POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Randomized Linear Algebra and its Applications

High Performance Computing Engineering - Advanced Methods for Scientific Computing

Jasmin Spinetto, Peng Rao, Anna Paola Izzo, Cao Wu, Jiali Claudo Haung,
10734813, 11022931, 10680171, 11036000, 11032111

Advisor:
Prof. Luca Formaggia

Academic year:
2024-2025

Abstract: In this work, we present an efficient implementation of the randomized Singular Value Decomposition (rSVD) algorithm, along with its parallelized versions and a benchmarking analysis. Our study explores the performance and scalability of these implementations, demonstrating their advantages in handling large-scale datasets. To show the practical applications of rSVD, we apply it to three key concrete problems: image compression, handwritten number recognition, and Principal Component Analysis (PCA), providing quantitative results that highlight their effectiveness. Additionally, we implement the CUR decomposition algorithm, both in its deterministic and randomized versions, as an alternative low-rank approximation method. We also analyzed its performance with respect to SVD. Our findings underscore the computational benefits of randomized techniques and their potential for real-world applications.

Key-words: Randomized Linear Algebra, RandomizedSVD, CUR Decomposition, Applications

1. Introduction

In this project we developed a custom implementation of the **Randomized Singular Value Decomposition (rSVD)** algorithm in C++ and conducted a benchmarking to compare its performance with the `BDCSVD` implementation provided by the `Eigen` library. Furthemore, we explored some **concrete application cases** of the RandomizedSVD decomposition method as:

- Image Compression
- Handwritten Numbers Recognition
- Principal Component Analysis

In addition to RSVD, we implemented and tested also the **CUR decomposition algorithm**, which is another randomized technique for approximating matrices, exploring both it's deterministic and randomized implementations.
This report outlines the theoretical background of RSVD and CUR decomposition, describes the implementation details of our algorithms and presents the results of our benchmarks and application experiments. Through this work, we aim to demonstrate the practical utility of randomized linear algebra techniques in solving computationally demanding problems while ensuring efficiency and accuracy.
The only pre-existing code utilized in our rSVD implementation was the matrix operations provided by the **Eigen library**, which facilitated efficient handling of matrix manipulations.

2. Randomized Linear Algebra

Random Linear Algebra is a branch of mathematics that combines principles from linear algebra with techniques from probability and statistics to analyze and solve problems involving large-scale or high-dimensional data. The base idea is to introduce **randomness** into computations to make algorithms faster, simpler, more efficient, more scalable, or more robust while maintaining accurate approximations of the desired results. This is needed because traditional linear algebra techniques can become computationally expensive for large datasets, such as those encountered in machine learning, data science, and numerical simulations. Randomized algorithms help mitigate these challenges by:

- Reducing computation complexity
- Lowering memory requirements
- Allowing for parallelization and distributed computing

The main techniques used are:

- **Randomized Sampling**: instead of processing an entire matrix, Randomized Linear Algebra uses random sampling to approximate matrix operations. For example by:
  - *Row/column sampling*: selecting a subset of rows or columns from a matrix based on a probability distribution to approximate its structure.
  - *Sketching*: compressing a large matrix into a smaller one ("sketch matrix") while preserving key properties, such as norms or singular values.
- **Low-Rank Approximation**: many large matrices encountered in practice are approximately low-rank, meaning their significant information can be captured by a smaller number of dimensions. Randomized techniques are used to compute these approximations efficiently:
  - *Randomized SVD (Singular Value Decomposition)*: Approximating the singular values and vectors using random projections.
  - *CUR Decomposition*: representing a matrix using a subset of its actual rows (C) and columns (R), and a smaller core matrix (U).
- **Random Projections**: high-dimensional data can be projected into a lower-dimensional space using random matrices, such as Gaussian random matrices and Sparse random matrices.
- **Montecarlo methods**: randomized algorithms often rely on Monte Carlo methods to provide probabilistic guarantees about the accuracy of approximations. For example, the probability of achieving a given approximation error is often a parameter of the algorithm.
- **Stochastic Iterative Methods**: iterative solvers for linear systems, such as gradient descent, can be randomized by incorporating stochastic components, which often improves convergence rates or reduces computational cost in large-scale problems.

2.1. Challenges

The main challenges of using Randomized Linear Algebra lie in **balancing efficiency and accuracy**. Since RLA methods rely on approximations, they introduce a trade-off between computational speed and the precision of results, which may not be acceptable in highly sensitive applications. Additionally, the performance of these methods depends on how the parameters are tuned, such as sampling size and pattern or random projections choice. For example, in a randomized SVD decomposition, using less random samples (meaning a lower rank) reduced computation time but can cause missing important features of the matrix, leading to an inaccurate low-rank approximation.

Randomized Linear Algebra can become ineffective when it's applied to highly ill-conditioned or extremely sparse matrices, to data with outliers or heavy noise, or when it is used for highly sensitive applications or small-scale problems.

3. Randomized Singular Value Decomposition

We have already implemented the following algorithms:

- QR factorization using Given's rotation: `GivensRotation`
- Basic SVD using Power Method: `PowerMethod`
- Randomized singular value decomposition: `RandomizedSVD`

3.1. QR Factorization

The Givens Rotation Q R decomposition is a method for decomposing a matrix A into an orthogonal matrix Q and an upper triangular matrix R, such that:

$$A = QR$$

A Givens rotation matrix is used to zero out specific elements of a matrix. For two elements a and b, the Givens rotation coefficients c and s are calculated such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $r = \sqrt{a^2 + b^2}$. The Givens rotation matrix is then used to zero out the element b by multiplying the matrix from the left:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} r & 0 \\ 0 & d' \end{bmatrix}$$

where $d' = cd - sb$. The Givens rotation matrix is a efficient way to perform QR decomposition, especially for sparse matrices. The pseudocode for the Givens rotation QR decomposition is shown in Algorithm 1.

