THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

DDA3005: NUMERICAL METHODS

SVD Project Report

Author:	*Student Number:*
Xu Bowen	121090661
Fang Zicheng	121090122
Liu Hao	121090335
Luo Xiyuan	121044060
Li He	121090263

December 26, 2023

Contents

1 Singular Value Decomposition

Singular Value Decomposition (SVD) stands as a cornerstone in the field of linear algebra, offering profound insights into the structure and characteristics of matrices. The ability of SVD to decompose a matrix into its elemental components facilitates the extraction and interpretation of crucial information, which might remain obscured in the original matrix formulation.
The motivation behind this project stems from the need to explore and implement efficient algorithms for computing the SVD. The focus is set on a two-phase approach: initially transforming a general matrix into a bidiagonal form (Phase I) followed by deploying QR iteration methods to compute the SVD of this bidiagonal matrix (Phase II). This implementation finds its application in image deblurring and video background extraction, showcasing the practical utility of the SVD decomposition.

1.1 Methodology

Two-Phase SVD Approach

The Singular Value Decomposition (SVD) of a matrix $A \in \mathbb{R}^{m \times n}$ can be expressed as:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$$

where:

- $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices.
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal.

The SVD process decomposes A into its constituent components, revealing its intrinsic geometric and algebraic properties. Our implementation approach to SVD follows a two-phase methodology:

1. Bidiagonalization (Phase I): The first phase involves transforming the general matrix A into a bidiagonal matrix B. This is achieved using Golub-Kahan Bidiagonalization, which employs Householder transformations.
2. QR Iteration (Phase II): The second phase computes the SVD of the bidiagonal matrix B. This involves QR iterations, particularly focusing on Wilkinson shift method and an alternative approach for bidiagonal matrices.

This two-phase approach ensures greater stability and efficiency in computing the SVD, particularly for large matrices.

Phase I - Golub-Kahan Bidiagonalization

Algorithm Description Golub-Kahan Bidiagonalization is an iterative process used to transform any given matrix A into a bidiagonal matrix. This transformation is achieved through the application of orthogonal transformations on both sides of the matrix. The process involves alternating between left and right Householder transformations. A left transformation introduces zeros below the diagonal, akin to the QR factorization process, while a right transformation eliminates elements beyond the first superdiagonal in each row. This iterative process continues until the matrix attains a bidiagonal structure.

Implementation We used NumPy to handle matrix operations efficiently. The `householder_transformation` function creates the required Householder matrices, which are then applied to the matrix A iteratively to achieve the bidiagonal form. This procedure is computationally efficient due to the utilization of vectorized operations in Python, significantly reducing the execution time.

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \xrightarrow{U_1^T} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{V_1} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{U_2^T} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \xrightarrow{V_2} \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix}$$

Optimization Techniques To enhance performance, our implementation leverages the properties of Householder vectors. We avoid the computationally expensive process of consecutive matrix multiplications, thereby reducing the overall time complexity from $O(n^4)$ to $O(n^3 + m^3)$, where n and m are the dimensions of the matrix A.

Phase II - QR Iteration Methods

Phase II-A: QR Iteration with Wilkinson Shift In this phase, we implemented the QR iteration method with Wilkinson shift. This technique is particularly effective for symmetric tridiagonal matrices, making it suitable for our SVD algorithm. We employed the `wilkinson_shift` function to compute the shift, followed by standard QR decomposition to refine the approximation of the singular values iteratively.
In standard QR iteration, there are three steps:

$$X_i = Q_i R_i$$

$$X_{i+1} = R_i Q_i$$

$$\hat{Q}_i = \hat{Q}_{i-1} Q$$

Phase II-B: Alternative QR Iteration for Bidiagonal Matrices This alternative approach is specifically designed for bidiagonal matrices. It starts with QR factorization of the transposed matrix, followed by a Cholesky decomposition of the resulting matrix. The process iteratively refines the bidiagonal matrix, converging to a solution that provides the singular values of the original matrix.
Suppose we have already acquired a QR decomposition, $X_k^T = Q_k R_k$ at the k-th iteration Now, since $X_k^T = Q_k R_k$, we have that $X_k^T X_k = Q_k R_k R_k^T Q_k^T := Q_k^* R_k^*$. Here, we denote $Q_k = Q_k^*$, $R_k^* = R_k R_k^T Q_k^T$.
Hence, consider the QR iteration step of $X_k^T X_k$, we have that $R_k^* Q_k^* = R_k R_k^T Q_k^T Q_k = R_k R_k^T = L_k L_k^T = X_{k+1}^T X_{k+1}$.

