Applying Linear Algebra for Image Manipulation, Filtering, and Compression in TinyIMG

Yousef Amirghofran, Anze Zgonc, Georgios Klonis, Lea Aboujaoude, Hala Yaghi

May 2025

Abstract

This paper details the implementation of TinyIMG, a web-based image processing application leveraging fundamental concepts from linear algebra. The application allows users to perform geometric transformations (rotation, scaling, shearing, translation), apply various image filters (blur, sharpen, edge detection, emboss), and compress images using Singular Value Decomposition (SVD). We discuss the matrix representations employed for images, the mathematical foundations of 2D affine transformations using 4x4 matrices in WebGL, the theory and application of convolution kernels for filtering effects executed in a Go backend compiled to WebAssembly (WASM), and the use of SVD for lossy image compression via low-rank approximation, also within the WASM module. The implementation utilizes a React frontend with WebGL for real-time transformations and relies on the `gonum/mat` library in Go for backend numerical computations. This work demonstrates the practical power and efficiency of linear algebra in the domain of computer graphics and image processing within a modern web architecture.

Keywords: Linear Algebra, Image Processing, WebAssembly, WebGL, Convolution, Singular Value Decomposition (SVD), Image Compression, Go, React.

1 Introduction

Digital image processing and computer graphics are fields deeply rooted in the principles of linear algebra. Matrices and vectors provide a powerful and computationally efficient framework for representing, manipulating, and analyzing image data and geometric structures. This project, TinyIMG, explores the practical application of these principles by developing an interactive web-based tool capable of performing common image manipulation tasks directly within the user's browser.

The primary motivation behind TinyIMG is to demonstrate how core linear algebra techniques can be effectively implemented and integrated into a performant, modern web application. Specifically, we focus on:

- Real-time geometric transformations using matrix multiplication within the WebGL pipeline.
- Image filtering effects achieved through 2D convolution with predefined kernels.
- Lossy image compression based on the mathematical technique of Singular Value Decomposition (SVD).

By leveraging WebAssembly (WASM) for computationally intensive backend logic (filtering and SVD) executed in Go, and WebGL for hardware-accelerated rendering and transformations in the frontend (React/TypeScript), TinyIMG aims to provide a responsive user experience.

The key functionalities explored, driven by linear algebra, are:

1. Representing image data using matrix structures.
2. Applying geometric transformations (rotation, scaling, shearing, translation) using 4x4 matrices.
3. Calculating the area scaling effect of transformations using determinants.
4. Implementing image filters (blur, sharpen, edge detection, emboss) via 2D convolution.
5. Performing image compression using SVD low-rank approximation.

This paper will delve into each of these areas, first explaining the underlying linear algebra theory, then discussing its specific application within the context of image processing, and finally detailing how it was implemented in the TinyIMG project, referencing the codebase where applicable. We will conclude by discussing the challenges encountered and potential limitations.

2 Matrix Representation of Images

Understanding how image data is structured is the first step in applying mathematical operations. Matrices provide a natural way to organize this data.

2.1 Theory: Matrices for Data Organization

A matrix is a rectangular array of numbers arranged in rows and columns. In the context of data representation, matrices can store multi-dimensional information in a structured format. For a 2D grayscale image, a single matrix can suffice, where each element M_{ij} represents the intensity of the pixel at row i, column j. For color images, multiple matrices or higher-dimensional structures are needed.

2.2 Application: Representing RGBA Channels

Digital color images, particularly in the RGBA model, have four components per pixel. While initially handled as a 1D array (`Uint8ClampedArray`) for efficient browser handling and data transfer, the conceptual model for applying many linear algebra techniques involves separating these channels. An $H \times W$ (height $\times$ width) image can be thought of as four distinct $H \times W$ matrices: M_R, M_G, M_B, M_A. Each matrix element $(M_{\text{channel}})_{ij}$ holds the intensity value (typically 0-255) for that specific channel at pixel (j, i) (column j, row i). This separation allows operations like SVD to be applied independently to each color component.

