Truncating the SVD via the Lanczos Algorithm for Fast-Data Visualization

Marco Bornstein

December 19, 2021

1 Introduction

Since the beginning of the digital age, the size and quantity of data sets have grown exponentially because of the proliferation of data captured by mobile devices, vehicles, cameras, microphones, and other internet of things (IoT) devices. One report, from the International Data Corporation (IDC), found that the amount of data created or replicated across the world in 2018 was estimated to be 33 zettabytes. The IDC report found that the total data created or replicated will balloon to 175 zettabytes by 2025 [8].

For many companies and researchers, this influx of data can be overwhelming to store, process, and analyze. Larger sets of data are more complex to interpret and less intuitive to explain or present to others. Data visualization is a powerful tool to provide meaning to the ever-increasing size and complexity of data. Statistician Antony Unwin well describes data visualization as a "useful for data cleaning, exploring data structure, detecting outliers and unusual groups, identifying trends and clusters, spotting local patterns, evaluating modeling output, and presenting results" [10]. Visualizing data may yield results that models miss, like outlying data, clusterings, and unusual distributions of data. In fact, multiple works have combined visual methods with classifiers to understand cluster structure in data and better classify unseen data [2, 5, 6].

Within my final course project, I had the goal to explore one of the most familiar and widely used algorithms for data visualization, Principal Component Analysis (PCA), and potentially improve its computational efficiency. Current state-of-the-art implementations of PCA, like in the "scikit-learn" machine learning library in Python, utilize a novel method covered in class: Randomized Singular Value Decomposition (RSVD) [3]. While I will provide an overview of PCA and the Singular Value Decomposition (SVD), I will not dive into detail on Halko's work [3] of RSVD as it was covered in lecture and is not the focus of this project. Instead, I detail a new implementation of PCA using *another* method covered in class: the Lanczos algorithm.

In this project, I propose using the Lanczos algorithm to approximate the truncated SVD of a large-data matrix. I then project this data matrix into a lower dimension to be visualized in two or three dimensions. Furthermore, I will be implementing the Lanczos algorithm in parallel to enhance its efficiency compared to a serial implementation. I will show that my Lanczos-based PCA algorithm is accurate and more efficient than computing the full SVD during PCA. Furthermore, my Lanczos-based PCA algorithm rivals (and potentially surpasses) the efficiency of the state-of-the-art RSVD implementation of PCA.

2 Principal Component Analysis

As detailed above, PCA is a common algorithm for data visualization. The goal of PCA is to reduce the dimensionality of a given problem. To accomplish this, PCA transforms a given data set, where features of the data are interrelated, into a reduced unique set of orthogonal axes (principal components) where the data has the largest variance. Variance is a measure of the variability of the data, and thus one wants to maximize the uniqueness or variability of the data when projecting it onto a smaller space. Thus, PCA aims to reduce the dimensionality of a data set so that when dropping higher dimensions, the loss of data is minimal.

2.1 Deriving the Principal Component Analysis

We begin with a data matrix $D \in \mathbb{R}^{m \times n}$ defined as

$$D = \big(d_1, \ldots, d_n\big) \tag{1}$$

where each column $d_i \in \mathbb{R}^m$ (for $i = 1, \ldots, n$) is a vector of observations of the ith feature for all m data samples. It is assumed that this data is normalized by removing the mean and scaling to unit variance. To determine a linear combination of the original features d_i which maximize the variance, we first define a normalized vector of constants $\alpha \in \mathbb{R}^n$ used to weight this linear combination. The corresponding linear combination is defined as:

$$\sum_{i=1}^{n} \alpha_i d_i = D\alpha, \quad ||\alpha||_2 = 1 \tag{2}$$

As mentioned in [4], the variance of any linear combination is defined as:

$$\text{var}(D\alpha) = \alpha^T C \alpha \tag{3}$$

The matrix $C = \frac{1}{n-1} D^T D \in \mathbb{R}^{n \times n}$ is the covariance matrix associated with the data matrix D. Therefore, maximizing the variance can be written as the following optimization problem:

$$\underset{\alpha}{\text{argmax}}\, \alpha^T C \alpha \;\; \text{s.t.} ||\alpha||_2 = 1 \longrightarrow \underset{\alpha}{\text{argmax}}\, \alpha^T C \alpha - \lambda(\alpha^T \alpha - 1) \tag{4}$$

The right-hand side of Equation (4) is the Lagrangian relaxation where λ is the Lagrange multiplier. Taking the derivative of the Lagrangian optimization formulation with respect to α and setting it equal to zero, in order to determine the maximum, yields:

$$2C\alpha - \lambda(2\alpha) = 0 \longrightarrow C\alpha = \lambda\alpha \tag{5}$$

This reveals an eigenvalue problem, with α being the eigenvector of C and λ being its corresponding eigenvalue. Returning to Equation (3), one can rewrite it using Equation (5):

$$\text{var}(D\alpha) = \alpha^T C \alpha = \alpha^T \lambda \alpha = \lambda \alpha^T \alpha = \lambda \tag{6}$$

We see from the equation above that the variance is equal to the eigenvalues of C. Therefore, the maximal variance will be the maximal eigenvalue of C: λ_1. A convenient property of covariance matrices, are that they are real symmetric matrices. Thus, their eigenvectors form an orthonormal set of vectors. Knowing this, the eigenvectors v_i (for $i = 1, \ldots, n$) of C are suitable candidates for the set of principal components. The suitability emerges because the principal components are sought to be uncorrelated (orthogonal) with one another. The principal component scores can be formally defined as:

$$i\text{th Principal Component Scores} = Dv_i, \quad i = 1, \ldots, n \tag{7}$$

2.2 Relationship to the Singular Value Decomposition

In the derivation of PCA above, the covariance of the data matrix C is equal to $\frac{1}{n-1}D^TD$. The data matrix D can be decomposed via the SVD such that:

$$D = USV^T \tag{8}$$

In the SVD above, U (of size $m \times m$) and V (of size $n \times n$) are orthogonal matrices. The columns of U are the left singular vectors of D (and the eigenvectors of DD^T). The columns of V are the right singular vectors of D (and the eigenvectors of D^TD). The diagonal matrix $S \in \mathbb{R}^{m \times n}$ contains the singular values σ of D and the non-negative square roots of the *non-zero eigenvalues of* D^TD. Knowing this, we can rewrite the covariance matrix C (via the SVD and orthogonality) as:

$$C = \frac{1}{n-1}D^TD = \frac{1}{n-1}(USV^T)^TUSV^T = \frac{1}{n-1}VS^2V^T \tag{9}$$

The eigenvectors of C are therefore the columns of V. Now, utilizing Equations (7) and (8), one can see that all of the principal components can be determined after computing the SVD of D as:

$$DV = USV^TV = US \tag{10}$$

Therefore, by computing the SVD of D, one can find the principal components by multiplying the left singular vectors U by the singular values S. To perform PCA, one computes the SVD of D and uses the outputted left singular vectors and singular values to find the principal component scores.