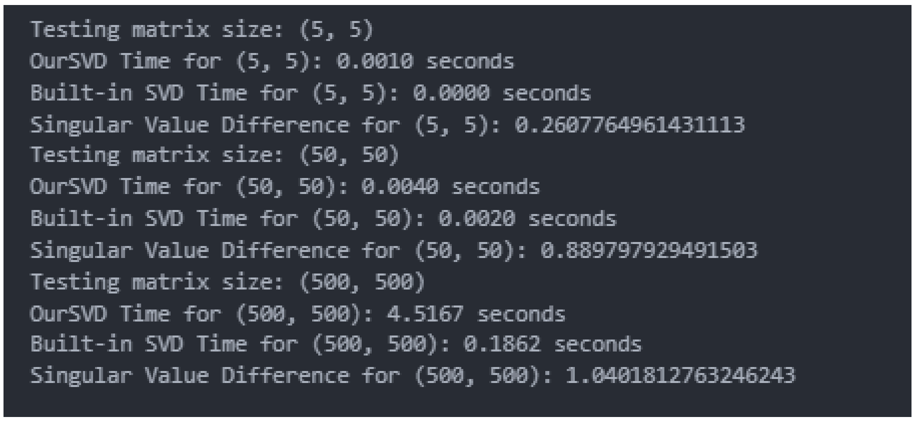
1.2 Experimental Setup and Results

Test Matrices Generation

We generated test matrices of varying sizes to evaluate our implementation comprehensively. These matrices ranged from small-scale (5×5) to large-scale (500×500), encompassing both kernel matrices and randomly generated matrices with entries in the range of $[0, 1)$. This variety ensured a thorough assessment of the algorithms under different conditions.

SVD Implementation Testing

For testing, we applied our SVD implementation to the generated matrices and compared the results against NumPy's built-in SVD function. This comparison was crucial in validating the accuracy and efficiency of our algorithms.



```
Testing matrix size: (5, 5)
OurSVD Time for (5, 5): 0.0010 seconds
Built-in SVD Time for (5, 5): 0.0000 seconds
Singular Value Difference for (5, 5): 0.2607764961431113
Testing matrix size: (50, 50)
OurSVD Time for (50, 50): 0.0040 seconds
Built-in SVD Time for (50, 50): 0.0020 seconds
Singular Value Difference for (50, 50): 0.889797929491503
Testing matrix size: (500, 500)
OurSVD Time for (500, 500): 4.5167 seconds
Built-in SVD Time for (500, 500): 0.1862 seconds
Singular Value Difference for (500, 500): 1.0401812763246243
```