2.3 Implementation in TinyIMG

- **Backend (Go/WASM):** When performing SVD, the Go backend explicitly converts the input flat `Uint8ClampedArray` data into four separate `mat.Dense` matrices provided by the `gonum/mat` library, one for each RGBA channel. This conversion happens within the `compressSVD` function before launching parallel SVD computations.

Listing 1: Go code snippet: Creating channel matrices (Illustrative)

```
// Create separate dense matrices for R, G, B, A channels
rMatrix := mat.NewDense(int(height), int(width), nil)
gMatrix := mat.NewDense(int(height), int(width), nil)
bMatrix := mat.NewDense(int(height), int(width), nil)
aMatrix := mat.NewDense(int(height), int(width), nil) // Compressing Al

// Parallelized filling of matrices from flat pixel data
numFillGoroutines := runtime.NumCPU()
rowsPerFillGoroutine := (int(height) + numFillGoroutines - 1) / numFill
fillDone := make(chan bool, numFillGoroutines)

for i := 0; i < numFillGoroutines; i++ {
    startY := i * rowsPerFillGoroutine
    endY := min(startY+rowsPerFillGoroutine, int(height))

    go func(startY, endY int) {
        defer func() { fillDone <- true }()
        for y := startY; y < endY; y++ {
            for x := 0; x < int(width); x++ {
                idx := (y*int(width) + x) * 4
                if idx+3 >= len(data) {
                    continue
                } // Bounds check
```

```
                    rMatrix.Set(y, x, float64(data[idx]))
                    gMatrix.Set(y, x, float64(data[idx+1]))
                    bMatrix.Set(y, x, float64(data[idx+2]))
                    aMatrix.Set(y, x, float64(data[idx+3]))
                }
            }
        }(startY, endY)
    }
    for i := 0; i < numFillGoroutines; i++ {
        <-fillDone
    }
    fmt.Println("Matrix filling complete.")
```

- **Frontend (WebGL):** In the frontend, the image is primarily treated as a single texture unit (`sampler2D`). The mapping of this texture onto the geometry is controlled by texture coordinates (UVs) associated with the vertices of the quad. These UV coordinates, typically ranging from (0,0) to (1,1), effectively tell the GPU which part of the 2D texture matrix corresponds to each point on the 3D geometry.

3 Geometric Transformations via Matrices

Linear algebra provides a powerful framework for manipulating the geometric presentation of an image (its position, orientation, size, and shape) without altering the underlying pixel data itself.

3.1 Theory: Linear Transformations and Homogeneous Coordinates

Linear transformations (like rotation, scaling, shearing) preserve vector addition and scalar multiplication ($T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$ and $T(k\mathbf{u}) = kT(\mathbf{u})$). They can be represented by matrix multiplication: $T(\mathbf{x}) = A\mathbf{x}$, where A is the transformation matrix.

Geometric Interpretation of 2D Transformations Linear transformations alter the geometry of an image by systematically modifying the position of each pixel according to a matrix rule. Here, we focus on three fundamental transformations: rotation, scaling, and shearing.

Rotation: A rotation transformation pivots every point of the image around the origin by an angle θ. The matrix

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

rotates the vector $\mathbf{x} = (x, y)^T$ by θ radians counterclockwise. All distances from the origin are preserved, but the orientation changes.

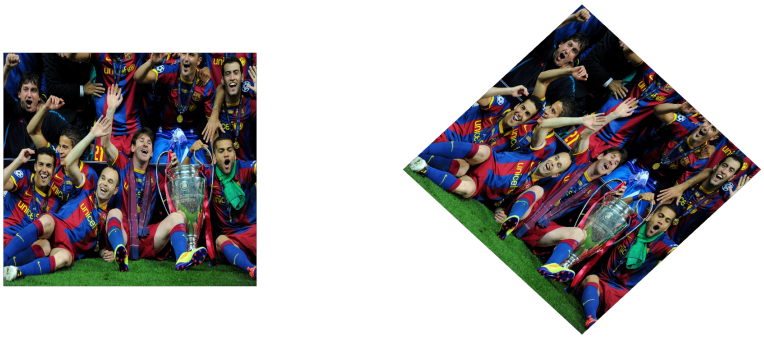

Figure 1: Effect of a rotation transformation on an image.

