STA 663 Final Project

Aihua Li, Yuxuan Chen

04/27/2021

Abstract

In this report, we try to realize biclustering by utilizing Sparse Singular Value Decomposition (SSVD). SSVD is a tool for biclustering by seeking the low-rank matrix approximation with sparsed left and right sigular vectors of original matrix. Note that a sparsity-inducing penaly is incorporated to get sparsity. We will decribe this algorithm in detail and try to optimize the process by implementing cython. After optimazation, applications of this algorithm on a simulated dataset and a real tumor dataset. Moreoever, comparative analysis with simulated dataset are also conducted to compare the performance of SSVD, SVD and SPCA. The full repository is available on https://github.com/YuxuanMonta/STA663-final-project-AY.

Key words: SSVD, Sparsity, Simulation, BIC, Penalty, Approximation

1 Introduction to biclustering via SSVD

The research we focus on is *Biclustering via Sparse Singular Value Decomposition* written by Mihee Lee, Haipeng Shen, Jianhua Z. Huang, and J.S. Marron from University of North Carolina at Chapel Hill [1].

In the paper, as what is shown in the title, a new exploratory analysis tool for biclustering is introduced. This tool is also designed to seek the assocaition between the colum and row of data matrices with high dimension (singularity). A checkerboard structured matrix approcimation to the original data matrices will be developed through this tool, achieved by sparsing both the left and right sigular vectors.

There are increasing number of dataset with high dimension in real life. In most case, those data with relatively higher dimension and smaller sample size is defined as "high-dimension low sample size (HDLSS)" in the paper.This kind of data could be a problem when being used to do classical statistical data analysis. Bicluserting method,a collection of unsupervised learning tools, is developed to solve this kind of problem. Under this background, SSVD is proposed in this paper as a new tool of bicluserting method. Besides SSVD, other algorithms like SVD and Sparse PCA (SPCA) can be also helpful when encountering the same problem. One typical application of SSVD algorithm is the microarray gene expression analysis, a limited number of individuals with numerous genes. SSVD can help to identify the groups of genes for different kind of diseases. Moreover, in text categorization, SSVD can be used to seek the pattern of words for different documents.

One advantage of SSVD compared to SVD is that SSVD can be useful to find sparsed structure pf HDLLSS data. Moreover, while Sparse PCA "only imposes sparse structure on one direction", SSVD can detect block structures in data matrices. However, because iteration method is needed to realize this algorithm, the computation cost can be expensive.

In our search, SSVD algorithm will be applied to get the approximation of sparse matrices of both simulated data set and real data set. Addtionally, we will also compare SSVD algorithm with SVD algorithm and SPCA algorithm.

2 Algorithm

In order to realise SSVD algorithm, a low-rank matrix approximation matrix (rank-k) to the original data matrix should be first computated. This rank-k matrix here is the combination of top k ranks SSVD layer. Then in the iteration process, each step of iteration can generate a SSVD layer from the residual matrix of previous layer, based on a penalized sum-of-squares criterion where the degree of sparsity in each iteration dependes on the BIC. This iteration process will continue until convergence. Afterward, the computated left and right singular vector of the SSVD can help to group the row and column space. Then an approximation matrix can be generated by sorting the singular vector in each group.

In the following part, we only focus on the rank-1 matrix approximation via SSVD.

Details of SSVD algorithm

- Step 1: Initialization

  Apply the standard SVD to X and get the initial SVD decomposition u_0, s_0, v_0. Note that $X = u_0 s_0 v_0^T$.

- Step2: Update

  1. update v:

     a: For each λ_v, get $BIC(\lambda_v) = \frac{||Y-\hat{Y}||^2}{nd \cdot \hat{\sigma_v}^2} + \frac{log(nd)}{nd}\hat{df}(\lambda_v)$. Note $df(\lambda_v)$ is the degree of sparsity of v with λ_v as the penalty parameter, and $\hat{\sigma_v}^2$ is the OLS estimate of the error variance from $||X - u\tilde{v}||_F^2 + \lambda_v P_2(\tilde{v})$, where $\tilde{v} = sv$.
     b: Get the λ_v that minimizes $BIC(\lambda_v)$ (penalty).
     c: Set $\tilde{v}_j = sign(X^T u_0)_j(|(X^T u_0)_j| - \lambda_v w_{2,j}/2)_+$, $\{j = 1, ..., d\}$, where $w_2 = |\hat{\tilde{v}}|^{-\gamma_2}$ and $\hat{\tilde{v}} = X^T u$
     d: Set $v_{new} = \frac{\tilde{v}}{s_v}$, where $\tilde{v} = (\tilde{v}_1, ..., \tilde{v}_d)^T$ and $s_v = ||\tilde{v}||$

  2. update u:

     a: For each λ_u, get $BIC(\lambda_u) = \frac{||Y-\hat{Y}||^2}{nd \cdot \hat{\sigma_u^2}} + \frac{log(nd)}{nd}\hat{df}(\lambda_u)$. Note $df(\lambda_u)$ is the degree of sparsity of u with λ_u as the penalty parameter, and $\hat{\sigma_u}^2$ is the OLS estimate of the error variance from $||X - \tilde{u}v||_F^2 + \lambda_u P_1(\tilde{u})$, where $\tilde{u} = su$.
     b: Get the λ_u that minimizes $BIC(\lambda_u)$ (penalty).
     c: Set $\tilde{u}_i = sign(X^T v_0)_i(|(X^T v_0)_i| - \lambda_u w_{1,i}/2)_+$, $\{i = 1, ..., n\}$, where $w_1 = |\hat{\tilde{u}}|^{-\gamma_1}$ and $\hat{\tilde{u}} = X^T v$
     d: Set $u_{new} = \frac{\tilde{u}}{s_u}$, where $\tilde{u} = (\tilde{u}_1, ..., \tilde{u}_n)^T$ and $s_u = ||\tilde{u}||$

  3. Iteration:

     Set $u_0 = u_{new}$ and $v_0 = v_{new}$ and go back to Step 2 (1) and Step 2 (2) until both the distances between $u_0 and u_{new}$ as well as $v_0 and v_{new}$ are smaller than tolerance.

- Step 3: Return $u = u_{new}$, $v = v_{new}$ and $s = u_{new}^T X v_{new}$ at convergence.

3 Coding and optimization

Starting from a most basic version of Python plain code, we further make 4-step improvement/modification. As shown in table 1, we have compared the computing time of the three key functions `update_uv()`, `SSVD_layer()`, and `SSVD()` in all the 5 versions. In this section, we will introduce the key idea of the optimization procedure. All codes are included in the appendix.

Version 1: The most basic plain codes

In the baseline version, we implement the SSVD algorithm according to the algorithm description. Most of the Python codes can be directly related to the mathematics in the paper, so that the codes is highly understandable.

Table 1: Average computing time comparisons in 10 runs 10 loops each (ms)

	update_uv()	**SSVD_layer()**	**SSVD()**	**Description**
Version1	351.8	1448.64	1268.01	basic
Version2	8.03	36.91	55.84	by linear algebra
Version3	3.84	15.52	15.12	by broadcasting
Version4	4.49	20.17	20.34	modify broadcasting
Version5	3.97	8.56	8.47	by Cython

Note that starting from this baseline version, there are three key functions:

- `update_uv()`, to update the vector u and v given the current values.
- `SSVD_layer()`, to compute the SSVD decomposition at the current layer.
- `SSVD()`, to compute the SSVD decomposition at all desired layers.

For the detailed full codes, see appendix A.

Version 2: Improve version 1 by linear algebra tricks

The baseline version, although intuitive, is terribly slow. After profiling the codes, a noticeable result is that the "ols calculation", (i.e., computing quantities like $\hat{\sigma}^2$), which involves the Kronecker product, dramatically slows down the computation, because working with the ols stuff indeed increases the matrix dimension from n or d into $n \times d$. Therefore, the computation of the ols are replaced in this version. As can be seen from table 1, this modication greatly improves the computing efficiency.

For the detailed full codes, see appendix B.

Version 3: Improve version 2 by broadcasting

Another thing that slows down the computation is the searching for the optimal λ from a given grid. In previous versions, it is done by loop with `map()`, while in this version, broadcasting is applied to replace all the loops. As a resut, with the help of broadcasting, we further speed up the computation by half, as shown in table 1.