Algorithm 1 Givens Rotation Algorithm

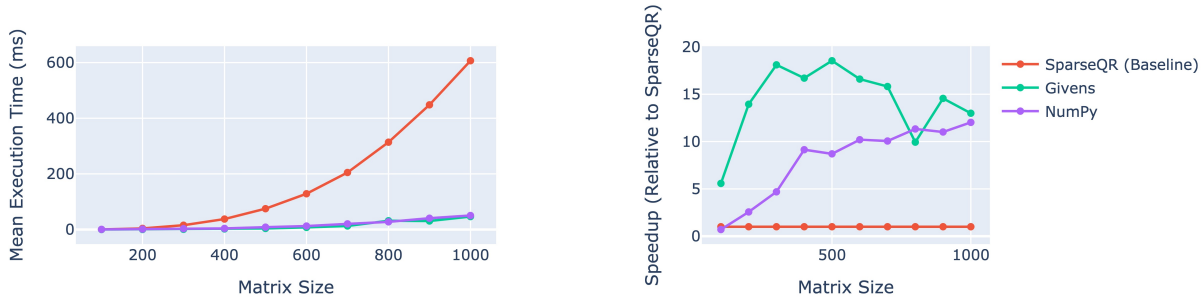
Require: Matrix $A \in \mathbb{R}^{m \times n}$, indices i, j, and column k $(1 \leq i < j \leq m)$
Ensure: Matrix A with the element $A[j, k]$ eliminated

1: Compute $r \leftarrow \sqrt{A[i, k]^2 + A[j, k]^2}$
2: Compute $c \leftarrow \frac{A[i,k]}{r}$ and $s \leftarrow \frac{A[j,k]}{r}$
3: **for** $l = k$ to n **do**
4: $\quad t \leftarrow A[i, l]$ {Temporary storage for row i}
5: $\quad A[i, l] \leftarrow c \cdot A[i, l] + s \cdot A[j, l]$
6: $\quad A[j, l] \leftarrow -s \cdot t + c \cdot A[j, l]$
7: **end for**

We implemented the Givens Rotation algorithm and utilized template metaprogramming techniques to achieve optimizations for both dense and sparse matrices. To construct this QR factorization solver, we can use both the following initializes:

```
Eigen::GivensRotationQR<Eigen::MatrixXd> givens_rotation_qr; // for dense
Eigen::GivensRotationQR<Eigen::SparseMatrix<double>> givens_rotation_qr; // for sparse
```

We conducted a benchmark of Algorithm 1, testing matrix sizes ranging from 100 × 100 to 1000 × 1000, and compared it with the `SparseQR` algorithm implemented in `Eigen`. And we also tested the computation results using `NumPy` with the same sparse matrices.. The results are presented in the following figure 1.
Our implementation outperformed `SparseQR`, achieved results comparable to `NumPy`, and even slightly surpassed `NumPy` by approximately 5 to 20 times. Since `NumPy` is built on `OpenBLAS` at its core but is slower due to the Python language compared to C++, this result aligns with expectations. The results show that Givens rotation implementation is a competitive and efficient approach compared to the baseline.



(a) Runtime. (b) Speedups.

Figure 1: Computational performance for sparse matrix

3.2. Singular Value Decomposition using Power Method

The singular value decomposition (SVD) is a fundamental matrix decomposition method that decomposes a matrix A into three matrices U, Σ, and V such that:

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

where U and V are orthogonal matrices and Σ is a diagonal matrix with the singular values of A. The SVD is widely used in various applications, including dimensionality reduction, data compression, and machine learning. The power method is an iterative algorithm that can be used to compute the singular values and vectors of a matrix. The power method works by repeatedly multiplying the matrix by a vector and normalizing the result. The pseudocode for the power method SVD is shown in Algorithm 2.

Algorithm 2 Power Method SVD (Rank-k Approximation)

Require: Matrix $A \in \mathbb{R}^{m\times n}$, target rank k, tolerance ϵ_δ, factor δ, and parameter λ
Ensure: Approximate singular values σ_i and singular vectors u_i, v_i for $i = 1 \dots k$

for $i = 1$ **to** k **do**
  $s \leftarrow log(4log(2n/\sigma/\epsilon\sigma)/2\lambda$
  $v \leftarrow \text{RandomNormal}(n)$
  for $\ell = 1$ **to** s **do**
    $v \leftarrow A^\top(Av)$
    $v \leftarrow v/\|v\|$
  end for
  $\sigma \leftarrow \|A\,v\|$
  $u \leftarrow (A\,v)/\sigma$
  if $\sigma < 10^{-14}$ **then**
    break
  end if
  $\sigma_i \leftarrow \sigma,\ u_i \leftarrow u,\ v_i \leftarrow v$
  $A \leftarrow A - \sigma\,u\,v^\top$
end for

3.3. RSVD Implementation

Let $A \in \mathbb{R}^{m\times n}$ be a matrix of low rank, and $m \geq n$. In the following, we seek the near-optimal low-rank approximation of the form

$$A \approx U_k\Sigma_k V_k^T$$

where k denotes the target rank. Instead of computing the singular value decomposition directly, we embed the SVD into the probabilistic framework. The principal concept is sketched in Figure 2.
We implemented the RSVD algorithm using `Eigen` library, the QR decomposition, and svd algorithm mentioned above. We defined the following explicit template instantiation and specialization:

```cpp
// Dense matrix
Eigen::RandomizedSVD<double, Eigen::Dynamic, Eigen::Dynamic> rsvd;
// Sparse matrix
Eigen::RandomizedSVD<Eigen::SparseMatrix<double>>
// Row-major matrix
Eigen::RandomizedSVD<Eigen::SparseMatrix<double, Eigen::RowMajor>>
```