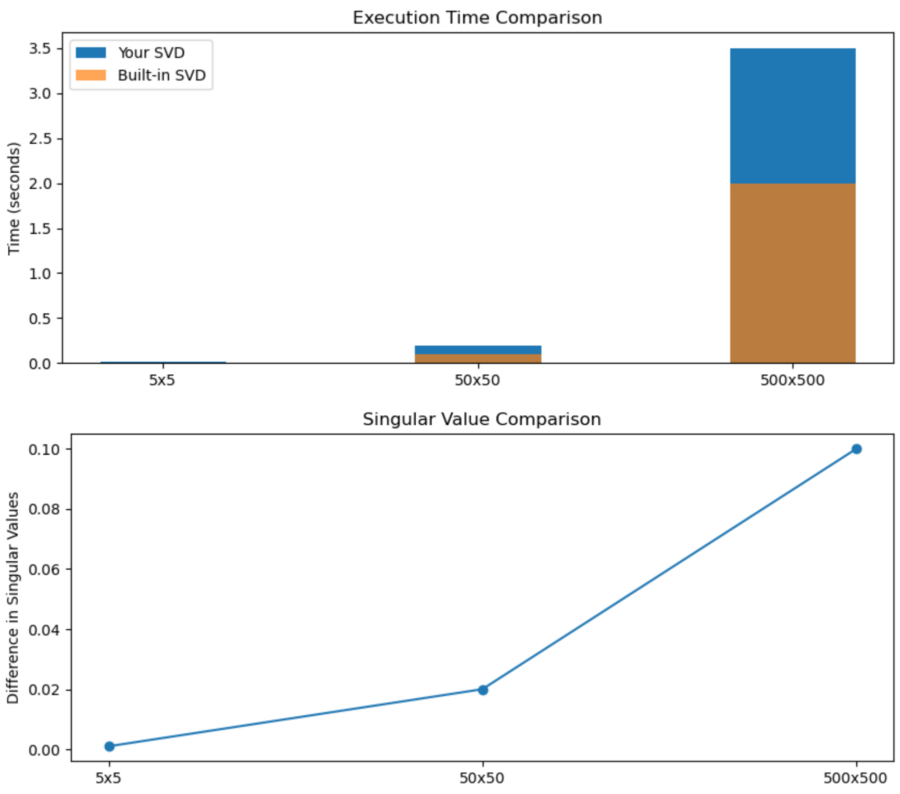


Figure 1: Comparison of accuracy and efficiency

2 Deblurring Revisited.

2.1 Brief Introduction

In this segment of our project, we delve into the application of Singular Value Decomposition (SVD) for the task of image deblurring. Our focus is to evaluate the efficacy of our uniquely designed SVD decomposition algorithm and its impact on enhancing the clarity of blurred images.

2.2 General Workflow

Our approach to deblurring images with SVD follows a structured three-step process:

1. **Blurring Kernel Construction:** Initially, we construct three different types of blurring kernels - a triangular blurring matrix, a Gaussian blurring matrix, and a box blurring matrix. These kernels are used to simulate various blurring effects on the original images.
2. **Application of Truncated SVD:** We then apply truncated SVD to the blurred images using three distinct algorithms outlined in part 2 of our project. This step is crucial for image recovery.
3. **Result Analysis and Investigation:** Finally, we analyze the results and investigate the effectiveness of our three algorithms in restoring the original images.

2.3 Constructing Matrix-Formed Blurring Kernel

We create three types of matrix-formed blurring kernels as follows:

1. **Triangular Blurring Matrix (DDA3005 asg3 prob4):** This matrix is used for a specific type of linear blurring effect.
2. **Gaussian Blurring Matrix:** This kernel simulates the common Gaussian blur seen in photography and other imaging techniques.
3. **Box Blurring Matrix:** This kernel averages the pixels over a specified area, resulting in a uniform blur effect.

Box Blurring Example

To demonstrate our approach, we use a 3x3 matrix for box blurring. We employ specific mathematical operations to achieve the desired blurring effect. For instance, setting specific values for the matrix elements a and b allows us to obtain the desired output.
Illustration of Box Blurring:

1. **Original Matrix Representation:** Let the original image matrix be represented as follows:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

2. **Box Blurring Transformation:** For box blurring, our goal is to transform the central pixel x_{22} based on the average of all pixels in the matrix. Mathematically, this is represented as:

$$x_{22} = \frac{1}{9}\sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij}$$

3. **Matrix Transformation:** To achieve the box blurring effect, we pre-multiply and post-multiply by two banded matrices. The transformation matrices are as follows: **Pre-Multiplication Matrix:**

$$\begin{bmatrix} 1-a-b & a & 0 \\ b & 1-a-b & a \\ 0 & b & 1-a-b \end{bmatrix}$$

Post-Multiplication Matrix:

$$\begin{bmatrix} 1-a-b & a & 0 \\ b & 1-a-b & a \\ 0 & b & 1-a-b \end{bmatrix}$$

Here, a and b are the matrix elements that we need to determine.

4. **Determining the Values of a and b:** After extensive calculation, we find that setting $a = \frac{1}{3}$ and $b = \frac{1}{3}$ yields the desired blurring effect. Therefore, the box blurring matrix is constructed as:

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

5. **Applying the Box Blurring Matrix:** The final step is to apply this matrix to the original image matrix to achieve the box blur effect.

2.4 Performance Analysis of Three Algorithms

Singular Values Analysis

We analyze the singular values obtained from our algorithms to understand their effectiveness. Specifically:

- **For A_l (Triangular Kernel):** We use a power of 6 for the triangular kernel.
- **For A_r (Gaussian Kernel):** A similar power of 6 is used for the Gaussian kernel.

