

# Alternating Manifold Proximal Gradient Method

Shixiang Chen, Justin Huang, Benjamin Jochem, Lingzhou Xue, Hui Zou

2021-09-10

## Contents

- Introduction
  - Algorithm
  - Convergence
  - Pseudocode
- Installation
- Documentation
  - R Usage
  - Python Usage
  - Arguments
  - Values
- Quick Start
  - R Quick Start
  - Python Example
- References

## Introduction

`sparsepca` and `amanpg` find sparse loadings in principal component analysis (PCA) via an alternating manifold proximal gradient method (A-ManPG). Seeking a sparse basis allows the leading principal components to be easier to interpret when modeling with high-dimensional data. PCA is modeled as a regularized regression problem under the elastic net constraints (linearized L1 and L2 norms) in order to induce sparsity. Due to the nonsmoothness and nonconvexity numerical difficulties, A-ManPG is implemented to guarantee convergence.

The package provides a function for performing sparse PCA and a function for normalizing data.

The authors of A-ManPG are Shixiang Chen, Shiqian Ma, Lingzhou Xue, and Hui Zou. The Python and R packages are maintained by Justin Huang and Benjamin Jochem. A MATLAB implementation is maintained by Shixiang Chen.

**Algorithm Description** The A-ManPG algorithm can be applied to solve the general manifold optimization problem

$$\min F(A, B) := H(A, B) + f(A) + g(B) \text{ subject to (s.t.) } A \in \mathcal{M}_1, B \in \mathcal{M}_2 \quad (1)$$

where  $H(A, B)$  is a smooth function of  $A, B$  with a Lipschitz continuous gradient,  $f(\cdot)$  and  $g(\cdot)$  are lower semicontinuous (possibly nonsmooth) convex functions, and  $\mathcal{M}_1, \mathcal{M}_2$  are two embedded submanifolds in the Euclidean space.

For sparse PCA, the following function definitions are used:

- $H(A, B) = \text{Tr}(B^T X^T X B) - 2\text{Tr}(A^T X^T X B)$
- $f(A) \equiv 0$

- $g(B) = \lambda_2 \sum_{j=1}^k \|B_j\|_2^2 + \sum_{j=1}^k \lambda_{1,j} \|B_j\|_1$
- $\mathcal{M}_1 = \text{St}(p, k)$
- $\mathcal{M}_2 = \mathbb{R}^{p \times k}$

where  $X$  is the  $n \times p$  data matrix or  $n \times n$  covariance matrix,  $A$  is the scores and  $B$  is the loadings,  $k$  is the rank of the matrices (in other words, how many principal components are desired),  $\lambda_1$  is the L1 norm penalty and  $\lambda_2$  is the L2 norm penalty. Both the L1 and L2 norm are used as elastic net regularization to impose sparseness within the loadings. Note that a different L1 norm penalty is used for every principal component, and the algorithm operates differently when the L2 norm penalty is set to a large constant (**np.inf** or **Inf**).

The A-ManPG algorithm uses the following subproblems with an alternating updating scheme to solve sparse PCA, computed in a Gauss-Seidel manner for faster convergence.

$$D_k^A := \arg \min_{D^A} \langle \nabla_A H(A_k, B_k), D^A \rangle + f(A^k + D^A) + \frac{1}{2t_1} \|D^A\|_F^2 \quad \text{s.t.} \quad D^A \in T_{A_k} \mathcal{M}_1 \quad (2)$$

$$D_k^B := \arg \min_{D^B} \langle \nabla_B H(A_{k+1}, B_k), D^B \rangle + f(B^k + D^B) + \frac{1}{2t_2} \|D^B\|_F^2 \quad \text{s.t.} \quad D^B \in T_{B_k} \mathcal{M}_2 \quad (3)$$

where  $A_{k+1}$  is obtained via a retraction operation (in this case, polar decomposition),  $t_1 \leq L_A$ , and  $t_2 \leq L_B$ .  $L_A$  and  $L_B$  are the least upper bounds of the Lipschitz constants for  $\nabla_A H(A, B)$  and  $\nabla_B H(A, B)$ , respectively. The subproblems are solved using an adaptive semismooth Newton method.