2.3 Using Principal Component Analysis for Dimensionality Reduction

As mentioned at the beginning of this section, the principal components are ordered with respect to how much each component maximizes the variance of the projected data. Shown in Equation (7) (and the relationship in Equation (10)), the first principal component score is $Dv_1 = \sigma_1 u_1$ (where u_1 is the first column of U and σ_1 is the largest singular value of D).

A low rank approximation of D can be constructed by using a small subset of the principal components. For example, a rank-c approximation D_c is constructed by taking the first c principal component scores. This is written (utilizing the orthogonal property of V) as

$$DV_c = USV^TV_c = US\begin{pmatrix} I \\ 0 \end{pmatrix} = U_cS_c \implies D = U_cS_cV_c^T \tag{11}$$

where U_c (of dimension $m \times c$) are the first c columns of U, S_c (of dimension $c \times c$) is a diagonal matrix containing the c largest singular values of D, and V_c (of dimension $n \times c$) are the first c columns of V. Thus, as shown in Equation (11), one can utilize the first c components of the SVD to create a rank-c approximation of D.

3 Lanczos Approximation to the Singular Value Decomposition

Section 2 ends by showing that a low rank approximation of D can be constructed with only a small subset of the principal components. Therefore, by Equation (11), a c-rank approximation of D only requires the first (largest) c singular values of D and the first c left singular vectors. In light of this, I chose to find an algorithm that can quickly find the c largest singular values and singular vectors of a matrix. One such algorithm is the Lanczos algorithm.

3.1 Overview of the Lanczos Algorithm

The Lanczos algorithm, developed by Cornelius Lanczos, is closely related to power methods for finding the extreme eigenvalues and eigenvectors of a Hermitian matrix A. Below, in Algorithm 1, I present the Lanczos algorithm as shown in [9].

Algorithm 1: Lanczos Algorithm

```
Choose an initial vector v_1 of 2-norm unity
Set β_1 = 0, v_0 = 0
for j = 1, 2, ..., k do
    w_j = A v_j − β_j v_{j−1}
    α_j = ⟨w_j, v_j⟩
    w_j = w_j − α_j v_j
    β_{j+1} = ||w_j||_2
    if β_{j+1} = 0 then
        Terminate Algorithm
    end
    v_{j+1} = w_j / β_{j+1}
end
```

This Lanczos algorithm is the symmetric Lanczos algorithm, as the input into this algorithm is a Hermitian matrix A of dimension n x n, and a specified number of iterations k. Typically, the value of k is smaller than the dimension of A and falls in the range $1 \leq k \leq n$. In practice, the extreme eigenvalues and eigenvectors of A can be approximated through the Lanczos algorithm for values of k that are much smaller than n. This saves time and is why the Lanczos algorithm is popular.

As described in [9], the Lanczos algorithm is a simplification of the Arnoldi method for symmetric matrices. Arnoldi's method arose as a means of reducing dense matrices into Hessenberg form through unitary transformations. Arnoldi believed that the eigenvalues of the resulting Hessenberg matrix obtained from a number of steps fewer than the number of columns "could provide accurate approximations to some eigenvalues of the original matrix" [9]. Applying the Arnoldi method for k iterations to a symmetric matrix A, as the Lanczos algorithm does, results in a tri-diagonal matrix T_k rather than a Hessenberg matrix.

This leads to a three-term recurrence in the Arnoldi process which the Lanczos algorithm utilizes. Therefore, the Lanczos algorithm differs from the Arnoldi method in that the generation of Krylov vectors is more computationally expensive for Arnoldi, since k previous vectors are needed at step k for Arnoldi rather than 2 for Lanczos, and the reduced matrix becomes tri-diagonal for Lanczos and not Hessenberg.

The outputted tri-diagonal matrix from the Lanczos algorithm is constructed as the following:

$$T_k = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_k \\ & & & \beta_k & \alpha_k \end{pmatrix} \tag{12}$$

Also outputted from the Lanczos algorithm are the orthogonal vectors v_j which form the matrix V_k (holding the k vectors). These are the vectors that transform A into a tri-diagonal matrix:

$$T_k = V_k^T A V_k \tag{13}$$

It is important to note, however, that exact orthogonality of these vectors is observed only at the begin-

ning of the Lanczos iterations. The vectors v_j then rapidly lose their orthogonality to one another in the algorithm [9]. To ensure orthogonality, one must fully re-orthogonalize the vectors at each iteration or partially re-orthogonalize. There has been much research on implementing and analyzing full and partial re-orthogonalization of the vectors, which I will not address here. Instead, I will address my own algorithm where I incorporate full re-orthogonalization to ensure the outputted matrix V_k is orthogonal. As I state later, using a partial orthogonalization scheme would likely further boost the efficiency of my algorithm and merits future research.

3.2 Approximating Eigenvalues and Eigenvectors via Lanczos

The power of the Lanczos algorithm, is that the eigenvalues of the outputted tri-diagonal matrix T_k provide a close approximation to the extreme eigenvalues (maximum eigenvalues) of the inputted matrix A even for a small value of k. The dimension of T_k, as shown in Equation (12), is k x k. Therefore, computing the eigenvalues of T_k can be very fast when using symmetric eigensolvers (like "eigh" in numpy) due to the small size of T_k. This is certainly much faster than using the eigensolvers on A, which are much larger in comparison to T_k (for small values of k).

Likewise, the eigenvectors of T_k and unitary transformation vectors V_k can approximate the eigenvectos of A. Below is a theorem from [1] that describes this eigenvector relationship.

Theorem 1. *Suppose we have run the Lanczos iteration for k steps without termination on its own (i.e. have produced T_k). Since $T_k \in S^n$, we can find an orthonormal matrix Q_k that diagonalizes T_k:*

$$Q_k^T T_k Q_k = \begin{pmatrix} \theta_1 & & \\ & \ddots & \\ & & \theta_k \end{pmatrix} = E_k$$

Let $Y_k = V_k Q_k \in \mathbb{R}^{n \times k}$; $Y_k = [y_1, \ldots, y_k]$, where y_i are column vectors. Then: y_i are "close" to the eigenvectors of A and θ_i are "close" to the eigenvalues of A. Specifically: $||Ay_i - \theta_i y_i||_2 = |\beta_k| \cdot |Q_{ki}|$, where Q_{ki} is the (k, i)th element of Q.

