Software Assignment: Image compression

EE25BTECH11041 - Naman Kumar

I. Introduction-Singular Values Decomposition

The SVD expresses any (rectangular or square) matrix $\mathbf{A}$ as a product of two orthogonal matrices and a diagonal matrix of non-negative scalars:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{1}$$

where $\mathbf{U}$ and $\mathbf{V}$are orthogonal (or unitary) and $\mathbf{\Sigma}$ is diagonal with non-negative entries called singular values. The decomposition always exists and generalizes the eigen-decomposition of symmetric positive (semi)definite matrices; for symmetric positive definite matrices the two orthogonal factors coincide.

- **Form and interpretation:**
  - $\mathbf{U}$ contains an orthonormal basis of the column space (left singular vectors).
  - $\mathbf{V}$ contains an orthonormal basis of the row space / domain (right singular vectors).
  - $\mathbf{\Sigma}$ has singular values $\sigma_1 \geq \sigma_2 \geq .. \geq 0$ on the diagonal; these are scale factors that show how A stretches unit vectors.
  - Geometrically, SVD finds orthonormal bases so that $\mathbf{A}$ maps each right singular vector $\mathbf{v_i}$ to a scaled left singular vector:$\mathbf{A}\mathbf{v_i} = \sigma_i\mathbf{u_i}$. Null-space directions corresponds to zero singular values.
- **Relation to the four fundamental sub-spaces:** The SVD arranges orthonormal bases for the row space, column space, null space and left-null space so $\mathbf{A}$ becomes diagonal between the row and column basis; the number of nonzero singular values equals the rank r.
- **Practical remarks**:
  - SVD is a canonical factorization-exists for every matrix-and is central because it simultaneously diagonalizes the action of $\mathbf{A}$ between specially chosen orthonormal bases in domain and co-domain.
  - It unifies ideas about orthogonality, eigen decomposition of symmetric positive matrices, and the geometry of linear maps.

There are various algorithms present to approximate eigenvalues of matrices. The best algorithms considering general $n \times n$ matrix are as following.

II. Jacobi Method for Eigenvalue and Singular Value Decomposition

A. Introduction

The Jacobi method is an iterative algorithm used to compute the eigenvalues and eigenvectors of a real symmetric matrix. It relies on a sequence of orthogonal similarity transformations that progressively eliminate off-diagonal elements of the matrix until it becomes diagonal. The diagonal entries then represent the eigenvalues, and the accumulated orthogonal transformations yield the corresponding eigenvectors.
For the Singular Value Decomposition (SVD), a *one-sided Jacobi method* is used, which operates directly on the original matrix A rather than on A^TA or AA^T, making it numerically stable and straightforward to implement.

Algorithm	Advantages	Disadvantages
QR Algorithm with Householder + Wilkinson Shift	• Very fast and reliable for symmetric or Hermitian matrices. • Wilkinson shift accelerates convergence significantly. • Householder reflections ensure numerical stability.	• Complex to code manually (requires multiple matrix factorizations). • Computationally expensive for large matrices due to repeated QR decompositions.
Divide-and-Conquer Algorithm	• Highly efficient for large symmetric matrices. • Well-suited for parallel computation and modern libraries.	• Complicated to implement from scratch. • Less stable for closely spaced or small eigenvalues. • Requires tridiagonal form preprocessing.
Jacobi Algorithm (One-Sided for SVD)	• Easiest to implement in C â requires only basic matrix operations. • High numerical stability due to use of orthogonal rotations. • Intuitive conceptually (pairwise annihilation of off-diagonal terms). • Produces accurate eigenvalues and eigenvectors without forming A^TA explicitly.	• Converges slowly for large or ill-conditioned matrices. • Not the fastest for very large datasets compared to QR-based methods.
Bisection Method	• Reliable for finding specific eigenvalues within an interval. • Efficient for tridiagonal matrices.	• Does not compute eigenvectors. • Slow convergence without accurate interval estimation. • Limited to symmetric matrices.
Standard QR Algorithm (without Shift)	• Simpler than the shifted version. • Works for general (non-symmetric) matrices.	• Much slower convergence. • Poor efficiency for large-scale problems.