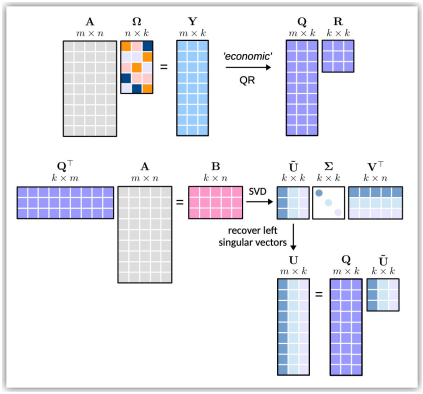
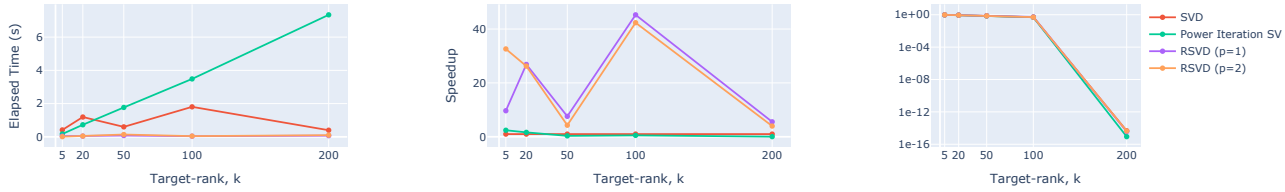


Figure 2: Randomized SVD [**?**]

We now investigate the performance of the implementation in more detail. Figures 3 shows the performance of varying target ranks. The speedups show the relative performance compared to the base `BDCSVD` in `Eigen`. The relative reconstruction error is computed as

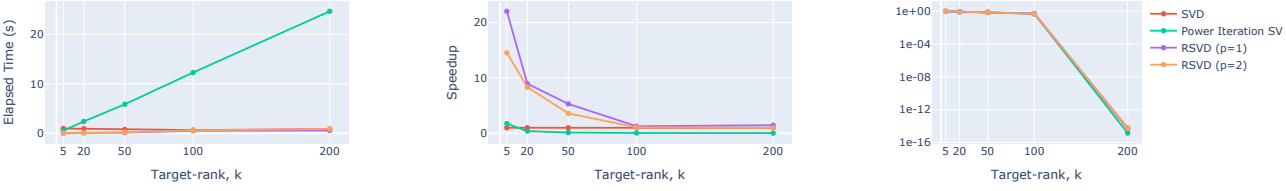
$$\frac{\|\mathbf{A} - \mathbf{A}_k\|_F}{\|\mathbf{A}\|_F}$$

The implementation of rsvd achieves substantial speedups over the other implementation of SVD. Obviously, increasing the number of power iteration can improve accuracy, we can control the trade-off between computational time and accuracy, depending on the application. Regarding errors, all methods can converge to **1e-15**, with `PowerSVD` being more accurate.



(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 3: Computational performance for a dense 1000×1000 matrix



(a) Runtime. (b) Speedups. (c) Relative errors.

Figure 4: Computational performance for a sparse 1000×1000 matrix

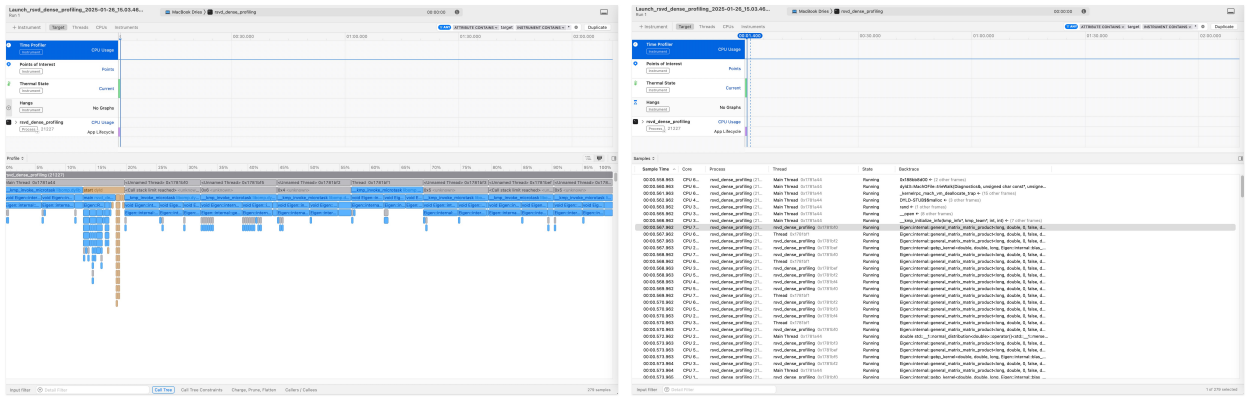
3.4. Optimization technique

Profiling

A profiler is a program that allows to examine the performance of an executable and find possible bottleneck. Since I am using the macOS operating system, Valgrind is not available for the ARM64 architecture. Therefore, I chose to use Instruments for profiling, utilizing the following command-line tool for profiling:

```
xcrun xctrace record --template 'Time Profiler' --launch ./rsvd_dense_profiling
```

Fig 5 shows the result. For my 8-core CPU, the image shows that `OpenMP` successfully started 8 threads, with the main thread out, each thread sharing about 10%-12% of the workload. Matrix multiplication is also shown to be the bottleneck of this implementation.



(a) Profile. (b) Samples.

Figure 5: Profile for dense rsvd

Parallelism

We use multiple threading for parallelization. For Eigen, the parallelization of matrix multiplication can be achieved by adding the following compilation directive, and we have enabled **level 03 optimization**, as well as using **-march=native** to instruct the compiler to generate optimized machine code based on the current hardware architecture.

```
add_executable(dense_svd dense_svd.cpp)
target_link_libraries(dense_svd PRIVATE Eigen3::Eigen)
target_link_libraries(dense_svd PRIVATE OpenMP::OpenMP_CXX)
target_compile_options(dense_svd PRIVATE -O3 -march=native)
```

Sorting

The key optimization is sorting the nonzero row indices in descending order before applying the rotations. Eliminates larger-indexed elements first, reducing interference in subsequent steps. Reduce redundant computations by systematically zeroing out elements from bottom to top.

```
// Sort in descending order
std::sort(rowList.begin(), rowList.end(), std::greater<Index>());
```

OpenBLAS

`OpenBLAS` is an open-source implementation of the `BLAS` and `LAPACK` libraries. `Eigen` implements `OpenBLAS` acceleration by enabling the `#define EIGEN_USE_BLAS` macro definition. On my computer (MacOS M1), the improvement of `OpenBLAS` for matrix operations is not significant, possibly due to the reasons of the chip architecture.

