6.5610 Spring 2026 Final Project

Adhitya Mangudy Venkata Ganesh, Nicola Lawford, Ishan Satish Pednekar, Li Xuan Tan

We implement singular value decomposition, an important tool in many data science and machine learning libraries, under the Cheon-Kim-Kim-Song scheme for fully homomorphic encryption. To do this, we use several different algorithms including iterative and randomized reduction methods, and evaluate their performance and accuracy.

1. Introduction

1.1. Introduction to Singular Value Decomposition

A classic linear algebra theorem states that "any matrix $A \in \mathbb{R}^{m \times n}$ can be factored into a singular value decomposition (SVD)

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

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices (i.e., $UU^T = VV^T = I$) and $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal with $r = \text{rank(A)}$ leading positive diagonal entries" (Martin and Porter, 2012).

1.1.1. Truncated Singular Value Decomposition (TSVD)

Truncated SVD is an approximation of a matrix using only the top k singular values.
i.e. where Full SVD writes a matrix A as $A = U \Sigma V^T$, Truncated SVD writes it as $A \approx A_k = U_k \Sigma_k V_k^T$.

Here, U_k contains the first k columns of U, Σ_k contains the top k singular values, and V_k contains the first k columns of V. Thus, truncated SVD represents just the most important rank-k structure of A.

1.2. Applications of Singular Value Decomposition

The following are some uses of SVD (and particularly Truncated SVD):

- **Image compression.** Encrypted image systems often need efficient representations of images before further encrypted computation. A low-rank approximation could reduce the amount of encrypted data that must be stored or processed.
- **Dimensionality reduction.** Truncated SVD reduces high-dimensional data to a smaller number of dominant components.
- **PCA / feature extraction.** If X is a data matrix, the top singular vectors capture the main directions of variation in the data. This is relevant to medical image feature extraction as truncated SVD could be used to compute compact low-dimensional features.

1.3. Fully Homomorphic Encryption Schemes

While most previous work on FHE had focused on exact computations of addition and multiplication, in 2016 Cheon, Kim, Kim, and Song proposed a method of FHE for arithmetic computation using approximate numbers, now evaluating operations including multiplicative inverses (Cheon et al., 2017). We opted for the CKKS scheme as the computation of singular value decomposition requires

floating point numbers, as well as being able to perform single-instruction multiple-data (SIMD) operations thanks to CKKS ciphertexts encoding many numbers simultaneously, which improves performance. CKKS can perform element wise addition and multiplication, and cyclic rotation of its slots homomorphically.

1.4. Uses for SVD in Fully Homomorphic Encryption

Truncated SVD can be used to compress data while preserving a good approximation, which is useful for frequently-proposed FHE applications that operate on large data such as privacy-preserving facial recognition (Ahmed and Yoshiura, 2025; Bowditch et al., 2020), private medical data analysis (Zhang et al., 2024), and encrypted image classification (Rovida and Leporati, 2024a).

2. Design Objectives

2.1. Accuracy

According to the original CKKS design, error is bounded by the depth of a circuit; it is at most one more bit than error in unencrypted arithmetic (Cheon et al., 2017). Thus, our chosen algorithm must minimize circuit depth. We can evaluate accuracy by comparison to the original matrix or, to get a better sense of losses due to encyption, comparison to the computational result in plaintext. Because some of our methods did not reconstruct the full singular value decomposition USV^T and only approximated the columns of U and/or V^T, we instead chose to check the accuracy of computed singular vectors in comparison to the native `numpy` plaintext implementations. In the cases where our optimizations or implementations favored the first singular vector, we measured the accuracy as

$$\text{Accuracy} = u_{0\text{FHE}} \cdot u_{0\text{plaintext}}$$

In the case where implementations computed all vectors at once, we measured accuracy as

$$\text{Accuracy} = \frac{\sum_{i=0}^{n} u_{i\text{FHE}} \cdot u_{i\text{plaintext}}}{n}$$

2.2. Efficiency

Computational efficiency is a consideration for practical uses of our implementation circuit, as well as the feasibility of running our tests on available computing resource. This metric will be impacted by computational depth, such that at high computational depth, accuracy can be improved by performing a bootstrapping operation; however, this operation is very computationally expensive (Gentry, 2009). Due to the large number of other design choices to explore and optimize, we scope out bootstrapping in this project.

We measure efficiency by tracking runtime on consistent hardware specified in Section 4.

