UCLouvain

FACULTE DES SCIENCES

Faculté des sciences

Comparaison entre différentes méthodes d'Analyse en Composantes Principales parcimonieuse

Auteur : Rghioui Mehdi

Promoteur : Professeur Segers Johan

Lecteur : Professeur Pircalabelu Eugen

Année académique 2023-2024

Table des matières

Résumé

L'Analyse en Composantes Principales (ACP) est une méthode très répandue permettant de réduire le nombre de dimensions d'un jeu de données. Les nouvelles variables créées sont appelées les composantes principales et elles vont contenir un maximum de variabilité du jeu de données d'origine. Il existe cependant un problème avec cette méthode: ces nouvelles variables sont difficilement interprétables dû au fait qu'elles sont des combinaisons linéaires de toutes les variables d'origine. Pour résoudre ce problème, il est possible de fixer à 0 certains coefficients des composantes principales les moins informatives, ce qui permet de sélectionner les variables qui participent le plus à l'explication de la composante principale. Cette extension de l'ACP est appelée l'analyse en composantes principales parcimonieuse, et il existe plusieurs façons d'y parvenir. Ce mémoire théorique sert de guide à la bonne utilisation des méthodes existantes appliquant la parcimonie dans les coefficients des composantes principales. Après un rappel sur l'ACP classique et un historique des méthodes de parcimonie, nous passerons en revue les méthodes SPCA, sPCA_rSVD et GPower d'abord d'un point de vue théorique, et ensuite par des simulations et des applications sur de réelles données.

1 Introduction

1.1 Contexte et question

L'Analyse en Composantes Principales, communément appelée l'ACP (Principal Component Analysis ou PCA en anglais), est une méthode de réduction de dimension des données inventée par Pearson (1901) et développée par Hotelling (1933). Elle vise à réduire la dimension, c'est-à-dire le nombre de variables, d'un ensemble de données pouvant comporter un grand nombre de ces variables, tout en préservant au maximum la variabilité du jeu de données d'origine. L'importance de capter un maximum de la variance des données réside dans le pouvoir informatif qu'elle apporte, comme par exemple la distribution des variables ou à quel point ces données sont dispersées. En supprimant des variables, nous allons perdre ces précieuses informations, mais en garder un grand nombre n'est pas une solution non plus car cela rend l'analyse complexe. En effet, un jeu de données à hautes dimensions engendre le fléau de la dimension qui empêche le bon fonctionnement des modèles que l'on veut appliquer et rend les résultats non-significatifs.

L'ACP va donc se charger de trouver un équilibre entre la préservation de la variance et un nombre de variables réduit. Elle consiste à réduire ce nombre de variables en un petit nombre de composantes principales. Ce sont de nouvelles variables qui sont des combinaisons linéaires des variables d'origine, c'est-à-dire que chacune des composantes principales comporte une part d'information de l'ensemble des variables d'origine. Une composante principale est représentée sous forme d'un vecteur où chaque élément correspond à une certaine valeur multipliée par un poids. Nous allons rentrer dans les détails des calculs dans la section suivante (section 2). Cependant, il n'existe aucune corrélation entre les composantes principales, ce qui en a fait une méthode de réduction de dimension très populaire, et celle qui a servie de base pour le développement de la plupart des méthodes récentes et de ses extensions. Un problème que nous pouvons reprocher à l'ACP, dans le cas d'un ensemble de données à hautes dimensions, est la difficulté à interpréter ces composantes principales due au fait qu'elles contiennent des informations de toutes les variables d'origine en même temps.

C'est à ce moment que l'analyse en composantes principales parcimonieuse (sparse Principal Component Analysis) devient intéressante. Il s'agit d'une extension de l'ACP classique qui va se charger d'induire un maximum d'éléments nuls dans les coefficients poids définissant les composantes principales, ce qui permettra de faire ressortir les éléments non-nuls du vecteur qui correspondront aux véritables variables jouant un rôle dans la composantes principales. Cela rendra cette nouvelle variable beaucoup plus facile à interpréter. Il existe plusieurs façons d'obtenir une version parcimonieuse des résultats de l'ACP, chacune ayant

ses avantages et ses inconvénients. Ainsi, nous pouvons nous poser des questions comme par exemple quelles sont les méthodes les plus efficaces de manière générale ? Ou alors dans une situation particulière ?

L'objectif de ce document sera donc de trouver des pistes de réponse à ces questions en comparant dans un premier temps les méthodes à travers la littérature, et ensuite par des simulations et des applications sur divers ensembles de données. Mon point de départ a été le cours de mon professeur et promoteur Mr.Segers (2021). Mes sources sont citées tout le long de mon mémoire, et la bibliographie complète peut être retrouvée en fin de document à la section 5. Le langage de programmation utilisé est R et j'ai codé sur RStudio. Les parties de code ne venant pas de moi sont également systématiquement citées, et mon script R complet peut être retrouvé en annexe. Concernant l'utilisation d'intelligence artificielle pour ce travail, seul ChatGPT 3.5 a été utilisé dans un but strictement informatif, et non de génération de texte ou de code.

1.2 Analyse en composantes principales

Afin de comprendre en détail l'analyse en composantes principales parcimonieuse, nous devons commencer par rappeler le fonctionnement de l'ACP classique.

Considérons une matrice X représentant un ensemble de données. Cette matrice possède n lignes et p colonnes, cela revient à dire qu'elle a n observations et p variables quantitatives. L'objectif de l'ACP est de réduire le nombre de dimensions (de variables) en trouvant les Composantes Principales (CP) qui captureront au mieux la variance des données d'origine. Le nombre de composantes principales choisies, noté k, doit suivre la condition que $k \leq p$ car notre but est de réduire la dimension. Souvent les premières composantes principales sont suffisantes pour expliquer la vue globale du jeu de données d'origine car une fois trouvées, elles sont ordonnées de manières décroissante d'importance.

Pour trouver les composantes principales, la première étape va consister à normaliser notre matrice de données X de manière à ce que les données soient centrées. Pour centrer les données, nous allons soustraire chaque élément x_{ij} par :

$$\bar{x_j} = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \qquad \text{où} \quad i = 1,\ldots,n, \quad j = 1,\ldots,p$$

Où $\bar{x_j}$ représente la moyenne de le colonne j. Appliquer l'ACP à des données centrées permet d'éliminer tout risque de biais, ainsi que de faciliter l'interprétation des résultats.

Le centrage des données est obligatoire en ACP, mais la standardisation reste optionnelle. Elle consiste à diviser chaque élément de la matrice X centrée par l'écart-type σ_j unique à

chaque colonne. La formule de l'écart-type est la suivante :

$$\sigma_j = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_{ij} - \bar{x_j})^2}$$

Nous allons voir que la standardisation sera tout de même appliquée lorsque l'on simulera les résultats des différentes méthodes car nous sommes dans un contexte de comparaison, il est donc important de comparer des résultats se situant sur la même échelle.

Ensuite, nous pouvons calculer la matrice de covariance Σ de taille $p \times p$ où chaque élément sera la covariance entre deux variables X_j et X_l. Comme nous avons standardisé les variables, nous trouverons la matrice de corrélation R de même taille, avec chaque élément compris en -1 et 1. Chaque élément est calculé par la formule suivante :

$$r_{jl} = \frac{\text{cov}\,(X_j, X_l)}{\sigma_j \sigma_l} \qquad \text{où} \quad j = 1, \ldots, p, \quad l = 1, \ldots, p$$

Avec la covariance $\text{cov}\,(X_j, X_l)$ calculée par :

$$\text{cov}\,(X_j, X_l) = \frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x_j})(x_{il} - \bar{x_l})$$

où n reste le nombre d'observations et les $\bar{x}$ sont les moyennes des colonnes en question. Les éléments de la diagonale de la matrice R seront tous égaux à 1. Plus la corrélation entre 2 variables sera proche de 1, plus les variables seront corrélées positivement entre elles. Ce sera le contraire pour une valeur proche de -1, et une corrélation proche de 0 signifie peu de relation entre ces variables.

Nous pouvons ensuite extraire de la matrice de corrélation les valeurs propres (ou eigenvalues) ainsi que les vecteurs propres (ou eigenvectors). La valeur propre λ_α indique la quantité de variance expliquée par chaque vecteur propre v_α, où α représente l'indice de la valeur propre et de son vecteur propre correspondant, de 1 jusqu'à p.

Les valeurs propres sont contenues dans un vecteur de taille p et sont ordonnées de manière décroissante, c'est-à-dire que l'élément λ_1 représente la valeur propre maximisant la variance expliquée. Pour trouver ces valeurs, il suffit de résoudre l'équation :

$$\det(R - \lambda I) = 0$$

Nous avons I la matrice identité de taille $p \times p$ qui permet à ce que le vecteur λ devienne une matrice carrée où les éléments de la diagonales sont les valeurs propres ordonnées de manières décroissantes. La différence avec la matrice de corrélation R est calculée, et det va ensuite calculer le déterminant de la matrice produite par cette différence. La solution de l'équation pour λ est le vecteur contenant les valeurs propres.

Les vecteurs propres quant à eux représentent les directions des axes principaux. Ils sont

contenus dans une matrice V de taille $p \times p$ où chaque colonne représente un vecteur propre et les lignes correspondent aux variables du jeu de données d'origine. Prenons par exemple le premier vecteur propre v_1, correspondant à la première colonne de V. Chaque élément de ce vecteur représente la contribution de la variable correspondante au vecteur propre, donc à quel point cette variable est informative pour v_1. Plus la valeur absolue de cet élément sera grande, plus la variable aura contribué à expliquer ce vecteur propre. Les vecteurs propres sont orthonormés, c'est-à-dire orthogonaux entres eux et ont une norme égale à 1. Les vecteurs propres sont trouvés en résolvant cette équation :

$$\begin{aligned} Rv_\alpha &= \lambda_\alpha v_\alpha \\ \Leftrightarrow (\Sigma - \lambda_\alpha I)v_\alpha &= 0 \end{aligned}$$

Nous trouvons un vecteur propre pour chaque $\alpha = 1, \ldots, p$. De ce fait, Nous constatons bien que chaque vecteur propre possède une valeur propre lui correspondant.

Comme les valeurs propres sont ordonnées de manière décroissante, c'est-à-dire que λ_1 possède la plus grande quantité de variance expliquée, alors le vecteur propre lui correspondant v_1 sera la direction qui capture la plus grande quantité de variance dans les données. Les éléments contenus dans ce vecteur sont les coefficients de la première composante principale. Soit k le nombre de composantes principales que l'on aimerait sélectionner, nous pouvons désormais garder les k premiers vecteurs propres, correspondant au k premières valeurs propres les plus grandes. Une fois la matrice V tronquée, nous allons opérer à un changement de notation. La nouvelle matrice des coefficients des composantes principales W de taille $p \times k$ aura comme colonne w_1 représentant le premier vecteur des coefficients sélectionnés, w_2 le second, et ainsi de suite jusqu'à w_k le k-ième vecteur des coefficients sélectionnés.

Désormais, nous pouvons trouver les valeurs des composantes principales u_α où dorénavant $\alpha = 1, \ldots, k$. La formule est la suivante :

$$U = XW \tag{1}$$

où U est la nouvelle matrice des composantes principales de taille $n \times k$ où chaque colonne représente une nouvelle variable u_α et les lignes sont les observations qui correspondent aux données d'origine. Ce que nous venons d'effectuer ici est une projection des variables d'origine vers l'espace réduit des composantes principales où la variance a été préservée du mieux possible. Par cette formulation, nous pouvons définir la matrice W comme étant la matrice des coefficients poids (appelés weights) appliquée à la matrice de données d'origine pour trouver les composantes principales.

Imaginons maintenant que nous sélectionnons l'ensemble des vecteurs propres pour composer la matrice W, elle se retrouve être de nouveau une matrice carré de taille $p \times p$ et

WW^T est égale à I. Alors dans ce cas-ci, nous pouvons réécrire :

$$UW^T = (XW)W^T = X(WW^T) = XI = X$$

où I représente toujours la matrice identité. Dans cette nouvelle formulation, la matrice transposée des weights W^T est multipliée par la matrice des composantes principales afin de trouver la matrice des données d'origine. Les coefficients contenus dans W^T sont appelés ici les loadings. Dans le cadre de ce travail, il est intéressant de faire la différence entre les weights et les loadings car notre objectif d'induire de la parcimonie dans la matrice des coefficients des composantes principales va différer selon la forme que nous choisissons. Diverses méthodes en découlent et peuvent donner des résultats différents (Guerra-Urzola et al., 2021). Nous allons nous étendre encore un peu sur le sujet de l'ACP mais cette fois en ciblant les points qui nous intéressent pour donner des pistes de réflexion pour la suite de ce mémoire.