From both Equation (13) and Theorem 1, one can decompose matrix A as:

$$A = V_k T_k V_k^T = V_k Q_k E_k Q_k^T V_k^T = (V_k Q_k) E_k (V_k Q_k)^T \tag{14}$$

From the equation above, approximations to the eigenvectors of A can be computed by multiplying the the two transformation matrices together $V_k Q_k$. Since T_k is a symmetric matrix, the orthonormal matrix Q_k which diagonalizes it are the eigenvectors of T_k. Therefore, one can find not only the eigenvalues of T_k when using the symmetric eigensolver, but also the eigenvectors Q_k. Knowing this, one call of "eigh" on numpy will compute the approximated eigenvalues of A as well as the eigenvectors of T_k necessary to compute the approximated eigenvectors of A.

3.3 Approximating the Singular Values and Singular Vectors via Lanczos

The Lanczos algorithm can easily approximate the singular values and singular vectors for any matrix $B \in \mathbb{R}^{m \times n}$, including if B *is not* symmetric. One method of doing this, is by performing the Lanczos algorithm of $B^T B$ (also known as the covariance matrix of B, shown in Section 2). In this method, only the singular values of B and the right singular vectors of B will be computed. The reason for this, is that the eigenvalues of $B^T B$ can be approximated via the Lanczos algorithm (as shown in Section 3.2). Thus,

the singular values of B can be determined by taking the square roots of the non-negative eigenvalues of B^TB. The right singular vectors of B are defined as the eigenvectors of B^TB, so these are found when approximating the eigenvectors of B^TB as in Section 3.2.

Another method for approximating the singular values and vectors of a matrix B, is to first pad the matrix to include B^T within it (and thus make it symmetric):

$$\bar{B} = \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} \tag{15}$$

While $\bar{B}$ is symmetric, the dimension of the matrix has increased to $(m+n)$ x $(m+n)$. An interesting property of this matrix, is that the positive eigenvalues of this matrix are equivalent to the singular values of B (the negative eigenvalues are simply the negative of the singular values). This is shown by solving for the eigenvalues by hand:

$$\det(\bar{B} - \lambda I) = \det \begin{pmatrix} -\lambda I & B^T \\ B & -\lambda I \end{pmatrix} = 0 \tag{16}$$

By definition of the determinant of block matrices with an invertible bottom right block (via Schur complements) [7] we can reduce the determinant above to be:

$$\det \begin{pmatrix} -\lambda I & B^T \\ B & -\lambda I \end{pmatrix} = \det(-\lambda I)\det(\frac{B^TB}{\lambda} - \lambda I) = 0 \rightarrow \det(\frac{B^TB}{\lambda} - \lambda I) = 0 \rightarrow \det(B^TB - \lambda^2 I) = 0 \tag{17}$$

Thus, the eigenvalues of $\bar{B}$ are in fact the singular values of B. The corresponding eigenvectors of this padded matrix will be the corresponding singular vectors of B (including both the left and right singular vectors). The eigenvectors of this matrix will have $(m+n)$ rows, and the first m rows form the left singular vectors and the next n rows form the right singular vectors. Therefore, applying the Lanczos algorithm to approximate the eigenvalues and eigenvectors of $\bar{B}$ (as shown in Section 3.2) will result in approximating the singular values and vectors of B. This is the method that I utilize in my fast data visualization algorithm.

4 Lanczos Fast-Data Visualization

After overviewing the PCA, SVD, and Lanczos algorithms, I will now present my algorithm for fast data visualization. Assume we are provided with a large data matrix $D \in \mathbb{R}^{m \times n}$, with $m >> n$ (D is a tall and skinny, rectangular-shaped matrix) which is normalized by removing the mean and scaling to unit variance. My algorithm combines PCA, SVD, and Lanczos to find the first two (2D) or three (3D) principal components of D efficiently before constructing a low-rank approximation, for visualization, of D using these components. In the following subsections, I detail the steps of the algorithm before providing the full algorithm in pseudo-code. My code for the algorithm is found in my GitHub repository: `https://github.com/Marcob1996/Parallel-Lanczos-PCA`.

4.1 Data Preparation and Lanczos Approximation

The first hurdle I encountered was to transform the data matrix into a symmetric matrix ready for application of the Lanczos algorithm, and without the use of costly matrix operations. As described in Section 3.3, I transformed the data matrix D into the padded and symmetric matrix $\bar{D}$. Figure 1 depicts this transformation. I utilized this method to avoid calculating the covariance matrix of D (which can be expensive when D is very large). The downside of this method to make D is symmetric is that the size of the matrix has

dramatically increased, eating up memory and making the matrix-vector products more expensive within the Lanczos algorithm. This showdown led to a solution as described in the next paragraph.

To avoid excess-memory usage, I realized that I did not have to explicitly form $\bar{D}$. Instead, I could utilize the block structure of $\bar{D}$. The inputted matrix within the Lanczos algorithm is only used once per iteration. This arises when computing the matrix-vector product between A and v_j, as shown in Line 4 of Algorithm 1. Instead of explicitly forming $\bar{D}$ and computing the matrix-vector product, I can split up the computations. I can compute the matrix-vector product of D^T and the last m rows of v_j and concatenate them with the matrix-vector product of D and the first n rows of v_j. Let v_j^m represent the last m rows of v_j ($v_j^m = [v_j]_{(n+1):(m+n)}$). Let v_j^n represent the first n rows of v_j ($v_j^n = [v_j]_{1:n}$). This simplification is depicted in the following equation.

$$Dv_j = \begin{pmatrix} 0 & D^T \\ D & 0 \end{pmatrix} v_j = \begin{pmatrix} 0 & D^T \\ D & 0 \end{pmatrix} \begin{pmatrix} v_j^n \\ v_j^m \end{pmatrix} = \begin{pmatrix} D^T v_j^m \\ Dv_j^n \end{pmatrix} \tag{18}$$

In Python I implemented this process in one line:

```
numpy.concatenate((numpy.dot(D.T, v[-m:]), np.dot(D, v[0:n])))
```

Thus, I may perform the Lanczos algorithm with $\bar{D}$ without explicitly inputting it into the algorithm. With this simplification, applying the Lanczos algorithm on $\bar{D}$ avoids excess-memory usage. The output of k iterations of the Lanczos algorithm, with $\bar{D}$ as the inputted matrix, results in the matrices $T_k \in \mathbb{R}^{k\times k}$ and $V_k \in \mathbb{R}^{(m+n)\times k}$ as described in Section 3.1. These matrices will be used in the following steps of my algorithm in a manner similar to Sections 3.2 and 3.3. Below is the Python code which performs this entire step of the algorithm.

```
def lanczos(D, k):
    m, n = D.shape
    tot = m + n
    V = numpy.zeros((tot, k))
    alphas = numpy.zeros(k)
    betas = numpy.zeros(k)
    v = numpy.random.rand(tot)
    v = v / numpy.linalg.norm(v)
    b = 0
    v_previous = numpy.zeros(tot).T
    for i in range(k):
        V[:, i] = v
        w = numpy.concatenate((numpy.dot(D.T, v[-m:]), numpy.dot(D, v[0:n])))
        a = numpy.dot(v, w)
        alphas[i] = a
        w = w - b * v_previous - a * v
        # Re-orthogonalization
        w = reorthogonalization(V, w, i)
        b = numpy.linalg.norm(w)
        betas[i] = b
        if b < numpy.finfo(float).eps:
            break
        v_previous = v
        v = (1 / b) * w
    T = numpy.diag(alphas) + numpy.diag(betas[0:-1], k=1) + numpy.diag(betas[0:-1], k=-1)
    return T, V
```