For the detailed full codes, see appendix C.

Version 4: Modify the broadcasting in version 3

Although broadcasting is a powerful tool to optimize the codes, there is a potential problem of its application in our context. In fact, one step that apply the broadcasting is to compute the outer product of vectors; then, conceivably, under the broadcasting, "outer product * lambda grid" can result in a quite huge matrix if the original data is of high dimension. Concerning this potential problem, in this versino, we consider appropriately combine the broadcasting and the map loop. It turns out that the performance of this version is slightly worse than version 3 for the simulation data of moderate dimension, as shown in table 1; but overall, the performance is somewhat comparable.

For the detailed full codes, see appendix D.

Version 5: Cythonize version 3

Based on the previous results, we decided to work with version 3, and further improve it by Cython. It will be too tedious to explain all the details about how we cythonize the codes, but we'd like to mention some key points:

1. First of all, we found that Cython *doesn't always* bring improvement to the codes, especially compared with the broadcasting in Numpy.

   In fact, we have tried to "fully" cythonize all the codes with as much parallelization as possible, and it turns out to be totally comparable, (or sometimes even slightly worse,) to the non-cythonized version. A possible reason is that there can be high overhead of parallelization, which sometimes exceed its benefits, especially compared to the powerful broadcasting in Numpy (which is already written in C).

   Therefore, we convert the codes into Cython line by line, and see if each-line modification can make improvements. Only those that can actually speed up the codes will be kept.

2. Another noticeable point is that in the plain python version, everytime we update the vectors u and v, we require new spaces in the computer to store the updated vectors. Then, Cython allows us to always use the same storage for the "old" and "new" vectors. We have found that this is a step greatly improving the efficiency.

   As shown in table 1, in fact, the performance of the Cython version is comparable to the version 3 in the function `update_uv()` (again, demonstrating the usefulness of basic Numpy), but Cython significantly improves the performance of the functions `SSVD_layer()` and `SSVD` (by saving storage).

For the detailed full codes, see appendix E.

4 Simulation

In this section, we realize the simulation in the original paper to demonstrate the clustering performance of our codes.

More specifically, we consider rank-1 true signal matrix $X^* = suv^T \in \mathbb{R}^{100\times 50}$. Let $s = 50$, and

$$\begin{aligned}\tilde{u} =& [10, 9, 8, 7, 6, 5, 4, 3, r(2, 17), r(0, 75)]^T, \quad u = \tilde{u}/||\tilde{u}||, \\ \tilde{v} =& [10, -10, 8, -8, 5, -5, r(3, 5), r(-3, 5), r(0, 34)]^T, \quad v = \tilde{v}/||\tilde{v}||,\end{aligned} \tag{1}$$

where $r(a, b)$ denotes a vector of length b, whose entries are all a. Then, the data matrix X is negerated by X^* plus a noise matrix ϵ, and here we consider standard normal noise.

Figure 1 shows the checkerboard plot (aka heatmap, SSVD first layer plot) of the true data decomposition, and figure 2 and 3 show the outputs of the 5 versions of codes.

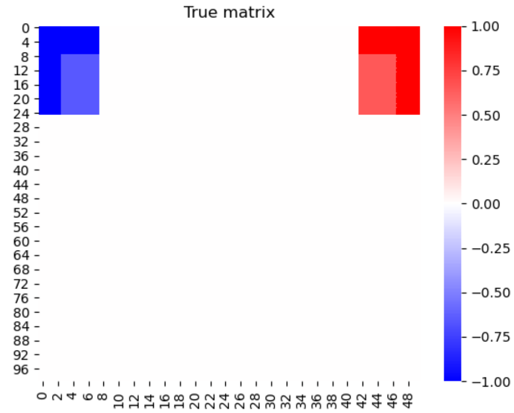

Figure 1: SSVD first layer plot of the true decomposition

As a result, only about first 25 entries are non-zero, which accords to our true normalized vector u with length 100 and last 75 data are all 0s.

Moreover, the original pattern can also be observed from the layer 1 approximation plot above, indicating SSVD reaches a good approximation of the true data by getting rid of the influence of noises.

5 Real data application

In this part, the real data sets we use is a gene expression cancer RNA-Seq data set. It is part of the RNA-Seq (HiSeq) PANCAN data set. This is a random extraction of gene expressions of patients having different types of tumor: BRCA, KIRC, COAD, LUAD and PRAD. (Data source: http://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq, see[2])

This dataset provides 801 subjects and 20531 genes. You may find the data in 'data.csv' and the 5 groups of gene in 'labels.csv'.

In the real data set application process, we may use only 5000 genes of all 801 sujects instead of the whole dataset given the consideration of computation cost. The application are realized by the version 5 codes (i.e., Cython)

The subject grouping can be seen above from the three scatterplots among the first three sparse left singular vectors. The first vector and the second vector suggest that tumor KIRC are identified. The second vector and the third vector reach the same results. The first vector and the third vector cannot separate the data well.

Therefore the three vectors may not provide good separation of the five tumor types. Our inital guess is that this could be because the data itself and out guess is proved in next step.