Memory access

Memory access is also one of the bottlenecks. We found that Eigen does not access matrix elements as fast as expected. Therefore, in the Givens Rotation algorithm, since this algorithm needs to frequently access matrix elements, for the access that requires eliminating elements, we directly use pointers for access, which reduces the computation time by about 5 times.

```
Scalar* data_ptr = matrix.data();
if constexpr (StorageOrder == RowMajor) {
    // RowMajor: (row, col) => row*cols + col
    for (Index row = 0; row < rows; ++row) {
        const Index offset_i = row * cols + i;
        const Index offset_k = row * cols + k;
        Scalar temp_i = data_ptr[offset_i];
        Scalar temp_k = data_ptr[offset_k];
        data_ptr[offset_i] = c * temp_i - s * temp_k;
        data_ptr[offset_k] = s * temp_i + c * temp_k;
    }
} else {
    // ColMajor: (row, col) => col*rows + row
    ...
    }
}
```

4. Applications

In this section, we present some useful concrete applications of the RandomizedSVD algorithm.

4.1. Image Compression

Image compression is the process of **reducing the storage size of an image while preserving its visual quality as much as possible**. This can be achieved both with *lossless* or *lossy* algorithms. In the specific compression done with the Singular Value Decomposition, the algorithm used is lossy, in particular, it removes redundant or less significant information inside the image.
In mathematical terms, an image can be represented as a matrix of the same size as the image, where each entry corresponds to the intensity of a pixel (in grayscale) or a combination of 3 color channels (in RGB images). Using Singular Value Decomposition (SVD), the matrix can be approximated as a **product of three smaller matrices**:

$$A \approx U_k \Sigma_k (V_k)^T$$

Where k is the rank of the approximation. By retaining only the largest k singular values and their corresponding vectors, we can achieve a low-rank approximation of the image that captures its most essential features while discarding finer details. Instead of storing the whole image, we store only the compressed representation (meaning the three smaller matrices). When needed, the image can be reconstructed from these components, with a slight loss of detail proportional to the rank k. Randomized SVD accelerates this process by efficiently estimating the top k singular values and vectors using random sampling and projections, rather than computing the full decomposition. This makes RSVD particularly well-suited for compressing large images.

4.1.1 Code Explanation

Brief explanation of the code:

- **Image Loading**: the program uses *stb_image* (which is a public domain image loader found at `http://nothings.org/stbto`) to load the input image, extracting its width, height and the number of channels. The pixel values are normalized to the range [0, 1] for numerical stability.
- **Grayscale Detection and Conversion**: if the image has three channels but is visually grayscale (meaning that all channels have equal intensity), it is reduced to a single grayscale channel for more efficient processing, eliminating redundant computations. This instance could happen due to compatibility reasons, or some applications' constraints, or for visual editing reasons. This specific instance is detected

by a customized *isGrayscaleRGB* function. Once the compression is done, the image will be brought back to its original number of channels.

- **Matrix Representation**: Each channel of the image is converted into an *Eigen MatrixXd*, where rows and columns correspond to the pixel grid.
- **Compression via RSVD**: For each channel, the code computes a low-rank approximation of the image matrix using the Randomized SVD algorithm with the rank given as input by the user:

```
rSVD.compute(channel_matrices[c], rank);
compressed_channels[c] = rSVD.matrixU() * rSVD.singularValues().asDiagonal()
                         * rSVD.matrixV().transpose();
```

Here, the rank is the most important parameter, specifying the number of singular values to retain and thus, controlling the level of compression.

- **Image Reconstruction**: The compressed matrices are scaled back to the [0, 255] range and converted into an *std::vector<unsigned char>* for saving.
- **Saving the Compressed Image**: use *stbi_ write_ png* (which is again a public domain image saver found at `http://nothings.org/stb` that writes out PNG/BMP/TGA/JPEG/HDR images to C stdio) to save the compressed image data to an output PNG file.
- **Relative Error Computation**: the program calculates the relative Frobenius norm error between the original and compressed image matrices for each channel:

```
double original_norm = std::sqrt(channel_matrices[c].squaredNorm());
double difference_norm = std::sqrt((channel_matrices[c]
                         - compressed_channels[c]).squaredNorm());
double relative_error = difference_norm / original_norm;
total_relative_error += relative_error;
```

Then, it computes the average total relative error across channels, allowing error comparison between compressed images with a different number of channels. This final value quantifies the compression accuracy.

- **Memory Usage Computation**: at last, the program calculates the memory used in terms of KBytes by the raw input matrix and by the three compressed matrices. With this values, it computes the memory compression ratio.

4.1.2 Results Analysis

The results were evaluated both visually and analytically. In particular, we compared the different results using the **relative error** computed as:

$$\text{Relative Error} = \frac{||A - A_{\text{compressed}}||_F}{||A||_F}$$

Where the F-norm (*Frobenius norm*) is defined as:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

And the **memory compression ratio**, computed as:

$$\text{Compression Ratio (CR)} = \frac{\text{bytes of input image}}{\text{total bytes of } U_k, \Sigma_k \text{ and } (V_k)^T}$$

The formula used was chosen to be independent of the memory saving format of both input and output, it computes the raw matrices sizes to understand how rSVD influences them.
Figure 6 displays a gray scale image of size 4000 x 6000 with 3 channels along with its compressed versions using different ranks. **Table 1** shows how increasing the rank also reduces the relative error, this makes sense because we are keeping more information about the matrix. However, if with smaller ranks the CR is larger than 1, meaning the compression is effective memory-wise, performed tests have proved that, after a certain rank (around 10% of the image width), the CR goes under 1, meaning the compressed matrices actually occupy more memory than the input image. This happens because the memory used by the compressed matrices grows linearly for U_k and V_k transposed while it grows quadratically for Sigma_k. However, our evaluation doesn't take into account the compression power of image saving formats. For example, using data storage