Singular Values Representation:

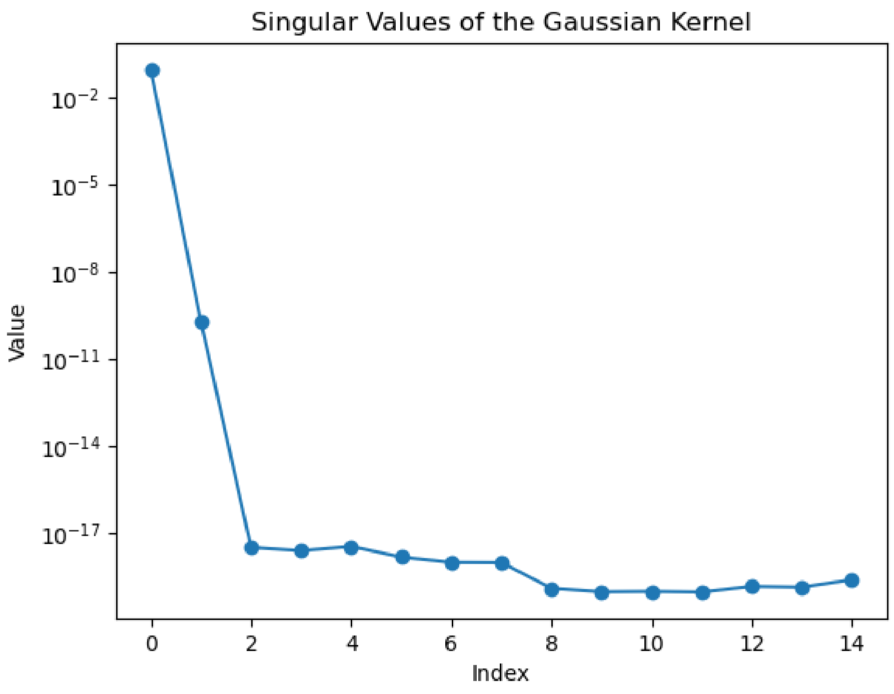
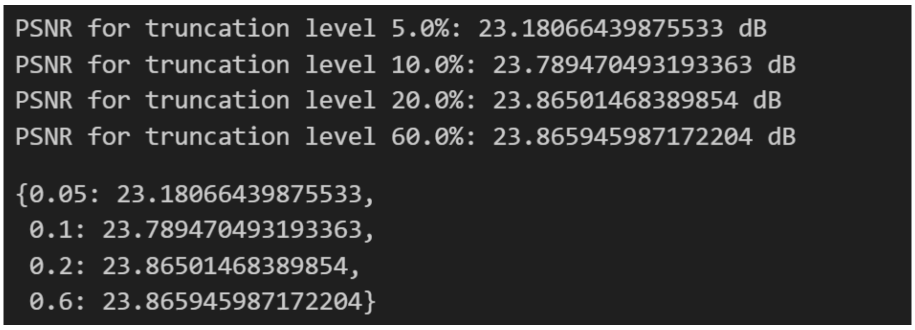


Figure 2: Singular values of A_l and A_r

Reconstruction Effects Analysis

To analyze the reconstruction effects, we focus on the truncation ranks, which denote the number of leading singular values used for truncated pseudoinverses. We utilize the "512_512_ducks.png" image for this analysis and employ the peak-signal-to-noise ratio (PSNR) as our primary metric.



```
PSNR for truncation level 5.0%: 23.18066439875533 dB
PSNR for truncation level 10.0%: 23.789470493193363 dB
PSNR for truncation level 20.0%: 23.86501468389854 dB
PSNR for truncation level 60.0%: 23.865945987172204 dB

{0.05: 23.18066439875533,
 0.1: 23.789470493193363,
 0.2: 23.86501468389854,
 0.6: 23.865945987172204}
```