Scaling: Scaling stretches or compresses the image along the x and y axes by factors s_x and s_y:

$$S = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

If $s_x > 1$, the image is stretched horizontally; if $0 < s_x < 1$, it is compressed. The same applies for s_y vertically.

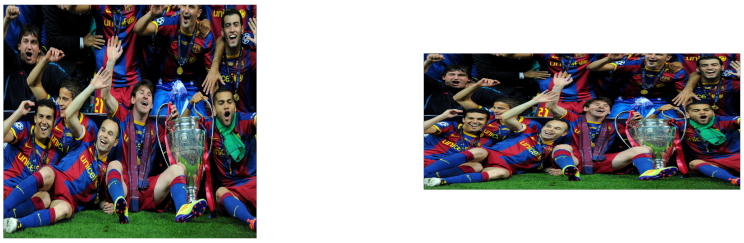

Figure 2: Comparison: original image and result of a scaling transformation.

Shearing: Shearing shifts each point in the image parallel to one axis, by an amount proportional to its coordinate on the other axis. Horizontal shearing by k is given by:

$$Sh_x = \begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$$

This transformation slants the image, turning rectangles into parallelograms.

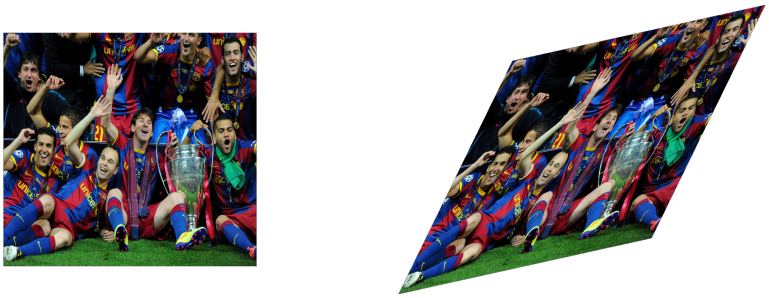

Figure 3: Effect of a shearing transformation on an image.

Final Transformation Formula The combined transformation applied to each point in the image can be represented as a single matrix multiplication in homogeneous coordinates. For 2D transformations (ignoring depth), the final transformation matrix M is:

$$M = T \cdot R \cdot Sh \cdot S$$

where:

- S is the scaling matrix,
- Sh is the shearing matrix,
- R is the rotation matrix,

- T is the translation matrix.

Explicitly, in homogeneous coordinates (3x3 form):

$$M = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Applying M to a point in homogeneous coordinates $(x, y, 1)^T$ yields the transformed position. In our WebGL implementation, this is extended to 4x4 matrices to support 3D and additional effects, but the principle and order of operations remain the same.

Translation (shifting position) is an *affine* transformation, not strictly linear. To incorporate translation and represent all affine transformations uniformly as matrix multiplications, we use homogeneous coordinates. A 2D point (x, y) becomes $(x, y, 1)$. Transformations are then represented by 3x3 matrices. In 3D graphics (and thus WebGL, even for 2D rendering), this extends to 4D homogeneous coordinates $(x, y, z, 1)$ and 4x4 matrices. A 2D transformation $(x', y') = (ax + cy + t_x, bx + dy + t_y)$ is represented in homogeneous coordinates as:

$$\begin{pmatrix} a & c & t_x \\ b & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \quad \text{or in 4x4 form:} \quad \begin{pmatrix} a & c & 0 & t_x \\ b & d & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix}$$

Multiple transformations can be combined by multiplying their respective matrices. The order matters: applying T_1 then T_2 corresponds to the matrix product $M_2 M_1$ applied to the vector.

3.2 Application: Transforming Image Display Geometry

In TinyIMG, these transformations are not applied to the image pixels directly, but to the vertices of the quadrilateral (quad) geometry onto which the image texture is mapped in WebGL. By transforming the quad's vertices (changing its position, size, orientation, or shape on the screen), we change how the mapped image appears. The 4x4 matrix representation is used because WebGL operates in a 3D context.