Dans ce sens, une autre formulation de l'ACP pourra nous intéresser ici :

$$X = TP^T + E \tag{2}$$

Cette formulation proposée par Whittle (1952) est le point de départ de la majorité des méthodes de parcimonie. La matrice des données d'origine X peut être obtenue par la multiplication entre la transposée de la matrice des loadings P^T (qui équivaut à W^T dans le cas d'une matrice des weights) et la matrice T (que nous appelons ici U) qui est appelée la matrice des variables latentes, c'est-à-dire des variables qui ne sont pas directement observées mais plutôt déduites par l'algorithme de l'ACP. En d'autres termes, ce sont les composantes principales. De plus, nous avons ici un terme additionnel E représentant une matrice d'erreurs non-corrélée au premier terme.

Par exemple, dans le cas où nous aimerions trouver le vecteur de la première variable x_1 :

$$x_1 = u_1 w_{1,1} + u_2 w_{2,1} + \cdots + u_p w_{p,1} + \text{Erreur}$$

où u_1 jusqu'à u_p représentent chacune des composantes principales contenues dans la matrice U, et $w_{1,1}$ jusqu'à $w_{p,1}$ les loadings correspondant contenus dans la matrice P^T. La variable du jeu de données d'origine est trouvée en additionnant les produits entre chaque vecteur des composantes principales et chaque élément du vecteur des loadings correspondant à la variable.

La matrice des composantes principales U peut aussi être formulée de la sorte : $U = XW$ où W représente désormais la matrice contenant les coefficients de type weights. L'équation pour trouver la première composante principale prendra donc cette forme :

$$u_1 = x_1 w_{1,1} + x_2 w_{2,1} + \ldots + x_p w_{p,1} + \text{Erreur}$$

Avec le même raisonnement que précédemment, les composantes principales sont trouvées en multipliant chaque vecteur des variables d'origine et chaque élément du vecteur des weights correspondant à la composante principale.

Nous comprenons maintenant clairement le problème de la relation linéaire entre les composantes principales et les variables d'origine. Elles sont toutes des combinaisons linéaires des variables d'origine. Avec des coefficients w non nuls, toutes les variables d'origine entrent en compte lors du calcul d'une composante principale, ce qui rend difficile l'interprétation de celle-ci. On ne voit pas distinctement quels sont les variables qui expliquent le plus la variance préservée dans la composante principale. Par conséquent, la solution que nous allons explorer dans ce document serait d'induire de la parcimonie dans cette équation, c'est-à-dire de rendre un maximum de weights, les w, égaux à 0. Ceux qui seront sélectionnés pour devenir des 0 seront ceux qui correspondent aux variables les moins informatives de la composante principale. De cette façon, les variables les plus importantes pour constituer la composante principale seront mises en avant, et nous pourrons clairement l'interpréter en l'expliquant par tels ou telles variables ayant leurs coefficients non-nuls.

Cette dernière équation prend la forme d'une régression linéaire, ce qui permet d'envisager l'utilisation d'une méthode de régularisation comme LASSO (Least Absolute Shrinkage and Selection Operator) comme première méthode de parcimonie appliquée sur les weights (Zou et al., 2006). Imposer une pénalité de type LASSO permettrait de rendre les coefficients des variables les moins importantes égaux à 0, tout en gardant les plus importantes. Lorsque l'on parle d'importance dans ce document, cela représente les termes capturant un maximum de variance des données d'origine, et donc un maximum d'information retenue. Nous reviendrons en détails sur cette méthode dans la partie SPCA 2.2.

Un autre point de théorie sur l'ACP pouvant être intéressant pour avoir des pistes de méthodes de parcimonie est son lien avec la décomposition en valeurs singulières (Singular Value Décomposition ou SVD). Toute matrice X quelconque, carré ou rectangulaire (nous prendrons pour l'exemple la matrice de taille $n \times p$), peut être décomposée en trois matrices distinctes :

$$X = UDV^T$$

Nous avons U représentant la matrice des vecteurs singuliers de gauche. Attention à ne pas confondre cette matrice U avec celles des composantes principales introduite précédemment, nous y revenons dans le paragraphe suivant. Celle-ci est de taille $n \times n$ et est orthogonale. Ses colonnes représentent les vecteurs propres de la matrice résultante de XX^T. Ensuite D est la matrice contenant les valeurs singulières de la matrice X sur sa diagonale. Une valeur singulière est la racine carrée d'une valeur propre de la matrice X^TX, et elles sont aussi ordonnées de manière décroissante d'importance. La matrice D est de taille $n \times p$ et est donc composée de $\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_p}$ sur la diagonale et 0 pour les autres coordonnées. On peut noter que D peut être une matrice carrée de taille $p \times p$ uniquement

dans le cas où $p > n$. Enfin, V^T est la transposée de la matrice des vecteurs singuliers de droite, avec une taille de $p \times p$ et est aussi orthogonale. Ses colonnes représentent les vecteurs propres de la matrice résultante de $X^T X$.

Le lien avec l'ACP est que les composantes principales peuvent être dérivées de la décomposition en valeurs singulières sans passer par la matrice de corrélation. Nous pouvons écrire

$$X = UDV^T = UP^T \tag{3}$$

La multiplication de U et D de la SVD donne la même matrice que celle des composantes principales U obtenue par l'ACP. Ainsi, V^T représente les loadings comme vu précédemment pour la matrice P^T.

La régularisation peut alors se faire à partir de ces résultats. Nous allons partir de la formulation de la SVD suivante :

$$UDV^T = \sum_{\alpha=1}^{p} \sqrt{\lambda_\alpha} u_\alpha v_\alpha^T \tag{4}$$

Nous pouvons choisir une valeur q se trouvant dans l'intervalle $1 \leq q \leq p$. Cette valeur q va remplacer p dans l'équation précédente 4, et comme $q \leq p$, cela veut dire que nous allons additionner moins d'éléments pour obtenir nos résultats, uniquement les premiers termes qui sont les plus importants. La matrice X^q trouvée par cette méthode sera l'approximation de rang q la plus proche des véritables données X. La parcimonie pourrait être induite grâce à un problème d'optimisation reprenant cette formulation, et en y ajoutant un terme de pénalité (Shen and Huang, 2008). Cette technique sera aussi développée et analysée dans la partie *sPCA-rSVD* 2.3. Maintenant que le rappel sur l'ACP et que quelques intuitions ont été présentées, nous pouvons entrer dans les détails de l'analyse en composantes principales parcimonieuse.

2 Analyse en composantes principales parcimonieuse

Comme nous avons vu dans l'introduction 1, le problème principale de l'ACP est que les composantes principales trouvées sont des combinaisons linéaires de l'ensemble des variables d'origine, ce qui rend difficile leurs interprétations. Par l'ACP parcimonieuse, nous pouvons rendre certains des coefficients égaux à 0, rendant les résultats plus parcimonieux et clairs, ce qui les rend au final plus facile à être interprétés. La composante principale sera plus facilement associée à des groupes de variables. Il n'existe pas une définition universelle de l'analyse en composantes principales appliquant la parcimonie. A l'heure actuelle, ce terme représente un ensemble de méthodes ayant le même but : égaliser à 0 un maximum des coefficients linéaires obtenus comme résultat de l'ACP. Chacune de ces méthode est alors une extension de l'ACP classique, permettant une meilleure interprétation de ses résultats. Comme vu au préalable dans l'introduction au moment de trouver 2 formulations différentes pour illustrer l'ACP (voir à partir de l'équation 1), nous pouvons classifier ces méthodes en deux groupes distincts : les méthodes rendant les weights parcimonieuses, et celles rendant les loadings parcimonieuses. Nous allons rentrer dans les détails de ces méthodes après un bref historique de l'ACP parcimonieuse.

2.1 Historique

Les premiers travaux dont l'objectif est d'induire de la parcimonie dans les coefficients des composantes principales commencent dans les alentours de 1950. Le problème d'interprétabilité n'a pas encore été relevé à cette époque, mais des pistes de recherche pour ce travail utilisées plus tard commençaient déjà à apparaître. Par exemple, nous avons Kaiser (1958) qui propose une rotation matricielle tenant en compte du critère VARIMAX. Cette méthode s'applique après que les composantes principales soient calculées. En reprenant les notations correspondant à l'équation 2, nous pouvons illustrer cette méthode par l'équation suivante :

$$X = U(Q^{-1})(PQ)^T + E.$$

Une fois les résultats de l'ACP obtenus, une matrice de rotation Q est calculée et est utilisée pour multiplier la matrice des loadings P^T, ainsi que la matrice des composantes principales U en prenant cette fois l'inverse de Q. La matrice Q est calculée par maximisation du carré de la variance des loadings afin d'obtenir des résultats nets (très proche de 1 ou de 0) pour les coefficients de chaque composante principale. En effet, un coefficient proche de 0

correspondra à une composante principale non-informative de nos données d'origine, et pourra être mise de coté pour l'explication de la composante principale.

Le problème d'interprétabilité des composantes principales lorsque l'on fait face à beaucoup de variables a été prouvée par Jeffers (1967) lors de son analyse du jeu de données `pitprops`, qui sont des données revenant souvent dans la littérature comme point de référence. Nous utiliserons ce jeu de données dans la partie application4 afin d'observer le comportement des méthodes sur un cas réel.

Plusieurs travaux sont apparus par la suite, reprenant l'idée de Kaiser de partir des résultats de l'ACP classique pour leur donner une structure plus simple via rotation. C'est le cas de Kiers (1994) qui propose un autre critère d'optimisation pour la matrice de rotation qui est la méthode SIMPLIMAX, ou la maximisation de la simplicité dans les résultats. La matrice de rotation aura cette fois-ci une forme oblique où la place des 0 sera méticuleusement choisie par un algorithme de minimisation de la somme des carrés des valeurs absolues des loadings, dans le but d'obtenir la forme la plus simple.

Cadima and Jolliffe (1995) montrent ensuite que peu importe la méthode de rotation, les coefficients des composantes principales trouvés peuvent être confrontés à un seuil. Si le loading en question est sous ce seuil, alors il peut être négligé dans l'écriture final des résultats, les rendant parcimonieux. Cette technique représente malgré tout certaines limites développées dans leur article.

L'année suivante, Tibshirani (1996) montre que la méthode de sélection de variables LASSO peut être utilisée afin d'obtenir des résultats parcimonieux. Cette idée sera développée plus tard pour l'analyse en composantes principales lorsque le problème pourra être écrit comme une régression. En attendant, Vines (2000) est le premier à ne pas partir des résultats obtenus par l'ACP classique, mais plutôt par la matrice de covariance. Il en dégage les composantes simples qui sont les vecteurs propres associés à des loadings entiers. Ces coefficients prendraient uniquement la forme de 1, 0 ou -1, ce qui simplifie les résultats.

La méthode du LASSO va être utilisée par Jolliffe et al. (2003) pour induire de la parcimonie sur les coefficients non pas des composantes principales, mais sur celles des variables des données d'origine, donc plus sur les loadings mais sur les weights. Cette première méthode de ce type sera nommée SCoTLASS Jolliffe et al. (2003) et sera travaillée par Zou and Hastie (2005), mais elle ne s'est pas réellement montrée efficace et des améliorations ont vite fait leurs apparitions. En effet, c'est dans ce dernier travail que la méthode de l'Elastic Net sera proposée. Elle regroupe tous les avantages de la méthode précédente, mais est capable de sélectionner en plus des groupes de variables corrélées entre elles. Elle montre également de meilleurs résultats, surtout dans des cas où le nombre de variables surpasse le nombre d'observations. Un algorithme de calcul efficace y est également proposé nommé LARS-EN. Mais ce n'est que l'année suivante où les mêmes auteurs, associés à Tibshirani cette fois-ci, vont appliquer leur méthode sur le problème de l'ACP, qui est maintenant vu comme un problème de régression (Zou et al., 2006). Ils vont nommer cette application la

Sparse Principal Component Analysis (SPCA) et il s'agit d'une méthode de régularisation de type Elastic Net où les pénalisations LASSO et Ridge sont toutes deux utilisées afin de sélectionner les termes les plus importants et d'obtenir des weights plus parcimonieux.