2.3. Security

The Homomorphic Encryption Standardization provides tables of attack speeds for common attacks based on the data dimension and key size. According to these tables, for data of dimension 32768, a $\log q$ ciphertext modulus of 881 bits provides > 128-bit security for all listed attacks, i.e. an attack will take $> 2^{128}$ operations (Albrecht et al., 2018). To achieve sufficient accuracy, we set our first

modulus to be 60 bits and the scaling modulus size to be 59 bits, meaning that approximately 14 levels of depth will exceed this 881-bit limit. Our implementations required > 14 levels of depth, and thus fell below the minimum 128-bit security setting offered by OpenFHE. While we had some level of security, it fell below standard settings. We decided to accept this tradeoff in favor of exploring other parameters and limitations within FHE.

3. Threat Model

For our threat model, we assume the following parties:

- The secret holder (client, in a client-server model) generates the FHE keys, encrypts data, and outsources computation of the FHE ciphertexts.
- The honest-but-curious adversary (server) which receives public key, evaluation key and ciphertexts, and performs FHE computations using its computational power.

We operate under the Ring Learning with Errors (RLWE) assumption, whose security is assumed under the parameters in the Homomorphic Encryption Standard (Albrecht et al., 2018). Note that because FHE schemes are malleable by design, a malicious adversary would require verifiable computation to defend against, which is extremely expensive. In addition, this malleability means FHE schemes cannot achieve IND-CCA security as it is trivial to produce related ciphertexts given the public key and evaluation key.

Security of the IND-CPA game follows from the fact that ciphertexts in CKKS are essentially RLWE samples, so having a non-negligible attacker's advantage allows distinguishing RLWE samples from uniform, which is assumed to be hard.

In 2020, Li and Micciancio proposed a new security notion IND-CPAD (IND-CPA Decryption) where the attacker also has access to a decryption oracle, and demonstrated that it is strictly stronger than IND-CPA for approximate encryption schemes by presenting a practical key recovery attack on CKKS libraries (Li and Micciancio, 2020). As such, it is necessary to also ensure that the adversary has no access to any of the decrypted data.

4. Implementation

Code is available on GitHub at https://github.com/matho-o/65610-sp26-project. The subspace iteration library and testing were built using LLM-assisted coding with Claude Sonnet 4.6 (Anthropic, 2025). A part of the code for testing was written with assistance from Gemini 3 Flash (DeepMind, 2026).The power iteration testing code was written with assistance from ChatGPT 5.5 (OpenAI, 2026).

Runtimes were measured on MIT ORCD's Engaging system, which is a mixed-use HPC cluster available to MIT affiliates. The hardware configuration used was 12 CPU cores and 64GB of memory on the `mit_normal` partition, running on Centos 7 for OpenFHE support.

4.1. Basic primitives

4.1.1. Rotations

In CKKS scheme, rotations can be performed on a ciphertext by a specific index i. For each index to rotate with, a Galois key needs to be generated for it. Thus, to save space, and time, we need to choose a set of keys which is small, and yet can rotate by the indi ces we use quickly. Thus, we use

powers of 2 as our rotation keys, in both the positive and negative directions. To compute the optimal rotations we need, we can use a breadth first search (BFS).

4.1.2. Matrices

For our methods, we will work with $n \times n$ matrices, where n is a power of 2. For smaller $m \times k$ matrices $(b_{ij})_{m\times k}$ with $m, k \le n$, we will encode this as $(a_{ij})_{n\times n}$ as

$$a[i][j] = \begin{cases} b[i][j] \text{ if } i \le m, j \le k \\ 0 \text{ otherwise} \end{cases}$$

If a $m \times k$ matrix is multiplied by a $k \times l$ matrix, this multiplication will still be valid when they are seen as $n \times n$ matrices.

In CKKS, cipher texts stores arrays of complex numbers, so we need to use a representation for our matrices. We will use a periodic row major form, which for a $2^k \times 2^k$ matrix $(a[i][j])$ gives a cipher text ct storing its slots as follows:

$$ct[m] = a[\lfloor m/2^k \rfloor \bmod 2^k][m \bmod 2^k]$$

In this format, the matrix is represented by a sequence of its rows one after the another, repeated periodically. For this representation, if there are N slots, we need $N \ge 2^{2k}$ to store the entire matrix in one step, which will be the case in our applications.
The row major form is chosen as in this representation, two encrypted matrices can be multiplied in depth 2. Storing a ciphertext as a sequence of diagonals is another method that we considered, as it takes 1 depth. However, for an $n \times n$ matrix it needs n ciphertexts, and the number of rotations in this form is $O(n^2)$ (in contrast with $O(n \log n)$ in the method described in Section 4.1.3), and its memory and time usage are much greater, so we decided to use the row major method for our purposes. The periodic form is chosen so that during rotations, there are no unintended zeroes in our computations.

