## Requirements

Installing zLib headers and libraries, and so is libpng is required to compile this project. On Ubuntu, you can install it using the following command:

```
sudo apt-get install zlib1g-dev libpng-dev
```

## Compilation

To compile the code, use the following command:

```
clang main.c lib/*/*.c -lm -lz -lpng
```

If you prefer using the clang compiler. This will link all the necessary libraries, and produce an executable named `a.out`.

## Usage

Use the following command to run the program:

```
./a.out <input_image.png> <k>
```

Where `<input_image.png>` is the path to the input PNG image file, and `<k>` is the number of singular values to retain during compression.

## Output

The program will generate a compressed image file named `out.png` in the current directory.

**Mathematical workings and explanations can be found in the `math.md` file included in this repository.**

**To understand the code structure and implementation details, refer to `code.md`**

## References

- Low-rank Approximation
- Singular Value Decomposition

## Basic Concepts and Terminologies

Let

$$A_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} a_{ij} \end{bmatrix}$$

Represent a matrix of size m x n.

The transpose of matrix A is represented as

$$A^T_{n\times m} = \begin{bmatrix} a_{ji} \end{bmatrix}$$

## Singular Value Decomposition (SVD)

Any real matrix A of size m x n can be decomposed into three matrices U, S, and V such that

$$A = USV^T$$

Where,

- U is an m x m orthogonal matrix whose columns are the left singular vectors of A
- S is an m x n diagonal matrix with non-negative real numbers on the diagonal, known as singular values of A
- V is an n x n orthogonal matrix whose columns are the right singular vectors of A

## K Low Rank Approximation

To approximate a matrix A using its top k singular values, we can truncate the matrices U, S, and V to retain only the first k columns of U, the first k rows and columns of S, and the first k columns of V. The k-rank approximation of A is given by

$$A_k = U_{m\times k} S_{k\times k} V^T_{k\times n}$$

## Image Representation

A grayscale image can be represented as a 2D matrix where each element corresponds to the intensity of a pixel, 0 representing black and 255 representing white. For a color image, it can be represented as three 2D matrices (one for each color channel: Red, Green, and Blue).

## Image Compression using SVD

To compress an image using SVD, we can follow these steps:

1. Represent the image as a matrix A.
2. Perform SVD on matrix A to obtain U, S, and V.
3. Retain only the top k singular values in matrix S and set the rest to zero. This reduces the rank of the matrix and hence compresses the image.
4. Reconstruct the compressed image using the modified matrices U, S, and V:

   $$A_{compressed} = US_kV^T$$

   Where $S_k$ is the diagonal matrix with only the top k singular values retained.
5. The resulting matrix $A_{compressed}$ represents the compressed image.

## Mathematical Justification

The effectiveness of SVD in image compression lies in the fact that many images have a lot of redundancy, and the singular values often decay rapidly. By retaining only the top k singular values, we can capture most of the important features of the image while significantly reducing the amount of data needed to represent it. This leads to a compressed image that is visually similar (albeit lesser in quality and detail) to the original, but requires less storage space.

# Reading PNGs

Reading PNG files byte by byte requires the understanding of the following sections of the file

### PNG File Signature

The first 8 bytes of a PNG file are always the same and serve as a signature to identify the file as a PNG image. The signature bytes are:

```
137 80 78 71 13 10 26 10
```

### PNG Chunks

After our first 8 bytes, PNG files are organized into chunks, each consisting of four parts:

1. Length (4 bytes): Indicates the length of the chunk's data field.
2. Chunk Type (4 bytes): A 4-character code that identifies the type of chunk.
3. Chunk Data (variable length): The actual data of the chunk.
4. CRC (4 bytes): A cyclic redundancy check value for error-checking the chunk type and data. Due to time constraints I have not implemented CRC checking in my code yet.