D'autres méthodes apparaissent par la suite, se basant non pas sur un principe de pénalisation des coefficients, mais sur une contrainte de cardinalité (d'Aspremont et al., 2008). C'est à dire que l'on va fixer dès le début le nombre de coefficients non-égaux à 0 dans nos résultats. Cette méthode va révéler une meilleur moyenne de performance que pour la méthode de la SPCA vue précédemment (Guerra-Urzola et al., 2023). Cette conclusion se base sur l'analyse des résultats de leurs algorithmes, que ce soit en terme de capacité à sélectionner les composantes principales les plus importantes, ou bien la précision des estimations des weights.

De retour sur les loadings, Shen and Huang (2008) proposent une nouvelle méthode induisant de la parcimonie par la forme tronquée de l'ACP formulée grâce à la décomposition en valeurs singulières. L'équation de la SVD vue dans l'introduction (voir équation 4) est le point de départ de cette méthode basée sur l'approximation par SVD régularisée à rang plus faible, de ce fait avec uniquement les premières composantes principales les plus importantes extraites.

Journée et al. (2010) proposent aussi une nouvelle méthode appliquée sur les weights. Elle est appelée la *generalized power method for SPCA* et nous la développerons ultérieurement dans le chapitre 2.4. Ces deux dernières techniques présentées se voient être les plus efficaces selon Guerra-Urzola et al. (2021) où leurs résultats sont apparus après analyse de quatre types de performance sur différents jeux de données.

Pour terminer, un bon nombre de nouvelles méthodes apparaissent relativement récemment dans la littérature, où chacune tente de résoudre le problème à partir d'un point de vue différent, comme la *Sparse Probabilistic Principal Component Analysis* qui va utiliser des probabilités bayesiennes pour vérifier l'importance des coefficients des composantes principales (Guan and Dy, 2009). Nous avons aussi la *SPCA via Iterative Tresholding* (MA, 2013) ou encore la *Joint Sparse Principal Component Analysis* (JSPCA) qui est une méthode plutôt robuste, sélectionnant dès le début des groupes de variables ayant les mêmes caractéristiques, ce qui aura tendance à relaxer la contrainte d'orthogonalité imposée par l'ACP (Yi et al., 2017).

2.2 SPCA

Analyser toutes ces techniques représenterait un travail colossal, c'est pourquoi uniquement les méthodes les plus intéressantes à comparer sont développées dans ce mémoire. Les méthodes les plus anciennes, comme la rotation matricielle avec seuil, ne sont pas reprises

car elles sont les bases des méthodes les plus récentes qui sont des améliorations de celles-ci. Même raisonnement pour les premières techniques appliquées aux weights comme SCoTLASS qui se montre peu performante et a un trop haut coût de calcul (Zou et al., 2006). Selon les simulations et applications menées par Guerra-Urzola et al. (2021), les méthodes *sPCA via regularized SVD* et *GPower* se sont retrouvées être les plus performantes. Dans ce mémoire, j'aimerai valider ces résultats en les comparant avec la méthode de l'ACP classique et la méthode *SPCA via Elastic Net regularization* qui a certes obtenu une moins bonne performance globale dans le travail de Guerra-Urzola et al. (2021), mais qui reste une méthode établie et incontournable lorsque l'on parle de parcimonie dans la structure des composantes principales.

La SPCA est une technique proposée par Zou et al. (2006) partant du principe que l'ACP peut être écrite sous forme d'un problème de régression, et donc qu'une pénalisation de type LASSO peut être appliquée sur les coefficients de régression (les weights) grâce à l'Elastic net, ce qui induira de la parcimonie dans les résultats.

Nous allons dans un premier temps observer le problème d'optimisation de cette méthode SPCA. Avec les matrices $A_{p\times k} = [\alpha_1, \ldots, \alpha_k]$, $B_{p\times k} = [\beta_1, \ldots, \beta_k]$ (les vecteurs α_k et β_k sont de tailles p) et le vecteur x_i de taille p de l'observation i du jeu de données d'origine centré et potentiellement réduit, nous avons

$$(\hat{A}, \hat{B}) = \underset{A,B}{\arg\min} \sum_{i=1}^{n} \|x_i - AB^T x_i\|^2 + \lambda \sum_{j=1}^{k} \|\beta_j\|^2 + \sum_{j=1}^{k} \lambda_{1,j} \|\beta_j\|_1 \tag{5}$$

$$\text{sous contrainte que } A^T A = I_{k\times k}. \tag{6}$$

Voyons ce que cela représente. Nous pouvons observer que ce problème d'optimisation tente de trouver une estimation des deux matrices A et B en voulant minimiser la somme de 3 grands termes distincts. L'astuce de la SPCA réside dans le fait que la matrice des weights est vue comme une matrice de projection permettant de trouver une approximation des données d'origine une fois multipliée par celles-ci. Cette matrice de projection est calculée par le produit AA^T, ce qui est démontré dans les travaux de Hastie et al. (2001). Nous avons ici AB^T qui représente ce terme, où A et B sont censés être égaux, mais l'article de référence nous montre que même si ces deux paramètres sont différents, nous pouvons retrouver exactement les k premiers loadings des composantes principales dans le cas où nous considérons uniquement k composantes principales.

Le produit $AB^T x_i$ représente cette même observation approchée par projection linéaire. En minimisant la somme des normes l_2 de la différence entre x_i et ce dernier terme, nous obtenons la meilleure projection possible en terme de fidélité aux données d'origine, ce qui nous donne une solution représentant les k premières composantes principales. La norme

l_2 est la norme Euclidienne et est calculée par :

$$\|x_i\|^2 = \sqrt{\sum_{j=1}^{p} x_{ij}^2}$$

qui est une distance que l'on souhaite minimiser afin de garder un maximum d'information.

Les deux termes suivant de ce problème d'optimisation (équation 5) sont des termes de pénalisation permettant d'induire de la parcimonie dans les résultats. Le terme $\lambda \sum_{j=1}^{k} \|\beta_j\|^2$ représente la pénalisation de Ridge. Ce type de pénalisation n'intervient pas dans le processus d'induction de 0 dans les coefficients des composantes principales, mais sert plutôt à assurer la bonne reconstitution des composantes principales lorsque nous sommes dans le cas où le nombre de variables p est supérieur au nombre d'observations n dans la matrice de données d'origine. Si c'est le cas, nous fixons $\lambda > 0$, sinon la pénalisation de Ridge a peu d'intérêt et on fixerait λ à 0.

Le terme $\sum_{j=1}^{k} \lambda_{1,j} \|\beta_j\|_1$ représente quant à lui la pénalisation LASSO qui va effectivement permettre de fixer certains coefficients à 0. Il est intéressant de noter que plusieurs valeurs peuvent être prises en compte pour les λ_j durant le calcul de minimisation, ce qui permet d'ajuster la pénalisation de type LASSO afin de contrôler la parcimonie induite sur les weights. Plus le terme λ_j sera grand, plus le nombre de coefficients des composantes principales égaux à 0 sera grand. Enfin, la contrainte $A^T A = I_{k\times k}$ signifie simplement que les k colonnes de A sont orthonormées.

Pour mieux comprendre, il est intéressant de voir comment nous en sommes arrivés là. Comme expliqué dans le paragraphe précèdent, l'astuce de la SPCA pour arriver à ce problème d'optimisation est donc bien une complexification du problème de régression. Le terme représentant la matrice des weights sera remplacé par le produit matricielle AB^T. L'article de référence de la méthode SPCA fait l'hypothèse que $\alpha = \beta$, nous pouvons écrire le problème de base, dans le cas où nous considérons uniquement la première composante principale, sous cette forme :

$$(\hat{\alpha}, \hat{\beta}) = \underset{\alpha,\beta}{\arg\min} \sum_{i=1}^{n} \|x_i - \alpha\beta^T x_i\|^2 + \lambda \|\beta\|^2$$
$$\text{sous contrainte que } \|\alpha\|^2 = 1 \text{ et } \alpha = \beta.$$

Pour un $\lambda \geq 0$, permettant de bel et bien prendre en compte la pénalisation de Ridge, ce qui est utile pour appliquer la méthode SPCA à tout type de données. Cette forme ne va calculer que les coefficients faisant référence à la première composante principale. Nous pouvons donc généraliser cette équation en prenant en compte cette fois-ci les k premières

composantes principales :

$$(\hat{A},\hat{B}) = \underset{A,B}{\arg\min} \sum_{i=1}^{n} \|x_i - AB^T x_i\|^2 + \lambda \sum_{j=1}^{k} \|\beta_j\|^2$$
$$\text{Sous contrainte que } A^T A = I_{k\times k} \text{ et } A = B$$

où A et B ont pu être décrits plus haut. Désormais, lorsque l'on voudra résoudre ce problème en trouvant B avec un A fixé, nous ferons face à un problème de régression où la pénalité de LASSO pourra être utilisé, permettant d'obtenir un nombre réduit de coefficients non-nuls.

La résolution de ce problème d'optimisation va se faire par un algorithme d'alternance entre A et B, où l'on va d'abord mettre à jour B grâce à un A fixé. Ceci prend bel et bien la forme d'un problème de régression et peut être estimé par la méthode de l'Elastic net :

$$\hat{\beta}_j = \underset{\beta_j}{\arg\min} \|Y_j^* - X\beta_j\|^2 + \lambda\|\beta_j\|^2 + \lambda_{1,j}\|\beta_j\|_1$$

où $\hat{\beta}_j$ représente une colonne de la matrice $\hat{B} = [\hat{\beta}_1,\ldots,\hat{\beta}_k]$ où k représente toujours le nombre de composantes principales que l'on désire garder. Ce vecteur, une fois normalisé par $\frac{\beta_j}{\|\beta_j\|}, j = 1,\ldots,k$, va donner le vecteur des weights parcimonieux $\hat{V}_j$. Nous avons aussi $Y_j^* = X\alpha_j$ qui est une simple transformation de X multipliée par la colonne α_j, un peu comme une variable réponse obtenue par les variables explicatives et leurs coefficients. La résolution de ce problème d'Elastic net peut être accomplie grâce à l'algorithme LARS-EN (Zou and Hastie, 2005).

Avec le $\hat{B}$ trouvé, nous pouvons désormais le fixer et mettre à jour A. Cette mise à niveau prend la forme d'un problème de rotation de Procrustes (Golub and Loan, 2013). L'objectif est de minimiser cette équation :

$$\sum_{i=1}^{n} \|x_i - AB^T x_i\|^2 = \|X - XBA^T\|^2$$
$$\text{Sous contrainte que } A^T A = I_{k\times k}$$

Ce problème de rotation de Procrustes est de rang réduit car encore une fois nous considérons les k premières composantes principales. La solution peut être trouvée grâce à la décomposition en valeurs singulières du produit matriciel $(X^T X)B$ qui est équivalant à UDV^T. Dans ce cas, $\hat{A}$ est égale à UV^T.

Nous pouvons résumer ces étapes en présentant l'algorithme 1 (Zou et al., 2006).

Algorithm 1 SPCA

Entrées : Matrice de données $X \in R^{n \times p}$
Termes de pénalisation de Ridge λ et de LASSO $\lambda_{1,j}$
Sortie : Matrice des weigths parcimonieuse des k premières composantes principales $\hat{V}_j \in \{-1, \ldots, 1\}^{p \times k}$

1. $A \leftarrow V[, 1:k]$, la matrice à p lignes et k colonnes contenant les weights des k premières composantes principales.

2. Etant donné que $A = [\alpha_1, \ldots, \alpha_k]$ est fixé, nous pouvons résoudre le problème d'Elastic net suivant pour $j = 1, 2, \ldots, k$
$\beta_j = \arg\min_\beta (\alpha_j - \beta)^T X^T X (\alpha_j - \beta) + \lambda \|\beta\|^2 + \lambda_{1,j} \|\beta\|_1$

3. Pour $B = [\beta_1, \ldots, \beta_k]$ fixé maintenant, nous pouvons calculer la décomposition en valeurs singulières $(X^T X)B = UDV^T$, et ensuite mettre à jour $A = UV^T$.

4. On répète les étapes 2 et 3 jusqu'à convergence.

5. Normalisation des résultats : $\hat{V}_j = \frac{\beta_j}{\|\beta_j\|}, j = 1, \ldots, k$ où $\hat{V}_j$ sont les weights parcimonieux des k premières composantes principales.

