Mini Project Report

Image Compression Using Truncated SVD

Submitted by: Rayidi Manohar Roll No: AI25BTECH11028

Date: November 8,2025

Summary of Gilbert Strang Lecture on Singular Value Decomposition(SVD)

Singular Value Decomposition (SVD) is the best way to factorize a matrix. According to SVD; Every matrix (A) can be written as

$$(A) = (U)(\Sigma)(V^T)$$

where (U) and (V) are orthogonal matrices and also orthonormal matrices, and (Σ) is a diagonal matrix that holds special values called **singular values**.
The main idea is that any matrix can be viewed as a vector multiplication of simpler matrices. First, the matrix turns the input space into new directions using (V^T). it stretches or compresses these directions by the singular values in (Σ). Finally, it rotates the result using (U). This process shows exactly how the matrix acts on space.

Mathematical Steps

To find the SVD of a matrix (A):

1) Compute (A^TA). This matrix is symmetric and positive semi-definite.
2) Find the eigenvalues and eigenvectors of (A^TA).
3) The square roots of the eigenvalues give the singular values. These are placed on the diagonal of (Σ).
4) The eigenvectors of (A^TA) become the columns of (V).
5) The columns of (U) are found using

$$(u_i) = \frac{1}{\sigma_i}(A)(v_i)$$

where (v_i) is a column of (V) and (σ_i) is the corresponding singular value.

Geometrically: SVD shows how a matrix changes shapes in space. stretches or compresses these directions by the singular values in (Σ). Finally, it rotates the result using (U). This process shows exactly how the matrix acts on space. So SVD gives a complete geometric picture of what the matrix does.

Truncated SVD and Compression

If we take only the first few largest singular values and their corresponding columns of $\left(U\right)$ and $\left(V\right)$, we get an approximate version of the original matrix:

$$\left(A_k\right) = \left(U_k\right)\left(\Sigma_k\right)\left(V_k^T\right)$$

This is called the **truncated SVD**. It keeps the most important information and removes small details. In image compression, this means we can keep the main structure of the image while using much less data.

how it is used in this Project

In this project, a gray-scale image is treated as a matrix of pixel values. Using truncated SVD, we keep only the top k singular values to build a smaller version of the image that still looks almost the same. The C program performs SVD, and reconstructs a compressed image using only those k values. This helps to reduce the storage size.

Conclusion

The truncated SVD gives a smart way to compress images by keeping only the most useful parts of the data.

Explanation of the Implemented Algorithm

The algorithm implemented in this program is a **Truncated Singular Value Decomposition (SVD)** using **Power Iteration with Deflation**. It is an efficient method to approximate the top k singular values and corresponding singular vectors of a matrix $A \in \mathbb{R}^{m\times n}$, which is useful for applications such as image compression, dimensionality reduction, and noise filtering.

Mathematical Formulation

A matrix $\left(A\right)$ can be decomposed using SVD as:

$$\left(A\right) = \left(U\right)\left(\Sigma\right)\left(V^T\right)$$

where

- $\left(U\right) \in \mathbb{R}^{m\times m}$ contains the left singular vectors,
- $\left(S\right) \in \mathbb{R}^{m\times n}$ is a diagonal matrix of singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$,
- $V \in \mathbb{R}^{n\times n}$ contains the right singular vectors.

For many appulications, it is enough to compute a *truncated SVD*, keeping only the top k singular values:

$$\left(A\right) \approx \left(U_k\right)\left(S_k\right)\left(V_k^T\right)$$

where $\left(U_k\right) \in \mathbb{R}^{m\times k}$, $\left(S_k\right) \in \mathbb{R}^{k\times k}$, and $\left(V_k\right) \in \mathbb{R}^{n\times k}$. This reduces storage and computational cost while preserving most of the information in $\left(A\right)$.

Power Iteration:

The top singular vectors can be approximated using an iterative method:

1) Initialize a random vector $(v) \in \mathbb{R}^n$.
2) Compute the left singular vector:

$$(u) = \frac{(A)(v)}{\|(A)(v)\|}$$

3) Compute the right singular vector:

$$(v) = \frac{(A^T)(u)}{\|(A^T)(u)\|}$$

4) Repeat steps 2–3 for several iterations until convergence.
5) Compute the singular value:

$$\sigma = \|(A)(v)\|$$

6) Remove the contribution of the current component (deflation):

$$(A) \leftarrow (A) - \sigma\,(u)\,(v^T)$$

Repeat this process for the next singular value until the top k singular values and vectors are computed.