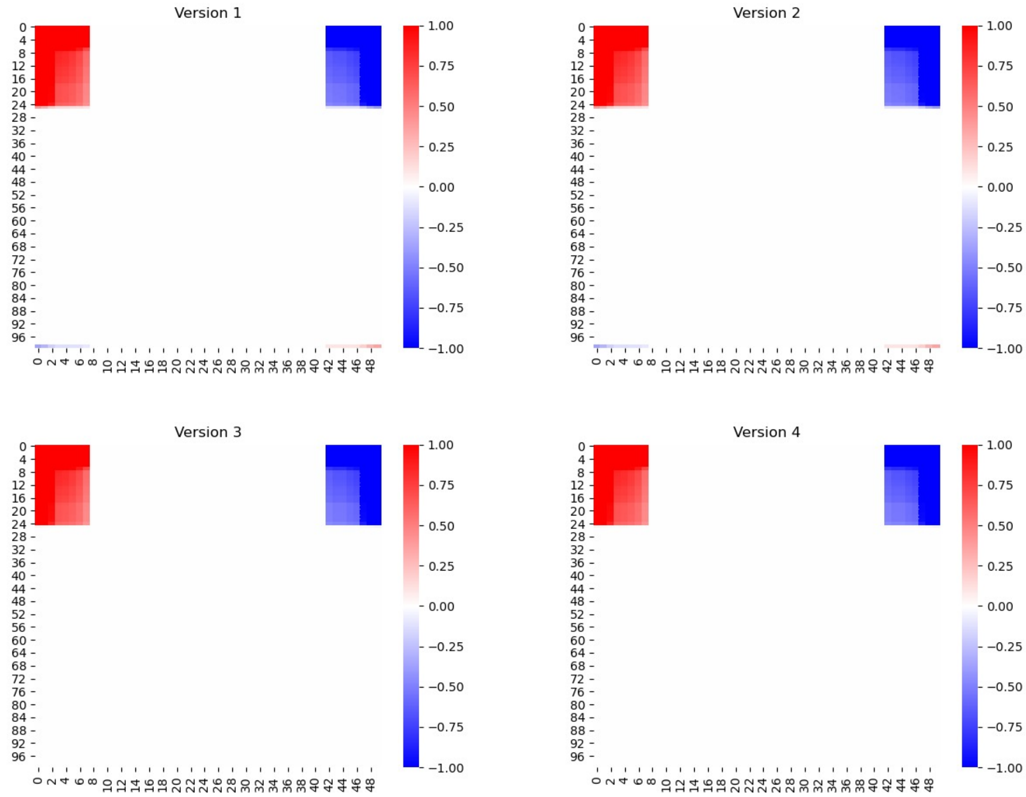

Figure 2: SSVD first layer plot of the outputs of version 1 to version 4

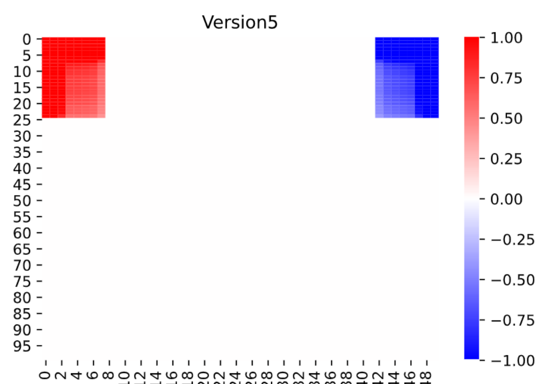

Figure 3: SSVD first layer plot of the outputs of version 5

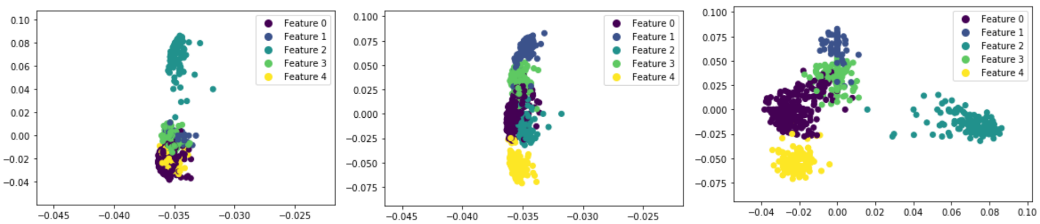

Figure 4: Tumor data: scatter plots of the first three entries in the SSVD decomposition. Left: u1 vs u2, middle: u1 vs u3, right: u2 vs u3

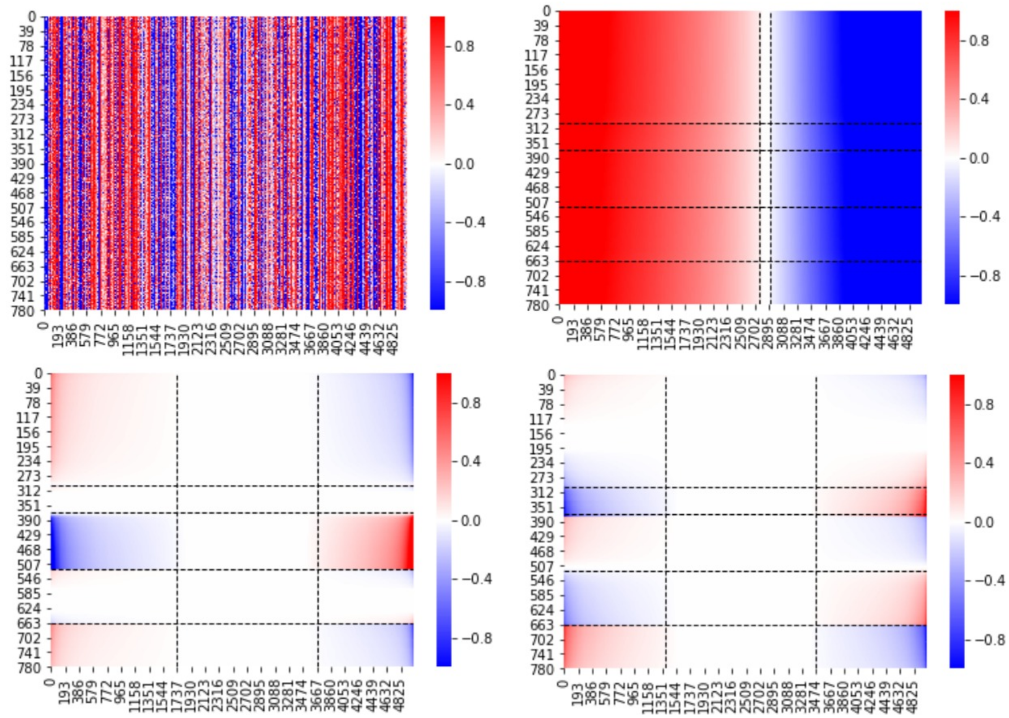

Figure 5: SSVD layer plot of the tumor data. Left top: heatmap of original dataset, right top: first layer of SSVD, left bottom: second layer of SSVD, right bottom: third layer of SSVD

In figure 5, the first 3 SSVD layers are clearly plotted. Based on the layer 1 plot, we can see a similar pattern for each tumor group. Thus, we will interpret the plot by simply ignoring the group. In this way, the figure can suggest that genes ordered from about 0 to round 2750 positively expressed for all tumors, whlie genes odered from 2900 negatively expressed for all tumors. Layer 2 suggests a contrast betweem 'BRCA'/'PRAD' and 'KIRC', also zeros out the other two types. Layer 3 suggests a approximately contrast betweem 'BRCA'/'PRAD'/'KIRC' and 'COAD'/'LUAD'.

By combining with the heatmap of original polt, it suggest that the SSVD algorithm works well for identifying the approximation of original dataset.

6 Comparative analysis

In this comparative analysis part, we will compare SSVD with SVD and Sparse PCA algorithm. Note that SPCA here uses 2 as weight parameter.

Rank 1 decomposition of this three methods will be compared. The true u and v are defined by the code in the next cell and normal distributed noises are incorporated to get test data.

For accuracy, we will compare four indicators.

1. Average number of zeros in u and v
2. Average number of correctly identified 0s in u and v
3. Average number of correctly identified non-0s in u and v
4. Rate of correctly identified all the positions of 0s AND non-0s (Correct classification rate) in u and v

Figure 6, 7 summarize the comparison results.

	Avg. # of 0s	Avg. # of correctly identified 0s	Avg. # of correctly identified non0s	Rate of correctly identified all 0s
u_ssvd	74.91	74.91	25.0	0.91
v_ssvd	33.88	33.88	16.0	0.88
u_svd	0.00	0.00	25.0	0.00
v_svd	0.00	0.00	16.0	0.00
u_spca	75.00	75.00	25.0	1.00
v_spca	34.00	34.00	16.0	1.00

Figure 6: Accuracy comparisons between algorithms

	Time(sec)
Time of SPCA	11.873480
Time of SSVD	3.345996
Time of SVD	0.603197

Figure 7: Running time comparisons between algorithms

Accuracy

The correct classification rate of SPCA indicates that SPCA successfully identifies all the positions of 0s and non-0s in both u and v. The correct classification rate of SSVD is also accepteable in both u and v. Moreover, the average number of 0s, average number of correctly identified 0s and average number of correctly identified non0s of SPCA and SSVD are all excellent for both u and v

However, given the nature of SVD algorithm, the correct classsification rate of SVD is 0 for both u and v, indicating there is no 0s in the approximations of all layers by also refering the 0s in column 1 and colmun 2 for both u and v. Nevertheless, the average number of correctly identified non-0s of all methods are perfect.

Running Time

As shown in the last table above, SPCA has the longest running time all SVD has least running time.

7 Conclusion and discussion

After going through this project, SSVD is clearly an useful tool to develop a low rank (rank = 1 for most of time in this project) martix approximation for those HDLSS datasets. It has relatively high accuracy and acceptable efficiency. For the usage of SSVD, this project mainly focuses on microarray gene expression analysis, and a good approximation is obtained.

There are some limitations of our product. Even though Cython has greatly improved the computation efficiency, a full C version is still worth trying to see if the computation can be fully optimized. Besides, in the tumor data application, for illustration purpose, it is sufficient to consider up to 5000 genes. Although, as we discussed before, the SSVD by the current 5000 genes has a powerful classification ability, for future study, we may expect to use the full data set to have a more detailed investigation. Last but not least, maybe a new sparsity-inducing penalty instead of the adaptive lasso penalty (weight parameter is 2 in this project) can be introduced.

8 References

[1] Lee, M., Shen, H., Huang, J., and Marron, J. (2010). Biclustering via sparse singular value decomposition. *Biometrics* **66**, 1087-1095.