(a) Original Image

(b) Rank=200

(c) Rank=100

(d) Rank=50

(e) Rank=10

Figure 6: Compression results with different ranks

formats after rSVD compression like CSR, CSC, HDF5 or MM that are specific for the sparse and low-rank matrices created by rSVD, would probably increase the threshold rank. The **limitations of rSVD**, and SVD, as cited in [7], come from the fact that it is **inherently redundant**, because U and V are *r-rank orthonormal*, inherently including constraints that prevent SVD from deeper compression and even making it frustrated as the data fidelity is strictly required. In this same paper, the solution found is to enhance the power of the SVD algorithm using **E-SVD**, which includes entropy coding like the JPEG compression format. Other papers like [2], find a better solution in **K-SVD**, which is an iterative algorithm that extends SVD by learning a sparse and adaptive dictionary for signal representation.

From a **visual point of view**, tests have proven that:

- Using a rank *below 5-10%* visually degrades the quality of the image and can introduce artifacts. Take into account that the ability to recognize compression mainly depends on how big the displayed image is (roughly speaking, if we zoom into the image, compression is much more obvious), so, depending on the use we want to make of the compressed image, we could be satisfied even with lower ranks.
- Using a rank *above 15%* outputs a good quality image, making compression hard to spot.

Rank	Relative Error	Memory Compression Ratio
10	0.160166	29.97
50	0.123699	5.97015
100	0.0741506	2.9703
200	0.050169	1.47059
300	0.030199	1.05408
400	0.024068	0.9739

Table 1: Table showing rank, relative error, and memory compression ratio of various test compressions performed with rSVD

4.1.3 Note

These results strongly depend on the specific structure of the image to compress, in particular on:

- **Redundancy**: how much pixels are correlated with each other. It can be *spatial* (smooth regions, large areas of similar colors, ...) or *spectral* (highly correlated channels). More redundant images are easier to compress because we can exploit correlation to make predictions.
- **Image Complexity**: it relates with how much the image structure is predictable, objects that have sharp edges, images with small details and fine textures or high color variability are harder to compress.
- **Noise**: strongly related to image complexity, if an image is noisy, it's less predictable because noise appears as random and with high-frequency.

What we can focus on, is the sufficient starting efficiency of the rSVD Image Compression implementation presented, giving a good starting point for further studies on better compression algorithms or versions of SVD and it's randomized counterpart.

4.2. Handwritten Numbers Recognition

We implemented a handwritten digit recognition system using the MNIST dataset and RandomizedSVD for dimensionality reduction. The MNIST dataset consists of 28x28 pixel grayscale images of handwritten digits (0-9), making each image a 784-dimensional vector. Our implementation combines dimensionality reduction through rSVD with nearest neighbor classification.

4.2.1 Implementation Details

The system consists of three main components:

- **Data Loading**: The `MNISTLoader` class handles loading and preprocessing of the MNIST dataset, converting the binary format into normalized Eigen matrices.
- **Dimensionality Reduction**: Using rSVD through our PCA implementation to reduce the 784-dimensional images to a lower-dimensional space while preserving essential features.
- **Classification**: A nearest neighbor classifier that works in the reduced dimensional space to predict digit labels.

The training process involves:

1. Computing the mean of the training data
2. Centering the data by subtracting the mean
3. Applying rSVD-based PCA to obtain the projection matrix
4. Projecting training data into the reduced space

For prediction, new images are:

1. Centered using the training mean
2. Projected into the reduced space
3. Classified using nearest neighbor comparison with the reduced training data

4.2.2 Code Explanation

The core implementation consists of training and prediction phases. During training, we first load and normalize the MNIST data, then use PCA for dimensionality reduction:

```
1 // Load and normalize MNIST data to [0,1]
2 auto [images, labels] = MNISTLoader::loadMNISTData(images_path, labels_path);
3
```

```
// Reduce dimensions using PCA
Eigen::PCA pca;
pca.computeByRSVD(training_data, num_components_, num_components_);
reduced_training_data_ = pca.reducedMatrix();
```

For prediction, we use nearest neighbor classification in the reduced dimensional space to identify digits.

4.2.3 Results Analysis

We conducted experiments using 10,000 training images and 1,000 test images, varying the number of components used in the dimensionality reduction. Table 2 shows the classification accuracy for different numbers of components:

Number of Components	Accuracy
2	37.9%
4	58.1%
8	85.1%
16	92.3%
32	93.7%
64	93.3%
128	93.1%
256	92.7%

Table 2: MNIST Classification Accuracy vs Number of Components

Key observations from the results:

- **Rapid Improvement**: Accuracy improves dramatically from 37.9% with 2 components to 85.1% with just 8 components, showing that even a small number of components can capture significant discriminative features.
- **Optimal Performance**: Peak accuracy of 93.7% is achieved with 32 components, suggesting this is the optimal balance between dimensionality reduction and information preservation.
- **Diminishing Returns**: Beyond 32 components, accuracy slightly decreases, possibly due to overfitting or the inclusion of noise components. This demonstrates that more components don't always lead to better performance.
- **Efficiency**: The system achieves over 92% accuracy while reducing dimensionality from 784 to just 16 dimensions, representing a 98% reduction in data dimensionality.

The implementation also includes a single image prediction test that successfully identifies individual digit images, demonstrating the practical applicability of the system for real-world handwritten digit recognition tasks.

These results demonstrate that rSVD can effectively reduce the dimensionality of image data while maintaining the discriminative features necessary for accurate classification. The high accuracy achieved with relatively few components highlights the efficiency of our implementation in capturing the essential characteristics of handwritten digits.

4.3. Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique widely used in data science, machine learning and statistics. **It transforms high-dimensional data into a lower-dimensional form while preserving as much of the original information as possible**.

