The Chinese University of Hong Kong, Shenzhen

DDA 3005

Numerical Methods

Report for Project

Author	**Student ID**
黎俊乐 Li, Junle	118010142
赵松霖 Zhao, Songlin	120090346
杨蕙菡 Yang, Huihan	120090438
王修远 Wang, Xiuyuan	120090478
林锦睿 Lin, Jinrui	120090527

Code links: https://github.com/Linnore/DDA3005_114514.git

Dec 18, 2022

1 Introduction and motivation

Singular value decomposition (SVD) is a mathematical method for decomposing a matrix into a product of three matrices. It is a widely used technique in data analysis and has various applications, including in the fields of data compression, noise reduction, and image processing.

The significance of SVD lies in its ability to reduce a matrix to its fundamental components, making it easier to understand and interpret the information it contains. By decomposing a matrix into its constituent parts, we can extract useful information and gain insights that would not be apparent from the original matrix. For example, SVD can be used to reduce a high-dimensional dataset to a lower-dimensional space while preserving the important patterns and relationships in the data. This can be useful for visualizing complex data, as well as for improving the performance of machine learning algorithms.

This inspired us to implement this project. We investigate and study different algorithms to compute SVD and utilize them in the field of image deblurring. We also utilize the simple svd decomposition on the video backgruond extraction to illustrate the powerful effects of svd decomposition.

2 Design of SVD algorithms: A two phase approach

To start, we want do a SVD decomposition

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

of a general matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$.

Without loss of generality, we only consider the tall and square matrix. For fat matrix, we compute the SVD of $\mathbf{A}^T = \widehat{\mathbf{U}}\Sigma\widehat{\mathbf{V}}^{\mathbf{T}}$ and returns $\mathbf{U} = \widehat{\mathbf{V}}, \mathbf{V}^{\mathbf{T}} = \widehat{\mathbf{U}}^{\mathbf{T}}$.

2.1 Introduction of two-phase method

To generalize and stabilize the SVD decomposition, two-phase approach is chosen to implement the decomposition. This approach involves two steps. First, general matrix is transformed into bidiagonal matrix. Second, we do the SVD decomposition on the bidiagonal matrix to get the desired results. (mechanisms and details will be discussed in the later content)

The two-phase approach is an iterative algorithm, meaning that it repeats these two steps until it has computed a sufficiently accurate approximation of the SVD. It is a popular method for computing SVD because it is relatively easy to implement and can handle a wide range of matrices.

2.2 Phase I - Global-Kahan Bidiagonalization

In this step, the algorithm uses Golub-Kahan bidiagonalization to transform the matrix $\mathbf{A}$ into a bidiagonal form.

In general, it utilizes distinct orthogonal transformation iteratively on the left and right to bidiagonalize the general matrix. The result of this step is a matrix that has non-zero entries only along the main diagonal and the superdiagonal or lowerdiagonal.

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

Improvement: forming Q and P inversely – see svd_phaseI() in ./utils/SVD.py

If Q and P are formed in a naïve way, i.e., using consecutive matrix multiplications, the overall time complexcity will be $\mathcal{O}(n^4)$. Our implementation exploits the properties of HouseHolder vectors and corresponding reflation matrices by the followings, and achieves $\mathcal{O}(n^3 + m^3)$. Suppose we seek to compute $Q = H_0 H_1 ... H_{n-1}$, we proof the following lemma: $B_k = H_k ... H_{n-1}$ has the form $\begin{bmatrix} I_{k\times k} & O \\ O & \hat{B}_k \end{bmatrix}$.

The proof is by induction on k.

Base case: $k = n, \hat{B}_k = I_{n\times n}$.

Inductive step: Assume the result is true for B_k. We now show that this is true for B_{k-1}.

$$B_{k-1} = H_{k-1} B_k = H_{k-1} \begin{bmatrix} I_{k\times k} & O \\ O & \hat{B}_k \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} I_{k-1\times k-1} & O \\ O & I - \frac{1}{\tau_k} \begin{pmatrix} 1 \\ u_k \end{pmatrix} \begin{pmatrix} 1 & u_k^T \end{pmatrix} \end{bmatrix} \begin{bmatrix} I_{k-1\times k-1} & 0 & O \\ 0 & 1 & 0 \\ O & 0 & \hat{B}_k \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} I_{k-1\times k-1} & O \\ O & (I - \frac{1}{\tau_k} \begin{pmatrix} 1 \\ u_k \end{pmatrix} \begin{pmatrix} 1 & u_k^T \end{pmatrix}) \begin{pmatrix} 1 & 0 \\ 0 & \hat{B}_k \end{pmatrix} \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} I_{k-1\times k-1} & O \\ O & \hat{B}_{k-1} \end{bmatrix} \tag{4}$$

Now we investigate the time complexity of such an iterative update procedure. In update i, it takes $\mathcal{O}(i^2)$ to perform matrix-vector multiplication once, and $\mathcal{O}(i^2)$ to calculate a vector-vector outer product and scaling by constant. Finally, it takes $\mathcal{O}(i^2)$ to subtract one matrix from another. Hence, the overall time complexity of forming Q for a tall matrix is $\sum_{i=1}^{n} \mathcal{O}(i^2) = \mathcal{O}(n^3)$. Now, noticing that we calculate the SVD of a fat matrix by calculating the SVD of its transpose, we deduce that the time complexity of forming Q inversely is $\mathcal{O}(\max(m, n)^3)$.