### Important Chunk Types

- IHDR: The first chunk in a PNG file, containing image width, height, bit depth, color type, compression method, filter method, and interlace method.
- IDAT: Contains the actual image data, which is compressed.
- IEND: Marks the end of the PNG file.
- pHYS: Contains physical pixel dimensions.
- tEXt: Contains textual information.

### IHDR Chunk Structure

The IHDR chunk is 13 bytes long and contains the following fields:

- Width (4 bytes): The width of the image in pixels.
- Height (4 bytes): The height of the image in pixels.
- Bit Depth (1 byte): The number of bits per sample or per palette index.
- Color Type (1 byte): Indicates the color type of the image (e.g., grayscale, truecolor, indexed-color).
- Compression Method (1 byte): The method used to compress the image data (always 0 for PNG).
- Filter Method (1 byte): The method used to filter the image data (always 0 for PNG).
- Interlace Method (1 byte): Indicates whether the image is interlaced (0 for no interlace, 1 for Adam7 interlace).

Due to time constraints, I have implemented only the reading of PNG files with the following specifications:

- Color Type: 0 (Grayscale with no alpha channel)
- Bit Depth: 8 bits per sample (or 16 bits per sample)
- No interlacing (Interlace Method: 0)

### IDAT Chunk and Image Data

The IDAT chunk contains the compressed image data. The data is compressed using the `DEFLATE` algorithm. We use the `zlib` library to decompress this data due to time-constraints. The decompressed image data is organized into scanlines, each preceded by a filter type byte, ie we obtain a flattened 2D array of bytes representing the image. Each row contains a filter type byte followed by the pixel data for that row.

**Filter Types** PNG uses five filter types to improve compression:

- Type 0: None: No filtering is applied, ie each byte represents the actual pixel value.
- Type 1: Sub: Each byte is replaced by the difference between it and the corresponding byte of the prior pixel in the same row. `Raw(x) = (Filtered(x) + Raw(x-1)) % 256`
- Type 2: Up: Each byte is replaced by the sum of it and the corresponding byte of the prior pixel in the same column. `Raw(x) = (Filtered(x) + Raw(x-Width)) % 256`
- Type 3: Average: Each byte is replaced by the average of the corresponding bytes of the prior pixel in the same row and the prior pixel in the same column. `Raw(x) = (Filtered(x) + (Raw(x-1) + Raw(x-Width)) / 2) % 256`
- Type 4: Paeth: Each byte is replaced by the result of the Paeth predictor, which chooses the pixel value that is closest to the sum of the values of the three neighboring pixels. `Raw(x) = (Filtered(x) + PaethPredictor(Raw(x-1), Raw(x-Width), Raw(x-Width-1))) % 256`

The Paeth predictor (coded in python here) is calculated as follows:

```python
def PaethPredictor(a, b, c):
    p = a + b - c
    pa = abs(p - a)
    pb = abs(p - b)
    pc = abs(p - c)
    if pa <= pb and pa <= pc:
        return a
    elif pb <= pc:
        return b
```

```
    else:
        return c
```

### IEND Chunk

The IEND chunk marks the end of the PNG file. It has no data and is always the last chunk in the file.

### Other Chunks

Other chunks like pHYS and tEXt can be present in PNG files, but they are not essential for reading the image data and can be ignored for basic PNG reading functionality.

## Performing SVD for the obtained matrix

To decompose the image matrix using Singular Value Decomposition (SVD), I have used the following algorithm:

1. Compute the covariance matrix $C = A^T A$, where $A$ is the image matrix.
2. Compute the eigenvalues and eigenvectors of the covariance matrix $C$ using the Jacobi method.
3. Sort the eigenvalues in descending order and arrange the corresponding eigenvectors accordingly.
4. The right singular vectors ($V$) are the eigenvectors of $C$.
5. The singular values ($\Sigma$) are the square roots of the eigenvalues of $C$.
6. Compute the left singular vectors ($U$) using the Gram-Schmidt process on the set of vectors $\{Av_i/\sigma_i\}$, where $v_i$ are the right singular vectors and $\sigma_i$ are the singular values.