**Convergence** Let  $\epsilon$  represent a tolerance level to detect convergence. An  $\epsilon$ -stationary point is defined as a point  $(A, B)$  with corresponding  $D^A$  and  $D^B$  that satisfy the following:

$$\|D^A/t_1\|_F^2 + \|D^B/t_2\|_F^2 \leq \epsilon^2 \quad (4)$$

The algorithm reaches an  $\epsilon$ -stationary point in at most

$$\frac{2(F(A_0, B_0) - F^*)}{((\gamma \bar{\alpha}_1 t_1 + \gamma \bar{\alpha}_2 t_2) \epsilon^2)}$$

iterations, where:

- $(A_0, B_0)$  are the initial values
- $F^*$  is the lower bound of  $F$ , from the general manifold optimization problem
- $\bar{\alpha}_1$  and  $\bar{\alpha}_2$  are positive constants

**Pseudocode** The following describes the algorithm used for solving the general manifold optimization problem using A-ManPG. For solving sparse PCA, the algorithm is implemented with the aforementioned definitions.

Input initial point (A0,B0) and necessary parameters for the required problem

for i=0,1,... do

    Solve the first subproblem for Da

    Set alpha = 1

    while F(Reetr(alpha \* Da),B) > F(A,B) - alpha \* norm(Da)^2 / (2 \* t1) do

        alpha = gamma \* alpha

    end while

Set A = Retr(alpha \* Da)

```

Solve the second subproblem for Db
Set alpha = 1

while F(A,Retr(alpha * Db)) > F(A,B) - alpha * norm(Db)^2 / (2 * t2) do
  alpha = gamma * alpha
end while

Set B = Retr(alpha * Db)
end for

Return A as the scores and B as the sparse loadings

```

## Installation

To install the R package, install `amanpg` directly from CRAN.

```
install.packages("amanpg")
```

To install the Python package, use `pip` to obtain `sparsepca` from PyPI.

```
pip3 install sparsepca
```

## Documentation

### R Usage

```

spca.amanpg(z, lambda1, lambda2,
            f_palm = 1e5, x0 = NULL, y0 = NULL, k = 0, type = 0,
            gamma = 0.5, maxiter = 1e4, tol = 1e-5,
            normalize = TRUE, verbose = FALSE)

```

### Python Usage

```

spca(z, lambda1, lambda2,
     x0=None, y0=None, k=0, gamma=0.5, type=0,
     maxiter=1e4, tol=1e-5, f_palm=1e5,
     normalize=True, verbose=False):

```

### Arguments

Name	Python Type	R Type	Description
<code>z</code>	<code>numpy.ndarray</code>	matrix	Either the data matrix or sample covariance matrix
<code>lambda1</code>	float list	numeric vector	List of parameters of length <code>n</code> for L1-norm penalty
<code>lambda2</code>	float or <code>numpy.inf</code>	numeric or Inf	L2-norm penalty term
<code>x0</code>	<code>numpy.ndarray</code>	matrix	Initial x-values for the gradient method, default value is the first <code>n</code> right singular vectors
<code>y0</code>	<code>numpy.ndarray</code>	matrix	Initial y-values for the gradient method, default value is the first <code>n</code> right singular vectors

Name	Python Type	R Type	Description
<code>k</code>	int	int	Number of principal components desired, default is 0 (returns min(n-1, p) principal components)
<code>gamma</code>	float	numeric	Parameter to control how quickly the step size changes in each iteration, default is 0.5
<code>type</code>	int	int	If 0, b is expected to be a data matrix, and otherwise b is expected to be a covariance matrix; default is 0
<code>maxiter</code>	int	int	Maximum number of iterations allowed in the gradient method, default is 1e4
<code>tol</code>	float	numeric	Tolerance value required to indicate convergence (calculated as difference between iteration f-values), default is 1e-5
<code>f_palm</code>	float	numeric	Upper bound for the F-value to reach convergence, default is 1e5
<code>normalize</code>	bool	logical	Center and normalize rows to Euclidean length 1 if True, default is True
<code>verbose</code>	bool	logical	Function prints progress between iterations if True, default is False

**Values** Python returns a dictionary with the following key-value pairs, while R returns a list with the following elements:

Key	Python Value Type	R Value Type	Value
<code>loadings</code>	numpy.ndarray	matrix	Loadings of the sparse principal components
<code>f_manpg</code>	float	numeric	Final F-value
<code>x</code>	numpy.ndarray	matrix	Corresponding ndarray in subproblem to the loadings
<code>iter</code>	int	numeric	Total number of iterations executed
<code>sparsity</code>	float	numeric	Number of sparse loadings ( <code>loadings == 0</code> ) divided by number of all loadings

Key	Python Value Type	R Value Type	Value
time	float	numeric	Execution time in seconds

## Quick Start

Consider the two examples below for running sparse PCA on randomly-generated data: one using a finite  $\lambda_2$ , and the other using a large constant  $\lambda_2$ .