3.3 Implementation in TinyIMG

- **WebGL Setup:** A quad is defined with vertex positions (e.g., -1 to +1) and texture coordinates (0 to 1). These are sent to the GPU.
- **Matrix Construction:** The React frontend uses the `gl-matrix` library to create individual 4x4 matrices for translation (`mat4.translate`), rotation (`mat4.rotateZ`), scaling (`mat4.scale`, incorporating flips via negative scales), and shearing (`mat4.fromValues` constructing the shear matrix) based on UI controls.
- **Matrix Combination:** The matrices are multiplied in the order: Scale → Shear → Rotate → Translate ($M_{\text{combined}} = T \cdot R_z \cdot Sh \cdot S$) to get a single transformation matrix.
- **Vertex Shader:** This combined matrix (`u_matrix`) is passed to the vertex shader. The shader multiplies each vertex position (extended to 4D homogeneous coordinates) by this matrix: `gl_Position = u_matrix * vec4(a_position, 0.0, 1.0);`. This computes the final transformed position of the vertex on the screen.
- **Fragment Shader:** The fragment shader uses the interpolated texture coordinates passed from the vertex shader to sample the image texture, effectively painting the (unaltered) image onto the (transformed) quad.

4 Determinants and Area Scaling

The determinant of a transformation matrix provides valuable geometric information, specifically how the transformation affects area (in 2D) or volume (in 3D).

4.1 Theory: Determinants and Geometric Interpretation

For a 2D linear transformation represented by the matrix $M = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$, the determinant is calculated as $\det(M) = ad - bc$. Geometrically, the absolute value $|\det(M)|$ represents the factor by which the area of any shape is scaled under the transformation $T(\mathbf{x}) = M\mathbf{x}$.

- $|\det(M)| = 1$: Area is preserved (e.g., pure rotation, shear). Note: Standard shear matrices have det=1. Shearing *can* be combined with non-uniform scaling in TinyIMG's implementation matrix, leading to area change.
- $|\det(M)| > 1$: Area increases (e.g., scaling up).
- $0 < |\det(M)| < 1$: Area decreases (e.g., scaling down).
- $\det(M) = 0$: The transformation collapses the area onto a line or point (degenerate).
- The sign of the determinant indicates whether the orientation is preserved (positive) or reversed (negative, e.g., a reflection).

For the 4x4 matrices used in homogeneous coordinates for 2D transformations, the area scaling factor is determined by the determinant of the upper-left 2x2 submatrix that represents the linear part (rotation, scaling, shear) of the transformation.

4.2 Application: Quantifying Transformation Effects

In TinyIMG, calculating the determinant of the combined transformation's linear part allows the application to display how much the user's chosen transformations have scaled the apparent area of the image on screen. This provides quantitative feedback alongside the visual change.

4.3 Implementation in TinyIMG

- After computing the combined 4x4 transformation matrix M_{combined} in JavaScript using `gl-matrix`, the elements corresponding to the upper-left 2x2 submatrix (a, b, c, d) are extracted.
- The determinant $ad - bc$ is calculated, potentially using a helper function from `gl-matrix` like `mat2.determinant` if applied to an extracted 2x2 matrix.
- The absolute value of this determinant is taken and displayed in the UI as the "Transformed Area" scaling factor (relative to an initial area, likely normalized).

5 Image Filtering via Convolution

Convolution is a fundamental operation in image processing used to apply various filters for effects like blurring, sharpening, and edge detection.

5.1 Theory: 2D Convolution

2D convolution involves sliding a small matrix, called a kernel (K), over an input image matrix (P_{in}). For each pixel position (x, y), the output pixel value $P'_{\text{out}}(x, y)$ is computed as the weighted sum of the input pixel values in the neighborhood defined by the kernel, where the kernel elements provide the weights. Mathematically, for a channel c and a kernel of size $(2k + 1) \times (2k + 1)$:

$$P'_{\text{out}}(x, y, c) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i, j) \cdot P_{\text{in}}(x + i, y + j, c)$$

The kernel's values determine the filter's effect. For example, an averaging kernel causes blurring, while a kernel emphasizing differences between the center pixel and its neighbors causes sharpening.