[2] UCI machine Learning Repository: Gene EXPRESSION CANCER rna-seq data set. (n.d.). Retrieved April 28, 2021, from http://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq

Appendix A Version 1

Basic plain codes

```
## This the most basic realization of the SSVD algorithm.
## Codes directly correspond to the paper context.

import numpy as np
import scipy.linalg as la


def get_tilde(lam, tilde_hat, w):
    """Get the tilde vector given the lambda, tilde_hat_vector, and w."""
    part1 = np.sign(tilde_hat)
    part2 = np.abs(tilde_hat)-lam*w/2
    part2[part2<np.zeros(part2.shape)] = 0
    tilde = np.multiply(part1, part2)
    return tilde

def get_BIC(lam, w, tilde_hat, fixed_vec, response, sigma2_hat, nd):
    """ Compute BIC given the lambda.
    lam: scalar
    For v:
        w = w2: (d, 1)
        tilde_hat = v_tilde_hat: (d, 1)
        fixed_vec = u_old: (n, 1)
        response = Y: (nd, 1)
        sigma2_hat = sigma2_hat_v: scalar
        nd: scalar, =n*d
    For u:
        w = w1: (n, 1)
        tilde_hat = u_tilde_hat: (n, 1)
        fixed_vec = v_new: (d, 1)
        response = Z: (nd, 1)
        sigma2_hat = sigma2_hat_u: scalar
        nd: scalar, =n*d
    Return: BIC, scalar
    """
    df = np.sum(np.abs(tilde_hat) > lam*w/2)
    tilde = get_tilde(lam, tilde_hat, w)  # get the tilde vector udner the current lambda
    response_hat = np.kron(np.eye(tilde.shape[0]), fixed_vec) @ tilde
    BIC = np.linalg.norm(response-response_hat)**2/(nd*sigma2_hat)+np.log(nd)/nd*df
    return BIC
def select_lam(lam_grid, w, tilde_hat, fixed_vec, response, sigma2_hat, nd):
    """ Select lambda given a lambda grid based on BIC.
    lam_grid: (S,)
    For v:
        w = w2: (d, 1)
        tilde_hat = v_tilde_hat: (d, 1)
        fixed_vec = u_old: (n, 1)
        response = Y: (nd, 1)
        sigma2_hat = sigma2_hat_v: scalar
        nd: scalar, =n*d
    For u:
        w = w1: (n, 1)
        tilde_hat = u_tilde_hat: (n, 1)
        fixed_vec = v_new: (d, 1)
        response = Z: (nd, 1)
        sigma2_hat = sigma2_hat_u: scalar
        nd: scalar, =n*d
    Return: scalar, the optimal lambda
    """
    BICs = list(map(lambda lam: get_BIC(lam, w, tilde_hat, fixed_vec, response, sigma2_hat, nd),
     lam_grid))
    return lam_grid[np.argmin(BICs)]
def update_uv(u_old, v_old, X, Y, Z, gamma1, gamma2, lam_grid):
    """Update u and v once given the current u and v.
```

```python
    Input:
        u_old: (n, 1)
        v_old: (d, 1)
        X: (n, d)
        Y: (nd, 1)
        Z: (nd, 1)
        gamma1, gamma2: scalar, tuning parameter
        lam_grid: ndarray, a grid of lambdas to be selected
    Return:
        u_new: (n, 1)
        v_new: (d, 1)
        lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
    """
    n, d = X.shape
    nd = n*d

    ## Update v using current u

    # ols for v, use current u
    v_tilde_hat = X.T @ u_old  # (d, 1), fixed ols estimate
    Y_hat = (np.kron(np.eye(d), u_old) @ v_tilde_hat).reshape((-1,1))  # (nd, 1), ols estimate
    sigma2_hat_v = ((Y-Y_hat).T @ (Y-Y_hat) / (nd-d))[0][0]  # scalar, fixed ols estimate

    # select lambda_v
    w2 = np.abs(v_tilde_hat)**(-gamma2)  # (d, 1)
    lambda_v = select_lam(lam_grid, w2, v_tilde_hat, u_old, Y, sigma2_hat_v, nd)

    # update v
    v_tilde = get_tilde(lambda_v, v_tilde_hat, w2)  # (d, 1)

    if np.all(v_tilde == 0):  # full shrinkage at v
        v_new = np.zeros(v_old.shape)
        u_new = np.zeros(u_old.shape)
        lambda_u = 0
    else:
        v_new = v_tilde / np.linalg.norm(v_tilde)

        ## Update u using current v

        # ols for u, use current v
        u_tilde_hat = X @ v_new  # (n, 1), fixed ols estimate
        Z_hat = np.kron(np.eye(n), v_new) @ u_tilde_hat  # (nd, 1), ols estimate
        sigma2_hat_u = ((Z-Z_hat).T @ (Z-Z_hat) / (nd-n))[0][0]  # scalar, fixed ols estimate

        # select lambda_u
        w1 = np.abs(u_tilde_hat)**(-gamma1)
        lambda_u = select_lam(lam_grid, w1, u_tilde_hat, v_new, Z, sigma2_hat_u, nd)

        # update u
        u_tilde = get_tilde(lambda_u, u_tilde_hat, w1)  # (n, 1)
        u_new = u_tilde / np.linalg.norm(u_tilde) if np.linalg.norm(u_tilde) != 0 else u_tilde

    return u_new, v_new, lambda_u, lambda_v


def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters grid.
    Input:
        X: (n, d), can be the original data matrix or the residual matrix
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iter, u, v, s, lambda_u, lambda_v
        n_iter: number of iterations
        u: (n, 1), the final u at convergence
        v: (d, 1), the final v at convergence
        s: scalar, the singular value at convergence
```

```
        lambda_u, lambda_v: the optimal tuning parameter at convergence
    """
    # SVD
    U, _, VT = la.svd(X)
    # prepare vector Y and Z, which are fixed after given X
    Y = X.T.reshape((-1,1))  # (nd, 1)
    Z = X.reshape((-1,1))  # (nd, 1)

    # initial value
    u_old = U[:,0][:,None]
    v_old = VT[0][:,None]

    for i in range(max_iter):
        u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, Y, Z, gamma1, gamma2, lam_grid)
        if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol:  # achieve the tolerance
            break
        if np.all(u_new == 0) or np.all(u_new == 0):  # full shrinkage (i.e., all zeros in the vector)
            print("Warning: Full shrinkage has been achieved. Iterations stops. No further decomposition. The desired number of layers may not be achieved. ")
            break
        u_old, v_old = u_new, v_new
    n_iter = i+1  # number of iterations
    u, v = u_new, v_new  # the final u and v at convergence
    s = (u_new.T @ X @ v_new)[0][0]
    if n_iter == max_iter:
        print("Warning: The maximum iteration has been achieved. Please consider increasing `max_iter`.")
    return n_iter, u, v, s, lambda_u, lambda_v

def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the SSVD given the data matrix X and the desired number of SSVD layers.
    Input:
        X: (n, d), the original data matrix
        num_layer: desired number of SSVD layers
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iters, us, vs, ss, lambda_us, lambda_vs
        n_iters: (num_layer,), number of iterations for each layer
        us: (n, num_layer), the final u at convergence for each layer
        vs: (d, num_layer), the final v at convergence for each layer
        ss: (num_layer,), the singular value at convergence for each layer
        lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each layer
    """
    n, d = X.shape
    n_iters = np.zeros(num_layer, dtype = int)
    ss = np.zeros(num_layer)
    lambda_us = np.zeros(num_layer)
    lambda_vs = np.zeros(num_layer)
    us = np.zeros((n, num_layer))
    vs = np.zeros((d, num_layer))
    # initial value
    u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
    for i in range(num_layer):
        resi_mat = resi_mat - s*u@v.T
        n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2, max_iter, tol)
        n_iters[i] = n_iter
        ss[i] = s
        lambda_us[i] = lambda_u
        lambda_vs[i] = lambda_v
        us[:,i] = u[:,0]
        vs[:,i] = v[:,0]
        if np.all(u == 0) or np.all(v == 0):  # full shrinkage (i.e., all zeros in the vector)
```

```
            break
    return n_iters, us, vs, ss, lambda_us, lambda_vs
```