### Jacobi Method for Eigenvalue Decomposition

The Jacobi method is an iterative algorithm used to compute the eigenvalues and eigenvectors of a symmetric matrix (this works because $A^{\mathrm{T}}A$ is symmetric). The algorithm works by performing a series of rotations to zero out the off-diagonal elements of the matrix. The steps are as follows:

1. Initialize the eigenvector matrix as the identity matrix.
2. Repeat until convergence:
    1. Find the largest off-diagonal element in the matrix.
    2. Compute the rotation angle to zero out this element.
    3. Apply the rotation to the matrix and update the eigenvector matrix.
3. The diagonal elements of the resulting matrix are the eigenvalues, and the columns of the eigenvector matrix are the corresponding eigenvectors.

Here we define the algorithm to converge when the maximum off-diagonal element is less than a small threshold value (e.g., $1 \times 10^{-12}$).

### Gram-Schmidt Process

The Gram-Schmidt process is used to orthogonalize a set of vectors. To compute the left singular vectors ($U$), we apply the Gram-Schmidt process to the set of vectors $\{Av_i/\sigma_i\}$:

1. Start with the set of vectors $\{Av_1/\sigma_1, Av_2/\sigma_2, \ldots, Av_n/\sigma_n\}$.
2. For each vector in the set:
    1. Subtract the projections of the vector onto all previously computed orthogonal vectors.
    2. Normalize the resulting vector to obtain an orthonormal vector.
3. The resulting set of orthonormal vectors forms the columns of the matrix $U$.

### K Low-Rank Approximation

To obtain a rank-$k$ approximation of the original image matrix $A$, we retain only the top $k$ singular values and their corresponding singular vectors:

1. Construct the diagonal matrix $\Sigma_k$ by keeping only the top $k$ singular values and setting the rest to zero.
2. Construct the matrices $U_k$ and $V_k$ by retaining only the first $k$ columns of $U$ and $V$, respectively.
3. The rank-$k$ approximation of the original matrix $A$ is given by:

$$A_k = U_k \Sigma_k V_k^T$$

## Saving the compressed image as PNG

To save the compressed image as a PNG file, we need to reverse the steps taken during the reading process:

1. **Reconstruct the Image Data**: Using the rank-$k$ approximation, reconstruct the image matrix.
2. **Apply PNG Filters**: Due to time constraints, I have implemented only the "None" filter (Type 0) for simplicity. We can obtain higher compression ratios by implementing other filter types in the future.
3. **Compress the Image Data**: Use the `zlib` library to compress the filtered image data using the DEFLATE algorithm.
4. **Create PNG Chunks**: Construct the necessary PNG chunks (IHDR, IDAT, IEND) with the appropriate data.
5. **Write to File**: Write the PNG signature followed by the constructed chunks to a new PNG file.

## References


- PNG Specification (Second Edition)

- Jacobi Method
- Gram-Schmidt Process

# Results

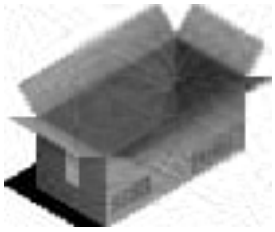
I have used my program on 4 images, 3 being the ones given and one of my own for debugging.

The original images can be found in the `figs/imgs` folder, while the output images are in the `figs/outs` folder.

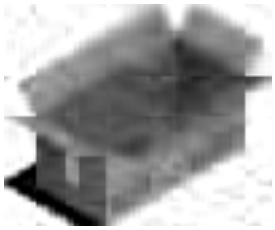
## Image 1 (test.png)

Original Image:

Compressed Image ($k = 20$):