Pseudocode

```
Start

Open the PGM file.
Read the header info (type, size, max value).
Load all pixel bytes into a 2D array.

Ask for k (how many singular values to keep).

Create arrays for U, S, and V.

For each component from 0 to k-1:
    Pick a random starting vector.
    Run a few power-iteration steps:
        u = A * v        -> make unit length
        v = A^T * u      -> make unit length
    Compute sigma from A * v.
    Store sigma, u, and v in S, U, and V.
    Remove this rank-1 part from A.

Rebuild an approximation of the image using U, S, V.
Clamp pixel values to the valid range.
```

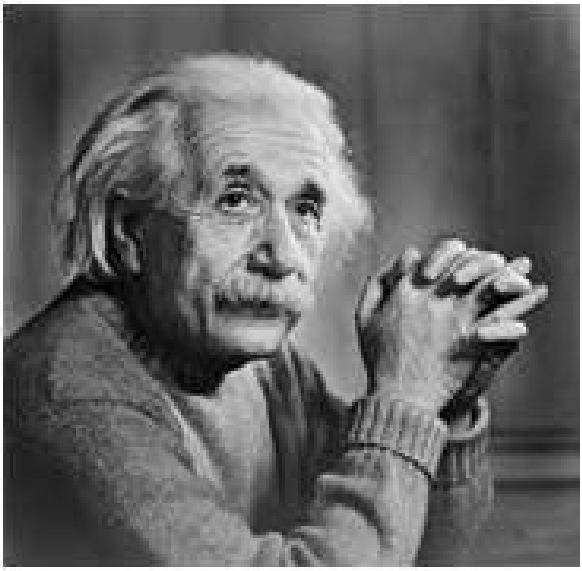


Fig. 6.1: einstein

```
Compute the Frobenius error between the original and the reconstructed imag
Show the error.

End
```

Intuition

- **Power Iteration:** Alternative the multiplication by A and A^T shows the dominant singular vectors without computing the full SVD.so,we this method is better because no need to calculate complete svd we can directly calculate truncated svd
- **Normalization:** Keeps the vectors u and v unit-length to ensure convergence.
- **Deflation:** Removes the already computed singular component from A to find the next largest singular vector.
- **Truncated SVD:** By keeping only k singular values, we can compress an image or reduce dimensionality while retaining most of the important information.

This method is efficient and works well for large matrices, in image compression tasks, where retaining only the top singular values significantly reduces storage of image.

1 Reconstructed Images for different k

Error Analysis

The error analysis is important to measure how much information is lost we analyze the accuracy of the image after performing truncated Singular Value Decomposition (SVD)

