

# Image Compression Using Singular Value Decomposition: A From-Scratch Implementation and Analysis

Noah Valenzuela, David Noboa, Ryan Graham, Azim Oseni  
University at Albany, SUNY

December 8th, 2025

## 1 Introduction

Digital image compression is essential in modern computing, from reducing storage costs to enabling faster data transmission. The Singular Value Decomposition (SVD) provides a mathematically elegant approach to image compression by exploiting the inherent redundancy in natural images. This project implements SVD from first principles (without using built-in linear algebra packages) and analyzes its effectiveness as a compression algorithm.

The SVD factors any matrix  $A$  into three components:  $A = U\Sigma V^T$ , where  $U$  and  $V$  are orthogonal matrices and  $\Sigma$  contains the singular values. By retaining only the  $k$  largest singular values, we can approximate the original image with significantly reduced storage requirements.

## 2 Motivation and Problem Statement

Standard image formats like JPEG achieve impressive compression ratios through sophisticated techniques like the Discrete Cosine Transform (DCT). However, understanding SVD-based compression offers valuable insights into how linear algebra concepts directly translate to practical data reduction problems.

**Central Research Question:** How effectively can SVD reduce image storage while maintaining acceptable visual quality? What are the quantitative trade-offs between compression ratio and image fidelity?

The key constraint of this project is implementing SVD entirely from scratch—no `np.linalg.svd` or `np.linalg.eig` functions allowed. This forces a deep understanding of the underlying numerical methods.

## 3 Technical Challenges

Building SVD from scratch presents significant challenges:

**Eigenvalue Computation:** For matrices larger than  $4 \times 4$ , the Abel-Ruffini theorem proves no closed-form solution exists for eigenvalues, necessitating iterative numerical solvers.

**Numerical Stability:** Implementation must handle floating-point errors, ensure orthogonality, and correctly match signs of left/right singular vectors.

**Computational Complexity:** For  $512 \times 512$  images,  $A^T A$  creates a  $512 \times 512$  matrix requiring iterative eigendecomposition with reasonable convergence time.

## 4 Methods

### 4.1 Mathematical Foundation

The SVD components are derived from eigendecompositions:

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

$$A A^T = U \Sigma \Sigma^T U^T \tag{2}$$

The columns of  $V$  are eigenvectors of  $A^T A$ , while singular values are  $\sigma_i = \sqrt{\lambda_i}$  where  $\lambda_i$  are the eigenvalues. Left singular vectors are computed via  $u_i = \frac{1}{\sigma_i} A v_i$  to ensure sign consistency.

## 4.2 Householder Reflections for QR Decomposition

The QR decomposition is implemented using Householder reflections. For each column  $k$ , we construct a reflection that zeros out subdiagonal elements:

```
1 def householder_qr(A):
2     A = np.array(A, dtype=float)
3     m, n = A.shape
4     Q = np.eye(m)
5
6     for k in range(min(m, n)):
7         x = A[k:, k].copy()
8         norm_x = _vector_norm(x)
9         if norm_x == 0.0:
10             continue
11
12         # Avoid cancellation
13         sign = 1.0 if x[0] >= 0.0 else -1.0
14         e1 = np.zeros_like(x)
15         e1[0] = 1.0
16         v = x + sign * norm_x * e1
17
18         v_norm_sq = (v * v).sum()
19         if v_norm_sq == 0.0:
20             continue
21
22         # Build Householder matrix
23         H_sub = np.eye(v.shape[0]) - 2.0 * np.outer(v, v) / v_norm_sq
24         H_k = np.eye(m)
25         H_k[k:, k:] = H_sub
26
27         A = H_k @ A
28         Q = Q @ H_k
29
30     return Q, A # A is now R
```

Listing 1: Householder QR Implementation Core (from Handwritten\_SVD.py)