Originally, PCA was achieved by obtaining a covariance matrix. Where covariance is defined as:

$$cov(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})\,(y_i - \bar{y})}{n-1} \tag{1}$$

and then we can get covariance matrix:

$$C = \begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(x,y) & cov(y,y) \end{bmatrix} \tag{2}$$

because $\bar{x}$ and $\bar{y}$ equal to 0 with centralization. we can transform equation to:

$$C = \begin{bmatrix} \frac{\sum_{i=1}^{n} x_i^2}{n-1} & \frac{\sum_{i=1}^{n} x_i y_i}{n-1} \\ \frac{\sum_{i=1}^{n} x_i y_i}{n-1} & \frac{\sum_{i=1}^{n} y_i^2}{n-1} \end{bmatrix} \tag{3}$$

convert it to the matrix multiplication form:

$$= \frac{1}{n-1} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \tag{4}$$

finally we can get:

$$C = \frac{1}{n-1} D^{\mathrm{T}} D \tag{5}$$

D is our data matrix. In order to preserve more information of our data matrix, when we want to get principal component of matrix, we need to get eigenvector of covariance matrix and sorting it with eigenvalue. In the meantime, we have this equation in SVD decomposition:

$$D^{\mathrm{T}} D = V L V^{\mathrm{T}} \tag{6}$$

it is equivalent to performing an SVD decomposition on D matrix. V matrix is consisted by covariance eigenvector. L matrix is a diagonal matrix and the value in the diagonal is the square of eigenvalue. In this way, we can perform PCA by SVD decomposition instead of computing covariance matrix which is more complexity. In this thesis, we are discoursing about RSVD, which can get better performance than SVD.

4.3.1 Code Explanation

Brief explanation of the code:

- **Data centralization**: In order to get best direction to project our data, we need to centralize our data.

```
Eigen::VectorXd mean = data.colwise().mean();  // get mean value of each col
Eigen::MatrixXd centered = data.rowwise() - mean.transpose();
```

- **Performing RSVD decomposition**: We need to perform RSVD decomposition on our centered matrix to get V matrix. where rank is preferably equal to final dimension and can't be smaller than dimension.

```
Eigen::RandomizedSVD<Eigen::MatrixXd> rSVD;
rSVD.compute(centered, rank);
```

- **Get principal component**: Get previous component of V matrix, the number is equal to our target dimension of data.

```
Eigen::MatrixXd principal_components = rSVD.matrixV().leftCols(dimension);
```

- **Data projection**: Finally we do matrix multiplication to project our data on target dimension.

```
this->reducedMat = centered * principal_components;
```

4.3.2 Results Analysis

We have performed variant implementation of PCA to compare the performance between them. As we can see, In Figure 7, the PCA implemented by RSVD is even worse than the common one. this depends on the dimensions of the data n (number of samples) and d (number of features). when the row is larger than the column, which means that the number of samples is more than the features, In this case, common PCA is better.

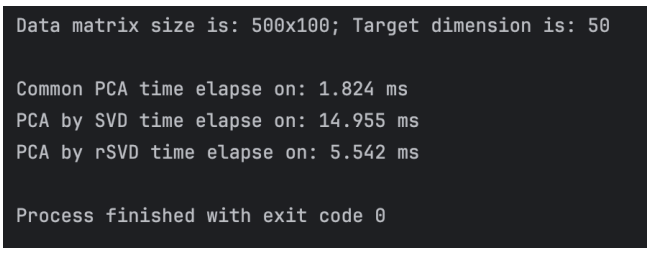


Figure 7: Test1

In Figure 8, when the number of column is larger than the row, the PCA implemented by RSVD is better than the common one.

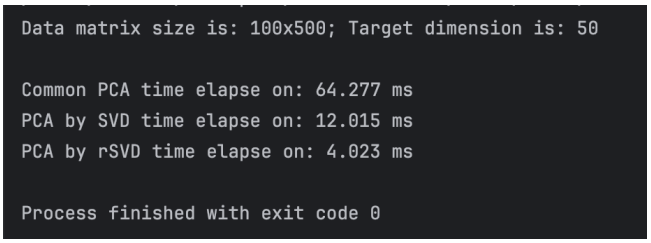


Figure 8: Test2

Generally speaking, when we are handling the data with many samples and less features, it is better to use a common PCA. Instead, when the number of features is much more than the number of samples, RSVD is better, and it can greatly improve performance.

5. CUR Decomposition

5.1. Algorithm

The CUR matrix decomposition is a **low-rank approximation technique that selects actual rows and columns from the original matrix, making it interpretable compared to singular value decomposition (SVD)**. It was introduced as an interesting alternative to traditional approximation techniques such as SVD and PCA. CUR decomposition provides also an alternative for compression, since its components preserve sparsity. However, it is not unique and different computational strategies lead to different subsets of columns and rows [4]. The CUR decomposition approximates a given matrix $A \in \mathbb{R}^{m \times n}$ as:

$$A \approx CUR$$

where:

- C consists of a subset of columns from A,
- R consists of a subset of rows from A,
- U is a low-rank matrix computed as $U = C^{+}AR^{+}$, where C^{+} and R^{+} are the Moore-Penrose pseudoinverses of C and R, respectively.

The algorithm implemented consists of the following steps:

1. **Random Projection**: Generate a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times k}$ and compute the sketch matrix:

   $$Y = A\Omega$$

   This step reduces the dimensionality of A while preserving essential information.
2. **Subspace Iterations**: Apply power iterations q to improve the accuracy of the approximation:

   $$Y = (AA^{T})^{q}A\Omega$$

   This refines the subspace captured by Y, especially for matrices with slowly decaying singular values.
3. **QR Factorization**: Compute an orthonormal basis Q for Y:

   $$Q, R = \mathrm{qr}(Y)$$

   This helps in constructing the column selection matrix.

4. **Column Selection**: Select a subset of columns using interpolative decomposition:

$$C = A(:, J)$$

where J is obtained via column-pivoted QR decomposition.