5.2 Theory: Common Kernels and Their Effects

Below are several standard kernels, each designed to achieve a particular filtering effect. We compare the result of applying each filter to an image against the original, and explain the mathematical intuition behind each kernel:

Original Image: The unfiltered image serves as a reference for comparison.

Blur (Box Filter): The blur kernel averages the values of each pixel with its neighbors, smoothing out sharp transitions and reducing noise. The kernel is:

$$K_{\text{blur}} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Each output pixel becomes the average of itself and its eight immediate neighbors, resulting in a softened image where fine details are diminished.

Figure 4: Comparison: original image and result of a blur filter.

Sharpen: The sharpen kernel enhances edges and fine details by amplifying the difference between a pixel and its neighbors. The kernel is:

$$K_{\text{sharpen}} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

The center pixel is multiplied by 5, while its immediate neighbors are subtracted. This accentuates regions with rapid intensity change (edges), making the image appear crisper.

Figure 5: Comparison: original image and result of a sharpen filter.

Edge Detection: The edge detection kernel highlights boundaries between regions of different intensity. The kernel is:

$$K_{\text{edge}} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

This kernel computes the difference between the center pixel and the sum of its neighbors. Where there is a strong contrast (edge), the output is high; otherwise, it is near zero, producing an image that emphasizes outlines.

Figure 6: Comparison: original image and result of an edge detection filter.

Emboss: The emboss kernel creates a 3D-like effect by highlighting edges in a specific direction. The kernel is:

$$K_{\text{emboss}} = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

By assigning positive weights to one side of the center pixel and negative to the other, the kernel simulates lighting from a particular angle, making features appear raised or recessed.

Figure 7: Comparison: original image and result of an emboss filter.

5.3 Implementation in TinyIMG

- **Kernels Defined:** The Go backend (`main.go`) defines specific 3x3 kernels (represented as flat `float64` slices) for each supported filter type within the `applyFilter` function.

```
// Inside applyFilter
switch filterType {
case "blur":
    filter = []float64{
        1 / 9.0, 1 / 9.0, 1 / 9.0,
        1 / 9.0, 1 / 9.0, 1 / 9.0,
```

```
            1 / 9.0, 1 / 9.0, 1 / 9.0,
        }
    case "sharpen":
        filter = []float64{
            0, -1, 0,
            -1, 5, -1,
            0, -1, 0,
        }
    // ... other filters ...
    }
```

- **Convolution Loop:** The `applyFilter` function iterates through each pixel (x, y) and each color channel (c=0, 1, 2 for R, G, B). Inside this loop, it iterates through the kernel's neighborhood (fx, fy). It calculates the corresponding source image coordinates (sx, sy), carefully handling image boundaries by clamping these coordinates to valid ranges using a helper function `clamp`. The weighted sum (`sum`) is accumulated.

```
for y := startY; y < endY; y++ {
    for x := 0; x < width; x++ {
        for c := 0; c < 3; c++ { // R, G, B channels
            sum := 0.0
            for fy := 0; fy < filterSize; fy++ {
                for fx := 0; fx < filterSize; fx++ {
                    sx := clamp(x + fx - filterSize/2, 0, width-1)
                    sy := clamp(y + fy - filterSize/2, 0, height-1)
                    sampleIndex := (sy*width+sx)*4 + c
                    sampleValue := float64(srcData[sampleIndex])
                    filterIndex := fy*filterSize + fx
                    sum += sampleValue * filter[filterIndex]
                }
            }
            resultIndex := (y*width+x)*4 + c
            resultData[resultIndex] = uint8(clamp(int(sum+0.5), 0, 255))
        }
        // Copy Alpha channel
        alphaIndex := (y*width+x)*4 + 3
        resultData[alphaIndex] = srcData[alphaIndex]
    }
}
```

- **Output and Clamping:** The final computed sum for each output pixel channel is clamped to the valid range [0, 255] using the `clamp` function again (after adding 0.5 for rounding) before being cast to `uint8` and stored in the result data array. The Alpha channel is copied directly from source to result.