4.1.3. Matrix multiplication

We want to perform an algorithm that can minimize the total depth of operations, while at the same time not using too many rotations or multiplications. Also, as our rotation r to use these rotations as much as possible. Also, as in our implementation we often use $AB/2$, and we do not want to use an extra depth on this, we modify our algorithm to handle these as well. This is done by multiplying the constant with the plain-text masks before the encryption step.
We will use the following masks in the Algorithm 1 to compute cAB for a plain-text number c and encrypted matrices A, B:

- **Column Mask (M_{col}):** Encryption of a plaintext vector where slots are 1 if $j \bmod n = 0$, else 0.
- **Row Mask (M_{row}):** Encryption of a plaintext vector where slots are c if $\lfloor j/n^2 \rfloor < n$, else 0. (Here the constant has been multiplied already).

The column and row masks M_{col}, M_{row} can be precomputed for the values of c and n we need. This algorithm takes $O(n \log n)$ rotations ($2n + 2n \log n$ exactly), and $O(n)$ multiplications ($2n$ exactly).

Algorithm 1 Homomorphic Matrix Multiplication (from Aikata (2024)

Require: Ciphertexts A, B, dimension n, scalar c

```
1:  C ← 0, ℓ ← log_2 n
2:  for i = 0 to n − 1 do
3:      Â ← Mul(M_col, Rotate(A, i))
4:      for j = 0 to ℓ − 1 do
5:          Â ← Add(Â, Rotate(Â, −2^j))
6:      end for
7:      B̂ ← Mul(M_row, Rotate(B, n · i))
8:      for j = 0 to ℓ − 1 do
9:          B̂ ← Add(B̂, Rotate(B̂, −n · 2^j))
10:     end for
11:     C ← Add(C, Mul(Â, B̂))
12: end for
13: return C
```

4.1.4. Matrix transpose

Here, we will implement a matrix transpose method described in Jiang et al. (2018). This is depth 1, and uses $O(n)$ rotations and multiplications. The algorithm used is given in Algorithm 2.

4.1.5. Normalization using Chebyshev Approximation

Some of our SVD methods require normalization, which involves a division by the norm of a vector or matrix: $||v|| = \frac{1}{\sqrt{v \cdot v}}$. Because division and square root are not allowed operations in FHE, we use a Chebyshev polynomial approximation, which has been used in a variety of FHE works (Adamek et al., 2025; Chen et al., 2019; Lee et al., 2020; Rovida and Leporati, 2024b). For a Chebyshev approximation of degree d, d points are chosen within an expected range, and the function to be approximated—in this case $\frac{1}{\sqrt{x}}$—is evaluated at these points. Then, a Chebyshev polynomial of the appropriate degree is fit through these points. Thus, a division becomes a polynomial evaluation, which is within the allowed FHE arithmetic operations (Trefethen, 2019). In our final implementation we used a 3-degree Chebyshev approximation, which consumed 3-5 levels of multiplicative depth: one to get the initial dot product, one to three to evaluate the polynomial, and one to multiply the result back into the dividend.

4.2. Power Iteration SVD

4.2.1. Rank-k SVD via Power Iteration and Deflation

To compute a truncated singular value decomposition of a matrix

$$A \in \mathbb{R}^{n \times n},$$

we use a power iteration method combined with deflation. The goal is to approximate the top k singular triplets

$$(\sigma_r, u_r, v_r), \qquad r = 1, \ldots, k,$$

Algorithm 2 Homomorphic Matrix Transposition from Jiang et al. (2018)

```
1:  procedure TRANSPOSE(A, n, pk)
2:      B ← 0
3:      for i = −(n − 1) to (n − 1) do
4:          Initialize pt ← 0^{n^2}
5:          for j = 0 to n − |i| − 1 do
6:              if i > 0 then
7:                  k ← ((n + 1)j + i) mod n^2
8:              else
9:                  k ← ((n + 1)j − i · n) mod n^2
10:             end if
11:             pt[k] ← 1
12:         end for
13:         pt ← replicate(pt, slot_size/n^2)
14:         T_i ← Encrypt_pk(pt)
15:         R_i ← Rotate(A, (n − 1)i)
16:         B ← Add(B, Mult(T_i, R_i))
17:     end for
18:     return B
19: end procedure
```

