# Software Assignment

Samyak Gondane

November 2025

# SVD Summary (Strang's Video)

The video provides a detailed explanation of the **Singular Value Decomposition (SVD)**, a powerful matrix factorization technique. SVD expresses any matrix $A$ as the product of three matrices: $A = U\Sigma V^T$, where:

$U$ and $V$ are **orthogonal matrices** (their columns are orthonormal vectors). $\Sigma$ is a **diagonal matrix** containing the **singular values** (non-negative real numbers).

**Key Concepts**:
**Goal of SVD**: To find orthonormal bases for the row space and column space of $A$, such that the transformation defined by $A$ maps one orthonormal basis to another, scaled by the singular values.
**Null Spaces**: The null space and left null space are handled by adding zero singular values in $\Sigma$.

**Connection to Eigenvalues**: The singular values are the square roots of the eigenvalues of $A^TA$ and $AA^T$, which are symmetric and positive semi-definite matrices.

**Examples**: The video walks through two examples:
A **non-singular matrix** (rank 2), where both $U$ and $V$ are computed.
A **singular matrix** (rank 1), where only one singular value is non-zero, and the rest are zero.

**Applications**: SVD is useful in many areas, including data compression, image processing, and solving linear systems. It generalizes the idea of diagonalization for non-square matrices and is a central concept in linear algebra.

This decomposition is a unifying concept in linear algebra, bringing together ideas from eigenvalues, orthogonal matrices, and the four fundamental subspaces of a matrix.

# Explanation of the Implemented Algorithm

## Mathematical Background

Given a grayscale image as a matrix $A \in R^{m \times n}$, the Singular Value Decomposition (SVD) expresses it as:

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

**U**: m × m orthogonal matrix (left singular vectors)
**Σ**: m × n diagonal matrix with singular values ($\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_T \geq 0$)
$\mathbf{V^T}$: n × n transpose of orthogonal matrix (right singular vectors)

To compress the image, we compute a low-rank approximation:

$$\mathbf{A_k} = U_k \times \Sigma_k \times V_k^T \tag{2}$$

$\mathbf{U_k}$: m × k matrix (top k left singular vectors)
$\mathbf{\Sigma_k}$: k × k diagonal matrix (top k singular values)
$\mathbf{V_k^T}$: k × n matrix (top k right singular vectors transposed)

This reduces storage while preserving most visual content.

## Pseudocode for Power Iteration SVD

**Input**: **A** ($m \times n$ matrix), $k$ (number of singular values), max_iter, tol
**Output**: **S** (top k singular values), **U** (left vectors), $\mathbf{V}^T$ (right vectors transposed)

1. Initialize $\mathbf{R} = \mathbf{A}$ (residual matrix)
2. For comp = 0 to $K - 1$:
   (a) Randomly initialize **v** (length $n$)
   (b) Repeat until convergence:
      i. $\mathbf{u} = \mathbf{R} \times \mathbf{v}$
      ii. Normalize **u**
      iii. $\mathbf{v} = \mathbf{R} \times \mathbf{u}$
      iv. Normalize **v**
      v. $\sigma = \mathbf{u^T} \times \mathbf{R} \times \mathbf{v}$
      vi. Check convergence: $\|\sigma - prev_\sigma\| < tol \times \sigma$
   (c) Store $\sigma$, **u**, **v** in **S**, **U**, $\mathbf{V^T}$
   (d) Deflate $\mathbf{R}$ : $\mathbf{R} = \mathbf{R} - \sigma \times \mathbf{u} \times \mathbf{v^T}$

**Input:**

$\mathbf{A} \rightarrow m \times n$ matrix
$\mathbf{k} \rightarrow$ number of singular values to compute
max_iter $\rightarrow$ maximum number of iterations
tol $\rightarrow$ convergence tolerance

**Output:**

$\mathbf{S} \rightarrow$ list of top $k$ singular values
$\mathbf{U} \rightarrow$ list of left singular vectors
$\mathbf{V}_T \rightarrow$ list of right singular vectors (transposed)

**Initialize:**

$\mathbf{R} \rightarrow A$ (residual matrix)
$\mathbf{S} \rightarrow$ empty list
$\mathbf{U} \rightarrow$ empty list
$\mathbf{V}_T \rightarrow$ empty list

**For comp = 1 to k:**

$\mathbf{v} \rightarrow$ random vector of length $n$
$prev_\sigma \rightarrow 0$

Repeat until convergence or max_iter:
$\mathbf{u} \rightarrow \mathbf{R} \times \mathbf{v}$
$\mathbf{u} \rightarrow \frac{\mathbf{u}}{\|u\|}$ (normalize)

$\mathbf{v} \rightarrow \mathbf{R}^t \times \mathbf{u}$
$\mathbf{v} \rightarrow \frac{\mathbf{v}}{\|v\|}$ (normalize)
$\sigma \rightarrow \mathbf{u}^t \times \mathbf{R} \times \mathbf{v}$