**R Example** As with other libraries, begin by loading **amanpg** in R.

```
library(amanpg)
```

Before proceeding, it is helpful to determine a few parameters. Let the rank of the sparse loadings matrix be  $k = 4$  (returning four principal components), the input data matrix be  $n \times p$  where  $n = 1000$  and  $p = 500$ ,  $\lambda_1$  be a  $4 \times 1$  “matrix” where  $\lambda_{i,1} = 0.1$ , and  $\lambda_2 = 1$ .

```
# parameter initialization
k <- 4
n <- 1000
p <- 500
lambda1 <- matrix(data=0.1, nrow=k, ncol=1)
lambda2 <- 1
```

For this example, the data matrix **z** is randomly generated from the normal distribution. Although it should be centered to mean 0 and normalized to Euclidean length 1, the function will automatically preprocess the input matrix when **normalize=TRUE**.

```
# data matrix generation
set.seed(10)
z <- matrix(rnorm(n * p), n, p)
```

Table 3: Generated Data Matrix

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
0.0187462	1.0500137	-0.3078650	0.4605151	-1.3456607	1.0115469	1.0914354	-0.1516380	-0.6442263	-0.9771862
-0.1842525	0.2860926	0.7580856	0.2350253	1.3901086	-0.2753449	-1.3122779	-0.0868394	-0.6683819	-1.1545605
-1.3713305	0.2405648	-0.5738634	0.6432573	0.4407198	0.1359363	1.4036478	-0.6577627	-0.1521340	-0.0557722
-0.5991677	0.8327052	-0.9387445	0.9131981	-1.5072985	0.9726657	1.0516302	0.3798498	-0.1487630	0.6177806
0.2945451	-0.2229832	-0.0276993	0.9882860	-0.2584527	0.1851170	-0.6740013	0.1318977	-1.3208829	1.3859589
0.3897943	0.2883442	-1.0662487	0.1127413	-1.6588799	-0.3955091	2.2357281	0.0554679	0.7280102	1.7293072

Alternatively, the data can be normalized beforehand and the parameter is set to **FALSE** in the function call. However, this example won’t do so, but the output of **normalize** is displayed below.

```
# see the effects of normalize()
head(as.data.frame(normalize(z))[, 1:10])
```

Now the function is called, passing through matrix **a**, **lambda1**, **lambda2**, and our desired rank **k**. The output is stored as a list in **fin\_sprout**. Note that if a different initial point is desired, **x0** and **y0** should be modified, but the default value as the first  $k$  right singular vectors is sufficient for this example.

If further printout is desired, set **verbose=TRUE** for progress updates (time, difference for convergence,  $F$  value) per iteration.

Table 4: Normalized Data Matrix

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
0.0003430	0.0481267	-0.0132514	0.0219345	-0.0621459	0.0458222	0.0523199	-0.0078868	-0.0285429	-0.0477722
-0.0089624	0.0123909	0.0357869	0.0112677	0.0641425	-0.0138376	-0.0586052	-0.0047972	-0.0292120	-0.0551659
-0.0647869	0.0105393	-0.0258082	0.0306516	0.0211169	0.0051185	0.0673177	-0.0316569	-0.0056872	-0.0049361
-0.0295190	0.0395046	-0.0442725	0.0446800	-0.0723942	0.0457365	0.0524440	0.0175012	-0.0057055	0.0274720
0.0121050	-0.0102003	-0.0001986	0.0427144	-0.0106224	0.0067722	-0.0273987	0.0048744	-0.0551507	0.0571280
0.0180233	0.0129888	-0.0496851	0.0058898	-0.0785342	-0.0201087	0.1080580	0.0017906	0.0361384	0.0800020

```
# function call
fin_sprout <- spca.amanpg(z, lambda1, lambda2, k=4)
print(paste(fin_sprout$iter, "iterations,", fin_sprout$sparsity, "sparsity,", fin_sprout$time))

## [1] "280 iterations, 0.491 sparsity, 1.3493480682373"
```