Listing 1: Variated Lanczos Algorithm

```
def reorthogonalization(V, w, i):
    for t in range(i):
        adj = numpy.dot(V[:, t], w)
        if adj == 0.0:
            continue
        w -= adj * V[:, t]
    return w
```

Listing 2: Re-orthogonalization Process

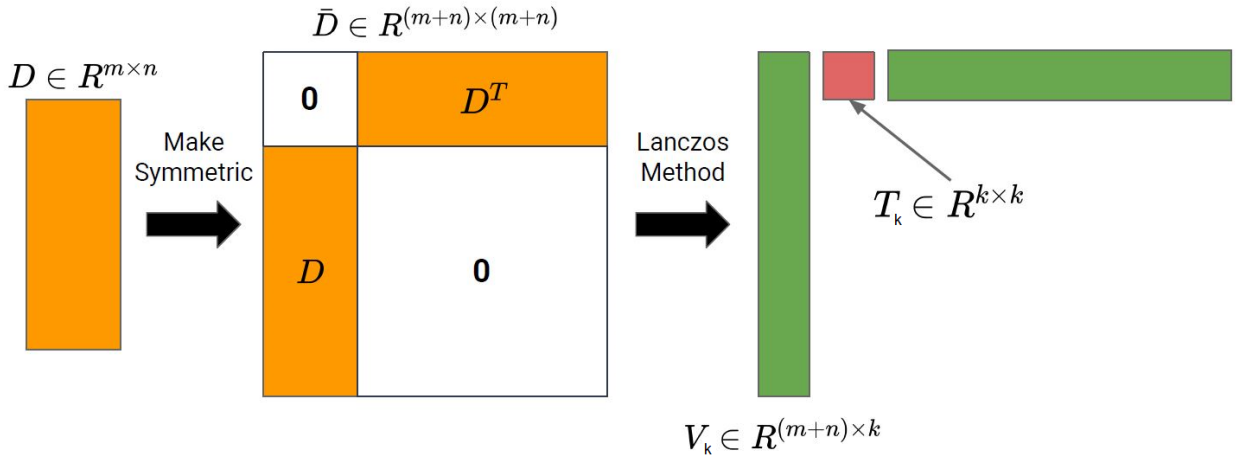

Figure 1: Preparing the Data Matrix and Performing Lanczos Algorithm

4.2 Approximating the Singular Value Decomposition

Once the tri-diagonal matrix T_k and the orthonormal matrix V_k have been computed through the steps detailed in Section 4.1, the SVD of D can be approximated. First, the eigenvalues and eigenvectors of T_k must be determined. As mentioned in Section 3.3, the positive eigenvalues of the prepared data matrix $\bar{D}$ are equivalent to the singular values of D. Therefore, computing the eigenvalues of the smaller tri-diagonal matrix T_k (which was constructed in Section 4.1 using $\bar{D}$) provides an approximation to the singular values of D. Likewise, the eigenvectors of T_k can be used to compute the singular vectors of D, as described in Section 3.3. Multiplying V_k with the eigenvectors of T_k provides an approximation to the singular vectors of D.

To compute the eigenvalues and eigenvectors of T_k, I utilized symmetric eigensolvers in the NumPy package within Python. The eigensolver "eigh" efficiently computes the eigenvalues and eigenvectors of symmetric matrices using LAPACK routines. This is implemented within Python in one line:

```
Eig_val, Eig_vec = numpy.linalg.eigh(T)
```

Once the eigenvalues and eigenvectors of T_k were found, I calculated the approximated singular vector of D: $Y = V_k Q_k$. As noted in Section 3.3, the first m rows of Y form the left singular vectors Y_l and the last n rows form the right singular vectors Y_r. This entire process is diagramed in Figure 2. My full computations of the singular values and singular vectors of D in Python are shown below.

```
def approx_svd(T, V, m, c):
    Sk, Qk = numpy.linalg.eigh(T)
    tempY = V @ Qk
    r = tempY.shape[0]
    Y_r = tempY[-m:,-c:] / numpy.linalg.norm(tempY[-m:,-c:], axis=0, keepdims=True)
    Y_l = tempY[0:(r-m),-c:] / numpy.linalg.norm(tempY[0:(r-m),-c:], axis=0, keepdims=True)
    return Y_l, Sk, Y_r
```

Listing 3: Approximating the Singular Value Decomposition

In my implementation of this process on my GitHub repository, the entire process is flipped due to a transpose within my inputted matrices. This did not affect the computations or results. I simply had to flip my resulting singular vectors for the reader to view them and the consequential results.

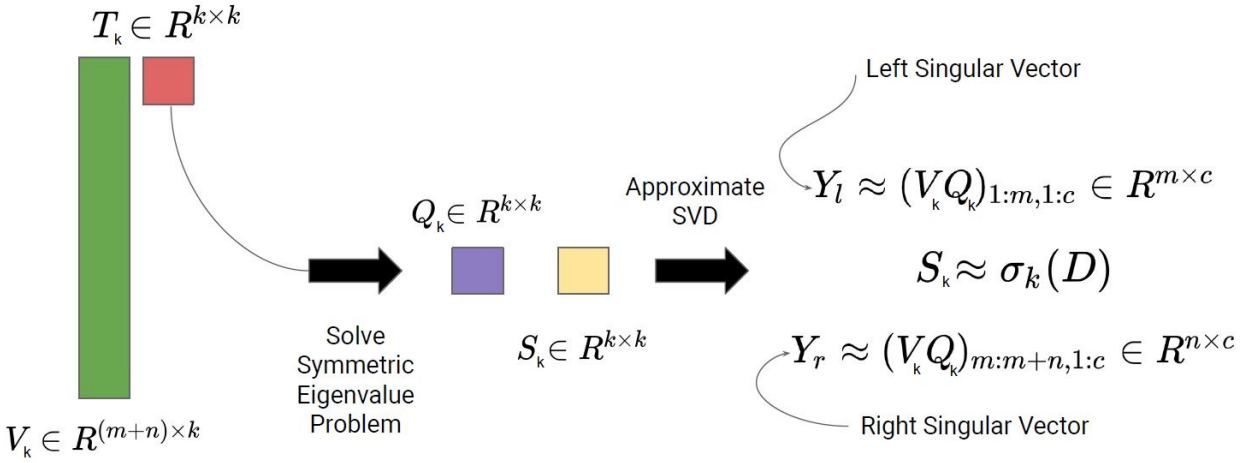

Figure 2: Approximating the Singular Value Decomposition of the Data Matrix

4.3 Computing the Principal Components Analysis

Finally, all that is left to do is to use the approximated SVD to compute the PCA. This process is documented in Sections 2.2 and 2.3. As shown in Equation (11), the first c columns of the left singular vector need to be multiplied by their corresponding c singular values in order to compute the first c principal component scores. This is equivalently a rank-c approximation of D. Therefore, all that has to be computed in Python (via element-wise multiplication) is the following single line:

```
projData = Y_l[:, 0:c] * Sk[0:c]
```

This process is done with $c = 2$ for 2D visualization of the data matrix D or with $c = 3$ for 3D visualization of the data matrix D.

