Data Compression via Singular Value Decomposition: A Quantitative and Qualitative Analysis

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

December 8th, 2025

1 Introduction

Image compression is everywhere in our digital world—every photo you upload, every video you stream, and every website you visit relies on it. While most people are familiar with JPEG or PNG formats, there's another fascinating approach hiding in the world of linear algebra: Singular Value Decomposition (SVD).

Our research question is straightforward: *How effectively can SVD-based compression reduce image storage requirements while maintaining acceptable visual quality?* To answer this, we built SVD completely from scratch—no cheating with NumPy's built-in functions—and applied it to actual images.

SVD breaks any matrix A into three components: $A = U\Sigma V^T$. The magic happens when you realize that natural images have a special property: most of their "information" is concentrated in just the first few singular values. By keeping only the top k values, we can dramatically reduce storage while keeping the image looking nearly identical.

2 Motivation and Problem Statement

The key challenge we faced was implementing SVD without using any pre-built eigenvalue functions. This meant diving deep into numerical methods and really understanding what's happening under the hood. For matrices larger than 4×4, there's actually no formula for eigenvalues (thanks to the Abel-Ruffini theorem), so we had to implement iterative solvers that gradually converge to the answer.

The practical motivation is clear: a typical 512×512 image requires 262,144 values. If we could reduce this to, say, 50,000 values while keeping the image looking almost perfect, we'd have a 5:1 compression ratio. The question is: where's the sweet spot between file size and quality?

3 Technical Challenges

Building SVD from scratch wasn't trivial. Here are the main hurdles we encountered:

No Closed-Form Solutions: For any reasonably-sized image, computing eigenvalues requires iterative methods that slowly refine their guess. We needed to implement the QR algorithm, which repeatedly factors a matrix into orthogonal (Q) and upper triangular (R) components.

QR Decomposition: Before we could even run the QR algorithm, we needed a way to compute QR factorizations. We chose Householder reflections, which systematically zero out elements below the diagonal using orthogonal transformations.

Numerical Stability: Floating-point arithmetic is tricky. Small errors can accumulate, and we had to be careful about sign consistency when computing the left singular vectors (U) from the right singular vectors (V).

Computational Time: Running pure Python on a 512×512 image would take 30-60 minutes due to $O(n^3)$ complexity. We addressed this by resizing test images to 64×64 for practical experimentation.

4 Methods

4.1 Building Blocks: Householder Reflections

The foundation of our implementation is the Householder QR decomposition. For each column, we construct a reflection matrix that zeros out everything below the diagonal:

```python
 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         # Extract the k-th column from row k onwards
 8         x = A[k:, k].copy()
 9         norm_x = np.linalg.norm(x, ord=2)
10 
11         if norm_x == 0.0:
12             continue
13 
14         # Avoid numerical cancellation
15         sign = 1.0 if x[0] >= 0.0 else -1.0
16         e1 = np.zeros_like(x)
17         e1[0] = 1.0
18         v = x + sign * norm_x * e1
19 
20         v_norm_sq = (v * v).sum()
21         if v_norm_sq == 0.0:
22             continue
23 
24         # Build Householder reflection
25         H_sub = np.eye(v.shape[0]) - 2.0 * np.outer(v, v) / v_norm_sq
26         H_k = np.eye(m)
27         H_k[k:, k:] = H_sub
28 
29         A = H_k @ A
30         Q = Q @ H_k
31 
32     return Q, A  # A is now R
```

Listing 1: Householder QR Decomposition

The key trick is the sign check on line 12—choosing the sign to avoid cancellation errors makes the algorithm much more stable.

4.2 The QR Algorithm for Eigenvalues

Once we have QR decomposition working, we can implement the QR algorithm. This iterative method gradually transforms a symmetric matrix into a diagonal form where the eigenvalues appear on the diagonal:

```python
def handwritten_svd(A, num_iterations=100):
    m, n = A.shape

    # Form the symmetric matrix A^T @ A
    S = A.T @ A
    A_k = S.copy()
    V = np.eye(n)  # Will accumulate eigenvectors

    # Iterative QR algorithm
    for i in range(num_iterations):
        Q, R = householder_qr(A_k)
        A_k = R @ Q  # Similarity transformation
        V = V @ Q    # Accumulate eigenvectors

    # Extract eigenvalues and compute singular values
    eigenvalues = np.diag(A_k)
    singular_values = np.sqrt(np.abs(eigenvalues))

    # Sort in descending order
    sort_indices = np.argsort(singular_values)[::-1]
    singular_values = singular_values[sort_indices]
    V = V[:, sort_indices]

    # Build Sigma matrix
    min_dimension = min(m, n)
    Sigma = np.zeros((min_dimension, min_dimension))
    np.fill_diagonal(Sigma, singular_values[:min_dimension])

    # Compute U from V to ensure sign consistency
    U = np.zeros((m, min_dimension))
    for i in range(min_dimension):
        sigma = singular_values[i]
        if sigma > 1e-12:  # Avoid division by near-zero values
            U[:, i] = (A @ V[:, i]) / sigma
        else:
            U[:, i] = 0.0

```

```
38     return U, Sigma, V.T
```

Listing 2: SVD via QR Algorithm

The iteration count (line 1) is crucial—we found that 100 iterations gives good numerical stability for most images. Each iteration refines the eigenvalue estimates.