Appendix B   Version 2

First-step improvement by linear algebra

```
## This is the first-step optimization, based on SSVDversion1.
## Based on linear algebra,
## computation about about Y, Z, and the Kronecker product are replaced.

import numpy as np
import scipy.linalg as la

def get_tilde(lam, tilde_hat, w):
    """Get the tilde vector given the lambda, tilde_hat_vector, and w."""
    part1 = np.sign(tilde_hat)
    part2 = np.abs(tilde_hat)-lam*w/2
    part2[part2<np.zeros(part2.shape)] = 0
    tilde = np.multiply(part1, part2)
    return tilde
def get_BIC(lam, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name):
    """ Compute BIC given the lambda.
    lam: scalar
    For v:
        w = w2: (d, 1)
        tilde_hat = v_tilde_hat: (d, 1)
        fixed_vec = u_old: (n, 1)
        response = Y: (nd, 1)
        sigma2_hat = sigma2_hat_v: scalar
        nd: scalar, =n*d
    For u:
        w = w1: (n, 1)
        tilde_hat = u_tilde_hat: (n, 1)
        fixed_vec = v_new: (d, 1)
        response = Z: (nd, 1)
        sigma2_hat = sigma2_hat_u: scalar
        nd: scalar, =n*d
    Return: BIC, scalar
    """
    df = np.sum(np.abs(tilde_hat) > lam*w/2)
    tilde = get_tilde(lam, tilde_hat, w)
    SSE = np.sum((X - np.outer(fixed_vec, tilde))**2) if fixed_name == "u" else\
        np.sum((X - np.outer(tilde, fixed_vec))**2)
    BIC = SSE/(nd*sigma2_hat)+np.log(nd)/nd*df
    return BIC
def select_lam(lam_grid, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name):
    """ Select lambda given a lambda grid based on BIC.
    lam_grid: (S,)
    For v:
        w = w2: (d, 1)
        tilde_hat = v_tilde_hat: (d, 1)
        fixed_vec = u_old: (n, 1)
        response = Y: (nd, 1)
        sigma2_hat = sigma2_hat_v: scalar
        nd: scalar, =n*d
    For u:
        w = w1: (n, 1)
        tilde_hat = u_tilde_hat: (n, 1)
        fixed_vec = v_new: (d, 1)
        response = Z: (nd, 1)
        sigma2_hat = sigma2_hat_u: scalar
        nd: scalar, =n*d
```

```python
    Return: scalar, the optimal lambda
    """
    BICs = list(map(lambda lam: get_BIC(lam, w, tilde_hat, fixed_vec, sigma2_hat, nd, X, fixed_name), lam_grid))
    return lam_grid[np.argmin(BICs)]

def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
    """Update u and v once given the current u and v.
    Input:
        u_old: (n, 1)
        v_old: (d, 1)
        X: (n, d)
        Y: (nd, 1)
        Z: (nd, 1)
        gamma1, gamma2: scalar, tuning parameter
        lam_grid: ndarray, a grid of lambdas to be selected
    Return:
        u_new: (n, 1)
        v_new: (d, 1)
        lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
    """
    n, d = X.shape
    nd = n*d

    ## Update v using current u

    # ols for v, use current u
    v_tilde_hat = X.T @ u_old  # (d, 1), fixed ols estimate
    SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2)  # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
    sigma2_hat_v = SSE_v / (nd-d)  # scalar, fixed ols estimate

    # select lambda_v
    w2 = np.abs(v_tilde_hat)**(-gamma2)  # (d, 1)
    lambda_v = select_lam(lam_grid, w2, v_tilde_hat, u_old, sigma2_hat_v, nd, X, "u")

    # update v
    v_tilde = get_tilde(lambda_v, v_tilde_hat, w2)  # (d, 1)

    if np.all(v_tilde == 0):  # full shrinkage at v
        v_new = np.zeros(v_old.shape)
        u_new = np.zeros(u_old.shape)
        lambda_u = 0
    else:
        v_new = v_tilde / np.linalg.norm(v_tilde)

        ## Update u using current v

        # ols for u, use current v
        u_tilde_hat = X @ v_new  # (n, 1), fixed ols estimate
        SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)
        sigma2_hat_u = SSE_u / (nd-n)  # scalar, fixed ols estimate

        # select lambda_u
        w1 = np.abs(u_tilde_hat)**(-gamma1)
        lambda_u = select_lam(lam_grid, w1, u_tilde_hat, v_new, sigma2_hat_u, nd, X, "v")

        # update u
        u_tilde = get_tilde(lambda_u, u_tilde_hat, w1)  # (n, 1)
        u_new = u_tilde / np.linalg.norm(u_tilde) if np.linalg.norm(u_tilde) != 0 else u_tilde

    return u_new, v_new, lambda_u, lambda_v


def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters grid.
    Input:
        X: (n, d), can be the original data matrix or the residual matrix
```

```
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iter, u, v, s, lambda_u, lambda_v
        n_iter: number of iterations
        u: (n, 1), the final u at convergence
        v: (d, 1), the final v at convergence
        s: scalar, the singular value at convergence
        lambda_u, lambda_v: the optimal tuning parameter at convergence
    """
    # SVD
    U, _, VT = la.svd(X)

    # initial value
    u_old = U[:,0][:,None]
    v_old = VT[0][:,None]

    for i in range(max_iter):
        u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
        if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol:  # achieve the tolerance
            break
        if np.all(u_new == 0) or np.all(u_new == 0):  # full shrinkage (i.e., all zeros in the vector)
            print("Warning: Full shrinkage has been achieved. Iterations stops. No further decomposition. The desired number of layers may not be achieved. ")
            break
        u_old, v_old = u_new, v_new
    n_iter = i+1  # number of iterations
    u, v = u_new, v_new  # the final u and v at convergence
    s = (u_new.T @ X @ v_new)[0][0]
    if n_iter == max_iter:
        print("Warning: The maximum iteration has been achieved. Please consider increasing `max_iter`.")
    return n_iter, u, v, s, lambda_u, lambda_v

def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the SSVD given the data matrix X and the desired number of SSVD layers.
    Input:
        X: (n, d), the original data matrix
        num_layer: desired number of SSVD layers
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iters, us, vs, ss, lambda_us, lambda_vs
        n_iters: (num_layer,), number of iterations for each layer
        us: (n, num_layer), the final u at convergence for each layer
        vs: (d, num_layer), the final v at convergence for each layer
        ss: (num_layer,), the singular value at convergence for each layer
        lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each layer
    """
    n, d = X.shape
    n_iters = np.zeros(num_layer, dtype = int)
    ss = np.zeros(num_layer)
    lambda_us = np.zeros(num_layer)
    lambda_vs = np.zeros(num_layer)
    us = np.zeros((n, num_layer))
    vs = np.zeros((d, num_layer))
    # initial value
    u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
    for i in range(num_layer):
        resi_mat = resi_mat - s*u@v.T
        n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2, max_iter, tol)
        n_iters[i] = n_iter
        ss[i] = s
```

```python
        lambda_us[i] = lambda_u
        lambda_vs[i] = lambda_v
        us[:,i] = u[:,0]
        vs[:,i] = v[:,0]
        if np.all(u == 0) or np.all(v == 0):  # full shrinkage (i.e., all zeros in the vector)
            break
    return n_iters, us, vs, ss, lambda_us, lambda_vs
```