4.4 Presenting the Full Algorithm

Below I provide pseudo-code of my algorithm, along with the Python code which connects all of the functions presented in the previous sub-sections.

Algorithm 2: Lanczos Fast Data Visualization (LFDV)

Input: Data matrix D, number of Lanczos iterations k, components c
$\bar{D} \leftarrow$ Prepare D (make symmetric)
$T_k, V_k \leftarrow \text{Lanczos}(\bar{D}, k)$
$Y_l, S_k, Y_r \leftarrow \text{ApproxSVD}(T_k, V_k, c)$
$D_c \leftarrow \text{PCA}(Y_l, S_k)$
Return: Dimension-reduced data matrix D_c for visualization

```
def lanczosSVD(D, k, c):
    m = D.shape[0]
    T, V = lanczos(D, k)
    U, Sk, Vt = approx_svd(T, V, m, c)
    projData = U[:, 0:c] * Sk[0:c]
    return projData, U, Sk, Vt
```

Listing 4: Computing the Principal Component Analysis

As mentioned previously, all code is found in my GitHub repository: `https://github.com/Marcob1996/Parallel-Lanczos-PCA`. The results of running this algorithm on a large data matrix is shown further on in the results section.

5 Parallelization

As part of this project, I parallelized the Lanczos Fast Data Visualization (LFDV) algorithm. To accomplish this, I coded the algorithm to be parallelized on CUDA. I utilized the CuPy package in Python to assist this process. The CuPy package performs all of the standard NumPy functions in parallel on a provided GPU. Therefore, to parallelize the code shown in Section 4, I first had to use CuPy commands to copy over all necessary matrices and vectors to the GPU. Once loaded, I began to use the power of CUDA to parallelize my code. These were the major changes to my serial code:

1. Parallelize the Lanczos algorithm
    - Replaced NumPy matrix-vector products with simple CuPy commands to parallelize matrix-vector products (necessary for Line 4 of Algorithm 1) on the GPU
    - Replaced NumPy dot products with CuPy dot products on the GPU (used for computing α_j and the re-orthogonalization process)
2. Parallelize the SVD computations
    - Performed the eigendecomposition of T_k on CUDA using the CuPy symmetric eigensolver (to compute the eigenvalues and eigenvectors of T_k)
3. Parallelize computation of the singular vectors
    - Performed parallel matrix multiplication on the GPU using CuPy to compute $Y = V_k Q_k$

The software necessary to perform this parallelization were CUDA 11.2.2, CuPy 10.0.0, and TensorFlow 2.7.0. I ran my experiements on the Center for Machine Learning (CML) cluster of computational resources (both CPUs and GPUs). A full overview of the CML cluster is found here: `https://wiki.umiacs.umd.edu/umiacs/index.php/CML`.

The CML cluster is home to 10 compute nodes. The nodes are equipped with 32 CPUs and 8 GPUs. The specific GPUs are NVIDIA GeForce RTX 2080 Ti. When running my parallel code, I only needed one GPU to hold all the memory needed for my benchmark problem. I also only requested one CPU during the course of running experiments. As described in my results section, I was able to see a substantial reduction in run-time when running my parallel code in comparison to the serial code. This confirmed that my parallel implementation in CUDA was working, and providing the parallelism I was seeking to speed-up the run-time. I will now present the results for both the serial and parallel algorithms.

6 Results

After coding both the serial and parallel versions of Lanczos Fast Data Visualization (LFDV), I set out to test its accuracy and efficiency. First, I needed a benchmark problem to test the implementations on. The problem I set out to solve was visualizing an extremely large dataset. The dataset I chose is known as the Modified National Institute of Standards and Technology (MNIST) dataset. Within the dataset, there are 70,000 28x28 pixel images of handwritten numbers (0 through 9). MNIST is commonly used for training and testing algorithms within machine learning. These handwritten images contain 28x28 = 784 pixels (a much greater resolution). Thus, the data matrix I inputted into my algorithm had dimension 70,000 by 784. In some instances, I took smaller subsets of the 70,000 images, in order to better visualize the data (since 70,000 samples is too dense and makes the visualizations hard to decipher).

6.1 Data Visualization Results

After inputting the MNIST dataset (with only 20,000 samples to keep the figures clean) into Lanczos Fast Data Visualization (LFDV) as described in Section 4, a reduced dimension data matrix is outputted. Therefore, each sample is reduced from 784 dimensions to just 2 or 3. While computing the dimensionality reduction, I keep track of the label of the digit for each sample. The label corresponds to which number the hand-written digit corresponds to. After reducing the dimensionality of the data, I plot each sample in 2 or 3 dimensions with a specific color corresponding to the label of that sample. For example, in Figure 3 all of the "1" hand-written digits were plotted with the color dark red. The reason for doing this is to hopefully view clusters of digits together. If this clustering occurs, then the data dimensionality has been effective at retaining the most important features of the data. Namely, in this case, if the dimensionaliy reduction from 784 to 2 or 3 is effective, then one can still tell apart a handwritten 1 from a 9. The results of the dimensionality reduction (data visualization) on MNIST are presented below in 2D and 3D.