If $\|\sigma - prev_\sigma\| < tol \times \sigma$:
break
$prev_\sigma \rightarrow \sigma$

Append $\sigma$ to $\mathbf{S}$
Append $\mathbf{u}$ to $\mathbf{U}$
Append $\mathbf{v}^t$ to $\mathbf{V}_T$

$\mathbf{R} \rightarrow \mathbf{R} - \sigma \times \mathbf{u} \times \mathbf{v}^t$ (deflation step)
Return $\mathbf{S}, \mathbf{U}, \mathbf{V}_T$

# Comparison of Algorithms

## Considered Algorithms

| Algorithm | Pros | Cons |
|---|---|---|
| **Power Iteration (used)** | Simple, memory-efficient, good for top-$k$ | Slow convergence, not full SVD |
| Golub–Reinsch | Accurate, full decomposition | Complex, heavy computation |
| Jacobi | Good for small matrices | Inefficient for large images |

Table 1: Comparison of SVD algorithms

**Chosen: Power Iteration** Why?

1. Easy to implement in C
2. Focused on top-k singular values
3. Suitable for image compression where full SVD isn't needed
4. Allows deflation and iterative refinement

# Reconstructed Images for Different k

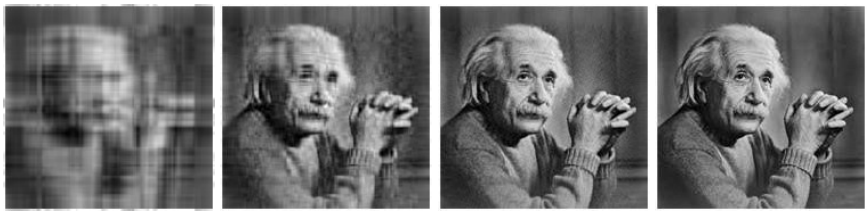

Figure 1: Reconstructed images for $k = 5, 20, 50, 100$

| k | Frobenius Error |
|---|---|
| 5 | 4713.35 |
| 20 | 2127.64 |
| 50 | 880.352 |
| 100 | 164.782 |

Table 2: Error analysis for Figure 1

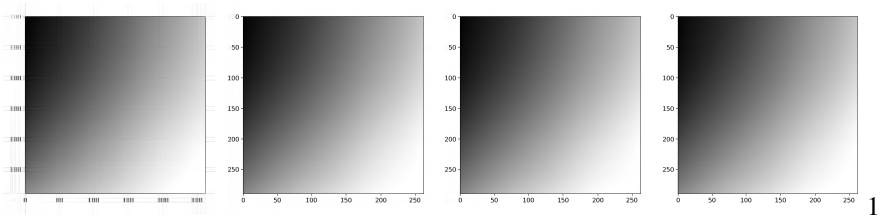

Figure 2: Reconstructed images for $k = 5, 20, 50, 100$

| k | Frobenius Error |
|---|---|
| 5 | 20703.9 |
| 20 | 10633.9 |
| 50 | 6185.22 |
| 100 | 3672.67 |

Table 3: Error analysis for Figure 2

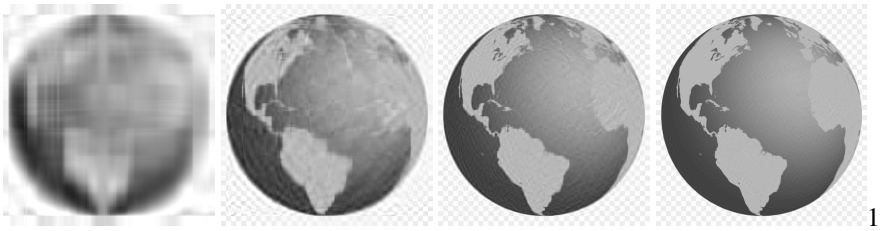

Figure 3: Reconstructed images for $k = 5, 20, 50, 100$

| k | Frobenius Error |
|---|---|
| 5 | 11155.9 |
| 20 | 10633.9 |
| 50 | 6185.22 |
| 100 | 3672.67 |

Table 4: Error analysis for Figure 3

# Error Analysis

Use Frobenius norm:

$$||A - Ak||_F = \sqrt{(\Sigma(a_{ij} - a_{ij}^k)^2)}$$

# Trade-offs and Reflections

## Trade-offs

1. Low k → High compression, poor image quality
2. High k → Better quality, less compression
3. Optimal k depends on acceptable error threshold

## Reflections

1. Full C implementation gave deeper control over memory and performance
2. Power iteration was intuitive and scalable
3. Modular design helped debug and extend easily
4. Writing to PGM format ensured compatibility with image viewers