TABLE 0: Comparison of eigenvalue and SVD algorithms with emphasis on implementation difficulty and stability. The Jacobi method, though slower, is preferred for its simplicity and robustness in C-based image compression applications.

B. Mathematical Foundation

Consider a real symmetric matrix $A \in \mathbb{R}^{n\times n}$. The goal is to find an orthogonal matrix V and a diagonal matrix Λ such that

$$A = V\Lambda V^T,$$

where Λ contains the eigenvalues and the columns of V are the orthonormal eigenvectors of A.
The Jacobi method achieves this by successively applying plane rotations that eliminate one off-diagonal element at a time. At each iteration, the matrix is transformed as:

$$A' = J^T AJ,$$

where J is a **Jacobi rotation matrix**, defined as an identity matrix modified in the (p, q)-plane:

$$J = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & c & s & & \\ & & -s & c & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix},$$

where the parameters $c = \cos\theta$ and $s = \sin\theta$ are chosen such that the (p, q) and (q, p) elements of A' become zero.
The rotation angle θ is determined using:

$$\tan(2\theta) = \frac{2A_{pq}}{A_{qq} - A_{pp}}.$$

Once θ is found, the updated elements of A are computed as:

$$\begin{aligned} A'_{pp} &= c^2 A_{pp} - 2csA_{pq} + s^2 A_{qq}, \\ A'_{qq} &= s^2 A_{pp} + 2csA_{pq} + c^2 A_{qq}, \\ A'_{pq} &= A'_{qp} = 0, \\ A'_{ik} &= A_{ik} \text{ for } i, k \neq p, q. \end{aligned}$$

Each rotation maintains the symmetry of A and preserves orthogonality. After multiple sweeps through the matrix (annihilating all off-diagonal terms within a tolerance ε), A converges to a diagonal matrix.

C. Algorithm Steps

1) Start with a symmetric matrix $A \in \mathbb{R}^{n \times n}$ and initialize $V = I_n$.
2) Repeat until convergence:
   a) Find the largest off-diagonal element A_{pq}.
   b) Compute $\tan(2\theta) = \dfrac{2A_{pq}}{A_{qq} - A_{pp}}$ and obtain $c = \cos\theta,\ s = \sin\theta$.
   c) Form the Jacobi rotation matrix $J(p, q, \theta)$.
   d) Update $A \leftarrow J^T A J$ and accumulate eigenvectors: $V \leftarrow VJ$.
3) The diagonal elements of A are the eigenvalues, and the columns of V are the eigenvectors.

D. One-Sided Jacobi Method for SVD

For non-symmetric or rectangular matrices $A \in \mathbb{R}^{m \times n}$, the **one-sided Jacobi SVD** method seeks orthogonal matrices U and V such that:

$$A = U\Sigma V^T,$$

where Σ is diagonal with non-negative entries.
The algorithm orthogonalizes the columns of A pairwise using Givens rotations, ensuring $A^T A$ becomes diagonal without forming it explicitly. This avoids potential numerical instability caused by squaring the condition number.

E. Convergence and Accuracy

The Jacobi method converges quadratically for symmetric matrices, ensuring high numerical accuracy. Although it converges more slowly than the QR algorithm for large-scale problems, its stability and simplicity make it ideal for small to medium-sized matrices, educational purposes, and implementations in low-level languages like C.

F. Advantages and Applications

- Conceptually simple and easy to implement without advanced linear algebra libraries.
- Produces highly accurate eigenvalues and orthogonal eigenvectors.
- Well-suited for symmetric or nearly symmetric matrices.
- Extensively used in image compression and principal component analysis (PCA) due to its stable orthogonal transformations.

G. Limitations

- Slow convergence for large matrices compared to QR or divide-and-conquer algorithms.
- Requires multiple sweeps for convergence, increasing computational time.

H. Summary

The Jacobi method offers a balance between numerical stability and implementation simplicity. While not the fastest, it is an excellent choice for cases where accuracy and clarity of computation outweigh raw speed, such as in educational implementations and image compression systems built in C.

III. Error analysis - JPEG

Value of k	Error in C	Error in Python
1	39045.21	38989.92
5	20501.10	20410.96
10	14918.67	14791.37
25	9355.04	9162.20
50	6124.03	5837.31
100	3630.36	3149.08
125	2969.53	2371.01
150	2469.45	1716.63
200	1775.52	198.63

TABLE 3: Error reduction for JPEG compression using Jacobi-based SVD in C and Python.