- **Parallelization:** To improve performance, the processing of image rows is parallelized. The image is divided into horizontal chunks (`CHUNK_SIZE`), and a separate Go goroutine processes each chunk concurrently. A wait group mechanism (using a channel `done`) ensures all chunks are processed before returning the result.

```
    numGoroutines := (height + CHUNK_SIZE - 1) / CHUNK_SIZE
done := make(chan bool, numGoroutines)
for i := 0; i < numGoroutines; i++ {
    startY := i * CHUNK_SIZE
    endY := min(startY+CHUNK_SIZE, height)
    go func(startY, endY int) {
        // ... convolution loop ...
        done <- true
    }(startY, endY)
```

```
}
for i := 0; i < numGoroutines; i++ {
    <-done
}
```

6 Image Compression via SVD

Singular Value Decomposition (SVD) is a powerful matrix factorization technique that can be applied to image compression by approximating the image data.

6.1 Theory: SVD and Low-Rank Approximation

Any real $m \times n$ matrix M can be decomposed into the product of three matrices: $M = USV^T$, where:

- U is an $m \times m$ orthogonal matrix (its columns are the left singular vectors).
- S is an $m \times n$ diagonal matrix containing the non-negative singular values (σ_i) in descending order ($\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$) along its diagonal. These are the square roots of the eigenvalues of M^TM or MM^T.
- V^T is the transpose of an $n \times n$ orthogonal matrix V (its columns are the right singular vectors, which are eigenvectors of M^TM).

The key idea for compression is that the most important information about the matrix M is often captured by the largest singular values and their corresponding singular vectors. We can approximate M by using only the first k singular values and the corresponding first k columns of U and V. This is the rank-k approximation:

$$\mathbf{A}_k \approx \left[\begin{array}{c|c|c|c} & & & \\ \vec{u}_1 & \vec{u}_2 & \dots & \vec{u}_k \\ & & & \end{array}\right] \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_k \end{bmatrix} \left[\begin{array}{c|c|c|c} & & & \\ \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_k \\ & & & \end{array}\right]^T$$

where U_k is $m \times k$, S_k is $k \times k$ (diagonal with $\sigma_1, \dots, \sigma_k$), and V_k^T is $k \times n$. The Eckart-Young theorem states that M_k is the best rank-k approximation of M in the least-squares sense. By choosing $k < \text{rank}(M)$, we can represent the approximation M_k using less data (storing U_k, S_k, V_k) than the original M, achieving compression.

6.2 Application: Compressing Image Channels

SVD can be applied to image compression by treating each color channel (R, G, B, A) as a separate matrix (M_R, M_G, M_B, M_A). We compute the SVD for each channel matrix and then reconstruct an approximation using a chosen rank k. The reconstructed channel matrices $M_{R,k}, M_{G,k}, M_{B,k}, M_{A,k}$ are then combined to form the compressed image. A lower rank k leads to greater compression (fewer singular values/vectors to store) but potentially more visible artifacts.

6.3 Implementation in TinyIMG

- **Channel Separation:** The `compressSVD` function in Go first separates the input image data into four $H \times W$ `mat.Dense` matrices for R, G, B, and A channels (this step is parallelized).
- **SVD Computation:** For each channel matrix, a separate goroutine calls the `compressMatrixSVD` function.
- `compressMatrixSVD` **Logic:**
  - It uses the `gonum/mat` library's SVD implementation: `svd.Factorize(m, mat.SVDFull)`.
  - It extracts the full U and V matrices and the slice of singular values s.