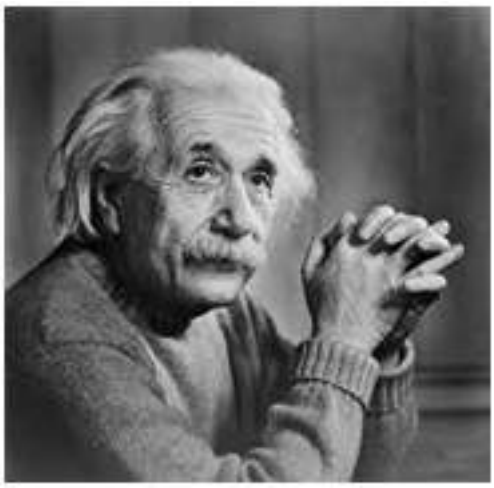
Compressed Image ($k = 10$):



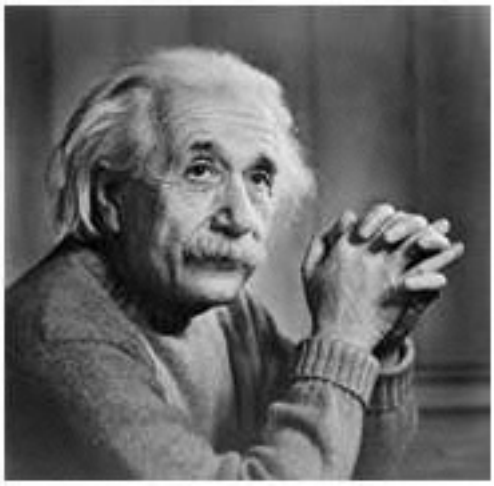
Here I have used the value $k = 20$ and $k = 10$. It is clear that using a lower value of $k$ gives us a higher lossy compression. Using $k \geq 80$ results in a compressed image that is very similar to the original image, as this is a `100x80` image.

## Image 2 (einstein.png)

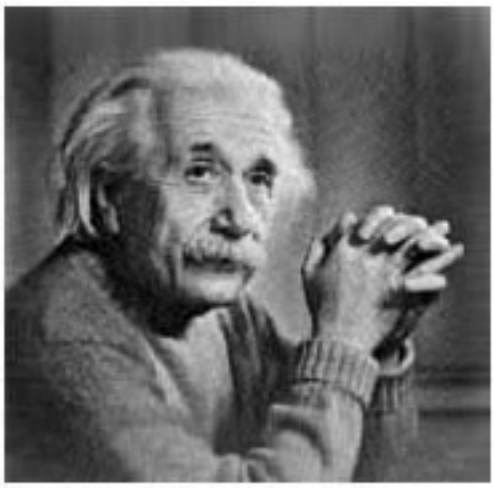
Original Image:



Compressed Image ($k = 80$):



Compressed Image ($k = 40$):



Compressed Image ($k = 30$):

Compressed Image ($k = 20$):

Compressed Image ($k = 10$):

### Image 3 (globe.png)

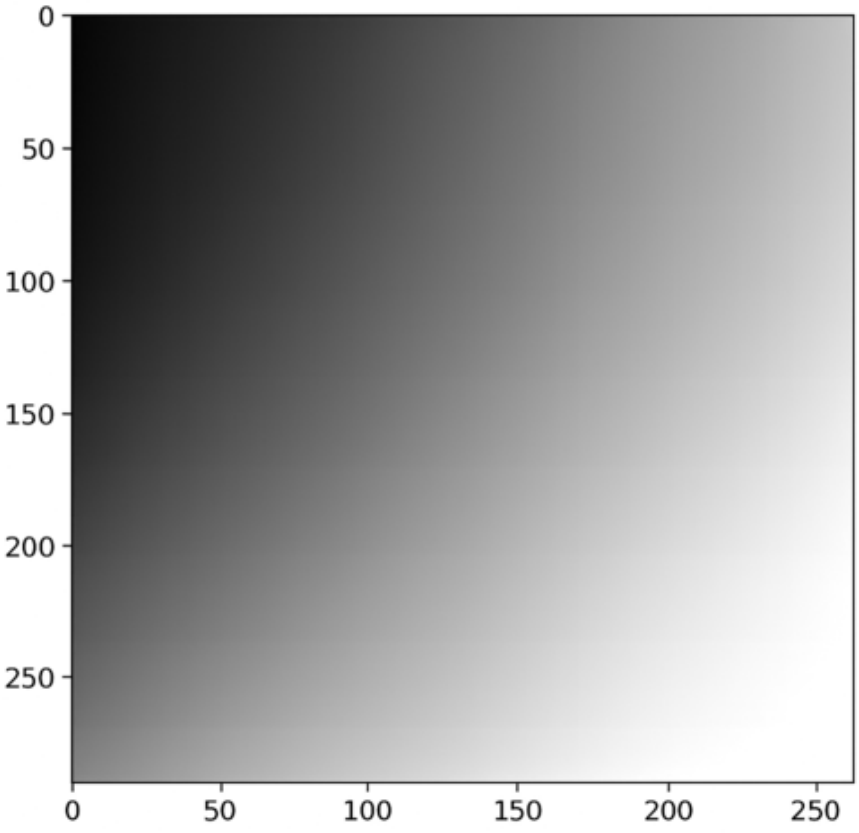
Original Image:

Compressed Image ($k = 20$):

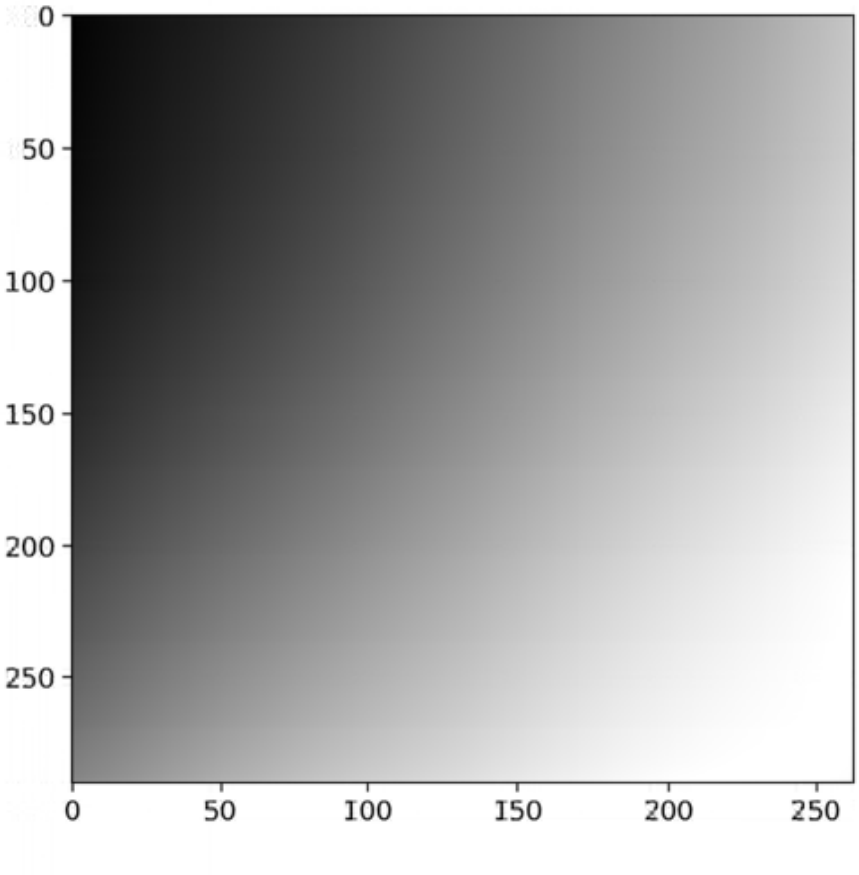
Compressed Image ($k = 10$):