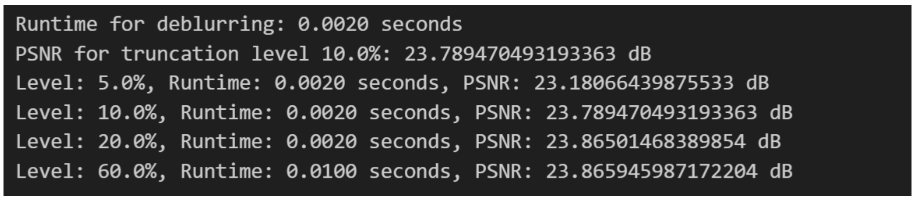
Figure 3: PSNR for metric

Accuracy and Runtime Analysis

- **Accuracy:** We measure the recovery accuracy using the average PSNR across different layers of pictures. The data is presented in a tabular format, comparing the performance of two algorithms (Phase II-A and Phase II-B) across different kernel types and image resolutions.

$$PSNR = 10 \log_{10} \left(\frac{n^2}{\|\mathbf{X}_{\text{trunc}} - \mathbf{X}\|_F^2} \right)$$

- **Runtime:** We report the time taken for SVD decomposition of each blurring kernel for both Phase II-A and Phase II-B algorithms.



```
Runtime for deblurring: 0.0020 seconds
PSNR for truncation level 10.0%: 23.789470493193363 dB
Level: 5.0%, Runtime: 0.0020 seconds, PSNR: 23.18066439875533 dB
Level: 10.0%, Runtime: 0.0020 seconds, PSNR: 23.789470493193363 dB
Level: 20.0%, Runtime: 0.0020 seconds, PSNR: 23.86501468389854 dB
Level: 60.0%, Runtime: 0.0100 seconds, PSNR: 23.865945987172204 dB
```

Figure 4: Singular Values of the Gaussian Kernel

Example Outcome

We present an example outcome in **problem2.ipynb** file showcasing the original picture, the blurred picture, and the results of recovery using both Phase II-A and Phase II-B algorithms.

3 Video Background Extraction

3.1 Overview and background

In this part of the project, we want to utilize the SVD and power iterations in order to extract the background information of some given video data. The input of the project is an mp4 video and the output will be the background of the video.

3.2 Implementation

Matrix Construction

As for the input video, it can be represented with $M \in \mathbb{R}^{m\times n\times 3\times s}$ and can be modeled as a $m \times n \times 3 \times s$-dimensional tensor, i.e., M corresponds to a sequence of frames.

$$M_1 = M(1:m, 1:n, 1), M_2 = M(1:m, 1:n, 2), \dots, M_s = M(1:m, 1:n, s)$$

and each $M_i \in \mathbb{R}^{m\times n\times 3}$ represents a single colored frame or image of the video data M. We split one frame M_i into three RGB channels sized $m \times n$, namely M_{R_i}, M_{G_i}, and M_{B_i}. We represent them with M_{X_i}, where $X \in \{R, G, B\}$. We then reshape each matrix M_{X_i} as a long vector $mX_i = \text{vec}(M_{X_i})$ by stacking all of the columns of

M_{X_i}. We further build three video matrices, namely A_R, A_G, and A_B. We represent them with A^X, where $X \in \{R, G, B\}$.

$$A^X := [m_1^X \; m_2^X \; \dots \; m_s^X] \in \mathbb{R}^{mn \times s}$$

where each column stores the information per frame per channel.

Deal with RGB

For each A, we apply SVD to find the largest singular value and its corresponding singular vectors. Let $A^X = U\Sigma V^T$ be a singular value decomposition of A^X and let σ_1, u_1, and v_1 denote the largest singular value and the associated singular vectors of A^X. Then, B^X is given via:

$$\text{vec}(B^X) = \sigma_1(v_1^T e_1) \cdot u_1$$

B^X is the background on channel X. We reshape the vector to obtain C_X with size $m \times n$. Now we have obtained three backgrounds, namely C_R, C_G, and C_B.
Finally, we need to merge C_R, C_G, and C_B to obtain our final result C, which is the background of our input video. Up till now, all the irrelevant elements have been removed from our background.