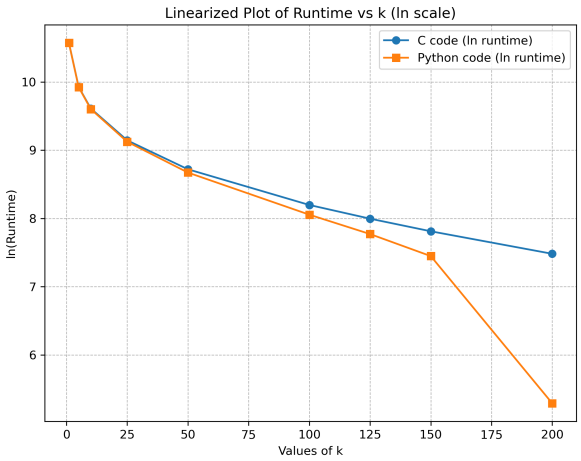

Fig. 3: Error in approximation with C and Python

Value of k	Error in C	Error in Python
1	39045.21	38989.92
5	20501.10	20410.96
10	14918.67	14791.37
25	9355.04	9162.20
50	6124.03	5837.31
100	3630.36	3149.08
125	2969.53	2371.01
150	2469.45	1716.63
200	1775.52	198.63

TABLE 3: Error reduction for JPEG compression using Jacobi-based SVD in C and Python.

IV. Error analysis - PNG

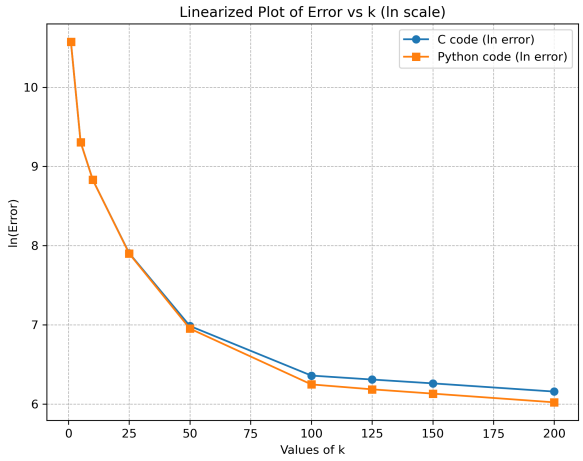

Fig. 3: Error in approximation with C and Python

V. Time to compress - JPEG

Value of k	Time Taken by C (s)	Time Taken by Python (s)
1	118.92	2.13
5	115.26	2.14
10	111.41	2.12
25	112.26	2.14
50	106.07	2.19
100	118.62	2.12
125	117.52	2.20
150	117.56	2.30
200	126.46	2.30

TABLE 3: Runtime comparison for JPEG compression using Jacobi-based SVD in C and Python.

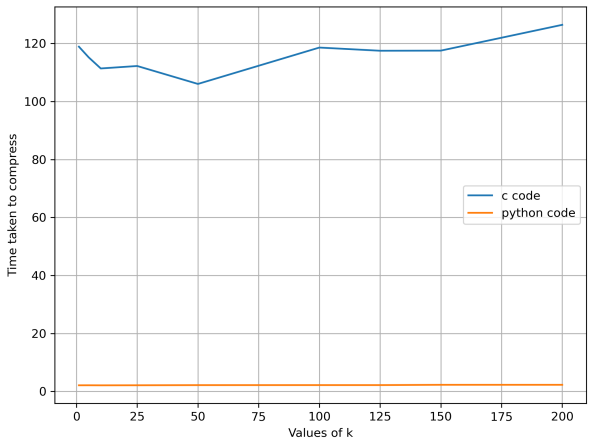

Fig. 3: Runtime comparison for JPEG compression in C and Python.

VI. Time to compress - JPEG

Value of k	Time Taken by C (s)	Time Taken by Python (s)
1	118.92	2.13
5	115.26	2.14
10	111.41	2.12
25	112.26	2.14
50	106.07	2.19
100	118.62	2.12
125	117.52	2.20
150	117.56	2.30
200	126.46	2.30

TABLE 3: Runtime comparison for JPEG compression using Jacobi-based SVD in C and Python.