**Image 4 (greyscale.png)**

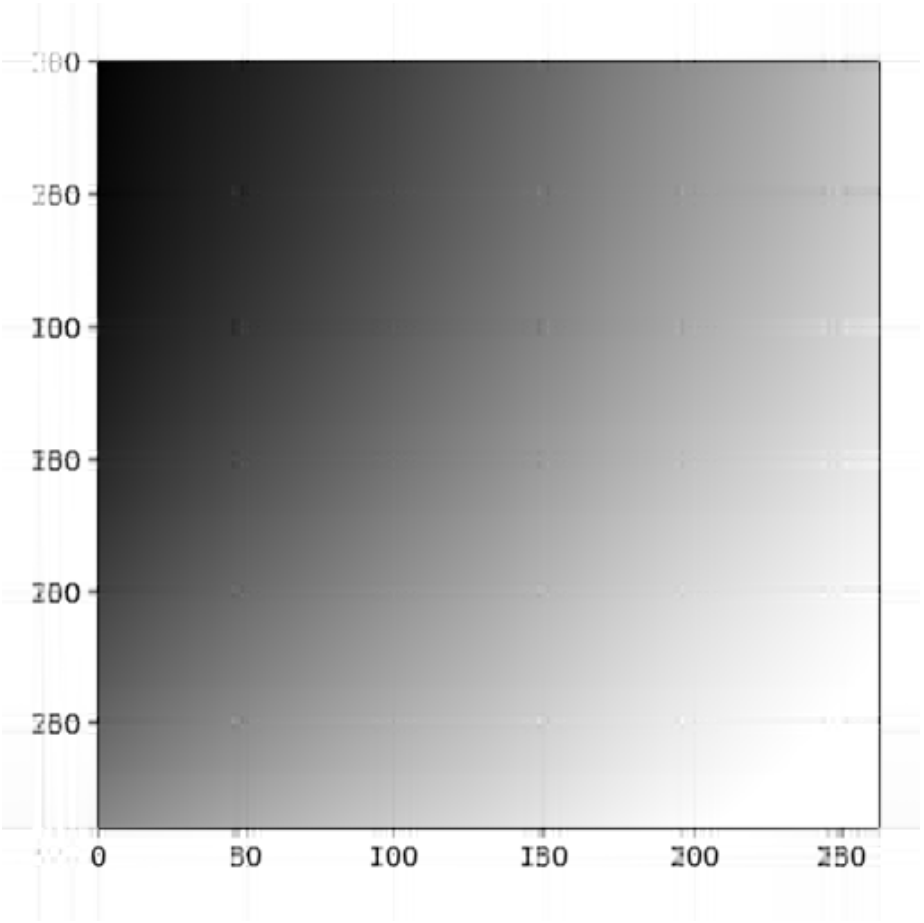
Original Image:




Compressed Image ($k = 20$):




Compressed Image ($k = 10$):




## Observations

Clearly, there is an inverse relationship between the value of $k$ and the amount of compression. A lower value of $k$ results in a higher compression, but also a loss in image quality.

# Tables and data analysis

Vivaan Parashar - AI25BTECH11040

November 8, 2025

Note: $\|\mathbf{A}\|$ of a matrix $\mathbf{A}$ denotes its Frobenius norm, ie:

$$\|\mathbf{A}\| = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

## 1 Tables

### 1.1 Error by Frobenius norm

| Image name | $k$ | $\|\mathbf{A} - \mathbf{A}_c\|$ |
|---|---|---|
| einstein.png | 80 | 363.79527 |
| einstein.png | 40 | 1168.53798 |
| einstein.png | 30 | 1572.21786 |
| einstein.png | 20 | 2129.10169 |
| einstein.png | 10 | 3250.57103 |
| globe.png | 20 | 3259.34411 |
| globe.png | 10 | 4934.48761 |
| greyscale.png | 20 | 1012.42629 |
| greyscale.png | 10 | 2512.75645 |
| test.png | 20 | 1004.19470 |
| test.png | 10 | 1546.89140 |