where

$$Av_r \approx \sigma_r u_r, \qquad A^\top u_r \approx \sigma_r v_r.$$

The algorithm extracts one singular triplet at a time. After each triplet is computed, we remove its rank-one contribution from the matrix. This is the deflation step, which allows the next power iteration to recover the next largest singular triplet.

4.2.2. Correctness:

If the initial vector v has a nonzero component in the direction of the top right singular vector v_1, then repeated multiplication by $A^\top A$ amplifies the v_1 component relative to the others. Indeed,

$$A^\top A v_i = \sigma_i^2 v_i.$$

Thus power iteration on $A^\top A$ converges toward v_1
Our implementation updates $v \leftarrow A^\top A v$ by computing

$$z = Av, \qquad u = \frac{z}{\|z\|}, \qquad w = A^\top u, \qquad v = \frac{w}{\|w\|}.$$

4.2.3. Deflation

After extracting one singular triplet, we remove its contribution from the matrix by setting $A \leftarrow A - \sigma_r u_r v_r^\top$.
After deflation we get $A^{(2)} = A - \sigma_1 u_1 v_1^\top = \sum_{i=2}^{n} \sigma_i u_i v_i^\top$. Thus the largest singular triplet of the deflated matrix $A^{(2)}$ is the second singular triplet of the original matrix A.

Algorithm 3 Rank-k SVD via Power Iteration and Deflation Guillemot et al. (2019)

Require: Matrix $A \in \mathbb{R}^{n \times n}$, target rank k, number of iterations T
Ensure: Approximate singular triplets $\{(\sigma_r, u_r, v_r)\}_{r=1}^{k}$

1: **for** $r = 1, \ldots, k$ **do**
2: Initialize random vector $v_r \in \mathbb{R}^n$
3: Normalize:

$$v_r \leftarrow \frac{v_r}{\|v_r\|}$$

4: **for** $t = 1, \ldots, T$ **do**
5: Multiply by A:

$$z \leftarrow A v_r$$

6: Normalize to approximate the left singular vector:

$$u_r \leftarrow \frac{z}{\|z\|}$$

7: Multiply by $A^\top$:

$$w \leftarrow A^\top u_r$$

8: Normalize to update the right singular vector:

$$v_r \leftarrow \frac{w}{\|w\|}$$

9: **end for**
10: Compute final left vector and singular value:

$$z \leftarrow A v_r, \qquad \sigma_r \leftarrow \|z\|, \qquad u_r \leftarrow \frac{z}{\sigma_r}$$

11: Store the singular triplet:

$$(\sigma_r, u_r, v_r)$$

12: Deflate the matrix:

$$A \leftarrow A - \sigma_r u_r v_r^\top$$

13: **end for**
14: **return** $\{(\sigma_r, u_r, v_r)\}_{r=1}^{k}$

Thus, after $r-1$ deflations, the matrix becomes

$$A^{(r)} = A - \sum_{i=1}^{r-1} \sigma_i u_i v_i^{\top}.$$

Thus, performing power iteration on $A^{(r)}$ returns the r-th singular triplet of the original matrix,and we can recover a rank-k approximation by repeatedly applying power iteration and deflation.

4.2.4. Normalization

The normalization steps $u_r \leftarrow \frac{z}{\|z\|}, \quad v_r \leftarrow \frac{w}{\|w\|}$ are necessary to prevent the norm of the vectors from growing or shrinking exponentially after repeated matrix multiplication.

In plaintext, it's feasible to perform normalization at every iteration. However, in the fully homomorphic encryption setting, normalization is expensive because it requires approximating an inverse square root, the Chebyshev approximation consumes significant multiplicative depth. In our implementation for example, each normalisation takes depth 5, whereas each matvec multiplication only takes depth 2. Therefore, we normalize only every few iterations rather than after every multiplication.