Figure 3: Two-Dimensional Visualization of MNIST Samples

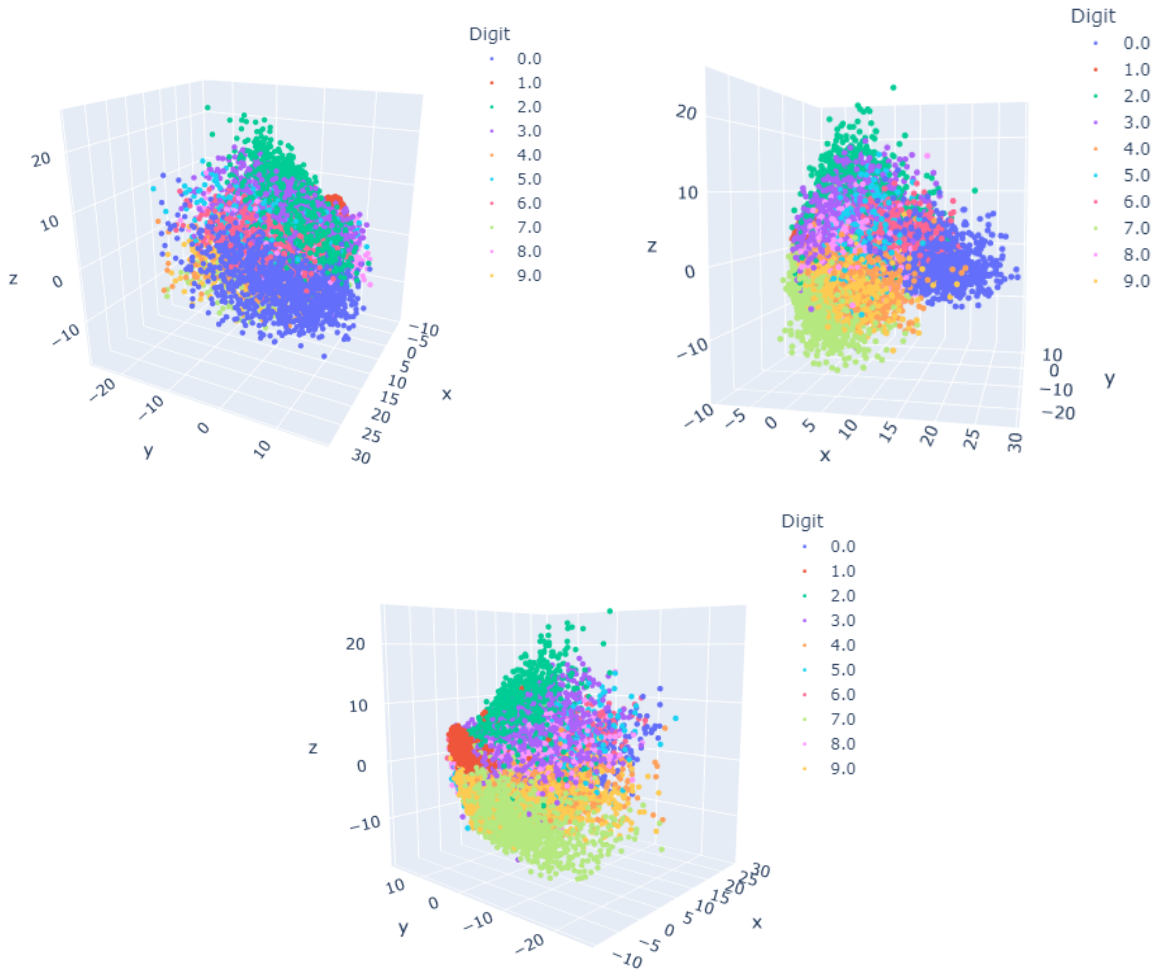

Figure 4: Three-Dimensional Visualization of MNIST Samples

As one can see in the figures above, clustering does occur. The Lanczos Fast Data Visualization algorithm succeeded in maintaining the structure of high-dimensional data in a lower dimension. It is revealing to see high-dimensional data, such as images of hand-written numbers, reduced to two and three dimensions yet still maintain enough information to differentiate between numbers successfully. When expanding from two to three dimensions, the clustering becomes more clear as less digits overlap with one another. Within my GitHub repository, one can download my Jupyter Notebook and play around with the 3D images as shown above in Figure 4 by rotating and panning through the visualization.

6.2 Data-Visualization Error

The accuracy, or error, of LFDV is determined by the norm of the difference (disregarding signs) between the reduced data outputted by my algorithm and the reduced data outputted by using the true SVD (calculated using the "svd" command in Python). It is important to note that the accuracy of LFDV is directly affected by the number of Lanczos iterations k. Therefore, I computed the accuracy of both serial and parallel LFDV for reducing the entire MNIST dataset (70,000 samples) to three dimensions over a range of values of k.

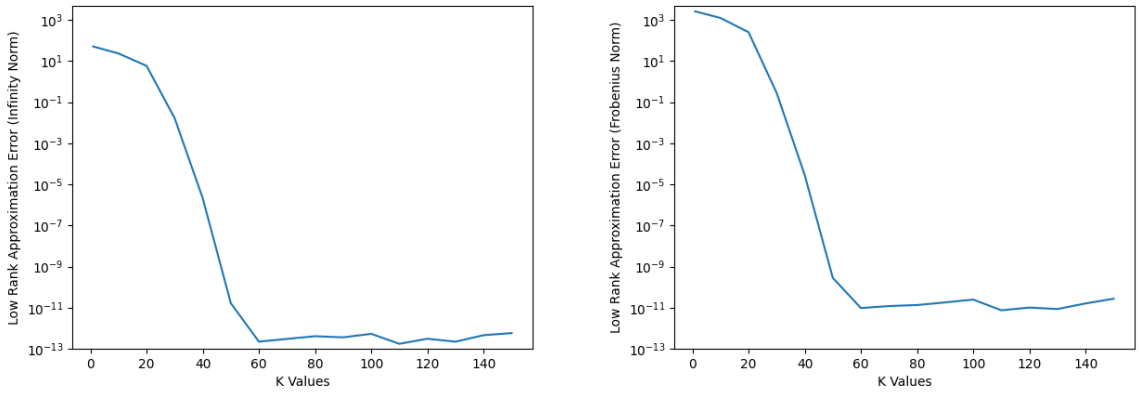

Figure 5: Error of Serial LFDV on MNIST Dataset for Varying k

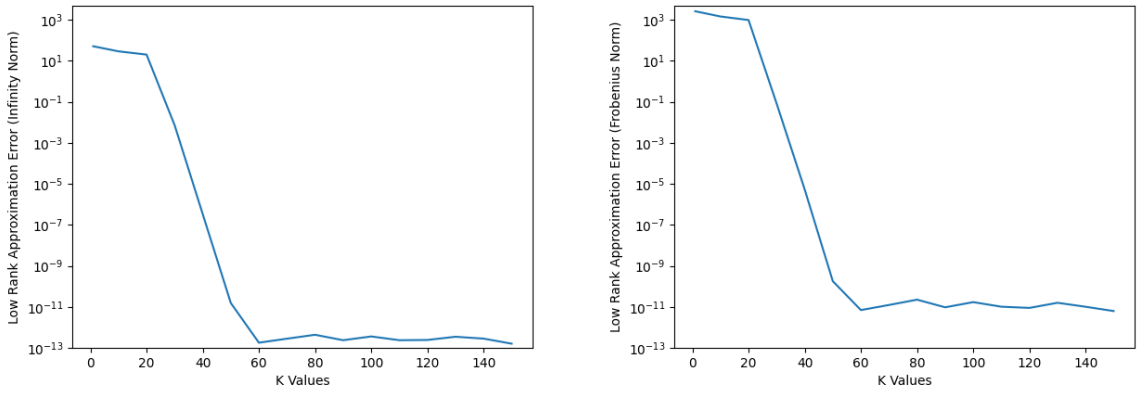

Figure 6: Error of Parallel LFDV on MNIST Dataset for Varying k

As anticipated, Figures 5 and 6 mirror each other. This shows that both the serial and parallel implementations correctly reduce the dimensionality of the MNIST dataset. The only slight variation arises from the randomness of the initial Lanczos vector (which occurs run-to-run). One can also see that both the serial and parallel implementations of LFDV both achieve a high degree of accuracy. As k grows to become large, the maximum of the absolute row sums (matrix infinity norm error) of the difference between the output of LFDV and the rank approximation using the true SVD converged close to machine precision $\sim 10^{-13}$. The Frobenius norm of this difference converged to $\sim 10^{-11}$ which is still extremely accurate.