- It determines the effective rank (`effectiveRank`) as $\min(k, \text{rows}, \text{cols})$.
- It creates the truncated matrices: U_k (`ur`) by taking a slice of the first k columns of U; S_k (`sr`) by creating a new $k \times k$ diagonal matrix with the top k singular values; and V_k (`vr`) by taking a slice of the first k columns of V.
- It reconstructs the rank-k approximation M_k by performing the matrix multiplications $M_k = U_k S_k V_k^T$ using `mat.Mul`.

- **Image Reconstruction:** The main `compressSVD` function collects the approximated matrices (R_k, G_k, B_k, A_k) from the goroutines. It then reconstructs the final compressed image pixel data into a flat `uint8` slice. This involves iterating through the pixel coordinates (x, y), reading the corresponding values from the approximated matrices, clamping them to $[0, 255]$ (using `clampFloat64` and rounding), and assembling the RGBA values back into the flat array (this step is also parallelized).
- **Comparison:**

Figure 8: SVD compression results: original and rank-70, rank-40, rank-10 approximations.

7 Challenges and Limitations

While the project successfully implemented the core functionalities, several challenges were encountered, and certain limitations exist.

7.1 Challenges

- **WASM Build and Integration:** Configuring the Go compiler to produce a WASM module compatible with the JavaScript environment, including handling dependencies like `gonum/mat`, required specific build flags and scripts (`build.sh`). Ensuring correct function exports and imports between Go and JavaScript using `syscall/js` needed careful attention.
- **JS/WASM Data Transfer:** Copying image pixel data between JavaScript and WASM memory using functions like `js.CopyBytesToGo` and `js.CopyBytesToJS` is necessary but introduces overhead, especially for large images. Minimizing these transfers is important for performance.

- **Parallelism in WASM:** Implementing parallel processing using Go goroutines for convolution and SVD aims to speed up computation. However, the actual concurrency achieved depends on the browser's WASM runtime and thread support. Debugging race conditions or performance bottlenecks in parallel WASM code can be challenging.
- **Debugging WASM:** Debugging Go code running inside WASM is less straightforward than native Go debugging. It often relies on logging (`fmt.Println`) within the Go code, which appears in the browser's developer console, or more advanced browser debugging tools with WASM support (which can vary in capability).

7.2 Limitations

- **Filter Variety:** The project implements only four basic convolution filters. Many other useful filters exist (e.g., Gaussian blur, median, bilateral).
- **SVD Compression Efficiency:** SVD is computationally intensive and, while demonstrating an important linear algebra concept, is generally outperformed by standard codecs like JPEG (DCT-based) and PNG (lossless) in terms of compression ratio versus visual quality for typical photographic images.
- **No True Compression Output:** The current implementation reconstructs the compressed image pixels but doesn't provide an option to save the compressed representation (the U_k, S_k, V_k factors), which would be necessary to realize actual file size savings.
- **Performance Benchmarking:** The project lacks formal performance analysis. While timing logs exist, a systematic evaluation across different image sizes, browsers, and hardware would be needed to quantify performance accurately.
- **User Interface:** The UI provides basic functionality but could be improved with features like zoom/pan, history states, or more sophisticated transformation controls.
- **Error Handling:** While some basic error checks exist (e.g., argument counts, data copying), the robustness of error handling could be improved, particularly for edge cases in WASM interactions or numerical computations.

8 Conclusion

This paper presented the TinyIMG project, a web application successfully demonstrating the practical implementation of fundamental linear algebra techniques for image manipulation, filtering, and compression. By integrating a React/WebGL frontend with a Go/WASM backend, the application provides users with tools for real-time geometric transformations using 4x4 matrices, image filtering via 2D convolution with various kernels, and lossy image compression through Singular Value Decomposition. The project highlights the power and versatility of matrix mathematics in computer graphics and showcases how computationally intensive tasks can be efficiently executed within the browser using WebAssembly. Following a structure that detailed the theory, application, and implementation for each core concept, we have illustrated how abstract mathematical ideas translate into tangible image processing functionalities. Despite limitations in scope and the practical efficiency of SVD for general-purpose compression, TinyIMG serves as a valuable educational example and a foundation for exploring more advanced image processing techniques in a web environment.