The procedures for forming P is essentially the same with that of forming Q. Since we multiply H_i on the left to get Q and G_i on the right to get P, they will not affect each other.

Special case handling: dealing with singular matrix $\in \mathbb{R}^{m\times n}$

For singular matrix $\mathbf{A}$, there will be cases that $a_{ik} = 0$ in the ith column for $k = i, i+1, ..., m$. In this case, we let $\mathbf{Q_i} = \mathbf{I}$ and $\mathbf{P_i} = \mathbf{I}$.

2.3 Phase II

Phase II - SVD computation: In this step, the algorithm uses the bidiagonal matrix from the first step to compute an approximation of the SVD. This is done by solving a set of linear equations, which allows the algorithm to refine its estimate of the singular values and singular vectors.

Little trick: Calculating U directly from V
In the standard SVD, we have that: $A = U\Sigma V^T$.
Now, suppose we do Golub-Kahan bidiagonalization on A to get a bidiagonal matrix B: $B = Q^T AP$, where Q and P are unitary.
Now consider $B^T B$ and BB^T. It is trivial to see that $BB^T \sim AA^T$, $A^T A \sim B^T B$:
$B^T B = P^T A^T QQ^T AP = P^T A^T AP, BB^T = Q^T APP^T A^T Q = Q^T AA^T Q$.
In the context of phase II, we perform eigendecomposition on BB^T and $B^T B$, which is,
$B^T B = STS^T = S\Sigma^2 S^T = P^T A^T AP, BB^T = GTG^T = G\Sigma^2 G^T = Q^T AA^T Q$.
The diagonal matrix of $B^T B$ containing eigenvalue is Σ^2 since $B^T B \sim A^T A$.
It is trivial that,
$A^T A = PS\Sigma^2 S^T P^T = V\Sigma^2 V^T, AA^T = QG\Sigma^2 G^T Q^T = U\Sigma U^T$.
Now, one can observe that $V = PS$, $U = QG$.
Therefore, suppose that now we have information of S. It is actually possible to obtain G in a much more efficient way: $B = Q^T AP = Q^T U\Sigma V^T P = G\Sigma S^T$.
Hence, $G = BS\Sigma^{-1}$.
Therefore, following the above procedure, we could calculate U from V.

2.3.1 Phase IIA

Suppose we have the matrix **A** has been reduced to bidiagonal form $\mathbf{B} \in \mathbb{R}^{n\times n}$.

Then, we implement QR iteration with Wilkinson shift and deflation to calculate an eigendecomposition of the symmetric tridiagonal matrix $\mathbf{B}^T\mathbf{B}$.

Then, we use the obtained eigenvalues and eigenvectors to construct a singular value decomposition.

Improvement1: Avoiding recursion – see eigh_by_QR() in ./utils/EVD.py
It might be tempting to implement the algorithm in a recursive way, but such implementation is hazardous because the RAM occupation could be monstrous when the matrix is huge. For every iteration, we need to store $\mathbf{Q_k}$ to wait for the result of the next iteration $\widehat{\mathbf{U}}_\mathbf{k}$. The storage space required for the kth iteration is $\mathcal{O}(k^2)$. Hence, the storage space required for the whole iteration is $\mathcal{O}(n^3)$. Hence, we resort to implementing the algorithm in a non-recursive way. Observing that after performing deflation for k times, only the left-upper $n \times n-k$ matrix is used. Hence, we use an index to record how many times the deflation procedure has been done and update the left-upper $n-k \times n-k$ matrix in-place in later iterations.

Improvement2: QR Factorization by Givens and Achieve Matrix Multiplications by Givens – see eigh_by_QR() in ./utils/EVD.py and applyGivenses() in ./utils/QR_Factorization.py

In standard QR iteration, there are three steps:

$$X_i = Q_i R_i \tag{5}$$

$$X_{i+1} = R_i Q_i \tag{6}$$

$$\hat{Q}_i = \hat{Q}_{i-1} Q \tag{7}$$

A naive implementation of (6) and (7) uses matrix multiplication, which costs $\mathcal{O}(n^3)$. To fully utilize the structural advantage of a symmetric tridiagonal matrix, we use QR factorization by Givens Rotations in (5). Note that for each Given rotation, two row operations are required to form the R factor and another two to form the Q factor, and we need to perform n-1 Givens Rotations. Thus QR factorization by Givens Rotations takes $\mathcal{O}(n^2)$. Further, to improve the complexity of (6) and (7), we may not need to form the Q factor explicitly in (5); instead, we record the series of Givens Rotations defined in (5) and apply these Givens rotations on the R factor and the cumulative Q product $\hat{Q}$ in (6) and (7) respectively. Series here mean n number of Givens rotations, and each Given rotation is simply two row or column operations. Therefore, using such a design, matrix multiplications of $\mathcal{O}(n^3)$ in (6) and (7) are avoided and replaced by these series of Givens rotations in $\mathcal{O}(n^2)$! Please check our codes for details!