2.3 sPCA - rSVD

La méthode *Sparse Principal Component Analysis via regularized low raw rank matrix approximation* ou *via regularized singular value decomposition* est proposée par Shen and Huang (2008), et a été introduite à la fin de la partie introductive de ce mémoire (voir partie 1.2). Elle va en effet utiliser la connexion entre l'ACP et la SVD pour obtenir un vecteur de loadings plus parcimonieux. Nous partons donc de l'équation $X = UDV^T$ où nous avons vu que UD représente la matrice des composantes principales, et V^T la matrice contenant les loadings. Grâce à la propriété de la diagonale de la matrice des valeurs singulières disant que $d_1 \geq d_2 \geq \ldots \geq d_r \geq 0$, où chaque d représente une valeur singulière et r est le rang de la matrice X, nous pouvons écrire l'équation de l'approximation de la matrice X par sa SVD d'un rang inférieur :

$$X^l \simeq \sum_{k=1}^{l} d_k u_k v_k^T$$

où k est désormais un indice représentant une composante principale parmi le nombre totale de composantes principales représenté par l où $l \leq r$ est le nouveau rang de la matrice approximée par SVD X^l. Tout comme pour la méthode SPCA, nous voulons garder un maximum d'information après l'approximation, nous allons donc ici utiliser la distance de Frobenius au carré car nous sommes dans un cas matriciel, et non vectoriel. La distance de Frobenius se calcule d'une manière similaire à une distance euclidienne pour un vecteur,

mais prend en compte le fait que l'on travaille avec une matrice de tel sorte que :

$$\|X\|_F^2 = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{p} x_{i,j}^2}$$

où chaque élément de la matrice sera pris en compte.

Imaginons que l'on souhaite trouver la meilleure approximation de la matrice X de rang $l = 1$, nous aimerions minimiser la distance de Frobenius au carré entre la matrice de données X et son approximation qui, dans le cas où l'on sélectionne uniquement la première composante principale, prendra la forme uv^T. Dans ce cas-ci, u est un vecteur de taille n (le nombre d'observations et v^T est quant à lui un vecteur de norme $\|v\| = 1$ et de taille p (le nombre de variables). Le problème d'optimisation prend donc cette forme :

$$(u,v) = \underset{u,v}{\arg\min} \|X - uv^T\|_F^2$$
$$\text{sous contrainte que } \|v\| = 1.$$

La contrainte de ce problème d'optimisation complique les choses pour notre objectif final, qui est d'induire de la parcimonie dans le vecteur des loadings v. Ce vecteur a une norme égale à 1, ce qui rend rigide les composantes du vecteur, et il n'est pas efficace d'y ajouter un terme de pénalisation. Cependant, nous pouvons transformer u et v^T en une forme réévaluée $\tilde{u}$ et $\tilde{v}^T$ où la contrainte de la norme unitaire est désormais appliquée sur $\tilde{u}$. Dans cette configuration, les composantes de $\tilde{v}^T$ pourront être modifiées, voir réduites par une pénalisation. Le vrai problème d'optimisation prend par conséquent cette forme :

$$(\hat{u},\hat{v}) = \underset{\tilde{u},\tilde{v}}{\arg\min} \|X - \tilde{u}\tilde{v}^T\|_F^2 + P_\lambda(\tilde{v}) \tag{7}$$
$$\text{sous contrainte que } \|\tilde{u}\| = 1 \tag{8}$$

où le premier terme est toujours une représentation de la distance de Frobenius, et le second terme $P_\lambda(\tilde{v}) = \sum_{j=1}^{p} \lambda_j(|\tilde{v}_j|)$ est le terme de pénalisation appliqué sur le vecteur des loadings $\tilde{v}$ (souvent une pénalisation de type LASSO, mais d'autres peuvent être appliquées). Le terme λ est un hyperparamètre contrôlant à quel point la parcimonie est induite. Afin de retrouver le vecteur unitaire des loadings, nous appliquons la formule de normalisation $v = \frac{\hat{v}}{\|\hat{v}\|}$ d'une façon similaire à la SPCA.

Pour résoudre ce problème d'optimisation, Shen and Huang (2008) proposent un algorithme itératif incluant un seuil à chaque itération. Après avoir calculé la décomposition en valeurs singulières de notre matrice, nous pouvons optimiser $\hat{u}$ et $\hat{v}$ en les mettant à jour avec l'un ou l'autre fixé alternativement. D'abord pour trouver $\tilde{v}$ avec u fixé, nous pouvons appliquer sur le vecteur $X^T\tilde{u}$ une fonction de seuil h_λ, où ce seuil sera ici de type soft avec comme formule $h_\lambda = \text{sign}(y)\max(|y| - \lambda, 0)$. A partir de quelques transformations du problème

d'optimisation (détaillées par Shen and Huang (2008)), nous avons

$$\arg\min_{\tilde{v}} \tilde{v}_j^2 - 2(X^T\tilde{u})_j\tilde{v}_j + P_\lambda(|\tilde{v}_j|) \tag{9}$$

où $j = 1,\ldots,p$. Le $\tilde{v}_j$ optimal sera trouvée en appliquant le fonction h_λ venant d'être définie sur le vecteur $X^T\tilde{u}$ du problème d'optimisation 9. Pour trouver le nouveau $\tilde{u}$ avec le nouveau $\tilde{v}$ cette fois-ci fixé, nous pouvons appliquer la formule :

$$\tilde{u} = \frac{X\tilde{v}}{\|X\tilde{v}\|}$$

Voici le pseudo-code de l'algorithme 2 (Shen and Huang, 2008).

Algorithm 2 sPCA-rSVD

Entrées : Matrice de données $X \in R^{n\times p}$
Terme de pénalisation λ
Sortie : Matrice des loadings parcimonieuse des k premières composantes principales $v \in \{-1,\ldots,1\}^{p\times k}$

1. Initialisation en appliquant la décomposition en valeurs singulières du jeu de données d'origine X. On obtient sa meilleure approximation de rang 1 de forme $d\hat{u}\hat{v}^T$ où $\hat{u}$ et $\hat{v}$ sont des vecteurs unitaires. Nous avons $\tilde{v}_{old} = d\hat{v}$ et $\tilde{u}_{old} = \hat{u}$.

2. On met à jour $\tilde{v}_{new} = h_\lambda(X^T\tilde{u}_{old})$ et $\tilde{u}_{new} = \frac{X\tilde{v}_{new}}{\|X\tilde{v}_{new}\|}$

3. On répète l'étape 2 en remplaçant les anciennes valeurs par les nouvelles jusqu'à convergence.

4. On standardise le final $\tilde{v}_{new}$ pour obtenir le vecteur correct des loadings plus parcimonieux par $v = \frac{\tilde{v}_{new}}{\|\tilde{v}_{new}\|}$.

2.4 GPower

Selon Guerra-Urzola et al. (2021), cette méthode de parcimonie se montre être la plus performante d'entre toutes, et c'est ce que nous vérifierons dans les parties suivantes. La GPower est développée par Journée et al. (2010) et comporte en fait 4 versions différentes se basant sur le même principe. Les 2 premières vont calculer la matrice des weights des composantes principales un vecteur à la fois. D'abord la composante principale avec la plus grande valeur propre, et ensuite les suivantes par déflation, c'est-à-dire la composante principale la plus importante sans compter celles qui ont été sélectionnées auparavant, en retirant la variance expliquée par les premières composantes principales de la variance totale du jeu de données. Les 2 suivantes sont des extensions des premières et vont quant à elles opérer sur toutes les composantes principales à la fois, en un bloc.

Chacune de ces méthodes vont toutes deux avoir des variantes au niveau du type de régularisation, soit l_1, soit l_0. La régularisation l_1 est une pénalisation se basant sur la somme des valeurs absolues des éléments du vecteur des weights, avec une norme $\|x\|_1 = \sum_{i=1}^{p} |x_i|$. La régularisation l_0 va quant à elle se baser sur le nombre d'éléments non-égaux à 0 dans le vecteur, avec une norme $\|x\|_0$ égale au nombre d'éléments non-égaux à 0, de la même façon qu'expliqué dans le travail de d'Aspremont et al. (2008). Dans l'article de référence de la méthode GPower, il a été montré que les versions en bloc ont légèrement une meilleure performance, surtout au niveau du temps de calcul. L'intérêt s'est porté dans un premier temps sur ces 2 versions lors des simulations, mais par simplicité nous sélectionnerons uniquement les 2 types de méthodes séquentielles qui montrent déjà de meilleures performances que les méthodes vues précédemment (voir la section 3.4 sur l'analyse des performances). L'explication de la méthode en bloc avec régularisation l_1 peut tout de même être trouvée à l'annexe A

Pour le premier cas (vecteurs des coefficients trouvés uns par uns et régularisation l_1), la méthode GPower va en fait consister à calculer une matrice des weights grâce à une nouvelle matrice que l'on va appeler schéma de parcimonie P de taille $p \times k$ à partir d'un problème d'optimisation où une fonction de seuil de type soft sera appliquée. Cette matrice aura une forme final de même taille que la matrice des weights, mais elle sera construite séquentiellement et aura d'abord la forme d'un vecteur p où chaque élément correspondant à un coefficient de la première composante principale aura une valeur de soit $p_j = 1$, soit $p_j = 0$, où $j \in \{1, \ldots, p\}$ correspondant au nombre de variables. Le deuxième vecteur ajouté à la matrice correspondra aux labels des weights de la deuxième composante principale et ainsi de suite. L'astuce ici est qu'une fois cette matrice remplie de 0 et de 1 obtenue, nous pouvons appliquer normalement l'analyse en composantes principales sur nos données, et la matrice des véritables weights parcimonieux $\hat{W}$ de taille $p \times k$ seront celles de l'ACP où les valeurs en communs avec un 0 seront négligées. Autrement dis, nous allons reprendre la matrice du schéma de parcimonie P et nous allons remplacer tous les éléments égaux à 1 par le coefficient calculé par l'ACP ayant le même indexe, les autres éléments restent à 0. Comme les notations de l'article de référence (Journée et al., 2010) sont plutôt complexes, nous allons utiliser celles reprises dans les articles Guerra-Urzola et al. (2021) et Verhoef (2021).

Nous allons commencer par définir l'équation de départ de la méthode GPower. Nous tentons de trouver $\hat{z}$ qui est un vecteur intermédiaire de taille n permettant de trouver le vecteur correspondant de la matrice P. Pour ce faire, nous allons définir une fonction S, qui tient pour soft, et qui est la fonction de seuil appliquant la régularisation aux weights. Elle va fonctionner de cette façon : $S(X^T z, \lambda) = \max(0, |X^T z| - \lambda)$. Tout élément négatif produit au sein de cette fonction, donc un coefficient inférieur au seuil λ, sera d'office fixé à 0. Le terme $|X^T z| - \lambda$ donnera les véritables valeurs des weights une fois le z optimal

trouvé. Le problème d'optimisation permettant de trouver le $\hat{z}$ optimal prendra cette forme

$$\hat{z} = \underset{\|z\|\leq 1}{\arg\max} \sqrt{\sum_{j=1}^{p} S(x_i^T z, \lambda)^2} \tag{10}$$

Comme nous sommes dans le cas d'une régularisation l_1, il est important que le terme de pénalisation λ soit inférieur à la plus grande des normes d'une des colonnes de X afin d'éviter d'obtenir une matrice des weights finale remplie de 0. Dès que $\hat{z}$ est obtenu après normalisation, nous pouvons nous concentrer sur l'équation de $\hat{w}$ qui est l'estimation du vecteur représentant ces véritables weights. Il est de base calculé par le problème d'optimisation suivant :

$$\hat{w} = \underset{\|w\|=1}{\arg\max} \|Xw\| - \lambda \|w\|_1 \tag{11}$$

Cette forme représente la maximisation de la norme du vecteur des composantes principales et peut être réécrite afin de pouvoir utiliser $\hat{z}$, et implicitement le schéma de parcimonie, dans le calcul des weights :

$$\hat{w_j} = \frac{\text{sign}(x_j^T z) S(x_j^T z, \lambda)}{\|S(x^T z, \lambda)\|}, j = 1, \ldots, p \tag{12}$$

où $\text{sign}(x_j^T z)$ représente le signe du coefficient et $S(x_j^T z, \lambda)$ est l'élément p_j du schéma de parcimonie prenant en compte ou non ce coefficient. Chaque élément du vecteur $\hat{w}$ est calculé de cette façon, avec comme résultat soit 0, soit le coefficient final de la composante principale qui a été normalisée. La normalisation est faite par le dénominateur de l'équation où la norme du résultat de la fonction seuil est calculée par :

$$\|S(x^T z, \lambda)\| = \sqrt{\sum_{h=1}^{p} S(x_h^T z, \lambda)^2}.$$

Ce vecteur est indexé à la matrice $\hat{W}$ où chaque colonne correspond à un de ces vecteurs, une pour chaque composante principale sélectionnée.