4.3. Subspace Iteration SVD

4.3.1. Algorithm

Subspace iteration does not compute the full singular value decomposition of $A = USV^T$; rather, it computes a *putative basis matrix* Q that approximates V^T, as Y approximates U. This is done by repeatedly multiplying a randomly drawn matrix Ω into A, and computing a QR factorization at each step, as shown in Algorithm 4.

Algorithm 4 Subspace Iteration Algorithm from Halko et al. (2010)

1: Draw an $n \times l$ standard Gaussian matrix Ω.
2: Form $Y_0 = A\Omega$ and compute its QR factorization $Y_0 = Q_0 R_0$.
3: **for** $j = 1, 2, ..., q$ **do**
4: Form $\tilde{Y}_j = A^T Q_{j-1}$ and compute its QR factorization $\tilde{Y}_j = \tilde{Q}_j \tilde{R}_j$.
5: Form $\tilde{Y}_j = A\tilde{Q}_j$ and compute its QR factorization $Y_j = Q_j R_j$.
6: **end for**
7: $Q = Q_q$

4.3.2. Implementation Decisions

QR factorization involves decomposing a matrix into $A = QR$, in which Q has orthonormal columns and R is weakly upper triangular (Halko et al. (2010)). To do this, we must perform orthogonalization and normalization within FHE. Zhang et al. (2019) implemented homomorphically encrypted QR factorization using a Gram-Schmidt process. This computes an orthogonal basis one vector at a time. Each consecutive orthogonal is generated by subtracting the projects of the vector onto the already-chosen orthogonal vectors. As a result, it is more accurate for the first vector, and more error accumulates for vectors selected later due to increasing multiplicative depth. Algorithm 5 shows

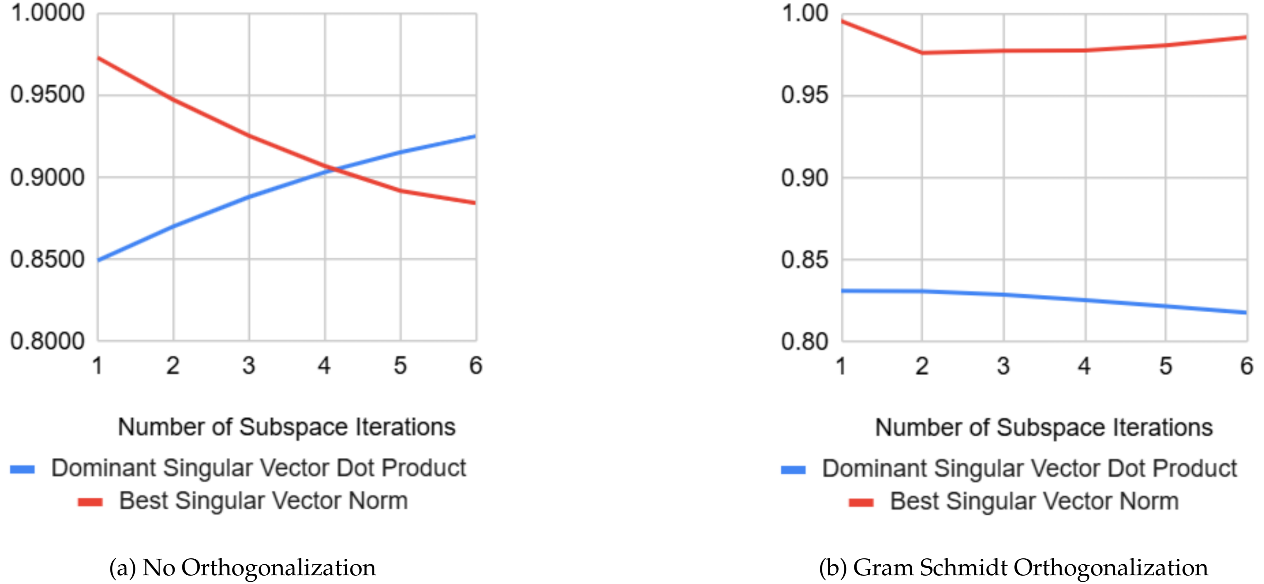
the Gram-Schmidt process to compute an orthonormal basis $\{q_1, ..., q_n\}$ of $Y = y_1, ..., y_n$. Note that instead of computing the division by the dot products at each step, we normalize in one step at the end. Normalization is done using the Chebyshev approximation, explained in 4.1. To control the range of inputs to this function, we normalize the matrix prior to encryption. Due to the large loss of depth in the latter columns in a Gram Schmidt orthogonalization, we implemented subspace iteration with and without the Gram Schmidt process, only normalizing.