Improvement3: Dealing with singular matrix – see svd_phaseII() in ./utils/SVD.py

Sometimes the bidiagonal matrix B we get from phase I is singular or nearly singular. We develop a specific procedure to handle those cases. Since B is bidiagonal, it is singular if and only if there are zeros on its diagonal. Now we have the following observation: $B' = \begin{bmatrix} B_1 & O & O \\ O & O & O \\ O & O & B_2 \end{bmatrix} = \begin{bmatrix} U_1 & O & O \\ O & O & O \\ O & O & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & O & O \\ O & O & O \\ O & O & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T & O & O \\ O & O & O \\ O & O & V_2^T \end{bmatrix}$

In other words, if there are zero entries in the middle of B's diagonal, we can do SVD on a matrix with those columns and rows discarded, which is a full-rank matrix, and insert zeros to U and V later.

Improvement4: Calculating BB^T quickly when B is a bidiagonal matrix– see ./utils/Bidiagonal_fastMult.py

Matrix multiplication generally costs $\mathcal{O}(n^3)$, but it is obvious that when B is a bidiagonal matrix, the matrix product BB^T can be interpreted as a recombination of the rows of B, with at most two rows of matrix B involved in the calculation of each row. Hence, the calculation can be done in $\mathcal{O}(n^2)$. Furthermore, if BB^T or B^TB is of our interest, the matrix multiplication can be done in $\mathcal{O}(n)$.

2.3.2 Phase IIB

Justification of PhaseIIB

Suppose we have already acquired a QR decomposition, $\mathbf{X_k}^T = \mathbf{Q_k R_k}$ at the kth itertaion.
Now, since $\mathbf{X_k}^T = \mathbf{Q_k R_k}$, we have that $\mathbf{X_k}^T \mathbf{X_k} = \mathbf{Q_k R_k R_k}^T \mathbf{Q_k}^T := \mathbf{Q_k}^* \mathbf{R_k}^*$.

Here, we denote $\mathbf{Q_k} = \mathbf{Q_k}^*, \mathbf{R_k}^* = \mathbf{R_k}\mathbf{R_k}^T\mathbf{Q_k}^T$.
Hence, consider the QR iteration step of $\mathbf{X_k}^T\mathbf{X_k}$, we have that,$\mathbf{R_k^*}\mathbf{Q_k^*} = \mathbf{R_k}\mathbf{R_k}^T\mathbf{Q_k}^T\mathbf{Q_k} = \mathbf{R_k}\mathbf{R_k}^T = \mathbf{L_k}\mathbf{L_k}^T = \mathbf{X_{k+1}}^T\mathbf{X_{k+1}}$.

Improvement1: Avoiding recursion
We use the similar method to avoid recursion as mentioned in the 2.3.1 Improvement2.

Improvement2: QR Factorization by Givens and Achieve Matrix Multiplications by Givens
We use a similar procedure as improvement 2 in phaseIIA. But we found such enhanced procedures, though reducing the time complexity, may introduce some accuracy issues when the matrix contains mostly very small numbers like $< 10^{-10}$. This is mainly due to the cancellation effects when computing the values of c and s that defines the Givens rotations. Note that the accuracy issues are severe in phaseIIB but acceptable in phaseIIA.

Improvement3: Dealing with singular matrix – see svd_phaseII() in ./utils/SVD.py

We use a similar procedure as improvement 3 in phaseIIA to handle singular matrices. Handling of singular matrices is crucial in phaseIIB because one can only apply Cholesky decomposition to nonsingular matrices.

2.4 Performance

The table below illustrates the performances of algorithms on some examples; for performances on other examples, please refer to the appendix. Kernel matrix refers to box blurring kernel with different sizes and power 66. All entries of random matrices are in $[0, 1)$. The numbers in matrix recovery are the percentages of entries of the original matrix recovered with SVD produced by the algorithm. The numbers in singular value recovery are percentages of singular values (we take that calculated by scipy.linalg as standard) of the original matrix recovered with SVD produced by the algorithm. It is easy to see that the algorithm performs well for matrices with different characteristics and sizes.

2.4.1 PhaseIIA

Dimensions and matrix type	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseIIA runtime
500, kernel	100%	97.28%	3.90E-08	1.10309s	0.299883s
1000, kernel	100%	97.02%	3.65E-08	8.222714s	1.415767s
100, random	100%	100%	9.03E-14	0.008s	0.145s
200, random	100%	100%	7.11E-14	0.074079s	0.587671s

2.4.2 PhaseIIB

Dimensions and matrix type	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseIIA runtime
500, kernel	100%	100%	1.14E-14	1.064225s	0.694181s
1000, kernel	100%	100%	3.56E-14	8.024716s	4.23336s
100, random	100%	100%	9.59E-14	0.008099s	2.35596s
200, random	100%	100%	1.61E-13	0.096014s	35.14804s

3 Image Deblurring: Based on our designed algorithm

3.1 Brief introduction

In this part of the project, we want to explore how our designed algorithm of SVD decomposition works in the field of picture deblurring.