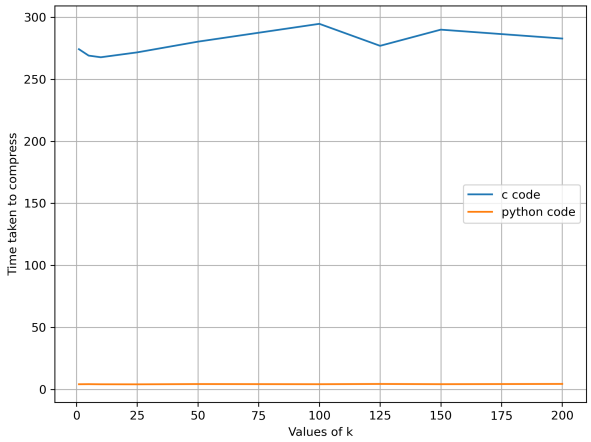

Fig. 3: Runtime comparison for JPEG compression in C and Python.

VII. C-Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <jpeglib.h>
#include <time.h>
#include <png.h>

#define eigens 25
#define error 1e-8

int matrix_multiply(const unsigned char *mat1, int w1, int h1,
                    const unsigned char *mat2, int w2, int h2,
                    int *result)
{
    if (w1 != h2) return 1;

    for (int i = 0; i < h1; i++) {
        for (int j = 0; j < w2; j++) {
            int sum = 0;
            for (int k = 0; k < w1; k++) {
                sum += mat1[i * w1 + k] * mat2[k * w2 + j];
            }
            result[i * w2 + j] = sum;
        }
    }
    return 0;
}


void transpose(const unsigned char *mat, int w, int h, int *result) {
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            result[j * h + i] = mat[i * w + j];
        }
    }
}
//jpeg read
unsigned char *read_image(const char *filename, int *width, int *height) {
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
    FILE *f = fopen(filename, "rb");
    if (!f) { perror("open"); return NULL; }

    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, f);
    jpeg_read_header(&cinfo, TRUE);
    cinfo.out_color_space = JCS_GRAYSCALE;
```

```
    jpeg_start_decompress(&cinfo);

    *width = cinfo.output_width;
    *height = cinfo.output_height;

    unsigned char *pixels = malloc((*width) * (*height));
    unsigned char *row = malloc(*width);

    while (cinfo.output_scanline < cinfo.output_height) {
        jpeg_read_scanlines(&cinfo, &row, 1);
        int y = cinfo.output_scanline - 1;
        for (int x = 0; x < *width; x++)
            pixels[y * (*width) + x] = row[x];
    }

    free(row);
    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose(f);

    return pixels;
}
//write jpeg
int write_jpeg(const char *filename, unsigned char *gray_data, int width, int height) {//,
    FILE *outfile = fopen(filename, "wb");
    if (!outfile) {
        perror("Cannot open output JPEG file");
        return 1;
    }

    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;

    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_compress(&cinfo);
    jpeg_stdio_dest(&cinfo, outfile);

    cinfo.image_width = width;
    cinfo.image_height = height;
    cinfo.input_components = 1;
    cinfo.in_color_space = JCS_GRAYSCALE;

    jpeg_set_defaults(&cinfo);
    //jpeg_set_quality(&cinfo, quality, TRUE);
    jpeg_start_compress(&cinfo, TRUE);

    int row_stride = width;
    while (cinfo.next_scanline < cinfo.image_height) {
        unsigned char *row_pointer = &gray_data[cinfo.next_scanline * row_stride];
        jpeg_write_scanlines(&cinfo, &row_pointer, 1);
```