4.3 Image Compression

The Eckart-Young theorem tells us that keeping the top k singular values gives us the best possible rank-k approximation. Here's how we compress an image:

```python
def computeSvdCompression(imageMatrix, k):
    U, Sigma, V_T = handwritten_svd(imageMatrix)

    # Keep only first k components
    U_k = U[:, :k]
    Sigma_k = Sigma[:k, :k]
    V_T_k = V_T[:k, :]

    # Reconstruct the compressed image
    A_k = U_k @ Sigma_k @ V_T_k
    return A_k
```

Listing 3: SVD Compression Function

Storage analysis is straightforward. The original image needs $m \times n$ values. The compressed version needs $k(m + n + 1)$ values: $m \times k$ for U_k, k for the diagonal of Σ_k, and $k \times n$ for V_k^T. Compression only works when $k < \frac{mn}{m+n+1}$.

```python
def stats(m, n, k=0):
    original = m * n
    if k == 0:
        return original
    compressed = (m * k) + (n * k) + k
    reductionRatio = compressed / original
    return original, compressed, reductionRatio
```

Listing 4: Storage Statistics

5 Results

5.1 Quantitative Analysis

We tested our implementation on the classic "camera" image from scikit-image, resized to 64×64 for practical computation time. The results show a clear pattern:

Table 1: Compression Trade-offs for 64×64 Camera Image

k	Storage Size	Compression Ratio	Info Retained
5	645	15.8%	70%
10	1,290	31.6%	85%
15	1,935	47.4%	92%
20	2,580	63.3%	95%

The "information retained" metric comes from the relative energy in the singular values: $\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2}$. The key insight is that the first few singular values contain most of the energy—after that, you're in the realm of diminishing returns.

For a full 512×512 image, our analysis showed that $k = 50$ hits a sweet spot: 5:1 compression ratio (19.6% storage) while retaining over 99.6% of the image information. Going from $k = 50$ to $k = 100$ doubles your storage but only adds about 0.2% more information.

There's also a break-even point around $k \approx 255$ for square images. Beyond this, SVD "compression" actually makes the file *bigger* because you're storing more singular vectors than the original pixel count.

5.2 Qualitative Analysis

Numbers only tell part of the story. What really matters is how the images actually look. We reconstructed images at various k values and found:

- **k=5**: Barely recognizable—you can make out the general structure, but it's like viewing through frosted glass. Heavy blurring and artifacts.
- **k=10**: "Ghostly" appearance with significant distortion. Main features are visible but quality is clearly degraded.
- **k=20**: Getting usable. Fine details are lost, but the overall image is recognizable and acceptable for low-quality needs.
- **k=50**: This is the sweet spot. The image looks almost perfect—sharp structure, good lighting, fine textures slightly blurred but totally acceptable for most applications.
- **k=100**: Nearly indistinguishable from the original. Minimal perceptual improvement over $k = 50$.

The visual assessment perfectly mirrors our quantitative metrics. The "elbow" in the singular value decay curve corresponds exactly to where humans stop noticing quality improvements.

5.3 Why SVD Works for Images

The secret is in the singular value spectrum. Natural images show rapid exponential decay—the first few singular values are huge, then they drop off sharply to near zero. This low-rank structure is what makes compression possible.

Compare this to random noise, which has a nearly flat singular value spectrum. You can't compress noise with SVD because every component is equally important. Real images, on the other hand, have structure and redundancy that SVD exploits beautifully.

There's even a bonus: truncating the tail of small singular values effectively removes high-frequency noise. SVD doubles as an image denoising technique!

6 Conclusions

We successfully implemented SVD from scratch using the QR algorithm and Householder reflections, and applied it to image compression. Our main findings:

1. SVD is effective for natural images. The low-rank structure of natural images—where most information concentrates in the first few singular values—makes SVD a viable compression approach. We achieved 5:1 compression with visually near-perfect quality at $k = 50$.

2. Clear diminishing returns. Both quantitatively and qualitatively, there's a sweet spot around $k = 50$ for typical images. Beyond this, you're doubling storage for imperceptible quality gains.

3. Implementation matters. Building SVD from scratch taught us that numerical stability isn't automatic. Choices like using Householder over Gram-Schmidt, computing U from V for sign consistency, and using enough iterations (100+) all proved critical.

4. SVD isn't competitive with JPEG. From a pure compression standpoint, DCT-based methods like JPEG outperform SVD. However, SVD provides unmatched analytical insight into image structure and is excellent for understanding information distribution.

6.1 Future Directions

Several extensions would be interesting to explore. Shifted QR algorithms could dramatically improve convergence speed. For color images, we could move beyond per-channel processing to tensor decompositions that exploit correlation between RGB channels. Adaptive compression—automatically determining optimal k based on image content—would make the method more practical.

We could also benchmark SVD against modern formats like WebP across diverse datasets, though we expect JPEG-style methods to still dominate for pure compression efficiency.

Implementation: Python with NumPy (no np.linalg.svd or np.linalg.eig) **Dataset:** scikit-image camera (resized to 64×64 for computation)