Appendix C   Version 3

Second-step improvement by broadcasting

```python
## This is the second-step optimation, based on SSVDversion2.
## Broadcasting are applied when selecting optimal lambda from a given grid.
## Function get_tilde(), get_BIC(), and select_lam() are no longer necessary.

import numpy as np
import scipy.linalg as la

def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
    """Update u and v once given the current u and v.
    Input:
        u_old: (n, 1)
        v_old: (d, 1)
        X: (n, d)
        Y: (nd, 1)
        Z: (nd, 1)
        gamma1, gamma2: scalar, tuning parameter
        lam_grid: ndarray, a grid of lambdas to be selected
    Return:
        u_new: (n, 1)
        v_new: (d, 1)
        lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
    """
    n, d = X.shape
    nd = n*d
    S = len(lam_grid)

    # initialize
    v_new = np.zeros(v_old.shape)
    u_new = np.zeros(u_old.shape)
    lambda_u = lambda_v = 0

    ## Update v using current u

    # ols for v, use current u
    v_tilde_hat = X.T @ u_old  # (d, 1), fixed ols estimate
    SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2)  # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
    sigma2_hat_v = SSE_v / (nd-d)  # scalar, fixed ols estimate

    # select lambda_v
    w2 = np.abs(v_tilde_hat)**(-gamma2)  # (d, 1)
    dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0)  # (S,), df for each lambda
    part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
    part1 = np.sign(v_tilde_hat)
    v_tilde_br = part1 * part2  # (d, S), each column is v_tilde under each lambda
    outer_prods = np.outer(u_old, v_tilde_br.T)  # (n, d*S), each chunk of size (n, d) is the outer product mat under each lambda
    outer_prods = np.array(np.hsplit(outer_prods, S))  # (S, n, d)
    SSEs_v = ((X-outer_prods)**2).sum(axis = (1,2))  # (S,), ||X-uv_tilde^T||_F^2=||Y-Y_hat||^2 for each lambda
    BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v  # (S,), BIC for each lambda
```

```
    lambda_v = lam_grid[np.argmin(BICs_v)]

    # update v
    part1 = np.sign(v_tilde_hat)
    part2 = np.abs(v_tilde_hat)-lambda_v*w2/2; part2[part2<0] = 0
    if not np.all(part2 == 0):  # not full shrinkage at v
        v_tilde = np.multiply(part1, part2)  # (d, 1)
        v_new = v_tilde / np.linalg.norm(v_tilde)

        ## Update u using current v

        # ols for u, use current v
        u_tilde_hat = X @ v_new  # (n, 1), fixed ols estimate
        SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)
        sigma2_hat_u = SSE_u / (nd-n)  # scalar, fixed ols estimate

        # select lambda_u
        w1 = np.abs(u_tilde_hat)**(-gamma1)  # (n, 1)
        dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0)  # (S,), df for each lambda
        part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
        part1 = np.sign(u_tilde_hat)
        u_tilde_br = part1 * part2  # (n, S), each column is u_tilde under each lambda
        outer_prods = np.outer(u_tilde_br.T, v_new)  # (n*S, d), each chunk of size (n, d) is the outer product mat under each lambda
        outer_prods = np.array(np.vsplit(outer_prods, S))  # (S, n, d)
        SSEs_u = ((X-outer_prods)**2).sum(axis = (1,2))  # (S,), ||X-u_tildev^T||_F^2=||Z-Z_hat||^2 for each lambda
        BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u  # (S,), BIC for each lambda
        lambda_u = lam_grid[np.argmin(BICs_u)]

        # update u
        part1 = np.sign(u_tilde_hat)
        part2 = np.abs(u_tilde_hat)-lambda_u*w1/2; part2[part2<0] = 0
        if not np.all(part2 == 0):  # not full shrinkage at u
            u_tilde = np.multiply(part1, part2)  # (n, 1)
            u_new = u_tilde / np.linalg.norm(u_tilde)

    return u_new, v_new, lambda_u, lambda_v


def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters grid.
    Input:
        X: (n, d), can be the original data matrix or the residual matrix
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iter, u, v, s, lambda_u, lambda_v
        n_iter: number of iterations
        u: (n, 1), the final u at convergence
        v: (d, 1), the final v at convergence
        s: scalar, the singular value at convergence
        lambda_u, lambda_v: the optimal tuning parameter at convergence
    """
    # SVD
    U, _, VT = la.svd(X)

    # initial value
    u_old = U[:,0][:,None]
    v_old = VT[0][:,None]

    for i in range(max_iter):
        u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
        if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol:  # achieve the tolerance
            break
```

```
        if np.all(u_new == 0) or np.all(u_new == 0):  # full shrinkage (i.e., all zeros in the vector)
            print("Warning: Full shrinkage has been achieved. Iterations stops. No further decomposition. The desired number of layers may not be achieved. ")
            break
        u_old, v_old = u_new, v_new
    n_iter = i+1  # number of iterations
    u, v = u_new, v_new  # the final u and v at convergence
    s = (u_new.T @ X @ v_new)[0][0]
    if n_iter == max_iter:
        print("Warning: The maximum iteration has been achieved. Please consider increasing `max_iter`.")
    return n_iter, u, v, s, lambda_u, lambda_v

def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the SSVD given the data matrix X and the desired number of SSVD layers.
    Input:
        X: (n, d), the original data matrix
        num_layer: desired number of SSVD layers
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iters, us, vs, ss, lambda_us, lambda_vs
        n_iters: (num_layer,), number of iterations for each layer
        us: (n, num_layer), the final u at convergence for each layer
        vs: (d, num_layer), the final v at convergence for each layer
        ss: (num_layer,), the singular value at convergence for each layer
        lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each layer
    """
    n, d = X.shape
    n_iters = np.zeros(num_layer, dtype = int)
    ss = np.zeros(num_layer)
    lambda_us = np.zeros(num_layer)
    lambda_vs = np.zeros(num_layer)
    us = np.zeros((n, num_layer))
    vs = np.zeros((d, num_layer))
    # initial value
    u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
    for i in range(num_layer):
        resi_mat = resi_mat - s*u@v.T
        n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2, max_iter, tol)
        n_iters[i] = n_iter
        ss[i] = s
        lambda_us[i] = lambda_u
        lambda_vs[i] = lambda_v
        us[:,i] = u[:,0]
        vs[:,i] = v[:,0]
        if np.all(u == 0) or np.all(v == 0):  # full shrinkage (i.e., all zeros in the vector)
            break
    return n_iters, us, vs, ss, lambda_us, lambda_vs
```

Appendix D Version 4

A comparable modication combining broadcasting and map

```
## This is a modified version of SSVDversion3.
## The broadcasting may result in extremely huge matrix (as we have to compute the outer product),
## which can be counterproductive if it indeed slows the computation.
## In this version, we appropriately combine broadcasting and the map loop.