Algorithm 5 Gram-Schmidt Process from Farahmand (2018)

1: $q_1 = y_1$
2: $q_2 = y_2 - \frac{y_2 \cdot q_1}{q_1 \cdot q_1} q_1$
3: $q_3 = y_3 - \frac{y_3 \cdot q_1}{q_1 \cdot q_1} q_1 - \frac{y_3 \cdot q_2}{q_2 \cdot q_2} q_2$
4: ...
5: $q_n = y_n - \frac{y_n \cdot q_1}{q_1 \cdot q_1} q_1 - \frac{y_n \cdot q_2}{q_2 \cdot q_2} q_2 - ... - \frac{y_n \cdot q_{n-1}}{q_{n-1} \cdot q_{n-1}} q_{n-1}$

4.3.3. Results

Results are shown in Figure **1**. We report the accuracy of the best singular vector u_1 due to the Gram Schmidt process accuracy declining for consecutive vectors; average accuracy also improved with iterations, but values ranged within the $[0.5, 0.6]$ range. Note that without orthogonalization, the best singular vector improved with more subspace iterations, but norms fell due to Chebyshev approximation error. In this sweep (Figure **1**a), multiplicative depth ranged from 20 to 90. Introducing the Gram Schmidt orthogonalization increased multiplicative depth substantially, such that in Figure **1**b multiplicative depth ranges from 41 to 141. While norms remained closer to 1— likely due to the orthogonalization keeping the components nearer to the center of the Chebyshev range—any gains in accuracy of the primary vector were canceled by this much larger multiplicative depth introducing greater error. We scoped out Power Iteration results from the final evaluation because of its high multiplicative depth(because of the normalization and deflation steps), which made it less scalable than the two other algorithms.



(a) No Orthogonalization

(b) Gram Schmidt Orthogonalization

Fig. 1: Subspace Iteration SVD Computation Accuracy and Norms, With and Without Orthogonalization. Computed on a 4×4 Frobenius-normalized matrix with $k = 4$ and a 3-degree Chebyshev polynomial approximation for normalization.

4.4. Randomized tSVD

In 2010, Halko, Martinsson and Tropp proposed a randomized algorithm for the truncated SVD problem that surpasses iterative algorithms in speed (especially with larger matrices) and is competitive in accuracy (Halko et al. (2010)). The additional slowdown imparted by FHE computations, especially matrix multiplications, makes it difficult to scale power or subspace iteration to larger matrices. In our results shown below, we observed a significant dropoff in subspace iteration accuracy in the truncated SVD case; compared to the previous section with full-rank results, we tested here with half-rank or quarter-rank SVD and saw predictably worse results. Randomized SVD performed slightly better at an initially large computational cost, but scales better as input matrices grow.

This randomized algorithm takes a random sample of the original matrix with a Gaussian test matrix sized based on the target number of singular vectors, then computes the orthonormal basis for this sample matrix (either using QR factorization as above, or Newton-Schulz iteration). Using the orthonormal basis, the original matrix is then projected into a smaller subspace that captures most of the action of the original, since the random sample is likely to retain the most dominant singular vectors. A full-rank SVD of the subspace is then computed using either of the iterative methods above, then projected back to the original subspace using the orthonormal basis. Since full-rank SVD is only required to be computed on a small matrix, this randomized approach saves significantly on computation, with Halko et al. (2010) finding a complexity of $O(mn \log k)$ for the top k singular vectors of a $m \times n$ matrix as opposed to $O(mnk)$ for classical algorithms.

PROTOTYPE FOR RANDOMIZED SVD

Given an $m \times n$ matrix $\boldsymbol{A}$, a target number k of singular vectors, and an exponent q (say $q = 1$ or $q = 2$), this procedure computes an approximate rank-$2k$ factorization $\boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is nonnegative and diagonal.

Stage A:

1 Generate an $n \times 2k$ Gaussian test matrix $\boldsymbol{\Omega}$.
2 Form $\boldsymbol{Y} = (\boldsymbol{AA}^*)^q \boldsymbol{A\Omega}$ by multiplying alternately with $\boldsymbol{A}$ and $\boldsymbol{A}^*$.
3 Construct a matrix $\boldsymbol{Q}$ whose columns form an orthonormal basis for the range of $\boldsymbol{Y}$.