De manière pratique, nous avons constaté 2 étapes distinctes : d'abord trouver le vecteur intermédiaire $\hat{z}$ afin d'obtenir une colonne du schéma de parcimonie P, et ensuite utiliser ces éléments en combinaison avec la matrice des données d'origine X afin de trouver le vecteur weights $\hat{w}$.

Pour la première partie, nous commençons par initialiser le vecteur z de façon à ce que la première colonne de P contienne au moins un élément égale à 1. Pour ce faire, z initial est trouvé par :

$$z = \frac{x_{j^*}}{\|x_{j^*}\|} \tag{13}$$

avec $j^* = \arg\max_j \|x_j\|$, ce qui permet de ne pas dépasser la plus grande des normes d'une des colonnes de X, ce qui entraînerait des vecteurs de weights remplis de 0 dans le même

esprit que la limite supérieur de λ. Ensuite, $\hat{z}$ est calculée par la formule que nous avons vu précédemment, et elle est recalculée jusqu'à convergence, c'est-à-dire jusqu'à ce que la différence entre le nouveau et l'ancien $\hat{z}$ soit négligeable. Une fois $\hat{z}$ obtenu, il ne reste plus qu'à trouver P tel que chaque élément est égale à 1 si $|x_j^T z| \geq \lambda$ et 0 sinon. Les algorithmes sont proposés dans l'article de Journée et al. (2010), et voici celui que nous venons d'expliquer (algorithme 3) :

Algorithm 3 Matrice P pour GPower par séquence et régularisation l_1

1. Initialisation :
Matrice de données $X \in R^{n \times p}$
Terme contrôlant la parcimonie $\lambda \geq 0$
$z = \frac{x_{j^*}}{\|x_{j^*}\|}$ où $j^* = \arg\max_j \|x_j\|$

2. $z \leftarrow \sum_{j=1}^{p} S(|x_j^T z| - \lambda) sign(x_j^T z) x_j$

3. $\hat{z} = \frac{z}{\|z\|}$

4. On répète les étapes 2 et 3 jusqu'à ce que $\hat{z}$ atteint un seuil de précision attendu ou que le maximum d'itération est atteint.

5. Construction du vecteur $P^n \in 0, 1$ tel que :
$p_j = 1$ si $|x_j^T z| \geq \lambda$ et $p_j = 0$ sinon.

Pour la seconde étape, nous pouvons utiliser le vecteur ou la matrice P trouvé précédemment afin de calculer la véritable matrice des weights parcimonieux des composantes principales de X. Il n'y a pas besoin d'un algorithme entier pour cette étape, il suffit simplement d'appliquer l'analyse en composantes principales classique par sa forme obtenue par décomposition en valeurs singulières, de prendre la première colonne de la matrice V^T qui correspond aux weights, et calculer $\hat{w}$ de tel sorte que chaque élément égal à 1 soit remplacé par la valeur de v_1 ayant le même indexe. Nous avons de ce fait :

$$\hat{w} = \{0\}^p, \quad \hat{w}_p = v_1,$$

où $\hat{w}_p$ sont les éléments égaux à 1 dans le vecteur $\hat{w}$, trouvé grâce au schéma de parcimonie. Une fois cela fait, nous obtenons le vecteur des coefficients parcimonieux de la première composante principale. Si nous souhaitons sélectionner plusieurs composantes principales, imaginons $k = 2$, alors il faudra répéter le processus précédent mais cette fois-ci en retirant l'information captée par la première composantes principales par déflation. Pour ce faire, la nouvelle matrice de données de départ X sera ajustée en la soustrayant par $X\hat{w}\hat{w}^T$. A ce moment, ce sera la deuxième colonne de V^T qui servira à calculer les nouveaux weights de la nouvelle colonne de $\hat{W}$.

Nous allons maintenant voir une autre façon d'appliquer la méthode GPower qui est la méthode séquentielle avec la régularisation l_0. A la différence de la régularisation l_1, la

limite supérieure des λ permettant de ne pas pénaliser abusément les coefficients va devenir la plus grande des normes d'une des colonne de X mise au carré. Le vecteur intermédiaire $\hat{z}$ est désormais trouvé par :

$$\hat{z} = \arg\max_{z} \sum_{j=1}^{p} S((x_j^T z)^2, \lambda)$$

Et les éléments du vecteur $\hat{w}$:

$$\hat{w_j} = \frac{\text{sign}(S[(x_j^T z)^2, \lambda])x_j^T z}{\sqrt{\sum_{h=1}^{p} \text{sign}(S[(X_h^T z)^2, \lambda])(x_j^T z)^2}}, j = 1, \ldots, p$$

L'algorithme pour trouver les vecteurs $\hat{z}$ et P est l'agorithme 4 suivant:

Algorithm 4 Matrice P pour GPower par séquence et régularisation l_0

1. Initialisation :
Matrice de données $X \in R^{n \times p}$
Terme contrôlant la parcimonie $\lambda \geq 0$
$z = [\frac{x_{j^*}}{\|x_{j^*}\|}]$ où $j^* = \arg\max_j \|x_j\|$

2. $z = \sum_{j=1}^{p} \text{sign}(S[(x_j^T z)^2, \lambda])x_j^T z x_j$

3. $\hat{z} = \frac{z}{\|z\|}$

4. On répète les étapes 2 et 3 jusqu'à ce que $\hat{z}$ atteint un seuil de précision attendu ou que le maximum d'itération est atteint.

5. Construction du vecteur $P^n \in 0, 1$ tel que :
$p_j = 1$ si $x_j^T z)^2 \geq \lambda$ et $p_j = 0$ sinon.

Pour trouver la matrice $\hat{W}$, nous n'avons pas besoin non plus dans ce cas-ci d'appliquer un algorithme pour trouver $\hat{w}$ optimal car ce vecteur ne dépend que du schéma de parcimonie P. Nous trouvons $\hat{w}$ par :

$$w_i = x_i \hat{z} p_i \quad \text{et} \quad \hat{w} = \frac{w}{\|w\|}.$$

Une fois de plus, cette méthode calcule les coefficients des composantes principales séquentiellement, alors si nous souhaitons fixer un k plus grand et obtenir la matrice des weights $\hat{W}$, nous devons reproduire le processus par déflation, avec la matrice des données soustraite par $\hat{z}\|w\|\hat{w}^T$. La matrice $\hat{W}$ finale sera la combinaison des différents vecteurs $\hat{w}$ trouvés après chaque itération.

Nous pouvons aussi étendre cette méthode où les vecteurs $\hat{w}$ ne seront plus calculés séquentiellement pour construire la matrice $\hat{W}$, mais celle-ci sera calculée en une seule fois. Nous ne développerons pas ces méthodes ici, mais quelques explications peuvent être trouvées dans l'annexe A.

Synthèse théorique			
Critères	SPCA	sPCA_rSVD	GPower
Type de coefficients	weights	loadings	weights
Régularisation	l_1 et l_2	l_0, l_1 ou l_2	l_0 ou l_1
Type d'algorithme	en bloc	séquentiel	séquentiel ou en bloc
Principe de l'algorithme	Mise à jour de A et B en alternance	Mise à jour de $\tilde{u}$ et $\tilde{v}$ en alternance	Optimisation de z pour trouver le schéma de parcimonie

Table 1: Tableau synthétique des méthodes de parcimonie.

2.5 Synthèse théorique

Dans cette section récapitulative, un tableau synthétique (table 1) est présenté afin d'avoir une comparaison théorique claire entre les trois méthodes de parcimonie choisies dans les sections 2.2, 2.3 et 2.4. Dans les méthodes existantes, nous avons dès le début fait la distinction entre les méthodes appliquant la parcimonie sur les loadings, c'est-à-dire les coefficients des composantes principales permettant de retrouver les données d'origine, et les méthodes appliquant la parcimonie sur les weights, les coefficients des données d'origine permettant de trouver la matrice des composantes principales. Dans les méthodes sélectionnées, seule la méthode sPCA_rSVD applique la parcimonie sur les loadings. Son problème d'optimisation se base sur la formule permettant de retrouver la matrice des données d'origine, alors que les autres permettent de trouver les composantes principales.

Les problèmes d'optimisation initiaux sont relativement semblables. Pour la SPCA (équation 5), la sPCA_rSVD (équation 7) et la GPower (équation 11), nous avons d'un côté les coefficients des composantes principales, et de l'autre côté une maximisation de variance ou une minimisation de la perte d'information selon la méthode, où un terme de pénalisation est appliqué afin d'obtenir des coefficients parcimonieux. La SPCA inclut une régularisation basée sur la norme l_1 et l_2, la sPCA_rSVD est capable d'utiliser tout type de régularisation, même si elle se concentre plus sur la diminution du rang de l'approximation des données d'origine. La régularisation l_1 est prise par défaut. Finalement la GPower possède une version pour la régularisation l_1 et une par contrainte de cardinalité l_0.

Une autre grande différence que nous pouvons faire est le fait que la matrice des coefficients soit calculée séquentiellement, c'est-à-dire que l'on va commencer par calculer le vecteur correspondant à la première composante principale, celle qui donne le plus d'information, et la suivante est calculée par déflation, sans prendre en compte la variance expliquée

fournie par la première composantes principales. C'est le cas pour la méthode séquentielle du GPower et la *sPCA_rSVD*. L'autre façon de calculer la matrice des coefficients consiste à prendre en compte l'ensemble des vecteurs la composant, ou dit aussi en bloc. Les méthodes GPower en bloc et SPCA résident sur cette manière de faire.

Les algorithmes de la SPCA et de la sPCA_rSVD reposent sur le même principe d'alternance de deux éléments : les matrices A et B pour la SPCA et les vecteurs $\tilde{u}$ et $\tilde{v}$ pour la sPCA_rSVD. Les éléments représentant les weights/les loadings sont ensuite normalisés. La méthode GPower semble différente, mais selon Journée et al. (2010) : *"Although rSVD and GPower were derived differently, it turns out that, except for the initialization and post-processing phases, the algorithms are identical."*. Les coefficients des composantes principales sont en effet trouvés dans les deux cas après normalisation d'un élément que nous avons optimisé jusqu'à convergence.

Pour terminer, nous pouvons relever quelques limitations pour ces méthodes. Les trois méthodes sont fortement sensibles aux choix des paramètres de pénalisation, qui est plutôt complexe. Dans tous les cas, nous ferons face à une perte d'information, mais tout réside dans ce principe de trouver un équilibre entre un minimum de perte d'information et un maximum de parcimonie induite dans les coefficients des composantes principales.

Des résumés théoriques des méthodes vues dans ce mémoire, ainsi que d'autres disponibles dans la littérature, sont présentés dans l'article de Zou and Xue (2018).

3 Simulation

3.1 Méthodologie

Ce qui différencie la partie simulation à celle de l'application est la capacité à jouer librement avec les différents paramètres des fonctions sans être contraint par un jeu de données défini. En effet, les données que nous allons travailler dans cette partie seront des données créées artificiellement sur base d'un processus de génération de données. Une fois cela fait, nous pourrons y appliquer les différentes méthodes vues précédemment afin de dégager des résultats pouvant être comparés.

La méthodologie de cette section s'inspire fortement de celle de Guerra-Urzola et al. (2021) qui est l'article de comparaison de méthodes d'ACP avec parcimonie la plus poussée trouvable à l'heure actuelle. Les points pertinents dans ce travail seront gardés, tout en y apportant des modifications, et en y délaissant ce qui a eu moins d'importance, moins d'impact sur les conclusions finales de l'article. Nous allons donc commencer à la section 3.2 par la génération de données où un algorithme de procédure de génération de données sera présenté. Puis dans la section 3.3 nous entrerons dans la partie simulation en tant que telle où les différentes méthodes que j'ai sélectionné, à savoir l'ACP, la SPCA, la sPCA-rSVD, la GPower avec régulation l_1 et avec régulation l_0, seront appliquées à nos données. Enfin, la dernière section 3.4 servira à présenter les différents critères de performances comme l'erreur relative au carré, le taux de mauvaises identifications, et le pourcentage de variance expliquée, pour finir sur une analyse complète des résultats.