## 4.3 QR Algorithm for Eigenvalues

The iterative QR algorithm converges to eigenvalues of symmetric matrices. Each iteration performs QR decomposition and recombines in reverse order:

```
1 def handwritten_svd(A, num_iterations):
2     # Form symmetric matrix A^T @ A
3     S = A.T @ A
4     n = S.shape[0]
5
6     A_k = S.copy()
7     V = np.eye(n) # Accumulate eigenvectors
8
9     for i in range(num_iterations):
10         Q, R = householder_qr(A_k)
11         A_k = R @ Q # Orthogonal similarity
12         V = V @ Q
13
14     eigenvalues = np.diag(A_k)
15     singular_values = np.sqrt(np.abs(eigenvalues))
16
17     # Sort in descending order
18     sort_indices = np.argsort(singular_values)[::-1]
19     singular_values = singular_values[sort_indices]
20     V = V[:, sort_indices]
21
22     # Compute U via stable method: u_i = Av_i / sigma_i
23     m, nA = A.shape
24     U = np.zeros((m, m))
25     for i in range(min(m, nA)):
26         if singular_values[i] > 1e-12:
27             U[:, i] = (A @ V[:, i]) / singular_values[i]
28
29     Sigma = np.zeros((m, nA))
30     minmn = min(m, nA)
31     Sigma[:minmn, :minmn] = np.diag(singular_values)
32
33     return U, Sigma, V.T
```

Listing 2: Core SVD Algorithm (from Handwritten\_SVD.py)

## 4.4 Image Compression Application

The compression scheme applies rank- $k$  truncation. Storage requirement:  $k(m + n + 1)$  vs.  $mn$  for original.

```
1 def computeSvdCompression(imageMatrix, k):
2     U, S, Vt = np.linalg.svd(imageMatrix, full_matrices=False)
3     Uk, Sk, Vtk = U[:, :k], S[:k], Vt[:, :k, :]
4     A_k = Uk @ np.diag(Sk) @ Vtk
5     return A_k, S
6
7 def compressionStats(m, n, k):
8     return m*n, k*(m+n+1), k*(m+n+1)/(m*n)
```

Listing 3: Compression Functions (ImgDecompExample.py)

## 5 Results

### 5.1 Quantitative Analysis

Tests on a  $512 \times 512$  grayscale image (skimage camera dataset) reveal clear trade-offs. The Frobenius norm error can be computed directly from discarded singular values:

$$\text{Relative Error} = \sqrt{\frac{\sum_{i=k+1}^r \sigma_i^2}{\sum_{i=1}^r \sigma_i^2}}$$

Table 1: Compression Trade-offs for  $512 \times 512$  Camera Image

$k$	Info Retained	Compression Ratio	Relative Error
5	51.2%	2.0%	69.8%
20	78.9%	7.9%	45.9%
50	90.3%	19.6%	31.1%
100	96.2%	39.1%	19.5%

Key finding:  $k = 50$  achieves approximately 5:1 compression while retaining over 90% of image energy. The break-even point occurs at  $k \approx 255$ ; beyond this, SVD "compression" actually expands file size because  $k(m + n + 1) > mn$ .

### 5.2 Visual Quality Assessment

Visual inspection of reconstructed images reveals:

- **k=5**: Heavily blurred, only major structures visible
- **k=20**: Recognizable but significant loss of detail
- **k=50**: Visually almost perfect, minor texture softness
- **k=100**: Nearly indistinguishable from original

The singular value spectrum shows rapid exponential decay—the first 50 values capture most energy while the remaining hundreds contribute minimally. This low-rank structure is characteristic of natural images and explains why SVD compression works.

### 5.3 Comparison with DCT/JPEG

SVD is outperformed by DCT (JPEG) for practical compression. DCT operates on  $8 \times 8$  blocks (localized artifacts) while SVD is global (widespread blurring). DCT is perceptually tuned to discard high-frequency components humans can't perceive; SVD discards mathematically low-variance components. However, SVD excels as a teaching tool for compression fundamentals.

## 6 Conclusions

This project successfully implemented SVD from scratch using the QR algorithm with Householder reflections. Key findings: (1) SVD achieves 5:1 compression with good quality at  $k \approx 50$  for  $512 \times 512$  images, (2) storage scales linearly with  $k$  while error follows non-linear singular value decay, (3) optimal  $k$  lies at the singular value curve's "elbow," and (4) DCT outperforms SVD practically but SVD provides superior analytical insights.

Implementation lessons: Householder reflections proved more stable than Gram-Schmidt; sign consistency between  $U$  and  $V$  requires computing  $U$  from  $V$ ; convergence depends critically on iteration count (typically 100+ for accuracy).

Future work: Implement shifted QR for faster convergence, explore tensor decomposition for direct RGB compression, and investigate SVD-based image denoising.

**Files:** Handwritten\_SVD.py, ImgDecompExample.py     **Dataset:** scikit-image camera (512×512)