The loadings can be viewed from `fin_sprout$loadings`. Note that many entries are set to zero as a result of the induced sparsity.

```
# View loadings
head(as.data.frame(fin_sprout$loadings))
```

Table 5: Finite Lambda Loadings

0.0880974	0.0000000	0	0.0000000
0.0000000	0.0000000	0	0.0000000
0.0285272	0.0324180	0	0.0784247
0.0000000	-0.0456984	0	0.0906653
0.0103526	-0.0010865	0	-0.2210455
0.0000000	0.0000000	0	0.0000000

The resulting scree plot looks like this.

```
pr.var <- (apply(fin_sprout$x, 2, sd))^2
pve <- pr.var / sum(pr.var)

par(mfrow=c(1,2))
plot(pve,
     xlab="Sparse PC",
     ylab="Proportion of Variance Explained",
     ylim=c(0,1),
     type="b")
plot(cumsum(pve),
     xlab="Sparse PC",
     ylab="Cumulative Proportion of Variance Explained",
     ylim=c(0,1),
     type="b")
```

The resulting biplot, with the zero loadings filtered out, can be obtained using the following:

```
y_sub = apply(fin_sprout$loadings, 1, function(row) all(row != 0))
loadings = fin_sprout$loadings[y_sub, ]
par(mfrow=c(1,1))
biplot(fin_sprout$x, loadings, xlab="PC 1", ylab="PC 2")
```

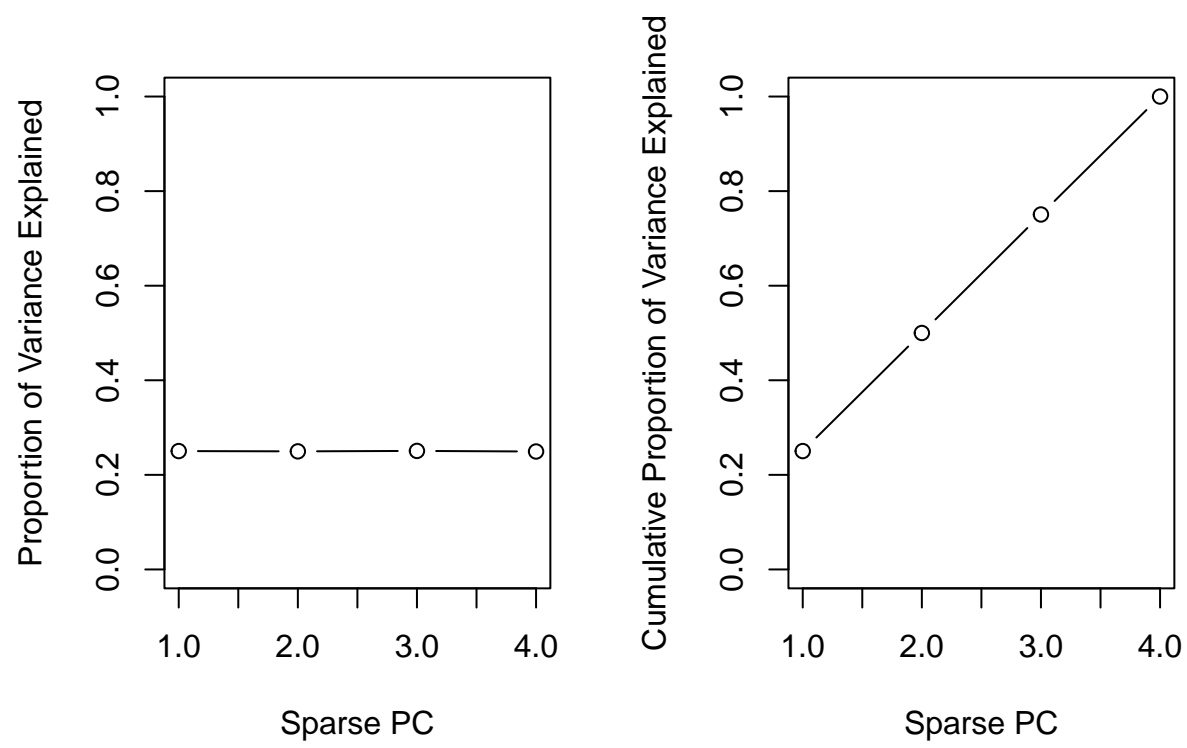
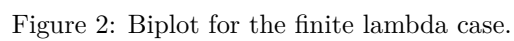


Figure 1: Scree plots for the finite lambda case.