3.2 General work flow

First, we construct three blurring kernel and blur the original pictures.
Second, we use truncated SVD with three different algorithms in the part2 to recover the pictures.
Third, based on the result, we further investigate and analyze the effect of our three algorithms.

3.3 Constructing Matrix-formed blurring kernel

We construct three matrix-formed blurring kernels: blurring matrix from DDA3005 asg3 prob4 (denoted as triangular blurring matrix for convenience in the later content), Gaussian blurring matrix, and box blurring matrix.

Examples on constructing blurring matrix

We demonstrate this using a 3x3 example with box blurring.
The original matrix is denoted as
$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$ .For box blurring, we want to have: $x_{22} = \frac{1}{9}\sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij}$.
Suppose we pre-multiply and post-multiply two banded matrices to the original picture,

$$\begin{bmatrix} 1-a-b & a & 0 \\ b & 1-a-b & a \\ 0 & b & 1-a-b \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 1-a-b & a & 0 \\ b & 1-a-b & a \\ 0 & b & 1-a-b \end{bmatrix}$$

After tedious calculation, we could conclude that setting $a = \frac{1}{3}, b = \frac{1}{3}$ can get the desiring result.
Therefore, we could get the box blurring matrix as

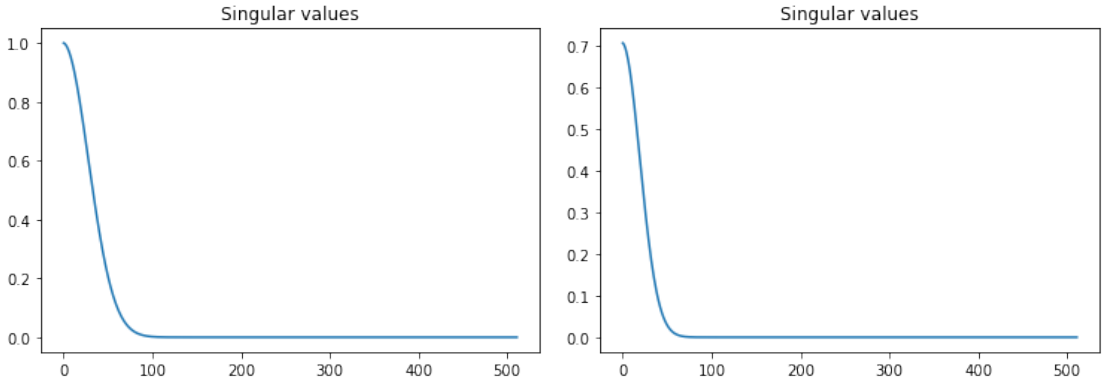
$$\begin{bmatrix} 1/3 & 1/3 & 0 \\ 1/3 & 1/3 & 1/3 \\ 0 & 1/3 & 1/3 \end{bmatrix}.$$

Things are essentially the same for Gaussian kernels.

3.4 Performance Analysis of three algorithms

3.4.1 Singular values analysis

Here for A_l, we use the triangular kernel with power of 66. for A_r, we use the Gaussian kernel with power of 66.



(a) Singular values of A_l (b) Singular values of A_r

Figure 1: Singular values

3.4.2 Reconstruction effects analysis

Truncation ranks in the following denotes the number of leading singular values used for truncated pseudoinverses.

Following the above singular value case, we use the "512_512_ducks.png" to do the analysis.

To compare quality of reconstruction, we use peak-signal-to-noise ratio,

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

As can been seen from Figure2, as the truncation rank increases, the PSNR increases, and the