3.2 Génération des données

Pour obtenir des résultats robustes, il est intéressant de générer plusieurs tables de données aux caractéristiques différentes afin d'analyser dans quelles situations une méthode serait plus performante qu'une autre. Voici les différents critères de distinction entre ces tables de données :

- La proportion de parcimonie induite (PS) : Nous pouvons, dès la création du jeu de données, induire nous-même un certain pourcentage de parcimonie dans les vecteurs des loadings ou des weights (qui sont les coefficients des composantes principales comme vu à la fin de l'introduction 1.2) pour vérifier à quel point les différentes méthodes ont un impact pour rendre les coefficients égaux à 0. Cette proportion pourra prendre les valeurs $PS = [0.7, 0.8, 0.9]$.

- Le nombre de variables (J) : Les méthodes de parcimonie de l'ACP sont censées obtenir de bons résultats, même lorsque le nombre de variables est très grand. C'est ce que nous confirmerons ou non par ce critère, prenant les valeurs $J = [10, 100, 1000]$.
- Le nombre d'observations (I): Ici, pour une raison de temps de calcul, il a été décidé de fixer le nombre d'observations de chaque table de données à 200. Dans l'article de référence, la taille des tables de données pouvait prendre les valeurs de 100 et 500, mais à aucun moment cela n'a fait ressortir une différence significative en terme de performance. On fixe donc $I = 200$, mais en réalité, les jeux de données auront une taille de 400 observations car les tables des deux types de données crées par l'algorithme ont été réunies. Nous reviendrons sur ce point juste après les 2 derniers critères.
- Le nombre de composantes principales (K) : Sachant que généralement, ce sont les premières composantes principales qui capturent la plus grande partie de la variance des données d'origine, nous pouvons nous limiter à en sélectionner uniquement un petit nombre, comme $K = [2, 3]$.
- La proportion de variance expliquée (VAF) : Ce critère apparaît uniquement lorsque l'on va créer artificiellement du bruit dans nos données afin qu'elles soient similaires à des données réelles qui ne sont généralement pas lisses avec une proportion de variance expliquée de 100%. Malgré l'initialisation de plusieurs valeurs pour ce critère dans le code, seul un pourcentage de 80% est utilisé dans l'article de référence. La valeur de 95% va de ce fait être prise en compte pour voir si l'on observe une différence ou non. Lorsque l'on va calculer la nouvelle table de données X avec comme $VAF = [0.8, 0.95]$, nous appliquons la formule $X + E * fx$ où E est le vecteur d'erreur de taille J où chaque élément e est généré à partir d'une loi normal de moyenne 0 et d'écart type 1, et fx est la contribution de variance expliquée où $fx = \sqrt{\frac{(\sum_{j=1}^{J}\sum_{i=1}^{I} x_{ij}) * (1-\text{VAF})}{(\sum_{j=1}^{J} e_j) * \text{VAF}}}$.

L'article de référence (Guerra-Urzola et al., 2021) propose trois algorithmes différents afin de générer les données. Le premier va générer des données où la parcimonie a été simulée sur les loadings. Le deuxième va quant à lui induire la parcimonie sur les weights. Enfin, le troisième applique la parcimonie aussi bien sur les loadings que sur les weights. Par simplification, j'ai décidé de ne présenter et ne travailler que ce dernier algorithme reprenant les informations des deux façons de faire afin d'éviter toutes répétitions. Il est à noter que malgré la création de cet algorithme "deux en un", l'article de référence ne va finalement utiliser que les jeux de données où la parcimonie est induite sur les weights. Le code a donc été amélioré afin que l'algorithme joigne les 2 types de tables de données, créant ainsi une base de données de 400 observations regroupant aussi bien les données où les weights sont parcimonieux, ainsi que les loadings.

L'algorithme va commencer par générer I observations de manière aléatoire suivant une loi normale de moyenne 0 et de variance 1 réparties entre les j variables. La décomposition

en valeurs singulières est ensuite effectuée sur les données générées en ne prenant que la version tronquée de rang K. Nous obtenons les matrices U, D et V telles que $X = UDV^T$. Comme vu précédemment avec l'équation 3 de la section 1.2, nous pouvons prendre la matrice V^T comme la matrice contenant les loadings, et donc selon le pourcentage de parcimonie induite *PS* choisi, nous pouvons remplacer les plus petites valeurs de la matrice par des 0.

A partir d'ici, l'algorithme va faire la différence entre deux cas de figure. D'abord le cas où la parcimonie est induite sur les loadings : Sur base de l'équation de Whittle (1952) $X = TP^T + E$, nous pouvons assigner la matrice U à T et VD à P afin que P représente effectivement les loadings et T la matrice des composantes principales orthogonales. Comme nous sommes dans un algorithme où la matrice des weights est aussi prise en compte, cette matrice W est dans ce cas fixé à VD^{-1}.

Le second cas concerne l'induction de parcimonie dans les weights : Ces vecteurs directement contenus dans la matrice V sont normalisés afin d'obtenir les véritables weights préservant le nombre de 0. Le but sera ici de maximiser la variance pour qu'elle se rapproche le plus possible de la proportion de variance expliquée désirée. La matrice des composantes principales T sera calculée par le produit entre la matrice des données d'origine et les weights normalisés XW. Les loadings attendu sont P dans l'équation $X = TP^T$. L'algorithme va terminer par induire de l'erreur artificiellement dans les données afin de représenter des comportements observables dans la vraie vie.

Voici le pseudo-code de l'algorithme 5 de Guerra-Urzola et al. (2021) pour générer des données avec loadings et weights parcimonieux. Le code R modifié se trouve à l'annexe H.

Cet algorithme va concrètement se répéter 36 fois. C'est le nombre totale de combinaisons possibles avec les différentes valeurs des paramètres d'entrées : $3 \times 3 \times 1 \times 2 \times 2$. Le code va de ce fait créer un tableau de taille 36×5 où chaque ligne représente un jeu de données, et les colonnes les différents paramètres. Chaque ligne revient donc à une combinaison différente de paramètres pour un même tableau de données. Nous pouvons observer ci-dessous les 5 premières lignes de ce tableau 2. L'algorithme va parcourir ce tableau pour créer au final un jeu de données par ligne. Pour augmenter la taille de ce tableau dans le but d'obtenir des résultats plus robustes, ce processus est répliqué 30 fois pour obtenir au final 1080 jeux de données.

3.3 Application des méthodes

Maintenant que nous avons nos données, nous pouvons simuler librement les méthodes que nous avons sélectionné. Le script R utilisé dans cette section (Voir annexe I) va prendre les jeux de données un par un et appliquer sur chacun d'eux l'analyse en composantes principales classique (ACP) vue dans la section 1.2, la *Sparse Principle Component Analysis*

Algorithm 5 Génération des données

Entrée : I , J , K, PS, et VAF
Sortie : $X \in \mathbb{R}^{I \times J}$

1. Génération d'un jeu de données X composé de I vecteurs suivant $\mathcal{N}(0_J, I_J)$.

2. Application de la SVD régularisée pour obtenir U, D, et V : $X = UDV^T$.

3. Remplacement des plus petits éléments en valeur absolue de V par des 0, en respectant la proportion de parcimonie PS désirée lors de l'initialisation.

4. Normalisation et orthogonalisation de V, tout en préservant les éléments nulles.

5. Si le modèle repose sur $X = TP^T + E$, alors
$T \leftarrow U$
$W \leftarrow VD^{-1}$
$P \leftarrow VD$

6. Si le modèle repose sur un problème de maximisation de la variance, alors
$W \leftarrow V$
$P = W$
$T = XW$
P est la solution de l'équation $X = TP^T$

7. $X \leftarrow TP^T + f(E)$ avec E contenant I vecteurs suivant $\mathcal{N}(0_J, I_J)$ et f est défini tel que $VAF = \|TP^T\|^2 / (\|TP^T\|^2 + f^2(\|E\|^2))$.

Coefficient des composantes principales					
jeu de données	J	I	PS	K	VAF
1	10	200	0.7	2	0.8
2	100	200	0.7	2	0.8
3	1000	200	0.7	2	0.8
4	10	200	0.8	2	0.8
5	100	200	0.8	2	0.8
...	...	...	...	...	...

Table 2: 5 premières lignes du tableau des différentes combinaisons des paramètres permettant de générer les données.

(SPCA) vue dans la section 2.2, la *sPCA via regularized SVD* (sPCA-rSVD) vue dans la section 2.3 et enfin la méthode GPower avec régularisation l_1 et l_0 vue dans la section 2.4. Une fois cela réalisé, nous pourrons calculer les différents critères de performances sur ces données pour chaque méthode.

Nous allons passer en revue les méthodes une par une. Le code R sera expliqué systématiquement pour voir en détail ce qu'il s'y passe et pour comprendre le raisonnement mathématique derrière. Une référence vers le code source de chaque méthode est disponible à chaque introduction de la méthode.

3.3.1 PCA

L'analyse en composantes principales classique est relativement simple à déployer sur R car elle ne nécessite pas forcément de package particulier à installer. La fonction `prcomp` existe dans la base R, et elle permet d'appliquer directement l'ACP sur la table de données mise en argument. Le code source de cette fonction est disponible sur Github en opensource (SurajGupta, 2016) (Voir annexe D) :

Les arguments de la fonction `prcomp` sont composés de :

- `x` : les données à laquelle on aimerait extraire les composantes principales. Nous pourrons y mettre directement les jeux de données générés précédemment.
- `retx` : initialement fixé sur `TRUE`, il permet de faire sortir la rotation des variables, donc de faire sortir les composantes principales dans l'objet final.
- `center` : permet de centrer les données.
- `scale` : permet de réduire les données en utilisant la fonction `scale`. Il est important que ces deux derniers arguments soient fixé sur `TRUE` car il est nécessaire de centrer les données pour appliquer tous types d'ACP, et nous sommes dans une optique de comparaison des méthodes, il est donc indispensable que les résultats finaux soient sur la même échelle, sinon la comparaison n'aurait pas de sens. C'est bien ce qu'il se passe dans les premières lignes de la fonction, où notre jeu de données est mis à l'échelle grâce à la fonction `scale` de la base R.
- `tol` : ce dernier argument peut quant à lui rester sur `NULL` car il s'agit d'un argument permettant d'omettre certaines composantes principales en fonction de leur écart-type, ce qui ne nous intéresse pas ici. De plus, le nombre de composantes principales que l'on va sélectionner a déjà été fixé selon la table de données que nous avons affaire.

Nous pouvons observer dans la fonction que l'ACP est réalisée grâce à la décomposition en valeurs singulières que nous avons vue lors de l'introduction à la section 1.2. Les valeurs singulières contenues dans la matrice D sont retenues et sont appelées ici les écart-types. La matrice des coefficients des composantes principales est la matrice V. La fonction produit

un objet de type `prcomp` qui contient donc les écart-types, les coefficients des composantes principales, les valeurs des données d'origine après être centrées et après avoir été mises à l'échelle. Enfin, vu que `retx` est fixé sur `TRUE`, nous avons aussi dans l'objet créé `x`, qui sont nos composantes principales obtenues par le produit entre les données d'origine et les coefficients (weights/loadings) des composantes principales.

3.3.2 SPCA

Pour la méthode SPCA, j'ai utilisé la fonction `spca` incluse dans le package `elasticnet` (Zou and Hastie, 2020). La fonction a été créée par les mêmes auteurs que l'article scientifique de la méthode de Zou et al. (2006), ce qui rassure fortement quant à la pertinence de la fonction. Le code source, directement observable sur R, est présent dans l'annexe E.

Passons en revue les arguments :

- `x` : les données d'origine, celles que nous avons donc générées.
- `K` : le nombre de composantes principales que l'on aimerait garder. Nous l'avons fixé plus tôt en fonction du jeu de données (2 ou 3 composantes principales).
- `para` : vecteur dont les éléments dépendent de la valeur de l'argument `sparse` mentionné ci-dessous. Si la valeur `'penalty'` est donnée à l'argument `sparse`, l'argument `para` va prendre la forme d'un vecteur de paramètres de pénalité de norme 1 et de taille `K`. Si la valeur `'varnum'` est donnée à la place de `'penalty'`, l'argument `para` sera dorénavant un vecteur toujours de taille `K`, mais cette fois-ci contenant le nombre de `weights` parcimonieux par composante principale. Dans notre cas, il nous suffit de vérifier le nombre d'éléments non-égaux à 0 dans `P` (la matrice des coefficients des composantes principales du jeu de données correspondant). `'varnum'` est donc choisi en parallèle pour l'argument `sparse`.
- `type` : permet de passer en argument soit `x` la matrice de données en tant que telle si on choisit `'predictor'`, soit la matrice de covariance ou de corrélation que l'on calcule au préalable si on choisit `'Gram'`.
- `sparse` : permet de changer la nature du vecteur de l'argument `para` selon que l'on choisisse `'penalty'` ou `'varnum'`.
- `use.corr` : permet de mettre à l'échelle nos données lorsqu'il va être fixé sur `TRUE`. En effet, nous pouvons constater au début du code source de la fonction, après vérification qu'il n'y ai pas d'erreur d'initialisation, que `x` est centrée réduite grâce à la fonction `scale`, qui prend comme argument `use.corr`.
- `lambda` : le paramètre de pénalisation initialisé à 0.00006.
- `max.iter` : le maximum d'itération si la convergence tarde à arriver, initialisé à 200 itérations.