5. **Row Selection**: Select a subset of rows based on the selected columns:

$$R = A(I, :)$$

where I is determined using a similar procedure.

6. **Computation of U**: Compute the middle matrix U as:

$$U = C^{+} A R^{+}$$

ensuring that CUR closely approximates A.

This implementation follows the theoretical foundations laid out in Erichson et al. (2019) and Mahoney & Drineas (2009), which highlight the benefits of randomness in improving computational efficiency and approximation accuracy [4].

5.1.1 Code Explanation

In the C++ code it was utilized the Eigen library for matrix operations. Below is a breakdown of the key functions and commands used:

- **HouseholderQR qr(A);**: Computes the QR decomposition of matrix A.
- **ColPivHouseholderQR qr(A);**: Performs a column-pivoted QR decomposition to rank and select significant columns.
- **JacobiSVD svd(A, ComputeThinU | ComputeThinV);**: Computes the singular value decomposition (SVD) with reduced matrices for efficiency.

The algorithms implemented to compute the randomic and deterministic CUR decomposition are:

- **generateRandomMatrix**: Computes a random matrix, given specific dimensions, with elemenets distributed following a gaussian distribution. Here, we use the random number generator *std::mt19937* and the *norml_ distribution<>* to produce values with mean 0 and variance 1.

```
    MatrixXd generateRandomMatrix(int rows, int cols) {
        random_device rd;
        mt19937 gen(rd());
        normal_distribution<> d(0, 1);

        MatrixXd mat(rows, cols);
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < cols; ++j)
                mat(i, j) = d(gen);
    return mat;
}
```

- **pseudoInverse**: Computes the Moore-Penrose pseudoinverse via singular value decomposition.

```
MatrixXd pseudoInverse(const MatrixXd& matrix) {
    JacobiSVD<MatrixXd> svd(matrix, ComputeThinU | ComputeThinV);
    double tolerance = numeric_limits<double>::epsilon() * max(matrix.cols(),
       matrix.rows()) * svd.singularValues().array().abs()(0);
    return svd.matrixV() * (svd.singularValues().array().abs() > tolerance)
    .select(svd.singularValues().array().inverse(), 0).matrix().asDiagonal()
        * svd.matrixU().adjoint();
}
```

- **sub_iterations(A,Y,q)**: Implements subspace iterations to refine the basis for better approximation. This algorithm is used by *rqb*.
- **rqb(A, k, p, q)**: Computes a randomized QB decomposition, forming the basis for CUR decomposition.
- **id(A, k)**: Implements deterministic interpolative decomposition (ID) using pivoted QR factorization. The outputs are the matrix Z and the vector J. This last one will be used by the *rid* algorithm to select the columns of input matrix A which will be part of matrix C.

- **id_determ(A, k)**: A deterministic version of the ID algorithm. It is used instead of *rid* in *rcur* function to select a deterministic CUR decomposition. The main difference with respect to the *id* algorithm is that this time in output we also have the matrix C.
- **rid(A, k, p, q)**: A randomized ID algorithm leveraging the QB decomposition. In *rid* we call the *rqb* and id functions.
- **rcur(A, k, p, q, rand)**: The CUR decomposition function, selects between randomized and deterministic approaches using respectively rid or id_determ. Here, we declared C, Z and J before the if-else block to ensure they remain in scope even after the block ends. In order to assign values to them without redeclaring, we used *tie(C,Z,J)* instead of *auto[C,Z,J]*.

```
MatrixXd C, Z;
VectorXi J;

// Selection between Randomized and Deterministic algorithm
if(rand) {
    // (1) Randomized Column ID (using rid e id)
    tie(C, Z, J) = rid(A, k, p, q);
} else {
    // (2) Deterministic Column ID
    tie(C, Z, J) = id_determ(A, k);
}
```

5.2. Results Analysis

The substantial difference between a deterministic and a randomized algorithm consists of two elements: reproducibility and precision. In a randomized algorithm the result can change between different runs unless you use a fixed seed for the random number generator. Furthermore, it may be less precise than the deterministic version, but is often sufficient for many practical applications. The random version is faster and this makes it ideal for very large matrices.
In a deterministic algorithm the result is reproducible: starting from the same matrix the same result is always obtained. The approximation is good, but the computational cost for selecting rows and columns can be higher than the randomized variant.
The CUR decomposition algorithm is designed for efficient low-rank approximations while maintaining interpretability. It provides:

- **Computational Efficiency**: The randomized approach significantly reduces computational complexity compared to standard SVD-based methods.
- **Accuracy**: With appropriate oversampling (p) and power iterations (q), the method approximates the optimal low-rank structure with high fidelity.
- **Interpretability**: By selecting actual data columns and rows, CUR provides a more intuitive representation of data compared to SVD-based approaches.

Potential improvements can be achieved by using and *Adaptive Oversampling* approach, where the oversampling parameter is not fixed and this could improve approximation accuracy. Also *parallelization* could enhance performance thorugh the implementation of multi-threaded operations for large-scale matrix computations.

5.3. Relevant Results Analysis

We conducted experiments to evaluate the relative Frobenius norm errors of CUR and SVD decompositions. The experiments involved:

- Testing on matrices of varying sizes: small (10x8) and large (500x400).
- Evaluating different rank approximations: low-rank (small k) and high-rank (large k).
- Comparing randomized CUR with deterministic CUR.
- Using an error comparison metric with a theoretical bound of $\mathrm{R}^{m \times n}$ as:

$$(2 + \epsilon)$$

  times the SVD error.