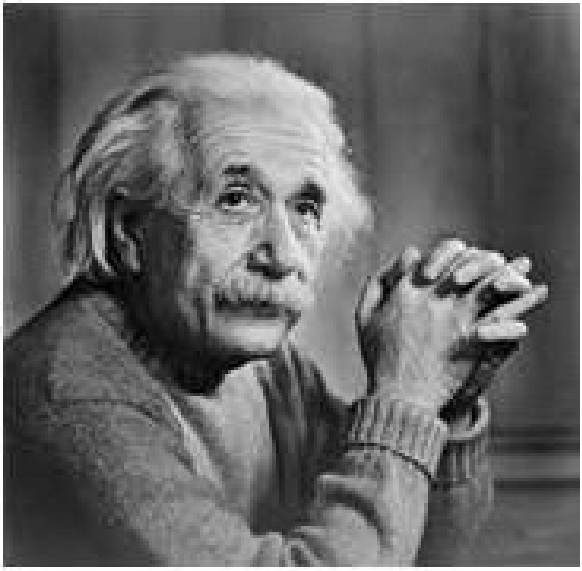


Fig. 6.2: einstein with k=100

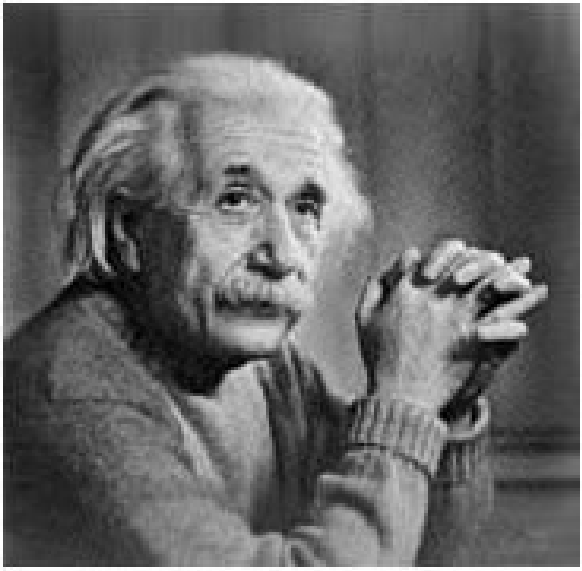


Fig. 6.3: einstein with k=50

Fig. 6.4: einstein with k=20

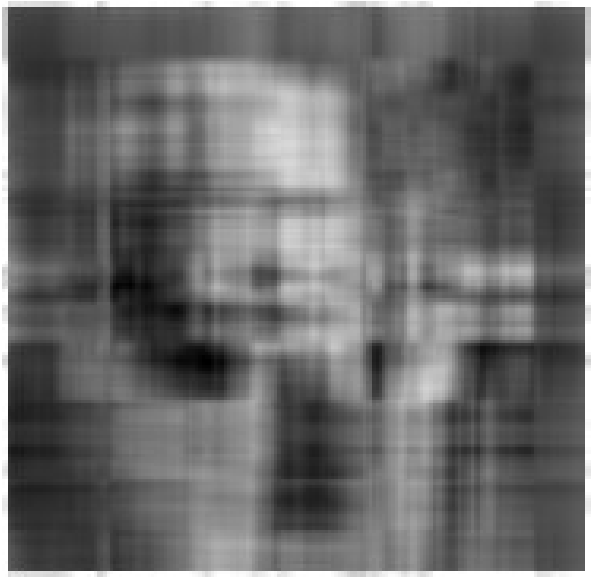


Fig. 6.5: einstein with k=5

Fig. 6.6: globe

Fig. 6.7: globe with k=100

Fig. 6.8: globe with k=50

Fig. 6.9: globe with k=20

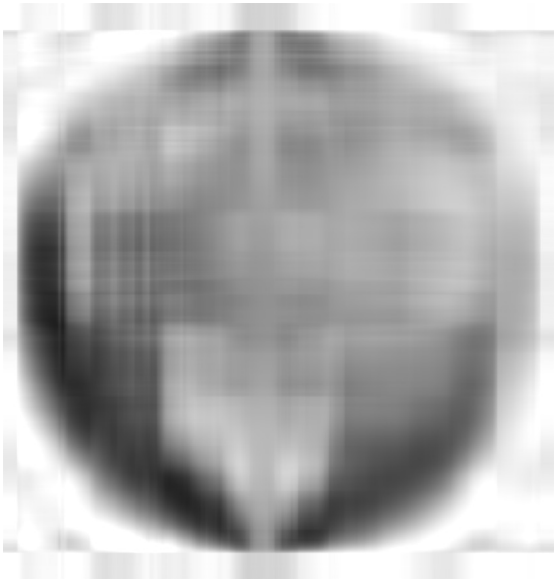


Fig. 6.10: globe with k=5

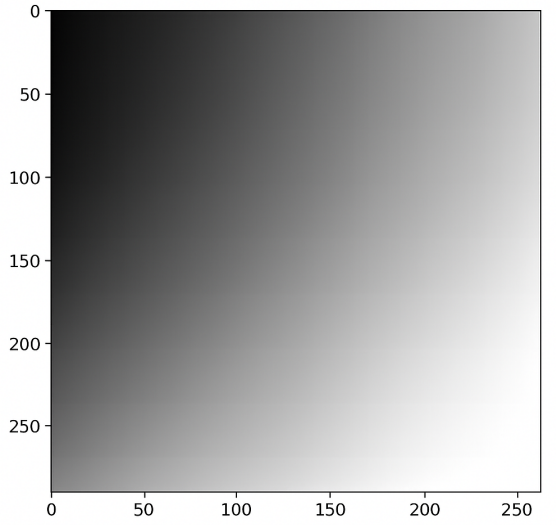


Fig. 6.11: greyscale

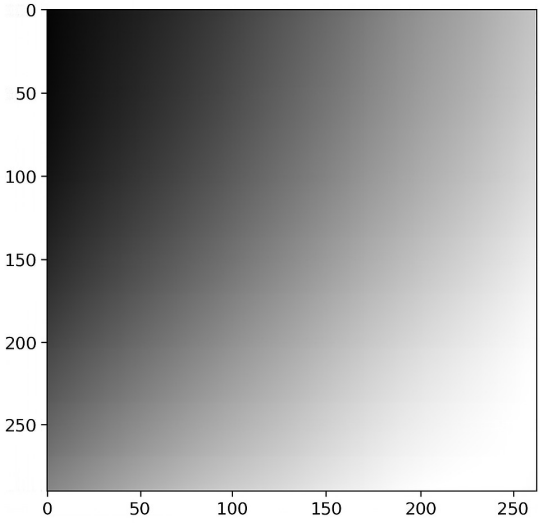


Fig. 6.12: greyscale with k=50

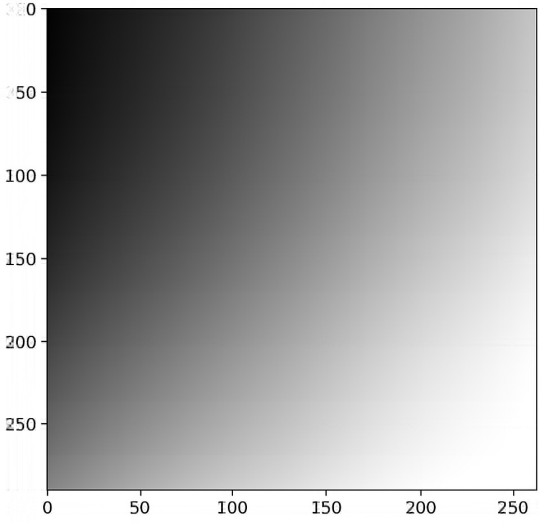


Fig. 6.13: greyscale with k=20

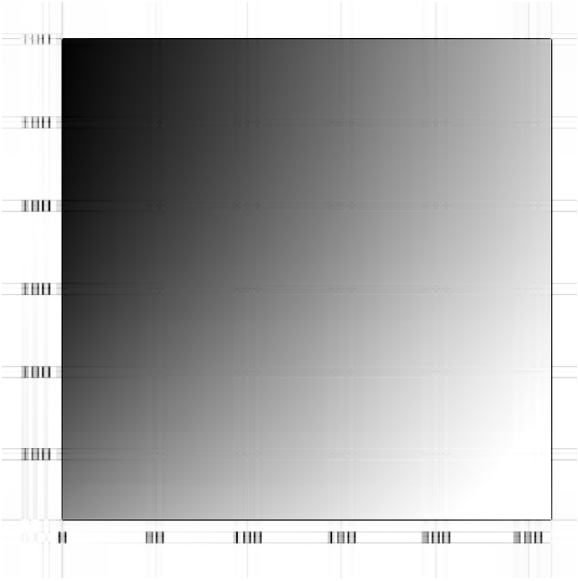


Fig. 6.14: greyscale with k=5

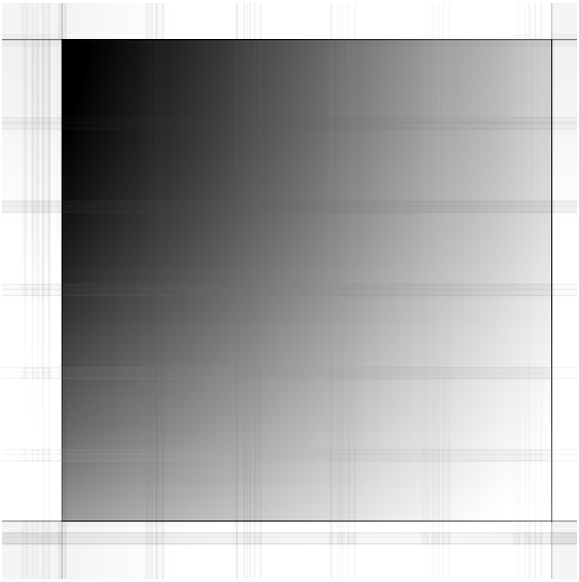


Fig. 6.15: greyscale with k=2

with varying numbers of singular values (k). the **Frobenius norm** measures the overall difference between the original image matrix $\left(A\right)$ and the reconstructed image matrix $\left(A_k\right)$.

Frobenius Norm Error:

The Frobenius norm of the error matrix is defined as:

$$E_F = \|\left(A\right) - \left(A_k\right)\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij} - \hat{A}_{ij})^2}$$

This metric gives a single scalar value representing the total pixel-wise deviation between the original and reconstructed images. A lower Frobenius norm indicates that the reconstructed image is closer to the original, implying better compression quality. As k increases, more singular values and vectors are used in reconstruction, leading to reduced error but increased computational cost and storage.