```

```python
import numpy as np
import scipy.linalg as la


def get_outer_prod(tilde_vec, fixed_vec, fixed_name):
    """Compute the outer product according to the formula. """
    if fixed_name == "u":
        return np.outer(fixed_vec, tilde_vec)
    else:
        return np.outer(tilde_vec, fixed_vec)


def update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid):
    """Update u and v once given the current u and v.
    Input:
        u_old: (n, 1)
        v_old: (d, 1)
        X: (n, d)
        Y: (nd, 1)
        Z: (nd, 1)
        gamma1, gamma2: scalar, tuning parameter
        lam_grid: ndarray, a grid of lambdas to be selected
    Return:
        u_new: (n, 1)
        v_new: (d, 1)
        lambda_u, lambda_v: scalar, the optimal lambda under the current u and v
    """
    n, d = X.shape
    nd = n*d
    S = len(lam_grid)

    # initialize
    v_new = np.zeros(v_old.shape)
    u_new = np.zeros(u_old.shape)
    lambda_u = lambda_v = 0
    SSEs_v = SSEs_u = np.zeros(S)

    ## Update v using current u

    # ols for v, use current u
    v_tilde_hat = X.T @ u_old  # (d, 1), fixed ols estimate
    SSE_v = np.sum((X - np.outer(u_old, v_tilde_hat))**2)  # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
    sigma2_hat_v = SSE_v / (nd-d)  # scalar, fixed ols estimate

    # select lambda_v
    w2 = np.abs(v_tilde_hat)**(-gamma2)  # (d, 1)
    dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0)  # (S,), df for each lambda
    part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
    part1 = np.sign(v_tilde_hat)
    v_tilde_br = part1 * part2  # (d, S), each column is v_tilde under each lambda

    SSEs_v = list(map(lambda tilde_vec: np.sum((X - get_outer_prod(tilde_vec, u_old, "u"))**2), v_tilde_br.T))  # (S,)
    BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v  # (S,), BIC for each lambda
    lambda_v = lam_grid[np.argmin(BICs_v)]

    # update v
    part1 = np.sign(v_tilde_hat)
    part2 = np.abs(v_tilde_hat)-lambda_v*w2/2; part2[part2<0] = 0
    if not np.all(part2 == 0):  # not full shrinkage at v
        v_tilde = np.multiply(part1, part2)  # (d, 1)
        v_new = v_tilde / np.linalg.norm(v_tilde)

        ## Update u using current v

        # ols for u, use current v
        u_tilde_hat = X @ v_new  # (n, 1), fixed ols estimate
        SSE_u = np.sum((X.T - np.outer(v_new, u_tilde_hat))**2)
```

```
        sigma2_hat_u = SSE_u / (nd-n)  # scalar, fixed ols estimate

        # select lambda_u
        w1 = np.abs(u_tilde_hat)**(-gamma1)  # (n, 1)
        dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0)  # (S,), df for each lambda
        part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
        part1 = np.sign(u_tilde_hat)
        u_tilde_br = part1 * part2  # (n, S), each column is u_tilde under each lambda
        SSEs_u = list(map(lambda tilde_vec: np.sum((X - get_outer_prod(tilde_vec, v_new, "v"))**2), u_tilde_br.T))  # (S,)
        BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u  # (S,), BIC for each lambda
        lambda_u = lam_grid[np.argmin(BICs_u)]

        # update u
        part1 = np.sign(u_tilde_hat)
        part2 = np.abs(u_tilde_hat)-lambda_u*w1/2; part2[part2<0] = 0
        if not np.all(part2 == 0):  # not full shrinkage at u
            u_tilde = np.multiply(part1, part2)  # (n, 1)
            u_new = u_tilde / np.linalg.norm(u_tilde)

    return u_new, v_new, lambda_u, lambda_v


def SSVD_layer(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters grid.
    Input:
        X: (n, d), can be the original data matrix or the residual matrix
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iter, u, v, s, lambda_u, lambda_v
        n_iter: number of iterations
        u: (n, 1), the final u at convergence
        v: (d, 1), the final v at convergence
        s: scalar, the singular value at convergence
        lambda_u, lambda_v: the optimal tuning parameter at convergence
    """
    # SVD
    U, _, VT = la.svd(X)

    # initial value
    u_old = U[:,0][:,None]
    v_old = VT[0][:,None]

    for i in range(max_iter):
        u_new, v_new, lambda_u, lambda_v = update_uv(u_old, v_old, X, gamma1, gamma2, lam_grid)
        if np.linalg.norm(u_new-u_old) < tol and np.linalg.norm(v_new-v_old) < tol:  # achieve the tolerance
            break
        if np.all(u_new == 0) or np.all(u_new == 0):  # full shrinkage (i.e., all zeros in the vector)
            print("Warning: Full shrinkage has been achieved. Iterations stops. No further decomposition. The desired number of layers may not be achieved. ")
            break
        u_old, v_old = u_new, v_new
    n_iter = i+1  # number of iterations
    u, v = u_new, v_new  # the final u and v at convergence
    s = (u_new.T @ X @ v_new)[0][0]
    if n_iter == max_iter:
        print("Warning: The maximum iteration has been achieved. Please consider increasing ‘max_iter‘.")
    return n_iter, u, v, s, lambda_u, lambda_v

def SSVD(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the SSVD given the data matrix X and the desired number of SSVD layers.
    Input:
        X: (n, d), the original data matrix
```

```
        num_layer: desired number of SSVD layers
        lam_grid: ndarray, a grid of lambdas to be selected
        gamma1, gamma2: scalar, tuning parameter
        max_iter: integer, the maximum iteration times
        tol: float, tolerance to stop the iteration
    Return: n_iters, us, vs, ss, lambda_us, lambda_vs
        n_iters: (num_layer,), number of iterations for each layer
        us: (n, num_layer), the final u at convergence for each layer
        vs: (d, num_layer), the final v at convergence for each layer
        ss: (num_layer,), the singular value at convergence for each layer
        lambda_us, lambda_vs: (num_layer,), the optimal tuning parameter at convergence for each layer
    """
    n, d = X.shape
    n_iters = np.zeros(num_layer, dtype = int)
    ss = np.zeros(num_layer)
    lambda_us = np.zeros(num_layer)
    lambda_vs = np.zeros(num_layer)
    us = np.zeros((n, num_layer))
    vs = np.zeros((d, num_layer))
    # initial value
    u = np.zeros((n, 1)); v = np.zeros((d, 1)); resi_mat = X; s = 0
    for i in range(num_layer):
        resi_mat = resi_mat - s*u@v.T
        n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer(resi_mat, lam_grid, gamma1, gamma2, max_iter, tol)
        n_iters[i] = n_iter
        ss[i] = s
        lambda_us[i] = lambda_u
        lambda_vs[i] = lambda_v
        us[:,i] = u[:,0]
        vs[:,i] = v[:,0]
        if np.all(u == 0) or np.all(v == 0):  # full shrinkage (i.e., all zeros in the vector)
            break
    return n_iters, us, vs, ss, lambda_us, lambda_vs
```

Appendix E Version 5

Final version optimized by Cython based on version 3

```
## This is the final version optimized by Cython.
## The following should be run in Jupyter notebook.

import cython
%load_ext cython

%%cython

import numpy as np
import scipy.linalg as la
import cython
from cython.parallel import parallel, prange

cdef extern from "math.h":
    double log(double x) nogil
    double pow(double x, double y) nogil
    double fabs(double x) nogil
    double sqrt(double x)
    double isless(double x, double y) nogil
    double fmax(double x, double y)
    double fma(double x, double y, double z) nogil