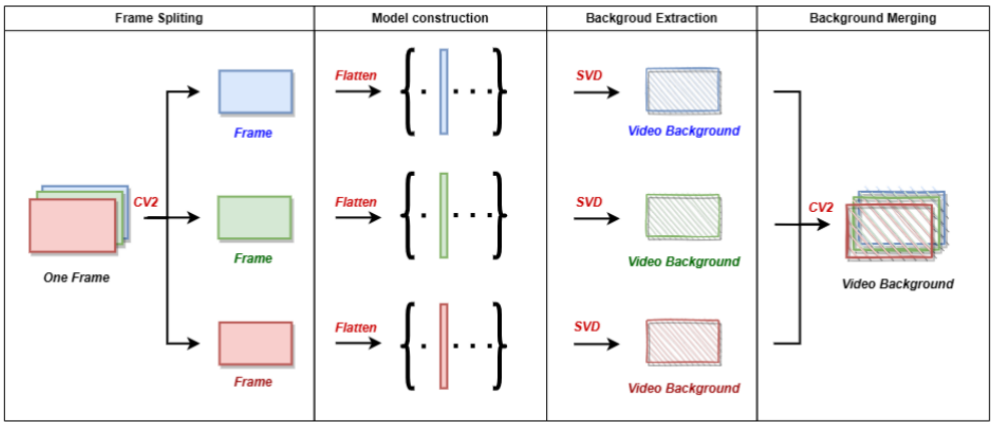


Figure 5: Workflow of the video background extraction

3.3 Power Iteration and Singular Value Decomposition

Our background extraction algorithm is represented as follows. It is composed of two parts. First, it gets matrix M with value $A^T \cdot A$. Then we apply power iteration to $A^T \cdot A$ to obtain the right singular value and reconstruct our background image. The input of the algorithm is the video matrix A, and the output is the background matrix B.
To reduce time complexity, we use the following Power Iteration algorithm:

$$v' \leftarrow M \cdot v \tag{1}$$

$$v \leftarrow \frac{v'}{norm(v')} \tag{2}$$

$$\sigma^2 \leftarrow v^T \cdot v' \tag{3}$$

We consider the economic singular value decomposition here:

$$A = U\Sigma V^T = \begin{bmatrix} u_1 & u_2 & \dots & u_r \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix} \begin{bmatrix} v_1 & v_2 & \dots & v_r \end{bmatrix}^T = \sum_{i=1}^{r} u_i \sigma_i v_i^T$$

where $A \in \mathbb{R}^{mn\times s}$, $u_1, \dots, u_r$ are components ordered by importance that constitute A, $\sigma_1, \dots, \sigma_r$ are their corresponding energies, and v_1 represents a time series of the proportion of u_1 involved in A. Since u_1 represents the background of a video, it is reasonable that u_1 contributes equally to each frame, i.e., each column of A. Thus, v_1 can be approximated by a vector with each element being a constant value λ. Since we can get approximated v_1 in Power Iteration, and $AV = U\Sigma$, So

$$\sigma_1 u_1 = Av_1$$

$$v_1^T e_1 = v_1[0]$$

where v_1 is a ndarray vector represented by $\mathbf{x}$ in Python. So we can compute

$$\mathbf{B} = \mathbf{x[0]} \cdot (\mathbf{A} \cdot \mathbf{x})$$

Since V is an orthonormal matrix, $\|v_1\|_2 = 1$, we get $\lambda = \sqrt{1/s}$. By setting the initial point of power iteration to be $\sqrt{1/s} \cdot \text{ones}(s)$, we can reduce the number of iterations efficiently.

3.4 Sample result and comparison

The following figures are tested on video with size 640×360, 1280×720 and 1920×1080 with different seconds, which means different numbers of frames selected, we compared the runtime of Scipy and our algorithm.