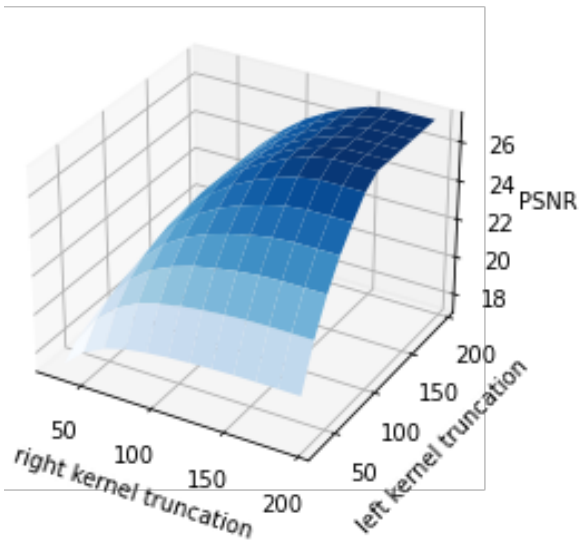
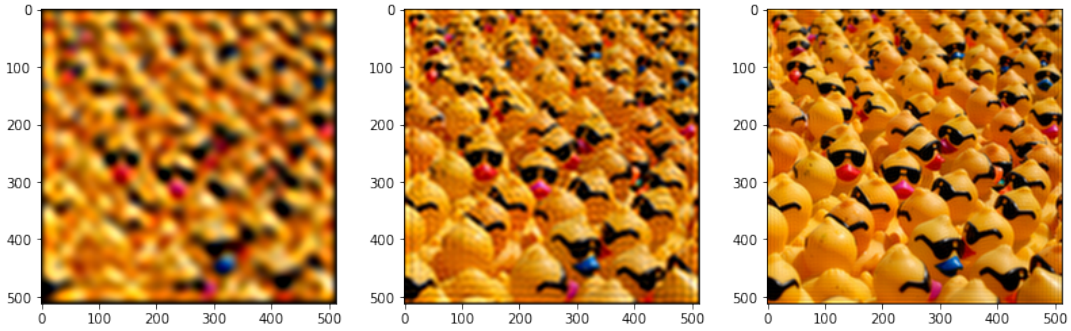


Figure 2: Comparison of different truncation ranks

reconstruction effect becomes better. In the last, the PSNR relatively stays the same as the truncation rank increases, indicating that the rest of the singular values are relatively small and are useless for reconstruction.

Figure3 are the visualization of the reconstruction effect with different truncation ranks.



(a) Recover with truncation 30 (b) Recover with truncation 70 (c) Recover with truncation 120

Figure 3: Reconstruction effect visualizing compared with different truncation ranks.

3.4.3 Accuracy and runtime analysis

Accuracy

Here we use the average PSNR of several layers of pictures to measure the recovery accuracy.

Algorithm	256_triangular	256_box	256_gaussian
PhaseII -A	22.4632	22.4357	21.2364
PhaseII -B	23.5775	22.4383	22.4632
Algorithm	512_triangular	512_box	512_gaussian
PhaseII -A	26.2261	26.2261	26.13838
PhaseII -B	26.2261	26.2261	21.9085
Algorithm	1024_triangular	1024_box	1024_gaussian
PhaseII -A	26.1304	26.1304	26.4722
PhaseII -B	26.1304	26.3457	26.4722

It seems that both of the two algorithms exhibit similar excellent performance on picture deblurring. To answer the question of which one returns more accurate singular values, we find that phase IIA returns the singular values with an accuracy of 1e-8 and phase IIB returns the singular values with accuracy of around 1e-15. More test results can be found in the appendix.

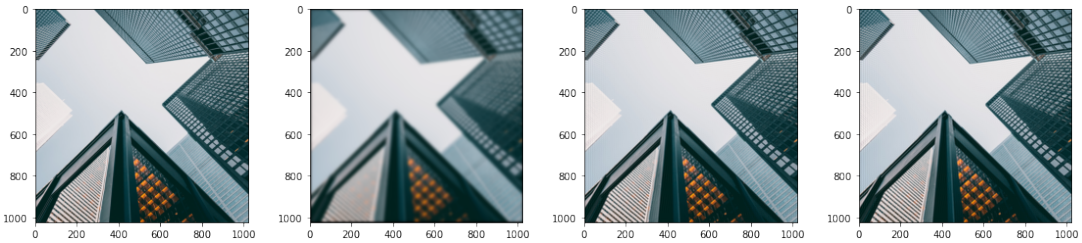
Runtime

Since the phase IIA and phase IIB algorithms are only used for SVD decomposition of blurring kernel, we report the time for solving SVD decomposition of one blurring kernel. (time:s)

Algorithm	256_triangular	256_box	256_gaussian
PhaseII -A	0.3225	0.2163	0.2516
PhaseII -B	0.4958	0.3709	0.2625
Algorithm	512_triangular	512_box	512_gaussian
PhaseII -A	1.7377	1.6063	1.3544
PhaseII -B	2.5153	2.1188	1.5401
Algorithm	1024_triangular	1024_box	1024_gaussian
PhaseII -A	14.8298	13.8463	12.3618
PhaseII -B	17.7953	16.9469	14.2002

In general, PhaseII -A takes less time then PhaseII -B. That is, PhaseII -A converges faster. More test result could be found in appendix.

3.4.4 One Example Outcome



(a) Original picture (b) Blur picture (c) Recover-PhaseII A (d) Recover-PhaseII B

4 Video Background Extraction

4.1 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.

4.2 General workflow

The general workflow is composed of four parts. First, we split a frame into three channels. Second, we flatten each channel and store them into three video matrices. Third, we do SVD on each matrix and extract the backgrounds on different channels. Finally, we merge three backgrounds and obtain our final result. The workflow is illustrated in the following figure.