```

```
cdef double vector_dist_sq(double[:,:] u, double[:,:] v):
    """Squared Euclidean distance between two vectors. Can also compute the squared norm of a vector if set one of the input being zero vector. """
    cdef int i
    cdef double s = 0
    for i in range(u.shape[0]):
        s += pow(u[i,0] - v[i,0], 2)
    return s

@cython.boundscheck(False)
@cython.wraparound(False)
cdef matrix_multiply(double[:,:] u, double[:, :] v, double[:, :] res, double c = 1):
    """Matrix multiplication, equivalent to c*u@v."""
    cdef int i, j, k
    cdef int m = u.shape[0], n = u.shape[1], p = v.shape[1]
    with cython.nogil, parallel():
        for i in prange(m):  # parallel
            for j in prange(p):  # parallel
                res[i,j] = 0
                for k in range(n):  # serial
                    res[i,j] += u[i,k] * v[k,j]
                res[i,j] = fma(res[i,j], c, 0)

@cython.boundscheck(False)
@cython.wraparound(False)
cdef elementwise_multiply(double[:,:] u, double[:,:] v, double[:,:] res):
    """Element multiplication of two matrices. """
    cdef int i, j
    cdef int m = u.shape[0], n = u.shape[1]
    with cython.nogil, parallel():
        for i in prange(m):  # parallel
            for j in prange(n):  # parallel
                res[i,j] = u[i,j] * v[i,j]


@cython.boundscheck(False)
@cython.wraparound(False)
cdef double[:,:] vec_outer_prod(double[:,:] u, double[:,:] v, double c = 1):
    """Outer product between two vectors, equivalent to c*u@v.T."""
    cdef int n = u.shape[0], m = v.shape[0]
    cdef int i, j
    cdef double[:,:] res = np.zeros((n, m))
    with cython.nogil, parallel():
        for i in prange(n):  # parallel
            for j in prange(m):  # parallel
                res[i,j] = u[i,0] * v[j,0] * c
    return res


@cython.boundscheck(False)
@cython.wraparound(False)
cdef double[:,:] get_w(double[:,:] tilde_hat, double gamma):
    """Get the w vector, equivalent to elementwise |tilde_hat|^(-gamma)"""
    cdef int i, l = tilde_hat.shape[0]
    cdef double[:,:] w = np.zeros((l, 1))
    with cython.nogil, parallel():
        for i in prange(l):  # parallel
            w[i,0] = pow(fabs(tilde_hat[i,0]), -gamma)
    return w



@cython.boundscheck(False)
@cython.wraparound(False)
cdef double[:,:] get_part2(double[:,:] tilde_hat, double[:,:] w, double lam):
    """Compute the part2 in the updating formula. """
    cdef int i, l = tilde_hat.shape[0]
    cdef double ele
    cdef double[:,:] part2 = np.zeros((l,1))
```

```

    with cython.nogil, parallel():
        for i in prange(l):  # parallel
            ele = fabs(tilde_hat[i,0])-lam*w[i,0]/2
            if ele > 0:
                part2[i,0] = ele
    return part2
cdef int is_full_shinkage(double[:,:] v):
    """Judge if the input vector if fully shrunk to 0. Return 1 if fully shunk, and 0 otherwise. """
    cdef int l = v.shape[0]
    cdef int flag = 1

    for i in range(l):
        if v[i,0] != 0:
            flag = 0
            break
    return flag  # flag = 1 if all elements are 0, = 0 otherwise

@cython.boundscheck(False)
@cython.wraparound(False)
cdef get_vec_new(double[:,:] tilde_vec, double[:,:] vec_new):
    """Get the vec_new according to tilde_vec/norm(tilde_vec). """
    cdef int i, l = tilde_vec.shape[0]
    cdef double norm = sqrt(vector_dist_sq(tilde_vec, np.zeros((l, 1))))
    with cython.nogil, parallel():
        for i in prange(l):  # parallel
            vec_new[i,0] = tilde_vec[i,0] / norm


cpdef update_uv3c(double[:,:] u_old, X, gamma1, gamma2, lam_grid, double[:,:] u_new, double[:,:] v_new):
    """Update u and v once given the current u and v."""
    cdef int n = X.shape[0], d = X.shape[1], S = lam_grid.shape[0]
    cdef int nd = n*d

    cdef double[:,:] v_tilde_hat = np.zeros((d,1)), v_tilde = np.zeros((d,1))
    cdef double[:,:] u_tilde_hat = np.zeros((n,1)), u_tilde = np.zeros((n,1))

    cdef double lambda_u = 0, lambda_v = 0


    ## Update v using current u

    # ols for v, use current u
    matrix_multiply(X.T, u_old, v_tilde_hat)  # v_tilde_hat: (d, 1), fixed ols estimate
    SSE_v = np.sum((X - vec_outer_prod(u_old, v_tilde_hat))**2)  # scalar, SSE_v = (Y-Y_hat).T @ (Y-Y_hat)
    sigma2_hat_v = SSE_v / (nd-d)  # scalar, fixed ols estimate

    # select lambda_v
    w2 = get_w(v_tilde_hat, gamma2)  # (d, 1)
    dfs_v = np.sum(np.abs(v_tilde_hat) > lam_grid*w2/2, axis=0)  # (S,), df for each lambda
    part2 = v_tilde_hat - lam_grid*w2/2; part2[part2<0] = 0
    part1 = np.sign(v_tilde_hat)
    v_tilde_br = part1 * part2  # (d, S), each column is v_tilde under each lambda
    outer_prods = np.outer(u_old, v_tilde_br.T)  # (n, d*S), each chunk of size (n, d) is the outer product mat under each lambda
    outer_prods = np.array(np.hsplit(outer_prods, S))  # (S, n, d)
    SSEs_v = ((X-outer_prods)**2).sum(axis = (1,2))  # (S,), ||X-uv_tilde^T||_F^2=||Y-Y_hat||^2 for each lambda
    BICs_v = SSEs_v/(nd*sigma2_hat_v) + np.log(nd)/nd*dfs_v  # (S,), BIC for each lambda
    lambda_v = lam_grid[np.argmin(BICs_v)]

    # update v
    part1 = np.sign(v_tilde_hat)
    part2 = get_part2(v_tilde_hat, w2, lambda_v)

    if is_full_shinkage(part2) == 0:  # not full shrinkage at v
```

```
        elementwise_multiply(part1, part2, v_tilde)  # v_tilde: (d, 1)
        get_vec_new(v_tilde, v_new)  # v_new: (d, 1)

        ## Update u using current v

        # ols for u, use current v
        matrix_multiply(X, v_new, u_tilde_hat)  # u_tilde_hat: (n, 1), fixed ols estimate
        SSE_u = np.sum((X.T - vec_outer_prod(v_new, u_tilde_hat))**2)
        sigma2_hat_u = SSE_u / (nd-n)  # scalar, fixed ols estimate

        # select lambda_u
        w1 = get_w(u_tilde_hat, gamma1)  # (n, 1)
        dfs_u = np.sum(np.abs(u_tilde_hat) > lam_grid*w1/2, axis=0)  # (S,), df for each lambda
        part2 = u_tilde_hat - lam_grid*w1/2; part2[part2<0] = 0
        part1 = np.sign(u_tilde_hat)
        u_tilde_br = part1 * part2  # (n, S), each column is u_tilde under each lambda
        outer_prods = np.outer(u_tilde_br.T, v_new)  # (n*S, d), each chunk of size (n, d) is the outer product mat under each lambda
        outer_prods = np.array(np.vsplit(outer_prods, S))  # (S, n, d)
        SSEs_u = ((X-outer_prods)**2).sum(axis = (1,2))  # (S,), ||X-u_tildev^T||_F^2=||Z-Z_hat||^2 for each lambda
        BICs_u = SSEs_u/(nd*sigma2_hat_u) + np.log(nd)/nd*dfs_u  # (S,), BIC for each lambda
        lambda_u = lam_grid[np.argmin(BICs_u)]

        # update u
        part1 = np.sign(u_tilde_hat)
        part2 = get_part2(u_tilde_hat, w1, lambda_u)
        if is_full_shinkage(part2) == 0:  # not full shrink at u
            elementwise_multiply(part1, part2, u_tilde)  # u_tilde: (n, 1)
            get_vec_new(u_tilde, u_new)  # u_new: (n, 1)

    return lambda_u, lambda_v


cpdef SSVD_layer3c(X, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the sparse SVD layer given the data matrix X at a SVD layer and the tuning parameters grid."""
    cdef int n = X.shape[0], d = X.shape[1]
    # SVD
    U, _, VT = la.svd(X)

    # initial value
    cdef double[:,:] u_old = U[:,0][:,None], v_old = VT[0][:,None]
    cdef double[:,:] u_new = np.zeros((n,1)), v_new = np.zeros((d,1))
    cdef int i  # number of iterations

    for i in range(max_iter):
        lambda_u, lambda_v = update_uv3c(u_old, X, gamma1, gamma2, lam_grid, u_new, v_new)  # update u_new, v_new
        if isless(fmax(vector_dist_sq(u_new, u_old), vector_dist_sq(v_new, v_old)), pow(tol, 2)):  # achieve the tolerance
            break
        if fmax(is_full_shinkage(u_new), is_full_shinkage(v_new)):  # full shrinkage (i.e., all zeros in the vector)
            print("Warning: Full shrinkage has been achieved. Iterations stops. No further decomposition. The desired number of layers may not be achieved. ")
            break
        u_old, v_old = u_new, v_new
    u, v = u_new, v_new  # the final u and v at convergence
    s = (u.T @ X @ v)[0][0]
    n_iter = i+1
    if n_iter == max_iter:
        print("Warning: The maximum iteration has been achieved. Please consider increasing `max_iter`.")
    return n_iter, np.array(u), np.array(v), s, lambda_u, lambda_v


cpdef SSVD3c(X, num_layer, lam_grid, gamma1, gamma2, max_iter=5000, tol=1e-6):
    """Get the SSVD given the data matrix X and the desired number of SSVD layers."""
```

```
    n, d = X.shape
    n_iters = np.zeros(num_layer, dtype = int)
    ss = np.zeros(num_layer)
    lambda_us = np.zeros(num_layer)
    lambda_vs = np.zeros(num_layer)
    us = np.zeros((n, num_layer))
    vs = np.zeros((d, num_layer))
    # initial value
    cdef double[:,:] res = np.zeros((n, d))
    cdef double s = 0
    resi_mat = X
    u = np.zeros((n, 1)); v = np.zeros((d, 1))
    for i in range(num_layer):
        resi_mat = resi_mat - s * np.outer(u, v)
        n_iter, u, v, s, lambda_u, lambda_v = SSVD_layer3c(resi_mat, lam_grid, gamma1, gamma2,
     max_iter, tol)
        n_iters[i] = n_iter
        ss[i] = s
        lambda_us[i] = lambda_u
        lambda_vs[i] = lambda_v
        us[:,i] = u[:,0]
        vs[:,i] = v[:,0]
        if np.all(u == 0) or np.all(v == 0):  # full shrinkage (i.e., all zeros in the vector)
            break
    return n_iters, us, vs, ss, lambda_us, lambda_vs
```