It is interesting to note how the error behaves with respect to k. As one can see in Figures 5 and 6, the norm of the error is extremely high for low values of k but rapidly decreases between $k = 20$ and $k = 40$. Once k reaches 60 iterations, the norm of the error is extremely low and the error plateaus. This shows that the LFDV converged to its best approximation at $k = 60$ and remained there even after further iterations of the algorithm. Thus, in only 60 iterations, LFDV found an extremely accurate low-dimensional representation of the MNIST dataset. Due to this low number of iterations relative to the original dimension of the MNIST dataset (784), LFDV is able to find low-dimensional representations of high-dimensional data efficiently. This efficiency is demonstrated in the next section.

6.3 Data-Visualization Performance

Similar to tracking the accuracy of LFDV, I also tracked the run-time (in seconds) of LFDV for varying values of k. For each value of k, both serial and parallel LFDV reduced the entire MNIST dataset (70,000 samples) to three dimensions. I stored the run-time of both serial and parallel LFDV for $k = 1$ to $k = 150$. Figure 7 showcases the run-times for serial LFDV and Figure 8 showcases the run-times for parallel LFDV.

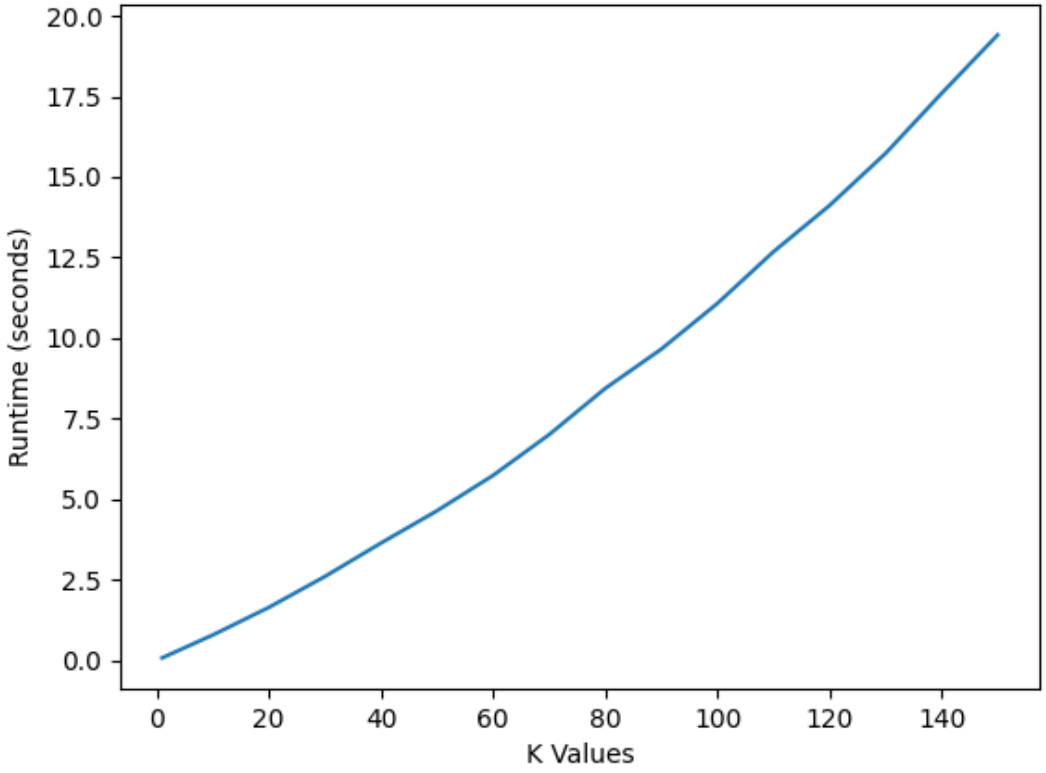

Figure 7: Run-time of Serial LFDV on MNIST Dataset for Varying k

As expected, more iterations of Lanczos (larger k) will increase the run-time of LFDV. Interestingly, the run-time starts to increase in a parabola-shaped manner. This reflects the cost of re-orthogonalization (k^2) as k grows to be larger. The larger the iteration count, the more vectors that need to be re-orthogonalized, and this process grows rapidly. This process has a major impact on the run-times of serial and parallel LFDV. In fact, a future improvement of LFDV would be to implement partial re-orthogonalization in order to spend less time orthogonalizing while still maintaining high levels of accuracy. This would boost the computational efficiency of LFDV. Below I plot the run-time of parallel LFDV as well as provide a tabular representation of the run-times comparing serial to parallel LFDV.