Stage B:

4 Form $\boldsymbol{B} = \boldsymbol{Q}^*\boldsymbol{A}$.
5 Compute an SVD of the small matrix: $\boldsymbol{B} = \widetilde{\boldsymbol{U}}\boldsymbol{\Sigma V}^*$.
6 Set $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.

Note: The computation of $\boldsymbol{Y}$ in Step 2 is vulnerable to round-off errors. When high accuracy is required, we must incorporate an orthonormalization step between each application of $\boldsymbol{A}$ and $\boldsymbol{A}^*$; see Algorithm 4.4.

Outline for the randomized SVD algorithm, reproduced from Halko et al. (2010).

Implementing this in FHE requires matrix multiplications and transposes (discussed in 4.1), a full-rank SVD algorithm (discussed in 4.2 and 4.3), and a method of computing the orthonormal basis. As opposed to the Gram-Schmidt orthogonalization used in 4.3, we apply the method of Newton-Schulz iteration to orthogonalize Y in the reduction from large A to small B. For a given matrix X, the Newton-Schulz iteration is given below:

$$X_{i+1} = \frac{3}{2}X_i - \frac{1}{2}X_i X_i^T X_i, \qquad X_0 = X.$$

This choice is because Gram-Schmidt requires large numbers of vector operations (namely dot products for projection) which aren't native to our SVD implementation and would be expensive to compute for the larger matrix Y, whereas Newton-Schulz depends solely on matrix multiplications and is thus simple to implement but requires several iterations to converge. Gram-Schmidt is retained in the full-rank subspace iteration SVD algorithm that we perform as a subroutine to recover the SVD of the small matrix B. A recent preprint also proposed a Chebyshev-optimized version of the Newton-Schulz iteration, but we were not able to implement this in our final project (Grishina et al. (2026)).

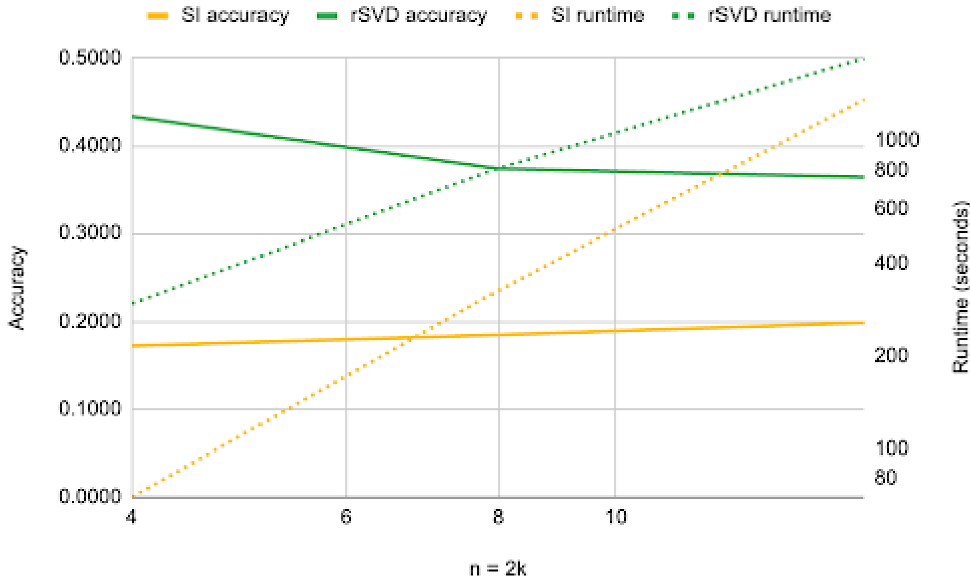


Fig. 2: Accuracy and runtime of subspace iteration compared to Halko et al.'s randomized SVD algorithm. Accuracy is measured by average cosine similarity to the singular vectors compared to a reference implementation from numpy, with one iteration of subspace iteration at Chebyshev degree 3.

5. Discussion

From the results above, it can be seen that the algorithms currently don't work well with larger matrices. The issue here is that in the normalization step, while doing the Chebyshev interpolation, we need to know the range of values of the vector being normalized. While running full SVD algorithms with randomized truncated SVD, as the intermediate matrix is unknown, the range of values of these are also unknown. For larger matrices, this range will also increase. Due to these issues, the accuracy of the approximation reduces, and to fix this, we need more multiplicative depth.
Many of our algorithms are limited by the multiplicative depth available. Algorithms such as the Gram Schmidt algorithm and the normalization steps consume a lot of multiplicative depth to work. With faster bootstrapping, there would be the possibility of running these more accurately and for more iterations, which can improve the accuracy.