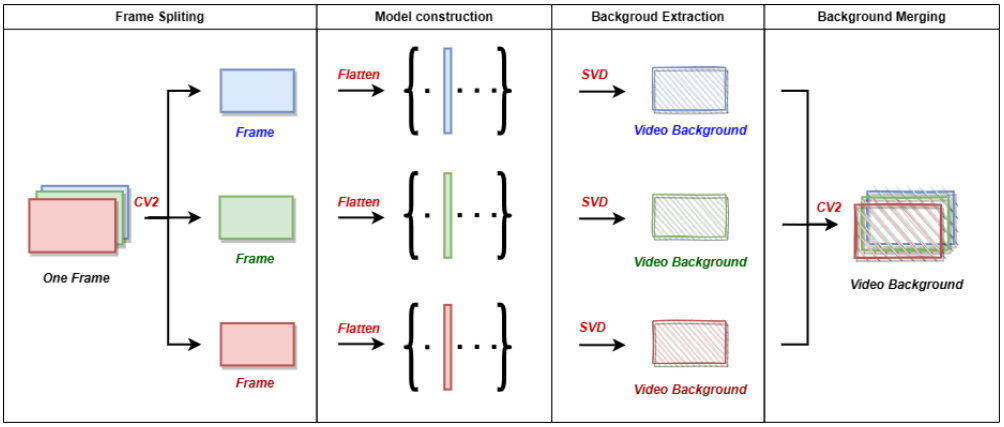


Figure 5: Workflow of the video background extraction

4.2.1 Frame Splitting

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

$$\boldsymbol{M}_1 = \boldsymbol{M}(1:\mathrm{m}, 1:\mathrm{n}, 1), \quad \boldsymbol{M}_2 = \boldsymbol{M}(1:\mathrm{m}, 1:\mathrm{n}, 2), \quad \ldots \quad \boldsymbol{M}_s = \boldsymbol{M}(1:\mathrm{m}, 1:\mathrm{n}, s)$$

and each $\boldsymbol{M}_i \in \mathbb{R}^{m \times n \times 3}$ represents a single colored frame or image of the video data $\boldsymbol{M}$. We split one frame $\boldsymbol{M}_i$ into three RGB channels sized $m \times n$, namely $\boldsymbol{M}_i^R$, $\boldsymbol{M}_i^G$, and $\boldsymbol{M}_i^B$. We

represent them with $\boldsymbol{M}_i^X$, where $X \in \{R, G, B\}$.

4.2.2 Matrix Construction

We then reshape each matrix $\boldsymbol{M}_i^X$ as a long vector $\boldsymbol{m}_i^X = \text{vec}\left(\boldsymbol{M}_i^X\right)$ by stacking all of the columns of $\boldsymbol{M}_i^X$. We further build three video matrices, namely $\boldsymbol{A}^R$, $\boldsymbol{A}^G$, and $\boldsymbol{A}^B$. We represent them with $\boldsymbol{A}^X$,where $X \in \{R, G, B\}$.

$$\boldsymbol{A}^X := \left[\begin{array}{cccc} \boldsymbol{m}_1^X & \boldsymbol{m}_2^X & \dots & \boldsymbol{m}_s^X \end{array} \right] \in \mathbb{R}^{mn \times s}$$

where each column store the information per frame per channel.

4.2.3 Background Extraction

For each A, we apply SVD to find the largest singular value and its corresponding singualar vectors. Let $\boldsymbol{A}^X = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ be a singular value decomposition of $\boldsymbol{A}^X$ and let $\sigma_1, \boldsymbol{u}_1$, and $\boldsymbol{v}_1$ denote the largest singular value and the associated singular vectors of $\boldsymbol{A}$. Then, $\boldsymbol{B}^X$ is given via:

$$\text{vec}(\boldsymbol{B}^X) = \sigma_1 \left(\boldsymbol{v}_1^\top \boldsymbol{e}_1\right) \cdot \boldsymbol{u}_1$$

$\boldsymbol{B}^X$ is the background on channel X. We reshape the vector to obtain C^x with size $m \times n$.Now we have obtained three background, namely $\boldsymbol{C}^R$, $\boldsymbol{C}^G$, and $\boldsymbol{C}^B$.

4.2.4 Background Merging

We then merge $\boldsymbol{C}^R$, $\boldsymbol{C}^G$, and $\boldsymbol{C}^B$ to obtain our final resualt C, which is the background of our input video. Up till now, all the irrelevant elements have been removed from our background.

4.3 Power iteration and singular value decomposition

Our background extraction algorithm is represented as follows. It is composed of two parts. First, it gets matrix $\boldsymbol{M}$ with value $\boldsymbol{A}^T \cdot \boldsymbol{A}$. Then we apply power iteration to $\boldsymbol{A}^T \cdot \boldsymbol{A}$ 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. Note that we apply the algorithm three times to reconstruct the colored image. The detailed explanation of A and B has been explained in 4.2.2 and 4.2.3.