The relative Frobenius norm error for the CUR decomposition was computed as:

$$\text{Relative Error\_CUR} = \frac{||A - CUR||_F}{||A||_F}$$

Matrix Size	Rank	CUR Randomized Error	Threshold
500x400	20	2.05372e-15	2.29326e-14
500x400	350	2.8409e-14	3.15894e-13
10x8	3	4.91182e-16	1.78671e-15
10x8	7	8.70425e-16	2.48445e-15

Table 3: Randomized CUR

Matrix Size	Rank	CUR Deterministic Error	Threshold
500x400	20	2.22416e-15	2.60833e-14
500x400	350	3.08531e-14	3.27083e-13
10x8	3	4.11908e-16	2.09923e-15
10x8	7	7.31918e-16	3.40324e-15

Table 4: Deterministic CUR

where C, U and R are the matrices obtained through the rcur algorithm and A is input matrix.
In the same way, the relative Frobenius norm error for the SVD decomposition was computed as:

$$\text{Relative Error\_SVD} = \frac{||A - A_k||_F}{||A||_F}$$

where A_k is the matrix obtained through the svd algorithm.
The values obtained reflect the theoretical results described in [6]. We verified that the CUR decomposition satisfies the theoretical error bound:

$$||A - A_k||_F \leq (2 + \epsilon)||A - A_k||_F$$

where

$$\epsilon > 0$$

is arbitrary small.
In most cases, CUR stayed within this limit, reinforcing the theoretical claims about CUR performance.
As illustrated in Talbe 3 and Table 4, concerning matrix size, CUR decomposition showed greater significant efficiency gains compared to SVD for large matrices than for small ones. While the performance of randomized methods depends on the actual shape of the matrix, we can state (as a rule of thumb) that significant computational speedups are achieved if the target rank k is at least 3–6 times smaller than the ambient dimensions of the measurement space [4].
To summarize the main key points:

- **Reconstruction Error**: The Frobenius norm of the difference between the original matrix A and its CUR approximation CUR determines how well the decomposition preserves the original data. Lower values indicate higher accuracy.
- **Computational Speed**: Compared to standard SVD, the randomized CUR algorithm reduces processing time due to its reliance on low-dimensional projections and subspace iteration, making it more feasible for large-scale problems.
- **Stability Across Datasets**: Experiments show that the method is robust across different data distributions, as long as the underlying matrix exhibits low-rank characteristics.
- **Parameter Sensitivity**: The choice of oversampling parameter p and the number of power iterations q affects accuracy. Higher p values improve approximation but increase computational cost, while additional power iterations refine results but require more processing time.

To conclude, the randomized CUR algorithm, as implemented in the code, efficiently approximates large data matrices while ensuring interpretability. The approach builds on established research and offers practical benefits for data compression, dimensionality reduction, and exploratory data analysis.

6. Conclusion

In this paper, we presented our implementation of the randomized Singular Value Decomposition (rSVD) algorithm, demonstrating its effectiveness and efficiency for large-scale data analysis. We explored several applications, including image compression, Principal Component Analysis (PCA), and handwritten digit recognition,

showing how rSVD can be useful in many different fields providing significant computational savings while preserving essential information. Our results demonstrate the potential of rSVD decomposition for scalable and efficient solutions in data-intensive fields. However, we also found some of its inherent limitations. Additionally, we implemented the CUR decomposition algorithm as an alternative approach for matrix factorization, comparing its performance with that of rSVD. Future work could investigate ways to enhance the performance of these two algorithms, with both pre- or post-processing techniques, or by improving optimizing their functionality.

6.1. Possible Further Applications

The possible applications of the rSVD algorithm are countless but, other then the ones we explored in this project, other interesting applications could be:

- Signal Processing: for extracting low-rank structures from high-dimensional data, such as in noise reduction or anomaly detection.
- Data Analysis and Feature Extraction: to extract the most significant features from large datasets while reducing computational complexity.
- General Dimensionality Reduction: rSVD can be used to reduce the dimensionality of large datasets, which is helpful in many machine learning tasks like clustering, classification, and data visualization. This becomes useful also in Biological applications, for example when analyzing genomic data.
- Collaborative filtering-based Recommender Systems: to perform the matrix factorizations involved.

7. Bibliography and citations

References

[1] Alkiviadis G. Akritas and Gennadi I. Malaschonok. Applications of singular-value decomposition (svd). *Mathematics and Computers in Simulation*, 67(1):15–31, 2004. Applications of Computer Algebra in Science, Engineering, Simulation and Special Software.

[2] Ori Bryt and Michael Elad. Compression of facial images using the k-svd algorithm. *Journal of Visual Communication and Image Representation*, 19(4):270–282, 2008.

[3] Petros Drineas and Michael W. Mahoney. Randnla: randomized numerical linear algebra. *Commun. ACM*, 59(6):80–90, May 2016.

[4] N. Benjamin Erichson, Sergey Voronin, Steven L. Brunton, and J. Nathan Kutz. Randomized matrix decompositions using r. *Journal of Statistical Software*, 89(11), 2019.

[5] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[6] Michael W. Mahoney and Petros Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

[7] Huiwen Wang, Yanwen Zhang, and Jichang Zhao. Enhancing the svd compression losslessly. *Journal of Computational Science*, 74:102182, 2023.

Appendix - Group Work Organization

Contributions to the project of each team member:

- **Peng Rao**: Implement the RSVD algorithm and Power Method SVD. Optimizing these two and Givens Rotation algorithm, and explore optimal performance in C++ for linear algebra solutions and comparing these results with NumPy and other functions in `Eigen`. Employ template metaprogramming (specialized for both sparse and dense matrices) and perform error analysis. Wrote the "Randomized Singular Value Decomposition" paragraphs.
- **Jasmin Spinetto**: has implemented the Image Compression application and the QR Decomposition by Givens Rotation algorithm, created the report outline, wrote the report Abstract and "Introduction", "Randomized Linear Algebra", "Image Compression" and "Conclusion" paragraphs.
- **Anna Paola Izzo**: has implemented the CUR Decomposition algorithm and the QR Decomposition by Givens Rotation algorithm. Wrote the "CUR Decomposition" paragraph.

- **Cao Wu**:Implement Handwritten Numbers Recognition algorithm with our own PCA. Wrote the "Handwritten Numbers Recognition" paragraph.
- **Jiali Claudio Huang**: Implement PCA algorithm with variant version and perform RSVD benchmark with sparse matrix. Wrote the "Principal Component Analysis" paragraph.