- `eps.conv` : le critère de convergence initialisé à 0.001. Nous allons voir tout de suite en quoi la convergence consiste. Ces trois derniers arguments ont des valeurs par défaut déjà optimisées pour ce genre de travail, je n'ai donc pas perdu de temps à essayer différentes combinaisons, mais cela pourrait être quelque chose d'intéressant à vérifier.
- `trace` : permet simplement de voir l'évolution du processus dans la console lorsqu'il est fixé sur `TRUE`.

Au sein de la fonction, nous avons vu que les données sont centrées et réduites. Après cela, nous pouvons voir que la décomposition en valeurs singulières est effectuée sur les données. Les matrices A et B sont dès lors initialisés. Elles contiennent respectivement k vecteurs α et k vecteurs β sur leurs colonnes et elles correspondent aux matrices A et B que nous avons décris à l'équation 5 de la section 2.2. La matrice A représente les k premiers weights de la matrice V correspondant aux k premières composantes principales, tout comme expliqué dans l'algorithme 1 de la SPCA (Zou et al., 2006). La matrice B va ensuite être calculée à partir de A et de la fonction `solvebeta` où le code source peut être retrouvé à l'annexe E.1. Cette fonction va se charger de résoudre le problème d'optimisation de l'elastic net que nous avons développé dans la section 2.2.

Après un travail de normalisation, nous pouvons rentrer au coeur de l'algorithme qui est une boucle répétant les mises à jours d'alpha et bêta jusqu'à convergence. La convergence signifie ici une différence assez basse entre la nouvelle valeur de bêta et l'ancienne valeur, pour qu'elle soit inférieur au critère de convergence fixé dès le début, à savoir ici 0.001. De plus, le nombre d'itérations ne doit pas dépasser le maximum d'itérations une nouvelle fois fixé au début, qui vaut ici 200. Pour calculer la différence entre ces bêtas, la fonction `convcheck` est utilisée. Son code source peut aussi être retrouvé à l'annexe E.2. Elle va sélectionner la plus grande différence entre chaque élément de l'ancien et du nouveau bêta. Dans cette boucle, nous retrouvons donc la mise à jour d'alpha avec bêta fixé, qui, comme vu dans la partie théorique de la section 2.2, est le produit entre les matrice u et v extraites de la décomposition en valeurs singulières de la matrice des weights, calculé au préalable par le produit $(X^T X)B$. Les bêtas sont mis à jour de la même façon qu'ils ont été initialisés, c'est-à-dire en passant par la fonction `solvebeta`.

Une fois sorti de la boucle, les bêtas sont une dernière fois normalisés en suivant la formule $\hat{V}_j = \frac{\beta_j}{\|\beta_j\|}$ pour obtenir notre matrice des weights. Un objet de type `spca` va finalement être créé, où les éléments les plus importants qu'il contient sont le nombre de composantes principales sélectionnés, la matrice des weights, le pourcentage de variance expliquée et la variance totale des données d'origine.

3.3.3 sPCA-rSVD

Pour cette méthode, je me suis d'abord dirigé sur la fonction `sPCA_rSVD` du package `itsspca` (Wang and Van Aelst, 2019) qui semble être la plus utilisée, mais malgré une

inspiration du travail de Shen H. et Huang J., le code source de la fonction ne suivait pas exactement l'algorithme proposé dans leur article scientifique. En continuant mes recherches, je suis tombé sur la fonction toute récente `ssvd` du package `irlba` (Baglama et al., 2022) qui rempli exactement ce rôle. Le code source de la fonction `ssvd` se trouve à l'annexe F.

Les arguments de la fonction `ssvd` sont :

- `x` : les données sur lesquelles nous allons travailler.
- `k` : le nombre de composantes principales que l'on veut extraire.
- `n` : le nombre d'éléments non-égaux à 0 dans les vecteurs des loadings.
- `maxit` : le nombre d'itérations maximum possible dans le cas où la convergence tarderai à arriver. Si ce maximum est atteint, la boucle prendra fin, mais cela n'empêchera pas la fonction de fournir une sortie. Un message d'avertissement sera affiché pour prévenir l'utilisateur que la convergence n'a pas été atteinte. Similairement à l'algorithme de la SPCA, la convergence ici correspond à une différence minime entre l'ancienne valeur du vecteur des loadings u et sa nouvelle valeur, suffisamment petite pour être inférieur au seuil `tol` mis en argument.
- `tol` : seuil de convergence initialisé par défaut à 0.001.
- `center` : permet de centrer les données si `TRUE`.
- `scale` : permet de mettre les données à l'échelle en fixant la valeur sur `TRUE` afin que les résultats soient comparables avec les autres méthodes.
- `alpha` : un hyperparamètre utilisé afin d'optimiser la pénalisation car il n'existe pas une unique valeur de λ pour atteindre le nombre d'éléments non-égaux à 0 désiré. Il est initialisé sur 0, ce qui veut dire que la solution obtenue correspondra au mieux aux valeurs singulières.
- `tsvd` : permet de directement proposer la décomposition en valeurs singulières de `x` en argument, ce qui permettrait de ne pas devoir la calculer en interne, mais ce n'est pas obligatoire.

Après avoir centré et réduit les données dans les premières lignes de la fonction, on vérifie que le nombre d'éléments non-égaux à 0 que l'on désire soit conforme avec ce que l'on met en paramètre. Ce nombre doit notamment être inférieur au nombre de colonnes de notre jeu de données, sinon la fonction va simplement donner la décomposition en valeurs singulières de `x` par la fonction `irlba` (Baglama and Reichel, 2005) pouvant être retrouvée à l'annexe F.1. Ce long code source permet de calculer la décomposition en valeurs singulières de la matrice de données en argument, mais contrairement à la fonction `svd` dans R de base qui va calculer l'ensemble des valeurs singulières, où nous pouvons seulement après cela sélectionner les plus importantes, la fonction `irlba` va estimer les valeurs singulières à

partir d'un nombre déjà fixé à l'avance de valeurs singulières désiré. `n` doit aussi être une liste de taille `k`, cet ajustement est fait en interne. Si la SVD n'a pas été donnée au départ, elle est calculée grâce à la fonction `irlba`.

Nous pouvons voir par la suite l'initialisation d'une fonction `soft` qui permet d'appliquer la pénalisation à partir d'un certain seuil de type soft. C'est l'inconvénient de la fonction `irlba`, il n'est pas possible d'appliquer une fonction de seuil d'un autre type, comme par exemple un seuil de type hard.

Après quelques dernières initialisations, nous rentrons enfin dans la boucle où les matrices `u` et `v` résultantes de la SVD vont être mises à jour l'une à la suite de l'autre avec l'un d'eux fixé, jusqu'à convergence similairement à α et β dans la fonction `SPCA` de la section 3.3.2. La différence entre la nouvelle matrice `u` et l'ancienne est ici `delta_u`. Cette variable est calculée en prenant l'élément le plus grand de la diagonale de la matrice résultante du produit croisé entre u_{old} et u_{new}. La matrice `v` est mise à jour avec la fonction de seuil de type soft appliquée aux données, avec `u` fixé et `n` qui rentre en compte pour savoir à quel point les loadings seront pénalisés. La matrice `u` est quant à elle mise à jour en suivant bien l'équation $\tilde{u}_{\text{new}} = \frac{X\tilde{v}_{\text{new}}}{\|X\tilde{v}_{\text{new}}\|}$ avec v_{new} fixé. Si le maximum d'itérations a été atteint, un message d'avertissement sera déployé.

La fonction se termine par la normalisation des résultats et la création d'un objet contenant les valeurs des matrices `u`, `v` et `d` la matrice des valeurs singulières, ainsi que les valeurs des paramètres choisis au préalable.

3.3.4 GPower

Comme expliqué dans la partie théorique à la section 2.4, Verhoef (2021) a repris le travail de Journée et al. (2010) et l'a simplifié, en plus de proposer une fonction R dans son package `gpowerr` afin de rendre la méthode directement applicable. Après analyse de son code source, les étapes de l'algorithme de l'article de référence de la GPower sont effectivement bien respectées. J'ai tout de même opéré quelques modifications dans le code, notamment des adaptations pour se concentrer que sur le cas de la GPower séquentielle (voir annexe G).

Les arguments de cette fonction `gpower` sont composés de

- `data` : la table des données que l'on veut analyser. Peut aussi prendre la forme d'une matrice de corrélation.
- `k` : le nombre de composantes principales que l'on souhaite extraire.
- `roh` : terme permettant de trouver λ le terme contrôlant la parcimonie dans les coefficients des composantes principales, ou appelé aussi la proportion de parcimonie désirée. La valeur que l'on assignera à `roh` sera multipliée par `roh_max`, qui est la plus grande norme parmi les colonnes du jeu de données d'origine, afin de trouver λ.

Dans le package `gpowerr`, il existe une fonction appelée `auto_gpower` permettant de trouver le bon `roh` à partir de la proportion de parcimonie désirée. J'ai de nouveau modifié son code source afin que la fonction ne retourne que la valeur du `roh` optimal, et non un texte explicatif. Cela me permet de placer la fonction `auto_gpower` comme argument pour `roh`, avec la proportion de parcimonie du jeu de données en question. Le code source de cette fonction est aussi disponible à l'annexe G.2.

- `reg` : permet de choisir le type de régularisation que l'on souhaite appliquer sur les coefficients, soit l_1, soit l_0.
- `center` : permet de dire si oui ou non nous aimerions centrer nos données. A noter que la réduction des données se fait ici par la normalisation du vecteur z initialisé.
- `block` : fixé sur `TRUE` si l'on souhaite utiliser les méthodes GPower en bloc.
- `mu` : moyenne appliquée sur chacun des vecteurs des composantes principales lors du calcul de leurs weights. Ces deux derniers arguments ne sont applicables que lorsque l'on souhaite appliquer la méthode GPower en bloc. Si c'est le cas, alors nous pouvons assigner une autre valeur que 1 à `mu`.
- `iter_max` : le nombre maximal d'itérations.
- `epsilon` : un critère de précision régulant le moment de la convergence.

Au sein même de la fonction désormais, nous avons dans un premier temps quelques initialisations d'erreurs, des vérifications sur la qualité des arguments donnés et l'initialisation de l'objet `gpower` qui sera produit par cette fonction. Après avoir centré les données, nous vérifions bien que nous sommes dans un cas séquentiel et l'algorithme se divise dès lors en deux parties distinctes : d'abord pour le cas l_1 et ensuite pour le cas l_0.

Pour le premier cas, on cherche d'abord la plus grande norme parmi les colonnes de la table des données afin de trouver λ. On initialise ensuite z (représenté dans la fonction par `x`) grâce à la normalisation que nous avons vu précédemment par l'équation 13.

Nous rentrons ensuite au sein même de l'algorithme où le $\hat{z}$ optimal sera calculé. La formule 10 sera appliquée et le vecteur résultant sera inclus dans une liste. Si ce vecteur ne contient que des 0, alors une erreur se produit car la pénalisation des coefficients est trop grande. Si tout se passe bien, la méthode du gradient est utilisée afin de se rapprocher du $\hat{z}$ optimal. Une fois ce nouveau z normalisé, nous vérifions que le maximum d'itération n'ait pas été atteint et que la différence normalisée entre le nouveau z et l'ancien soit supérieur à `epsilon`. Si ce n'est pas le cas, alors la boucle s'arrête ici.