Algorithm 1 EXTRACT-BACKGROUND

Require: Video matrix $A \in R^{mn \times s}$, threshold ϵ, initial point v_0
Ensure: Background vector $\text{vec}(B) \in R^{mn}$
$M \leftarrow A^T \cdot A$
$v \leftarrow v_0$
$\sigma^2 \leftarrow v^T \cdot M \cdot v$
while $\left\| \frac{Mv - \sigma^2 v}{s} \right\|_2 \geqslant \varepsilon$ **do**
  $v\_ \leftarrow M \cdot v$
  $v \leftarrow \frac{v\_}{\text{norm}(v\_)}$
  $\sigma^2 \leftarrow v^T \cdot v\_$
end while
$B \leftarrow v[0] \cdot (A \cdot v)$
Return B

4.4 Improvement comparing with the basic algorithm

4.4.1 Selection of frames

From numerical experiments we find that we can greatly reduce the run time of our algorithm while preserving the quality of background extraction by reducing the number of frames selected. In the following figure, we use the video of path "test_videos/1280_720/city_02.mp4". we stack one frame into matrix A every $1 \sim 20$ frames of the video and compare the run time of them.

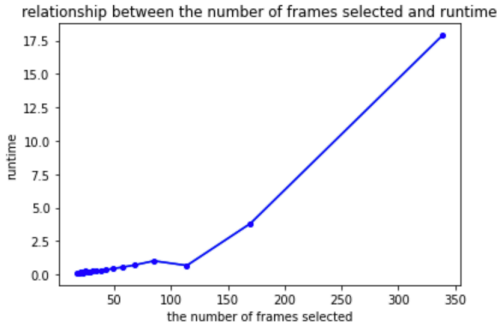


Figure 6: relationship between the number of frames selected and runtime

4.4.2 Selection of initial values of power iteration

We consider the economic singular value decomposition here:

$$A = U\Sigma V^\top = \begin{bmatrix} u_1 & u_2 & \cdots u_r \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix} \begin{bmatrix} v_1 & v_2 & \cdots & v_r \end{bmatrix}^\top = \sum_{i=1}^{r} u_i \sigma_i v_i^\top \tag{8}$$

where $A \in \mathbb{R}^{mn \times s}$, $u_1, \ldots, u_r$ are components ordered by importance that constitute A, $\sigma_1, \ldots, \sigma_r$ are their corresponding energies, and v_1 represents a time series of proportion of u1 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 V is an orthonormal matrix, $\|v_1\|_2 = 1$, we get $\lambda = \sqrt{\frac{1}{s}}$. By setting initial point of power iteration to be $\sqrt{\frac{1}{s}} * \text{ones}(s)$, we can reduce the number of iteration a little bit. In the following figure, we compared the number of iterations of when applying initial value to be $e1$ and $\sqrt{\frac{1}{s}} * \text{ones}(s)$, we find that it converges faster in most cases.

4.5 Comparison with scipy

The following figure is tested on video with size 640×360 and 1280×720, we compared the runtime of scipy and our algorithm:

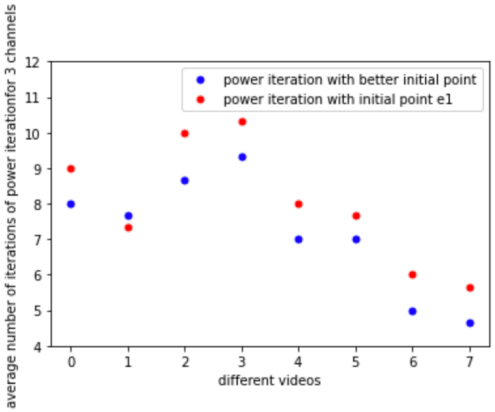
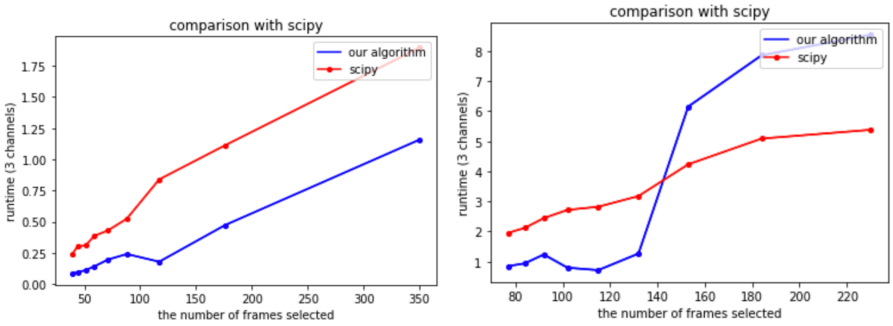


Figure 7: comparison of different initial value strategies



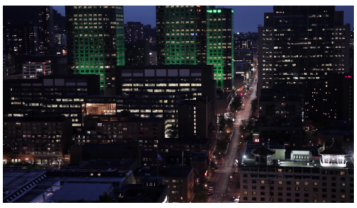
(a) 640x360 (b) 1280x720

As the graph shows, our algorithm is slower than scipy if the matrix A is too large. In our algorithm, power iteration converges very fast. The bottleneck of the algorithm is calculating $A.T \cdot A$. This step is time-consuming and occupies the most time in our algorithm, especially for videos with large sizes. We have tried random SVD, but in vain. We have also tried hermitian dilation, but the resulting matrix B is too large to construct and is to be stored in RAM. The simplest method to avoid this problem while not affecting the algorithm result is to decrease the number of frames selected.

