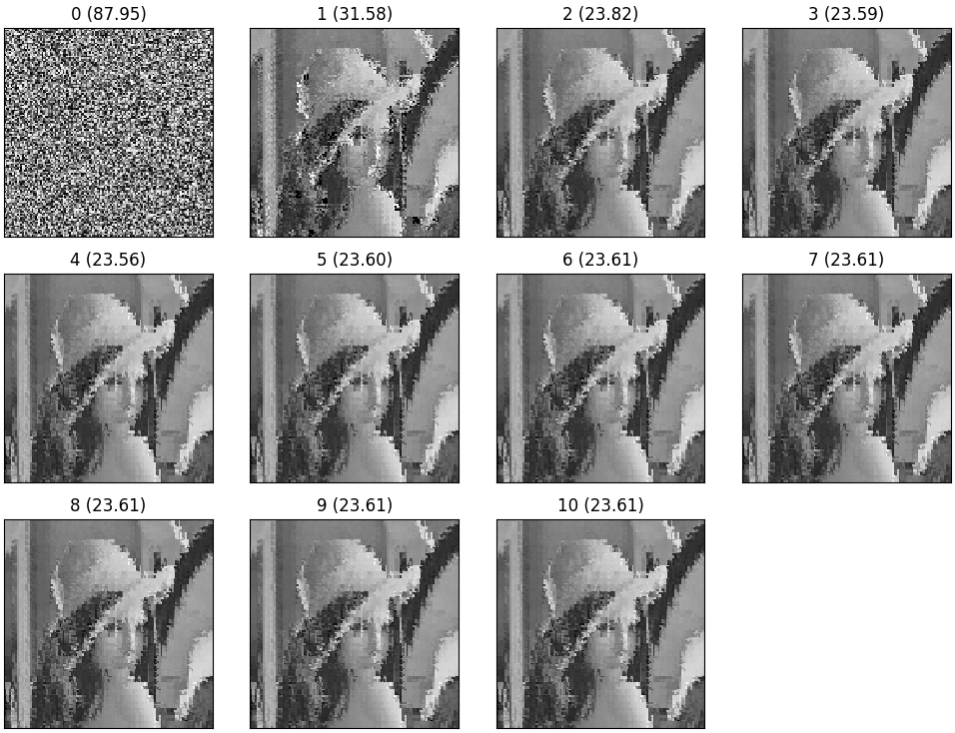

Figure 15: Grayscale fractal image compression iterations

3.3.2 RGB Image

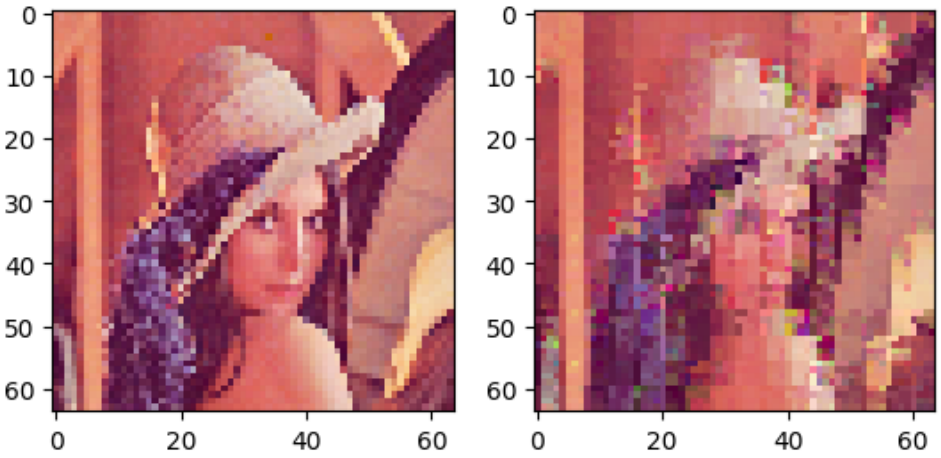

Figure 16: RGB image fractal image compression comparision

4 References

1. MathWorks - Affine Transformation
2. pvigier's blog
3. An Introduction to Fractal Image Compression - Texas Instruments Europe