Now consider an alternative situation where we set  $\lambda_2$  to a large constant `Inf`. The algorithm changes by directly retracting B without using a while loop to determine an appropriate retraction step size and only iterating A.

```
# infinite lambda2
inf_sprout <- spca.amanpg(z, lambda1, lambda2=Inf, k=4)
print(paste(inf_sprout$iter, "iterations,", inf_sprout$sparsity, "sparsity,", inf_sprout$time))

## [1] "344 iterations, 0.253 sparsity, 1.5885910987854"
head(as.data.frame(inf_sprout$loadings))
```

Table 6: Finite Lambda Loadings

0.0612782	0.0070632	0.0148628	0.0000000
0.0000000	0.0080497	0.0000000	0.0000000
0.0000000	0.0670739	-0.0796881	0.0616913
0.0000000	0.0000000	0.0050157	0.1115940
0.0674123	-0.0194809	0.0490605	-0.1250086
0.0000000	0.0000000	0.0000000	-0.0207933

We obtain the scree plot and the biplot using the same method.

```
pr.var <- (apply(inf_sprout$x, 2, sd))^2
pve <- pr.var / sum(pr.var)

par(mfrow=c(1,2))
plot(pve,
     xlab="Sparse PC",
     ylab="Proportion of Variance Explained",
     ylim=c(0,1),
     type="b")
plot(cumsum(pve),
     xlab="Sparse PC",
     ylab="Cumulative Proportion of Variance Explained",
     ylim=c(0,1),
     type="b")
```

For data with high dimensionality, the biplot is still much harder to read with a lower sparsity value.

```
y_sub = apply(inf_sprout$loadings, 1, function(row) all(row != 0))
loadings = inf_sprout$loadings[y_sub, ]

par(mfrow=c(1,1))
biplot(inf_sprout$x, loadings, xlab="PC 1", ylab="PC 2")
```

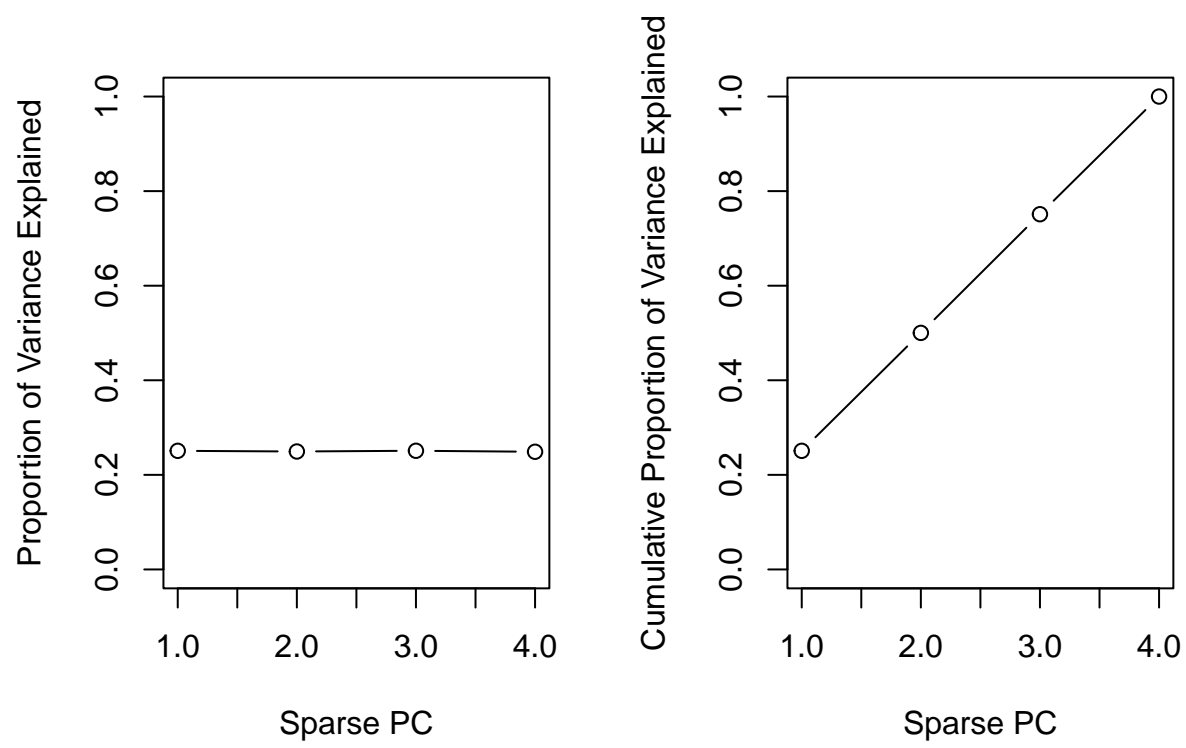


Figure 3: Scree plot for  $\lambda = \infty$  case.

