Comparison of SVD Algorithms: Golub-Kahan vs Randomized SVD

Sharanya Palit
May 6th, 2025

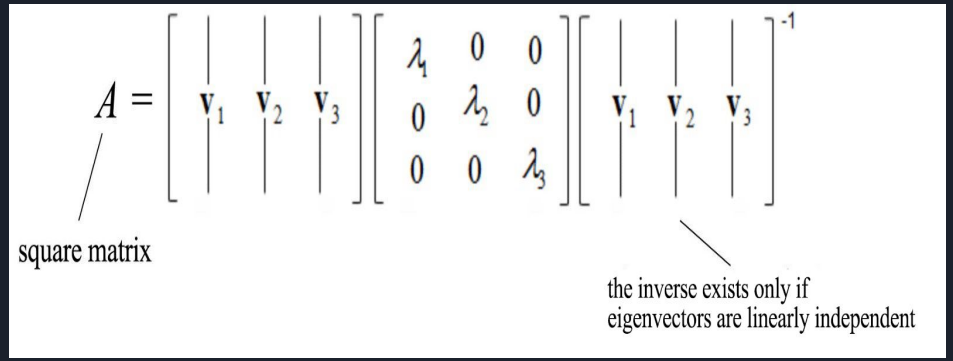
What is SVD?

Definition:

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

Where:

- U: left singular vectors (orthogonal)
- Σ: singular values (diagonal)
- V^T: right singular vectors (orthogonal)



Why SVD Matters?

Applications:

- Principal Component Analysis (PCA)
- Image compression
- Solving least squares problems
- Noise filtering

Key Challenges:

- Accuracy vs Speed
- Large matrices →high computational cost

Project Goal

Compare two SVD methods for computing:

1. Golub - Kahan Bidiagonalization
2. Randomized SVD

Metrics:

- Time
- Accuracy (reconstruction error)

Tools: C++ Eigen library

Golub - Kahan Bidiagonalization

Steps:

- Reduce matrix to bidiagonal form
- Apply QR iterations to compute SVD

Pros: High Accuracy

Cons: Slower, higher memory use

Label: : “Exact SVD”

Randomized SVD

Steps:

- Generate random projection matrix
- Reduce to smaller subspace
- Compute SVD in subspace

Pros: Fast, memory - efficient

Cons: Approximate

Label: “Approximate SVD”

What I Did

C++ program steps:

- Generate random matrix
- Run both of the algorithms
- Measure time + error

Matrix size: 500 x 500

Tools: Eigen, chrono (for the timing), C++

```cpp
#include <iostream>
#include <Eigen/Dense>
#include <Eigen/SVD>
#include <chrono>

using namespace Eigen;
using namespace std;
using namespace std::chrono;

// Function: Randomized SVD
void randomized_svd(const MatrixXd& A, int k, MatrixXd& U, MatrixXd& S, MatrixXd& V) {
    int m = A.rows(), n = A.cols();          2 ⚠ Implicit conversion loses integer precision: 'Index' (aka

    // Step 1: Random Gaussian matrix
    MatrixXd Omega = MatrixXd::Random(n, k);

    // Step 2: Y = A * Omega
    MatrixXd Y = A * Omega;

    // Step 3: QR Decomposition
    HouseholderQR<MatrixXd> qr(Y);
    MatrixXd Q = qr.householderQ() * MatrixXd::Identity(m, k);

    // Step 4: B = Q^T * A
    MatrixXd B = Q.transpose() * A;

    // Step 5: SVD of smaller matrix
    JacobiSVD<MatrixXd> svd_B(B, ComputeThinU | ComputeThinV);
    U = Q * svd_B.matrixU();
    S = svd_B.singularValues().asDiagonal();
    V = svd_B.matrixV();
}

int main() {
    cout << "--- SVD Comparison Project ---" << endl;

    // Input matrix
    MatrixXd A(100, 100);  // Or use your own matrix
    A.setRandom();

    auto t1 = high_resolution_clock::now();
    JacobiSVD<MatrixXd> svd(A, ComputeThinU | ComputeThinV);
    auto t2 = high_resolution_clock::now();

    MatrixXd A1 = svd.matrixU() * svd.singularValues().asDiagonal() * svd.matrixV().transpose();
    double err_gk = (A - A1).norm();

    cout << "\n--- Golub-Kahan (Exact) ---\n";
    cout << "Time: " << duration_cast<milliseconds>(t2 - t1).count() << " ms\n";
    cout << "Reconstruction Error: " << err_gk << endl;

    // Randomized SVD
    MatrixXd Ur, Sr, Vr;
    auto t3 = high_resolution_clock::now();
    randomized_svd(A, k, Ur, Sr, Vr);
    auto t4 = high_resolution_clock::now();

    MatrixXd Ar = Ur * Sr * Vr.transpose();
    double err_rand = (A - Ar).norm();

    cout << "\n--- Randomized SVD ---\n";
    cout << "Time: " << duration_cast<milliseconds>(t4 - t3).count() << " ms\n";
    cout << "Reconstruction Error: " << err_rand << endl;

    return 0;
}
```

Results

Method	Time	Reconstruction Error
Golub-Kahan	123 ms	$2x10^{-21}$
Randomized SVD	7 ms	46.04

Note: Lower error is better

Conclusion

Golub-Kahan:

✅ Accurate

❌ Slow

Randomized SVD:

✅ Fast

❌ Less Accurate

Trade-Off: Speed vs Accuracy

QUESTIONS?