Table 1: Table with images and error

### 1.2 Error by Frobenius norm per pixel

| Image name | $k$ | Resolution | Pixels | $\frac{\|\mathbf{A}-\mathbf{A}_c\|}{\text{Pixels}}$ |
|---|---|---|---|---|
| einstein.png | 80 | 186x182 | 33852 | 0.01075 |
| einstein.png | 40 | 186x182 | 33852 | 0.03452 |
| einstein.png | 30 | 186x182 | 33852 | 0.04644 |
| einstein.png | 20 | 186x182 | 33852 | 0.06289 |
| einstein.png | 10 | 186x182 | 33852 | 0.09602 |
| globe.png | 20 | 300x314 | 94200 | 0.03460 |
| globe.png | 10 | 300x314 | 94200 | 0.05238 |
| greyscale.png | 20 | 512x512 | 100000 | 0.00386 |
| greyscale.png | 10 | 512x512 | 100000 | 0.00959 |
| test.png | 20 | 100x80 | 8000 | 0.12099 |
| test.png | 10 | 100x80 | 8000 | 0.18637 |

Table 2: Previous table, but with error per number of pixels

## 2 Plots and analysis

### 2.1 Frobenius error

From Fig. 1, we can see that as $k$ increases, the error decreases. This is expected, as a higher value of $k$ means more singular values are retained in the compressed image, leading to a closer approximation of the original image. The relationship appears to be non-linear, with diminishing returns as $k$ increases. One thing to notice though is that this is not consistent across image sizes, which makes sense. As a larger resolution image has a greater number of entries, both its Frobenius norm and the Frobenius norm of the error array should be higher as more entries are included in the sum, and there is no term (for example dividing by the number of pixels) to account for the size of the arrays.

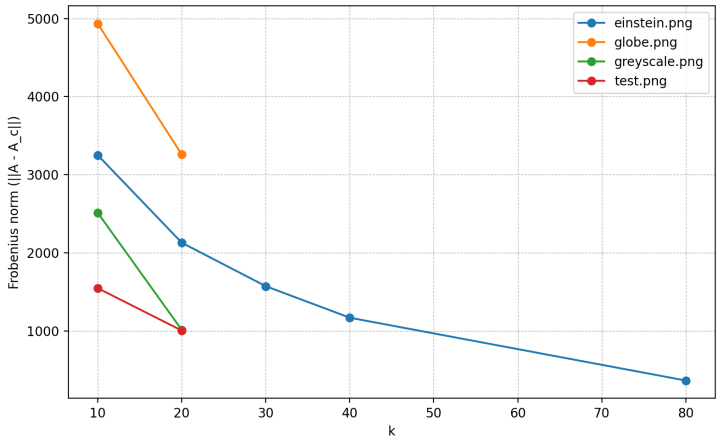



Figure 1: Error by Frobenius norm

## 2.2 Frobenius error per pixel

From Fig. 2, we can see that as $k$ increases, the error decreases, and this is uniform for all images. This is expected for the same reasons, except in this case no two lines intersect, whereas previously the lines represented by `greyscale.png` and `test.png` seemed to intersect.

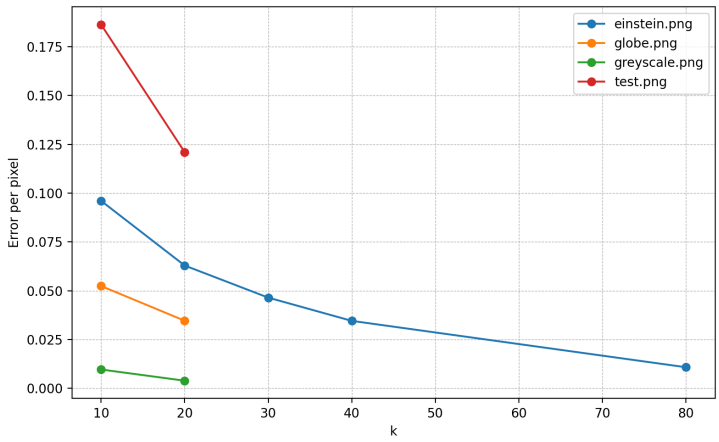



Figure 2: Error by Frobenius norm per pixel