Compression Ratio:= it is the ratio of original size of the file to size of the file after truncated svd. Therefore,Compression ratio for a matrix $\left(A\right)$ of size m x n is defined as :

$$C.R = \frac{m * n}{k(m + n + 1)};$$

The table below summarizes the Frobenius norm errors obtained for different values of k for each of the three images.

TABLE 6: Frobenius Norm Error for Different Values of k for Einstein Image

Number of Singular Values (k)	**Frobenius Error**	**Compression ratio**
100	164.856348	0.9174
50	880.547952	1.8348
20	2126.747429	4.5870
5	21809.1015	18.3480

TABLE 6: Frobenius Norm Error for Different Values of k for Globe Image

Number of Singular Values (k)	**Frobenius Error**	**Compression Ratio**
100	3673.577547	4.2928
50	6185.836458	8.5856
20	10635.167780	21.4640
5	20703.892815	85.8558

To visualize the effect of truncation, reconstructed images for different values of k are shown above. As k increases, the visual quality of the reconstructed image improves, and fine details become clearer.

Observations

From the results, it can be observed that:

- For small values of k, the reconstruction is blurry and the Frobenius error is high.

TABLE 6: Frobenius Norm Error for Different Values of k for Greyscale Image

Number of Singular Values (k)	**Frobenius Error**	**Compression Ratio**
50	1190.836021	10.2350
20	3819.334979	25.5875
5	11155.921013	102.3500
2	17171.694821	255.8751

- As k increases, the reconstructed image retains more structural and intensity details, and the error decreases.

Hence, an optimal k can be chosen to balance between image quality and compression efficiency.

Discussion of Trade-offs and Reflections on Implementation Choices

Implementation Choices

In this project, we implemented truncated Singular Value Decomposition (SVD) for image compression.

- Using power iteration algorithm for truncated SVD.
- making functions for PGM I/O, matrix operations, normalization, SVD, reconstruction.
- Storing compressed data in a binary '.bin' format containing only U_k, S_k, V_k, instead of saving the reconstructed image directly.

Trade-offs Observed

1. The power iteration SVD is simple and easy to understand, but it requires multiple iterations per singular vector. It is suitable for small to medium-sized images but may be slow for very large images.

Reflections:

- The choice of k is a key parameter that directly affects both storage efficiency.
- Saving compressed matrices separately allows future reconstruction and experimentation with different k values without recomputing the SVD.if we do it with the original matrix without making a copy then we may get the repeated values of errors.so,we need to use a copy of matrices.

so,this project implements low-rank image compression using SVD.