```
    }

    jpeg_finish_compress(&cinfo);
    jpeg_destroy_compress(&cinfo);
    fclose(outfile);

    return 0;
}
//read pgm
unsigned char *read_pgm(const char *filename, int *width, int *height)
{
    FILE *f = fopen(filename, "rb");
    if (!f) {
        perror("Cannot open PGM file");
        return NULL;
    }

    char format[3];
    if (fscanf(f, "%2s", format) != 1) {
        fprintf(stderr, "Invalid PGM header\n");
        fclose(f);
        return NULL;
    }

    if (format[0] != 'P' || (format[1] != '5' && format[1] != '2')) {
        fprintf(stderr, "Unsupported PGM format (use P2 or P5)\n");
        fclose(f);
        return NULL;
    }
    int c = fgetc(f);
    while (c == '#') {
        while (fgetc(f) != '\n');
        c = fgetc(f);
    }
    ungetc(c, f);

    int maxval;
    if (fscanf(f, "%d %d %d", width, height, &maxval) != 3) {
        fprintf(stderr, "Invalid PGM metadata\n");
        fclose(f);
        return NULL;
    }
    fgetc(f);

    unsigned char *data = malloc((*width) * (*height));
    if (!data) {
        perror("malloc");
        fclose(f);
        return NULL;
    }
```

```
    if (format[1] == '5') {
        fread(data, 1, (*width) * (*height), f);
    } else {
        for (int i = 0; i < (*width) * (*height); i++) {
            int val;
            fscanf(f, "%d", &val);
            data[i] = (unsigned char)val;
        }
    }

    fclose(f);
    return data;
}
//write pgm
int write_pgm(const char *filename, const unsigned char *data, int width, int height)
{
    FILE *f = fopen(filename, "wb");
    if (!f) {
        perror("Cannot open output PGM file");
        return 1;
    }

    fprintf(f, "P5\n%d %d\n255\n", width, height);
    fwrite(data, 1, width * height, f);
    fclose(f);
    return 0;
}

// PNG READ
unsigned char *read_png(const char *filename, int *width, int *height) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("Cannot open PNG file");
        return NULL;
    }

    unsigned char header[8];
    fread(header, 1, 8, fp);
    if (png_sig_cmp(header, 0, 8)) {
        fprintf(stderr, "Error: %s is not a valid PNG file.\n", filename);
        fclose(fp);
        return NULL;
    }

    png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png_ptr) { fclose(fp); return NULL; }

    png_infop info_ptr = png_create_info_struct(png_ptr);
    if (!info_ptr) { png_destroy_read_struct(&png_ptr, NULL, NULL); fclose(fp); return NUL
```

```
    if (setjmp(png_jmpbuf(png_ptr))) {
        png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
        fclose(fp);
        return NULL;
    }

    png_init_io(png_ptr, fp);
    png_set_sig_bytes(png_ptr, 8);
    png_read_info(png_ptr, info_ptr);

    *width = png_get_image_width(png_ptr, info_ptr);
    *height = png_get_image_height(png_ptr, info_ptr);
    int color_type = png_get_color_type(png_ptr, info_ptr);
    int bit_depth = png_get_bit_depth(png_ptr, info_ptr);

    if (bit_depth == 16)
        png_set_strip_16(png_ptr);

    if (color_type == PNG_COLOR_TYPE_PALETTE)
        png_set_palette_to_rgb(png_ptr);

    if (color_type == PNG_COLOR_TYPE_RGB ||
        color_type == PNG_COLOR_TYPE_RGB_ALPHA)
        png_set_rgb_to_gray_fixed(png_ptr, 1, -1, -1); // Convert RGB to Gray

    if (color_type == PNG_COLOR_TYPE_GRAY_ALPHA)
        png_set_strip_alpha(png_ptr);

    png_read_update_info(png_ptr, info_ptr);

    unsigned char *pixels = malloc((*width) * (*height));
    png_bytep *row_pointers = malloc((*height) * sizeof(png_bytep));
    for (int y = 0; y < *height; y++)
        row_pointers[y] = pixels + y * (*width);

    png_read_image(png_ptr, row_pointers);

    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    free(row_pointers);
    fclose(fp);

    return pixels;
}
// A mxn
void jacobi_svd(double *A, double *V ,double *sig, int m, int n){
    // V= I
    for(int i=0;i<n*n;i++) V[i]=0;
    for(int i=0; i<n; i++) V[i*n+i]=1;
    for(int i=0; i<eigens ;i++){
```