4.6 Result and sample output

This subsection offers some sample output of our algorithm by selecting $\frac{1}{5}$ of the frames. The background image is clear and brilliant, which shows that our algorithm works well and achieves its basic functionality.

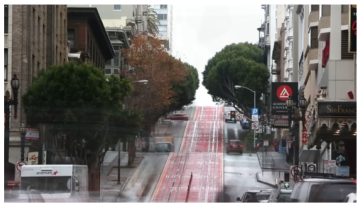
(a) walking



(b) city01

(c) highway

(d) road



(e) sanfrancisco

(f) waterwheel

Figure 9: Sample output background pictures based on input videos.

5 Contribution

Task	**Member** (in order of student ID)
Problem1 & Problem2	黎俊乐, 杨蕙菡, 王修远
Problem3	赵松霖, 林锦睿
Report	All

6 Appendix

6.1 Performance analysis: problem1 phaseIIA

The numbers in matrix recovery are the percentages of entries of the original matrix recovered with SVD produced by the algorithm. The numbers in singular value recovery are percentages of singular values (we take that calculated by scipy.linalg as standard) of the original matrix recovered with SVD produced by the algorithm.

6.1.1 Kernel matrix

Box blurring kernel with different sizes and power 66 are used in this test.

Dimensions	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseII runtime
100	100%	97.05882%	1.68E-08	0.008s	0.011001s
200	100%	97.22222%	2.40E-08	0.076997s	0.048001s
300	100%	97.24771%	2.70E-08	0.216599s	0.102984s
400	100%	96.59864%	4.04E-08	0.629876s	0.196988s
500	100%	97.28261%	3.90E-08	1.10309s	0.299883s
600	100%	97.28507%	3.82E-08	1.806973s	0.432401s
700	100%	97.28682%	3.76E-08	2.772715s	0.625986s
800	100%	97.28814%	3.71E-08	4.297864s	0.818416s
900	100%	96.997%	3.68E-08	5.927234s	1.106071s
1000	100%	97.02703%	3.65E-08	8.222714s	1.415767s

6.1.2 Random matrix

Random matrices with entries in $[0, 1)$ are used in this test.

Dimensions	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseII runtime
20	100%	100%	3.77E-15	0.001002s	0.008998s
40	100%	100%	1.51E-14	0.002001s	0.020999s
60	100%	100%	1.78E-14	0.008s	0.051012s
80	100%	100%	1.42E-14	0.005028s	0.097602s
100	100%	100%	9.03E-14	0.008s	0.145s
120	100%	100%	6.39E-14	0.019001s	0.192s
140	100%	100%	4.26E-14	0.032998s	0.281001s
160	100%	100%	9.59E-14	0.04377s	0.371642s
180	100%	100%	8.53E-14	0.059999s	0.452636s
200	100%	100%	7.11E-14	0.074079s	0.587671s

6.2 Performance analysis: problem1 phaseIIB

The numbers in matrix recovery are the percentages of entries of the original matrix recovered with SVD produced by the algorithm. The numbers in singular value recovery are percentages of singular values (we take that calculated by scipy.linalg as standard) of the original matrix recovered with SVD produced by the algorithm.

6.2.1 Kernel matrix

Box blurring kernel with different sizes and power 66 are used in this test.

Dimensions	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseII runtime
100	100%	100%	6.06E-15	0.012023s	0.036977
200	100%	100%	1.89E-15	0.134022s	0.181669s
300	100%	100%	5.44E-15	0.331684s	0.526514s
400	100%	100%	5.77E-15	0.740913s	1.036176s
500	100%	100%	1.14E-14	1.064225s	0.694181s
600	100%	100%	1.45E-14	1.82602s	1.076133s
700	100%	100%	1.21E-14	2.781548s	1.641877s
800	100%	100%	2.30E-14	4.207093s	2.311061s
900	100%	100%	3.11E-14	5.863215s	3.155976s
1000	100%	100%	3.56E-14	8.024716s	4.23336s

6.2.2 Random matrix

Random matrices with entries in $[0, 1)$ are used in this test.

Dimensions	Matrix recovery	Singular value recovery	Max error of singular value	PhaseI runtime	PhaseII runtime
20	100%	100%	7.11E-15	0.000985s	0.049129s
40	100%	100%	2.58E-14	0.002001s	0.322099s
60	100%	100%	1.69E-13	0.002001s	0.778991s
80	100%	100%	4.09E-14	0.004s	1.245006s
100	100%	100%	9.59E-14	0.008099s	2.35596s
120	100%	100%	5.33E-14	0.019998s	5.828254s
140	100%	100%	2.25E-13	0.032001s	8.892522s
160	100%	100%	5.68E-14	0.046177s	9.858239s
180	100%	100%	1.64E-13	0.06205s	14.56252s
200	100%	100%	1.61E-13	0.096014s	35.14804s