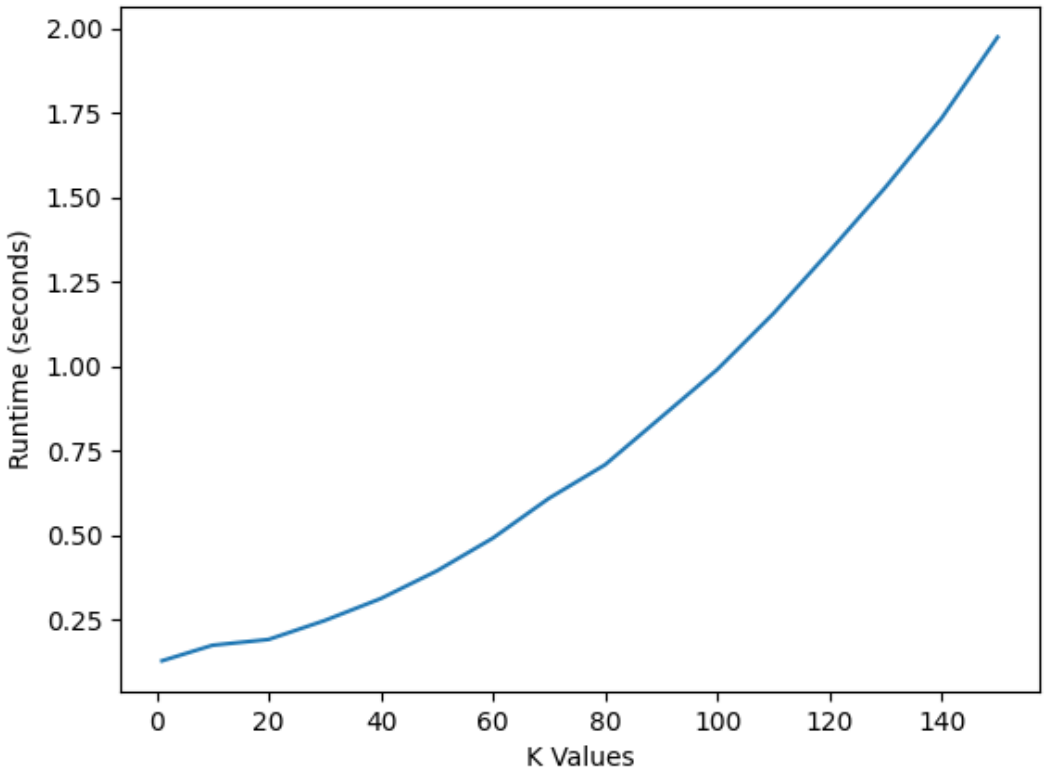

Figure 8: Run-time of Parallel LFDV on MNIST Dataset for Varying k

Table 1: Run-time (s) of Serial and Parallel LFDV for 70,000 MNIST Images

Value of k	Serial LFDV	Parallel LFDV
1	0.07850	0.12926
10	0.78621	0.17524
20	1.64144	0.19239
30	2.60018	0.24838
40	3.63792	0.31336
50	4.63963	0.39528
60	5.73712	0.49240
70	7.00186	0.61017
80	8.43485	0.70947
90	9.65428	0.85005
100	11.07082	0.99118
110	12.67410	1.15661
120	14.10928	1.34078
130	15.72588	1.53056
140	17.58909	1.73490
150	19.40405	1.97504

As depicted in Figure 8 and Table 1, parallel LFDV outperformed serial LFDV in terms of run-time. This was expected, as utilizing the power of a GPU should increase the computational power and greatly decrease the run-time. The run-times for parallel LFDV were generally 10x faster than those for serial LFDV (on Figure 8, the y-axis is 10x smaller than Figure 7). Parallel LFDV took only 0.492 seconds to accurately reduce 70,000 MNIST images to three dimensions ($k = 60$). I selected this time, as Section 6.2 showcased that only 60 iterations are needed to reduce the cumulative error to $\sim 10^{-11}$. In comparison, serial LFDV took approximately 5.737 seconds to do the same, which is 11.5x greater in time.

While running this experiment, I also timed how long it would take to compute the full SVD of only 30,000 MNIST images serially (using more images led to a memory issue and an extremely long waiting time). Likewise, I used the state-of-the-art "sklearn" PCA function to reduce the dimensionality of MNIST for 70,000 images. The "sklearn" implementation uses the ARPACK implementation of the truncated SVD for the most accurate results. By default, the PCA function utilizes randomized truncated SVD [3] as mentioned earlier in this paper. This method is only slightly faster than ARPACK, yet yields errors that are magnitudes larger (around 10^{-1} for 70,000 MNIST images). The "sklearn" PCA function utilizing ARPACK is just as fast yet much more accurate for this test problem, and is the fastest PCA algorithm in an open-source package. Here are the run-time results for the full serial SVD on 30,000 images and "sklearn" on 70,000 images:

MNIST Images	Full SVD Run-time (s)	"Sklearn" PCA Run-time (s)
30,000	241.302	—
70,000	—	3.784

Table 2: Run-times for MNIST Image Reduction to Three Dimensions Using Common Algorithms

From Table 2, calculating the full SVD for a low-rank approximation is extremely expensive in time and computational resources. For only 30,000 MNIST images, the full SVD took over 4 minutes to compute. That is much slower than LFDV and other truncated SVD methods. On the flip side, the "sklearn" function for PCA runs extremely quick and efficiently, taking only 3.784 seconds. This is faster than serial LFDV for values of k larger than 40.

Thus, for this problem, the "sklearn" function achieves a faster time than serial LFDV when attaining extremely high accuracy. However, for a dataset that has dimensions 70,000 x 784, having an error of 10^{-2} may be sufficient. In this case, serial LFDV takes only approximately 2.60 seconds to find a good low-rank approximation. That is about 46% faster than the "sklearn" PCA method. Since these functions all run serially, I don't believe "sklearn" utilizes GPUs by default when available, it is unfair to compare against parallel LFDV. Nevertheless, parallel LFDV blows away "sklearn" and the full SVD methods.

7 Conclusion

The Lanczos Fast Data Visualization (LFDV) algorithm is an efficient and accurate method to visualize high-dimensional data. The algorithm is also highly parallelizable, with parallel results showcasing that massive data sets can be visualized in three-dimensions in under one second. Leveraging the power of the standard Lanczos algorithm, LFDV is able to compete with the state-of-the-art data visualization algorithms. While results are already promising, greater exploitation of the algorithm remains. Utilizing partial re-orthogonalization and further optimizing the existing code would boost the efficiency of LFDV.

I wish to thank Dr. Elman for his instructive teaching of Iterative Methods and Computational Methods for Eigenvalue Problems. This course has sparked an interest in the power of Numerical Linear Algebra for me. Finally, all of my code can be found publicly on my Github repository: `https://github.com/Marcob1996/Parallel-Lanczos-PCA`.

References

[1] Constantine Caramanis and Sujay Sanghavi. Lectures on large scale learning, 2013.

[2] Dianne Cook, Doina Caragea, and Vasant Honavar. Visualization for classification problems, with examples using support vector machines. In *Proceedings of the COMPSTAT*. Citeseer, 2004.

[3] Nathan Halko, Per-Gunnar Martinsson, and Joel Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2010.

[4] Ian Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[5] Minca Mramor, Gregor Leban, Janez Demšar, and Blaž Zupan. Visualization-based cancer microarray data classification analysis. *Bioinformatics*, 23(16):2147–2154, 2007.

[6] Fábio Henrique Oliveira, Alessandro Machado, and Adriano Andrade. On the use of t-distributed stochastic neighbor embedding for data visualization and classification of individuals with parkinson's disease. *Computational and mathematical methods in medicine*, 2018, 2018.

[7] Philip Powell. Calculating determinants of block matrices. *arXiv preprint arXiv:1112.4379*, 2011.

[8] David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, page 16, 2018.

[9] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[10] Antony Unwin. Why is data visualization important? what is important in data visualization? *Harvard Data Science Review*, 2(1), 1 2020. https://hdsr.mitpress.mit.edu/pub/zok97i7p.