```
    double max_off_diag=0;
    for(int j=0; j<n-1 ; j++){
        for(int k=j+1;k<n;k++){
            double alpha =0,beta=0,gamma=0;
            for (int l = 0; l < m; l++)
            {
                double ap = A[n*l+j];
                double aq = A[n*l+k];

                alpha +=ap*ap;
                beta += aq*aq;
                gamma+= aq*ap;
            }
            max_off_diag = fmax(max_off_diag,fabs(gamma));
            if ((fabs(gamma) < error * sqrt(alpha * beta))|| gamma==0.0) continue;

            double tao = (beta - alpha) / (2.0 * gamma);
            double t = (tao>=0?1:-1)/(fabs(tao)+sqrt(1+tao*tao));
            double c = 1/sqrt(1+t*t);
            double s = c*t;

            for (int l = 0; l < m; l++)
            {
                double ap = A[l*n+j], aq = A[l*n+k];

                A[l*n+j]=c*ap-s*aq;
                A[l*n+k]=s*ap+c*aq;
            }
            for (int l = 0; l < n; l++)
            {
                double vp = V[l*n+j], vq = V[l*n+k];

                V[l*n+j]=c*vp-s*vq;
                V[l*n+k]=s*vp+c*vq;
            }
        }
    }
    if(max_off_diag<error) break;
}
for (int j = 0; j < n; j++) {
    double norm = 0;
    for (int i = 0; i < m; i++) {
        norm += A[i*n + j] * A[i*n + j];
    }
    sig[j] = sqrt(norm);
}
for (int j = 0; j < n; j++) {
    if (sig[j] > 1e-12)
        for (int i = 0; i < m; i++)
            A[i*n + j] /= sig[j];
```

```
    }
    for (int i = 0; i < n - 1; i++) {
        int max_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (sig[j] > sig[max_idx]) max_idx = j;
        }
        if (max_idx != i) {
            // swap Ï
            double temp = sig[i];
            sig[i] = sig[max_idx];
            sig[max_idx] = temp;

            for (int r = 0; r < m; r++) {
                double tmp = A[r * n + i];
                A[r * n + i] = A[r * n + max_idx];
                A[r * n + max_idx] = tmp;
            }

            // swap V columns
            for (int r = 0; r < n; r++) {
                double tmp = V[r * n + i];
                V[r * n + i] = V[r * n + max_idx];
                V[r * n + max_idx] = tmp;
            }
        }
    }
}

//forbenius norm
double forbenius(double *A, int m, int n){
    double sum=0;
    for(int i=0; i<m;i++){
        for (int j = 0; j < n; j++)
        {
            sum+=A[i*n+j]*A[i*n+j];
        }
    }
    return sqrt(sum);
}

int main() {
    int width, height;
    char filename[256];
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    int response;
    printf("Enter image is jpeg(1) or png(0): ");
    scanf("%d", &response);
```

```
if(response){
    printf("Enter JPG path (e.g., ../../figs/name.jpg): ");
    scanf("%255s", filename);

    unsigned char *jpg = read_image(filename, &width, &height);
    if (!jpg) return 1;

    if (write_pgm("../../figs/input_pgm/input.pgm", jpg, width, height) == 0)
        printf("Successfully converted JPEG to PGM!\n");
    else
        printf("Failed to write PGM.\n");
        free(jpg);
}
else{
    printf("Enter PNG path (e.g., ../../figs/name.png): ");
    scanf("%255s", filename);

    unsigned char *png =  read_png(filename, &width, &height);
    if (!png) return 1;

    if (write_pgm("../../figs/input_pgm/input.pgm", png, width, height) == 0)
        printf("Successfully converted PNG to PGM!\n");
    else
        printf("Failed to write PGM.\n");
        free(png);
}
unsigned char *img = read_pgm("../../figs/input_pgm/input.pgm", &width, &height);
if (!img) return 1;
// making char matrix into double matrix for jacobi
double *A = malloc(width * height * sizeof(double));
double *error_matrix = malloc(width * height * sizeof(double));
for (int i=0;i<height*width;i++) A[i]=img[i];

int minm = (height < width) ? height : width;
double *sig = calloc(minm, sizeof(double));

double *V = malloc(width*width*sizeof(double));
jacobi_svd(A,V, sig, height, width );

// now im=U , V=V, sig=sigma
int k;
printf("Enter values of k ");
scanf("%d", &k);
double *Ak = calloc(height*width, sizeof(double));
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        double sum=0;
        for (int r = 0; r < k; r++)
```