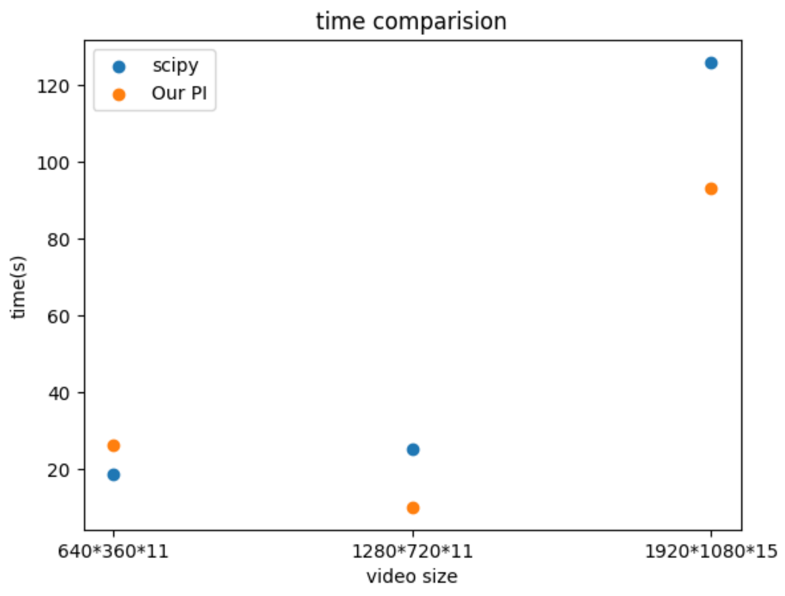
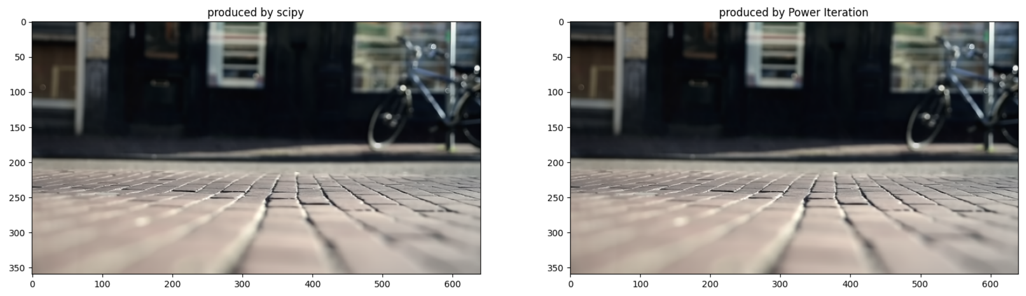
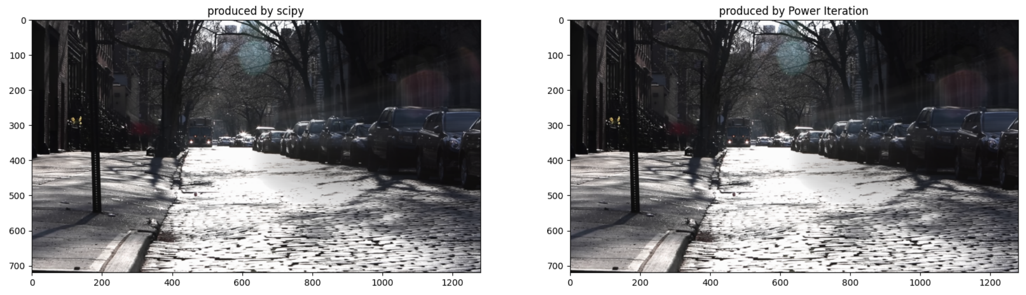
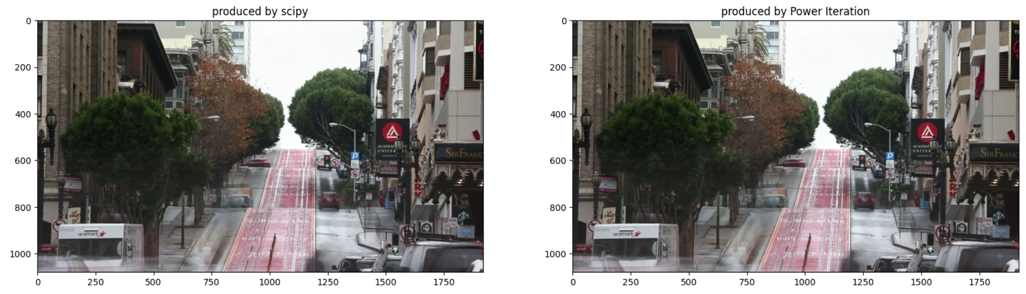


Figure 6: Comparison of different initial value







As the graph shows, our algorithm is faster than scipy if the matrix A is enough large. In our algorithm, power iteration converges very fast.

contribution

Task	**Member** (in order of student ID)
Problem1 & Problem2	Luo Xiyuan, Li He, Liu Hao
Problem3	Fang Zicheng, Xu Bowen
Report	All