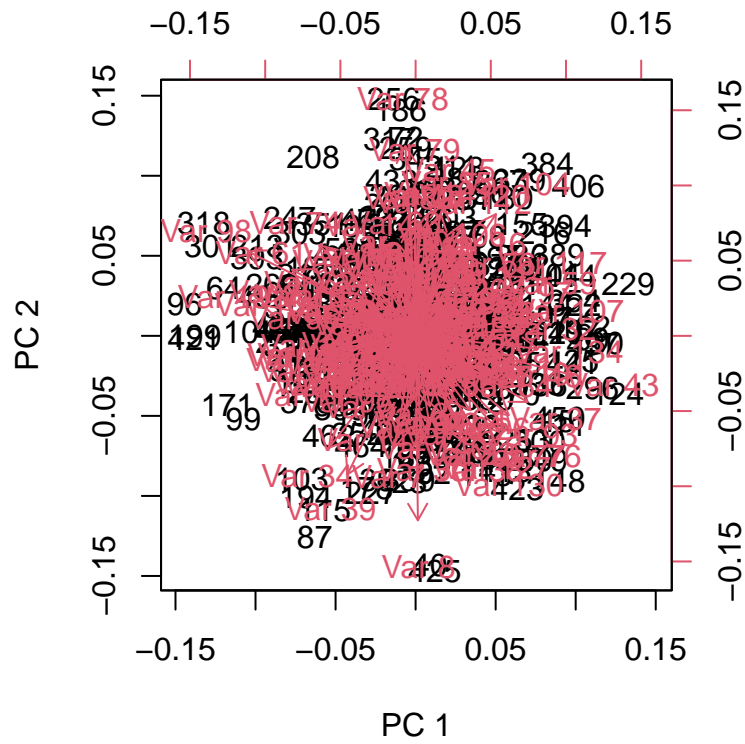


Figure 4: Biplot for  $\lambda = \infty$  case. Observe that with lower sparsity in the loadings and high-dimensional data, the biplot becomes less readable.

**Python Example** Note that the Python package depends on numpy.

The following example accomplishes the same situation (down to the same randomly-generated data) but in Python.

```
import numpy as np
from sparsepca import spca

k = 4 # rank
p = 500 # dimensions
n = 1000 # sample size
lambda1 = 0.1 * np.ones((k, 1))
lambda2 = 1

np.random.seed(10)
z = np.random.normal(0, 1, size=(n, p)) # generate random normal 1000x500 matrix

fin_sprout = spca(z, lambda1, lambda2, k=k)
print(f"Finite: {fin_sprout['iter']} iterations with final value
      {fin_sprout['f_manpg']], sparsity {fin_sprout['sparsity']],
      timediff {fin_sprout['time']}.")

fin_sprout['loadings']

inf_sprout = spca(z, lambda1, np.inf, k=k)
print(f"Infinite: {inf_sprout['iter']} iterations with final value
      {inf_sprout['f_manpg']], sparsity {inf_sprout['sparsity']],
      timediff {inf_sprout['time']}.")

inf_sprout['loadings']
```

## References

- Chen, S., Ma, S., Xue, L., and Zou, H. (2020) "An Alternating Manifold Proximal Gradient Method for Sparse Principal Component Analysis and Sparse Canonical Correlation Analysis" *INFORMS Journal on Optimization* 2:3, 192-208 <doi:10.1287/ijoo.2019.0032>.
- Zou, H., Hastie, T., & Tibshirani, R. (2006). Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, 15(2), 265-286 <doi:10.1198/106186006X113430>.
- Zou, H., & Xue, L. (2018). A selective overview of sparse principal component analysis. *Proceedings of the IEEE*, 106(8), 1311-1320 <doi:10.1109/JPROC.2018.2846588>.