```
            {
                sum+=A[i*width+r]*sig[r]*V[j*width+r];
            }
            if(sum<0) sum=0;
            if(sum>255) sum=255;
            Ak[i*width+j]=sum;
        }
    }
    unsigned char *recon = malloc(height*width);
    for (int i=0;i<height*width;i++) recon[i]=(unsigned char)(Ak[i]+0.5);

    for (int i=0;i<height*width;i++) error_matrix[i]=fabs(recon[i]-img[i]);

    write_pgm("../../figs/output_pgm/compressed_output.pgm", recon, width, height);
    printf("Compressed image written.\n");
    write_jpeg("../../figs/output_jpg_png/compressed_output.jpg",recon, width,height);

    printf("%lf\n", forbenius(error_matrix, height,width));

    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Runtime: %.6f seconds\n", cpu_time_used);

    free(img);
    free(A);
    free(Ak);free(V);free(recon);free(sig);

    return 0;
}
```

VIII. Python-Code

Using numpy to solve SVD faster:

```
import numpy as np
import matplotlib.pyplot as plt
import time


def read_pgm(filename):
    with open(filename, 'rb') as f:
        header = f.readline().decode().strip()
        if header not in ['P2', 'P5']:
            raise ValueError("Unsupported PGM format (only P2 or P5 supported).")

        line = f.readline().decode()
        while line.startswith('#'):
            line = f.readline().decode()

        width, height = map(int, line.split())
```

```
        f.readline()  # skip maxval line
        if header == 'P5':
            data = np.frombuffer(f.read(width * height), dtype=np.uint8)
        else:
            data = np.loadtxt(f, dtype=np.uint8)

    return data.reshape((height, width))


def write_pgm(filename, img):
    h, w = img.shape
    img = np.clip(img, 0, 255).astype(np.uint8)
    with open(filename, 'wb') as f:
        f.write(f'P5\n{w} {h}\n255\n'.encode())
        f.write(img.tobytes())


def compress_image_svd(A, k):
    U, S, VT = np.linalg.svd(A, full_matrices=False)
    A_k = U[:, :k] @ np.diag(S[:k]) @ VT[:k, :]
    return A_k, S


def frobenius_norm(A, B):
    return np.sqrt(np.sum((A - B) ** 2))

filename = input("Enter PGM file path (e.g., ../../figs/input_pgm/input.pgm): ").strip()
img = read_pgm(filename).astype(float)
height, width = img.shape
print(f"Loaded image: {width}x{height}")

k = int(input("Enter k: "))
start = time.time()

print("Performing SVD compression...")
compressed, sigma = compress_image_svd(img, k)

compressed = np.clip(compressed, 0, 255)
write_pgm("../../figs/output_pgm/compressed_output_python.pgm", compressed)

err = frobenius_norm(img, compressed)
print(f"Frobenius norm of error matrix = {err:.6f}")
print("Compressed image saved as 'compressed_output_python.pgm'")

end = time.time()
print(f"Runtime: {end - start:.6f} seconds")

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("Original")
```

```
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title(f"Compressed (k={k})")
plt.imshow(compressed, cmap='gray')
plt.axis('off')

plt.tight_layout()
plt.show()
```

IX. Reconstructed images

Fig. 3: Original

Size: 146Kb

(a) For $k = 50$ (C code) Size: 49.4 KB

(b) For $k = 25$ (C code) Size: 43.4 KB

(c) For $k = 5$ (C code) Size: 28 KB

Fig. 3: Comparison of compressed images for different k values using C code.

The quality of the reconstructed image in SVD-based compression depends on the number of singular values retained, denoted by k. As the value of k increases, more information about the image structure and details is preserved, leading to a clearer and more accurate reconstruction. When k is small, such as k=5, only the most dominant features are captured, and the image appears blurred with significant loss of detail. For moderate values like k=25, the main structure of the image becomes recognizable, but fine details are still missing. When k is larger, such as k=50, the reconstructed image closely matches the original with better sharpness and texture. Therefore, k controls the balance between compression and quality: a small k gives higher compression but lower visual quality, while a larger k reduces compression efficiency but preserves more detail.

X. References

- G. H. Golub and C. F. Van Loan, Matrix Computations, 4th Edition,
- Strang, Gilbert, Linear Algebra and Its Applications. New York, Academic Press, 1976.
- Online resources: https://www.wikipedia.org

Thank You