Optimal Extension:

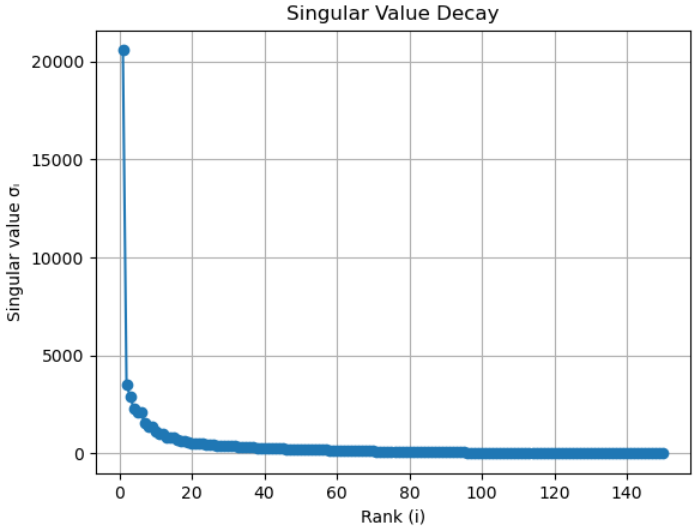
Visualising singular values to show how image information is distributed across ranks

Visualization of Singular Values and Information Distribution

To understand how information is distributed within the image, we plotted the singular values obtained from the Singular Value Decomposition (SVD). For an image matrix A, the SVD is written as

$$A = \sum_{i=1}^{r} \sigma_i \, u_i \, v_i^T ,$$

where σ_i are the singular values (ordered as $\sigma_1 \geq \sigma_2 \geq \cdots$), u_i and v_i are the left and right singular vectors, and $r = \text{rank}(A)$.



By plotting the singular values σ_i against their rank index i, we can observe that how much information each rank contributes. In the plot, the singular values decay rapidly at the beginning and then gradually level off. The first few singular values are significantly larger, indicating that these components take most of the important image information. therefore, the most of the information of the image is concentrated in the largest singular values, the image can be approximated well by keeping only the first k components: Thus, the singular value visualization clearly shows that image information is not evenly distributed across all ranks. The early ranks contain the majority of the information, while higher ranks contribute very little. This tells why low-rank SVD approximation is an effective method for image compression.

Codes:

Complete C Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
double **read(char *filename, int *rows, int *cols, int *maxVal, char *format) {
    FILE *fp = fopen(filename, "rb");
    if (fp == NULL) { printf("Error: Cannot open %s\n", filename);
```

```
            exit(1); }

    fscanf(fp, "%2s", format);
    int c = fgetc(fp);
    while (c == '#')
    { while (fgetc(fp) != '\n'); c = fgetc(fp); }
    ungetc(c, fp);

    fscanf(fp, "%d %d", cols, rows);
    fscanf(fp, "%d", maxVal);
    fgetc(fp);

    double **imgdata = (double **)malloc((*rows) * sizeof(double *));
    for (int i = 0; i < *rows; i++)
    {imgdata[i] = (double *)calloc(*cols, sizeof(double));}

        for (int i = 0; i < *rows; i++)
        {
            for (int j = 0; j < *cols; j++)
            {
                unsigned char v;
                fread(&v,1,1,fp);
                imgdata[i][j] = v;
            }
        }


    fclose(fp);
    return imgdata;
}
void write(char *file, double **img, int rows, int cols, int maxVal, char *format)
{
    FILE *fp = fopen(file, "wb");
    if (fp == NULL)
    {
            printf("Error: Cannot write to %s\n", file);
            exit(1);
    }
    fprintf(fp, "%s\n%d %d\n%d\n", format, cols, rows, maxVal);
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            int val = (int)round(img[i][j]);
            if (val < 0) val = 0;
```

```
            if (val > maxVal) val = maxVal;
                unsigned char c = val;
                fwrite(&c,1,1,fp);
        }
    }
    fclose(fp);
}
double **copymat(double **A, int row, int cols)
{
    double **B = (double **)malloc(row * sizeof(double *));
    for (int i = 0; i < row; i++)
    {
        B[i] = (double *)malloc(cols * sizeof(double));
        for (int j = 0; j < cols; j++)
        {B[i][j] = A[i][j];}
    }
    return B;
}
void matVec(double **A, double *v, double *res, int r, int c)
{
    for (int i = 0; i < r; i++)
    {
        double s = 0.0;
        for (int j = 0; j < c; j++)
        {s += A[i][j] * v[j];}
        res[i] = s;
    }
}

void normalize(double *v, int n) {
    double s = 0;
    for (int i = 0; i < n; i++)
        {s += v[i]*v[i];}
    double norm = sqrt(s);
    if (norm < 1e-12) return;
    for (int i = 0; i < n; i++)
        {v[i] /= norm;}
}