6. Team member contributions

- Adhitya researched algorithm ideas for SVD and matrix operations, and implemented matrix multiplication and transposition (Section 4.1).
- Nicola researched design objectives and implemented the subspace iteration SVD algorithm.
- Ishan researched SVD algorithms and implemented the power iteration algorithm.
- Li Xuan researched and implemented algorithms specifically for truncated SVD (Section 4.4), and wrote up instructions for setting up the OpenFHE toolchain on the ORCD Engaging cluster.

References

J. Adamek, A. Aikata, A. Al Badawi, A. Alexandru, A. Arakelov, G. Arakelov, P. Binfet, V. Correa, J. Dumezy, S. Gomenyuk, et al. Fherma cookbook: Fhe components for privacy-preserving applications. In *Proceedings of the 13th Workshop on Encrypted Computing & Applied Homomorphic Computing*, pages 68–76, 2025.

S. Ahmed and N. Yoshiura. Beyond face blurring: Privacy-preserving surveillance via homomorphic encryption and encrypted facial representations. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 43–56. Springer, 2025.

Aikata. Encrypted matrix multiplication, 2024. URL https://fherma.io/content/65de4152bfa5f4ea4471701e.

M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

Anthropic. Claude sonnet 4.6 [large language model], 2025. URL https://claude.ai/. Accessed: May 12, 2026.

W. Bowditch, W. Abramson, W. J. Buchanan, N. Pitropakis, and A. J. Hall. Privacy-preserving surveillance methods using homomorphic encryption. In *ICISSP*, pages 240–248, 2020.

H. Chen, I. Chillotti, and Y. Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.

J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

G. DeepMind. Gemini 3.1 flash [large language model], 2026. URL https://gemini.google.com. Accessed: May 12, 2026.

A. Farahmand. 6.4 the gram-schmidt procedure, 2018. URL https://math.berkeley.edu/~arash/54/notes/6_4.pdf.

C. Gentry. A fully homomorphic encryption scheme, 2009.

E. Grishina, M. Smirnov, and M. Rakhuba. Accelerating newton-schulz iteration for orthogonalization via chebyshev-type polynomials, 2026. URL https://arxiv.org/abs/2506.10935.

V. Guillemot, D. Beaton, A. Gloaguen, A. Strang, F. Carbonell, T. E. Nichols, A. C. Evans, R. Adalat, J. P. Lerch, and M. M. Chakravarty. A constrained singular value decomposition method that integrates sparsity and orthogonality. *PLOS ONE*, 14(3):e0211463, 2019. doi: 10.1371/journal.pone.0211463.

N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2010. URL https://arxiv.org/abs/0909.4061.

X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1209–1222, 2018.

Y. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No. Near-optimal polynomial for modulus reduction using l2-norm for approximate homomorphic encryption. *IEEE Access*, 8:144321–144330, 2020.

B. Li and D. Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Paper 2020/1533, 2020. URL https://eprint.iacr.org/2020/1533.

C. D. Martin and M. A. Porter. The extraordinary svd. *The American Mathematical Monthly*, 119(10): 838–851, 2012.

OpenAI. ChatGPT (GPT-5.5 Thinking). https://chat.openai.com/, May 2026. Response to prompt: “Explain CKKS for an FHE SVD project”.

L. Rovida and A. Leporati. Encrypted image classification with low memory footprint using fully homomorphic encryption. *International journal of neural systems*, 34(05):2450025, 2024a.

L. Rovida and A. Leporati. Transformer-based language models and homomorphic encryption: An intersection with bert-tiny. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*, pages 3–13, 2024b.

L. N. Trefethen. *Approximation theory and approximation practice, extended edition*. SIAM, 2019.

J. Zhang, X. Xiao, W. Ren, Y. Zhang, et al. Privacy-preserving feature extraction for medical images based on fully homomorphic encryption. *Journal of Advanced Computing Systems*, 4(2):15–28, 2024.

Y. Zhang, P. Zheng, and W. Luo. Privacy-preserving outsourcing computation of qr decomposition in the encrypted domain. In *2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, pages 389–396. IEEE Computer Society, 2019.