void simpleSVD(double **A, int r, int c, double **U, double *S, double **V, int k) {
    for (int comp = 0; comp < k; comp++) {
        double *v = (double *)malloc(c * sizeof(double));
        for (int i = 0; i < c; i++) {v[i] = (double)rand() / RAND_MAX;}

        for (int it = 0; it < 20; it++) {
```

```
            double *u = (double *)calloc(r, sizeof(double));
            matVec(A, v, u, r, c);
            normalize(u, r);

            double *newV = (double *)calloc(c, sizeof(double));
            for (int j = 0; j < c; j++)
                for (int i = 0; i < r; i++)
                    newV[j] += A[i][j] * u[i];
            normalize(newV, c);

            for (int i = 0; i < c; i++) v[i] = newV[i];
            free(u); free(newV);
        }

        double *Av = (double *)calloc(r, sizeof(double));
        matVec(A, v, Av, r, c);
        double sigma = 0;
        for (int i = 0; i < r; i++) sigma += Av[i]*Av[i];
        sigma = sqrt(sigma);
        S[comp] = sigma;

        for (int i = 0; i < r; i++) U[i][comp] = (sigma == 0 ? 0 : Av[i] / sigma);
        for (int i = 0; i < c; i++) V[i][comp] = v[i];

        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++)
                A[i][j] -= sigma * U[i][comp] * V[j][comp];

        free(Av); free(v);
    }
}

void reconstruct(double **U, double *S, double **V, double **R, int r, int c, int k) {
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++) {
            double s = 0;
            for (int t = 0; t < k; t++) s += S[t] * U[i][t] * V[j][t];
            R[i][j] = s;
        }
}
void frobeniuserror(double **orig, double **recon, int r, int c) {
    double sum = 0.0;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++) {
            double d = orig[i][j] - recon[i][j];
```

```
            sum += d * d;
        }
    double err = sqrt(sum);
    printf("\nFrobenius norm of reconstruction error: %.6f\n", err);
}
void compressionratio(int row, int col, int k) {
    double ogSize = (double)(row * col);
    double newSize = (double)(row * k + col * k + k);
    double ratio = ogSize / newSize;

    printf("\nCompression ratio : %.4f\n", ratio);
}


int main() {
    char inFile[200], outFile[200], fmt[3];
    int rows, cols, maxVal, k;

    printf("Enter input PGM file: "); scanf("%199s", inFile);
    double **A_orig = read(inFile, &rows, &cols, &maxVal, fmt);
    double **A_work = copymat(A_orig, rows, cols);
printf("Enter number of singular values to keep (k): "); scanf("%d", &k);
    if (k > cols) k = cols;

    double **U = (double **)malloc(rows * sizeof(double *));
    double **V = (double **)malloc(cols * sizeof(double *));
    for (int i = 0; i < rows; i++) U[i] = (double *)calloc(k, sizeof(double));
    for (int i = 0; i < cols; i++) V[i] = (double *)calloc(k, sizeof(double));
    double *S = (double *)calloc(k, sizeof(double));
    simpleSVD(A_work, rows, cols, U, S, V, k);
FILE *sf = fopen("singular_values.txt", "w");
for (int i = 0; i < k; i++)
    fprintf(sf, "%d %f\n", i+1, S[i]);
fclose(sf);

printf("Singular values saved to singular_values.txt\n");

    double **R = (double **)malloc(rows * sizeof(double *));
    for (int i = 0; i < rows; i++) R[i] = (double *)calloc(cols, sizeof(double));
    reconstruct(U, S, V, R, rows, cols, k);
        printf("Enter output PGM file: "); scanf("%199s", outFile);
    write(outFile, R, rows, cols, maxVal, fmt);
    frobeniuserror(A_orig, R, rows, cols);
    compressionratio(rows,cols,k);
    for (int i = 0; i < rows; i++) { free(A_orig[i]); free(A_work[i]); free(U[i]); free(R
```

```
        [i]); }
    for (int i = 0; i < cols; i++) free(V[i]);
    free(A_orig); free(A_work); free(U); free(V); free(R); free(S);
        return 0;
}
```