Le schéma de parcimonie est ensuite calculé où chaque élément de ce vecteur sera `TRUE` ou `FALSE` selon le résultat de la fonction seuil. L'étape suivante consiste donc à trouver les véritables weights parcimonieux grâce à la fonction du package `gpowerr` `pattern_filling` (voir le code à l'annexe G.1). Cette fonction va utiliser comme argument la table de données d'origine, le schéma de parcimonie calculé et le vecteur $\hat{z}$ optimal normalisé. La

formule de $\hat{w}$ (voir l'équation 12) y sera appliquée, et nous obtiendrons notre vecteur des weights qui sera inclus dans la matrice finale des weights parcimonieux dans le cas où nous sélectionnons plusieurs composantes principales.

Avant de répéter l'algorithme, la déflation est opérée afin que la nouvelle table de données ne contiennent plus l'information captée par les précédentes composantes principales trouvées.

Pour la méthode de régularisation l_0, le code reste globalement le même mise à part les différences mathématiques au sein des fonctions, et la non-utilisation de la fonction `pattern_filling` car le vecteur $\hat{z}$ et l'inverse du schéma de parcimonie vont directement être fusionnées afin de donner le vecteur des weights.

Le code source se termine par l'assignation de différents éléments à l'objet `gpower` comme la matrice des weights, les composantes principales, les données approchées par ces composantes principales, la proportion de parcimonie et la variance expliquée.

3.4 Analyse des performances

Maintenant que les méthodes ont été appliquées sur nos données fictives, il est temps de débuter la comparaison des performances entre ces 5 méthodes, à savoir l'ACP, la SPCA, la sPCA_rSVD, la GPower avec régularisation l_1 et avec régularisation l_0. Nous utiliserons 3 critères différents pour mener à bien ces comparaisons. Ces critères sont repris dans l'article de référence (Guerra-Urzola et al., 2021).

3.4.1 Présentation des critères de performance

Nous avons en premier lieu l'erreur relative au carré (squared relative error - SRE) qui est une mesure permettant de vérifier à quel point il y a eu perte d'information entre deux matrices, par exemple entre la matrice des composantes principales d'un jeu de données et la matrice des composantes principales estimées grâce à la méthode choisie. La formule de la SRE est la suivante :

$$\text{SRE} = \frac{\|\hat{Z} - Z\|_F^2}{\|Z\|_F^2} \tag{14}$$

où Z est la matrice des composantes principales de la matrice de données d'origine X, $\hat{Z}$ est la matrice des composantes principales trouvées à partir de la méthode en question, et la norme de Frobenius est donnée par la formule $\|X\|_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{p} x_{ij}^2$. C'est donc bien un terme d'erreur que l'on aimerait minimiser pour juger d'une bonne reconstitution de la matrice des composantes principales. Une SRE égale à 0 signifierait qu'il n'y a pas eu de perte d'information lors de l'estimation de la matrice en question, alors qu'une SRE élevée signifierait une grande perte d'information, et donc une mauvaise reconstitution dans le cas d'une matrice de composantes principales. Le code R afin de calculer la SRE dans notre cas est donné par :

```
out_error =  RelativeError(Out$W, W1)

W1 = W1[,out_error$permutation]%*%diag(out_error$s)
Z1 = Z1[,out_error$permutation]%*%diag(out_error$s)

ErrorScores = (norm((Out$Z-Z1),type = "F")/norm(Out$Z,type = "F"))^2
```

où `Out$W` est la matrice des loadings ou weights de la table de données générée, `W1` est la matrice des loadings ou weights trouvée par la méthode que l'on a appliqué et la fonction `RelativeError` est une fonction créée par Rosember Guerra dans l'article de référence, le code source peut être trouvé à l'annexe I.1. Elle permet de calculer les différences colonnes par colonnes entre 2 matrices. La fonction va produire un objet contenant l'erreur minimale trouvée, la liste des permutations possible des composantes principales ayant effectivement cette erreur minimale, et le signe de l'erreur. Ces éléments vont être utilisés afin de mettre à jour la matrice des loadings ou des weights `W1`, ainsi que la matrice des composantes principales calculée `Z1`. Enfin, nous pouvons trouver `ErrorScores` qui applique bien la fonction statistique 14 vue précédemment, où les matrices comparées sont celles des composantes principales.

Nous avons ensuite le taux de mauvaises identifications (misidentification rate - MR) qui est un taux entre 0 et 1 montrant la fréquence à laquelle la méthode va effectivement reconnaître les 0 dans les vecteurs des loadings ou des weights dans la structure de notre jeu de données généré où la parcimonie à été induite artificiellement. La formule du MR est la suivante :

$$\text{MR} = 1 - \frac{\text{Nombre d'éléments 0 correctement classifiés}}{\text{Nombre d'éléments 0}} \tag{15}$$

où le numérateur correspond au nombre de 0 retrouvés au même endroit dans les matrices des loadings ou des weights de la table de données générée et de celle reconstituée. Le dénominateur est simplement le nombre totale d'éléments égaux à 0 que l'on désire. Un MR proche de 0 va démontrer une grande habilité à retrouver la même structure de parcimonie dans les loadings/weights alors qu'un MR proche de 1 veut dire que peu de coefficients ont été interprétés comme des 0 par la méthode alors que c'est bien le cas dans notre structure de base. Voici maintenant le code R pour trouver ce critère de performance :

```
ZeroInd = which(as.vector(Out$W)==0)

W1_vec = as.vector(W1)

if (Out$Propsparse == 0){
    MR0 = 0
}else{
    wz = W1_vec[ZeroInd]
    MR0 = 1-sum(wz == 0)/length(ZeroInd)
}
```

`ZeroInd` va stocker l'indice des éléments égaux à 0 de la matrice des coefficients du jeu de données généré. Si nous n'avions pas mentionné de parcimonie désirée dans la génération des données, le MR serait d'office fixé à 1 vu qu'il n'y aura pas d'élément égal à 0 pouvant être retrouvé par une méthode, et donc pas d'erreur possible. Ce n'est pas notre cas vu que l'on induit soit 70%, 80% ou 90% de parcimonie dans les loadings ou weights. C'est pour cela que la suite du code va d'abord sélectionner les éléments de la matrice des coefficients calculée où l'indice correspond à celle où un 0 est présent dans la matrice des coefficients des données générée. Le résultat *wz* est vu ici comme un vecteur. On applique enfin la formule 15 vue précédemment, avec comme numérateur le nombre d'éléments effectivement égaux à 0 sélectionnés plus tôt.

Le troisième critère de performance est le pourcentage de variance expliquée (percentage of explained variance - PEV). C'est bien un pourcentage, une valeur entre 0 et 1, qui permet d'observer à quel point la variance du jeu de données d'origine est bien gardée dans la reconstitution des données sur base des composantes principales calculées. La formule est la suivante :

$$PEV = 1 - \frac{\|\hat{X} - X\|_F^2}{\|X\|_F^2}. \tag{16}$$

Cette formule va nous sortir une valeur correspondant à la variance des données reconstituées, et pour arriver à une conclusion que la méthode est efficace, il faut que cette valeur soit le plus proche possible de la variance expliquée que nous avons initialisé dans nos données générées, qui correspond au VAF, de 80% ou 95%. Un PEV différent de cette valeur montrerait de l'overfitting ou de l'underfitting, c'est-à-dire que la méthode prendrait trop en compte ou pas assez en compte le bruit que nous avons induit. Le code que nous utilisons pour obtenir le PEV est;

```
P = t(Out$X)%*%ginv(t(Z1))

Xs = Z1%*%t(P)

Svar = 1 - (norm((Out2$X-Xs1),type = "F")/norm(Out2$X,type = "F"))^2
```

Nous cherchons dans un premier temps à trouver la matrice des coefficients des composantes principales P en multipliant la matrice des données d'origine avec l'inverse de la matrice des composantes principales $Z1$ trouvées par la méthode choisie. En multipliant cette matrice P par $Z1$, nous obtenons la nouvelle matrice des données reconstituée. Il ne nous reste plus qu'à appliquer la formule précédente 16 pour obtenir le PEV.

3.4.2 Résultats

Ces 3 critères de performance sont calculés pour chacune des 5 méthodes que nous souhaitons comparer, une fois pour chaque jeu de données que nous avons généré. Avec 30 réplications des données, le temps de calcul s'est trouvé fort long, mais le risque d'observer un phénomène irrégulier dans les résultats est désormais minime. Un tableau est produit, où

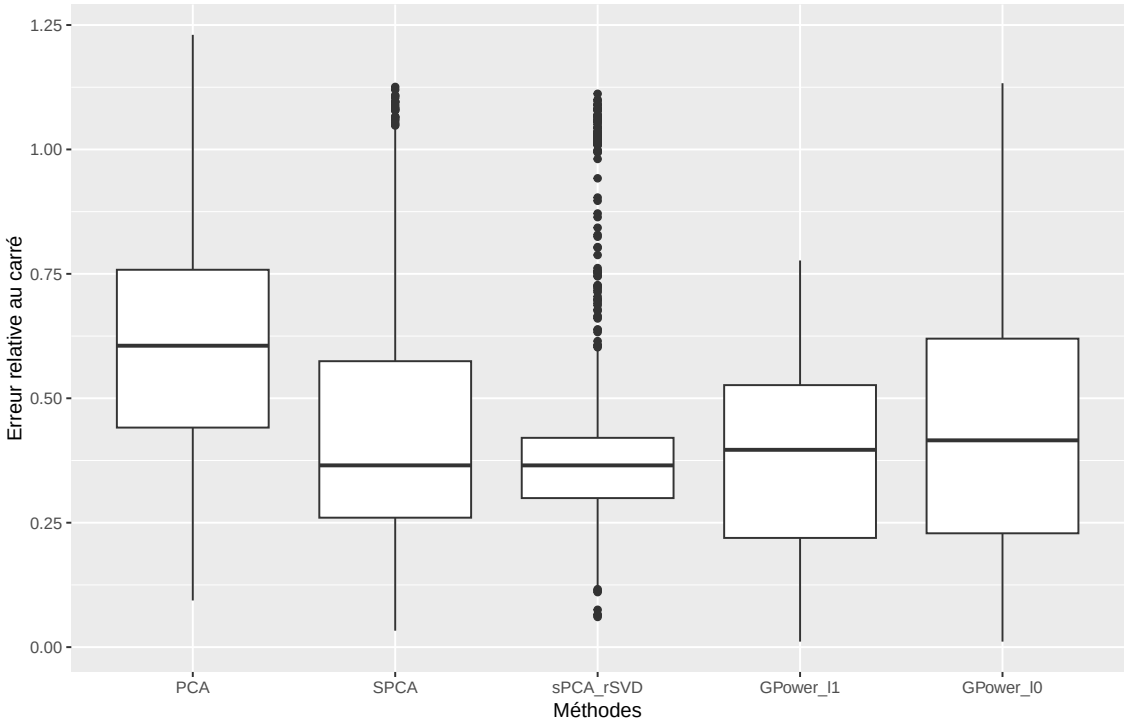

Figure 1: Comparaison de l'erreur relative au carré entre les matrices des composantes principales par méthode.

chaque ligne représente une table de données, et les colonnes les différents critères de performance pour chaque méthode. Le tableau a été retravaillé afin que les résultats soient plus visible, où nous avons maintenant une séparation claire des différentes méthodologies. Sur chaque ligne nous avons une table de données, ses caractéristiques, la méthode appliquée et les performances obtenues. Le code complet des simulations se trouve à l'annexe I.

A partir de ce tableau, nous pouvons en dégager des graphiques afin de nous aider à comparer les résultats. Le type de graphique que j'ai choisi est le boxplot par sa visibilité claire lors de comparaisons. Nous allons d'abord nous pencher sur l'erreur relative au carré (figure 1). Ce premier graphique nous montre une tendance générale de la SRE selon la méthode choisie. Les résultats sont plutôt variables, pouvant se rapprocher très fortement de 0, donc avoir une différence très minime entre les matrices de composantes principales, ou alors dépasser la valeur de 1 montrant une plus grande erreur, mais qui n'est pas aberrante non plus. Chaque méthode est plutôt efficace pour retrouver les composantes principales de la table de données générée. Nous pouvons cependant voir que l'ACP obtient une moins bonne performance avec une moyenne légèrement supérieur à 0.6. Les quatre autres méthodes ont une moyenne assez similaire dans les alentours de 0.375 mais deux sortent du lot : la GPower avec régularisation l_1 qui ne possède pas d'erreurs dépassant 0.8, et la sPCA_rSVD où les valeurs sont fortement rapprochées de la moyenne. Pas mal de valeurs aberrantes sont quand même à observer pour cette méthode.

Il est maintenant intéressant d'approfondir l'analyse afin de juger quelle méthode sera la

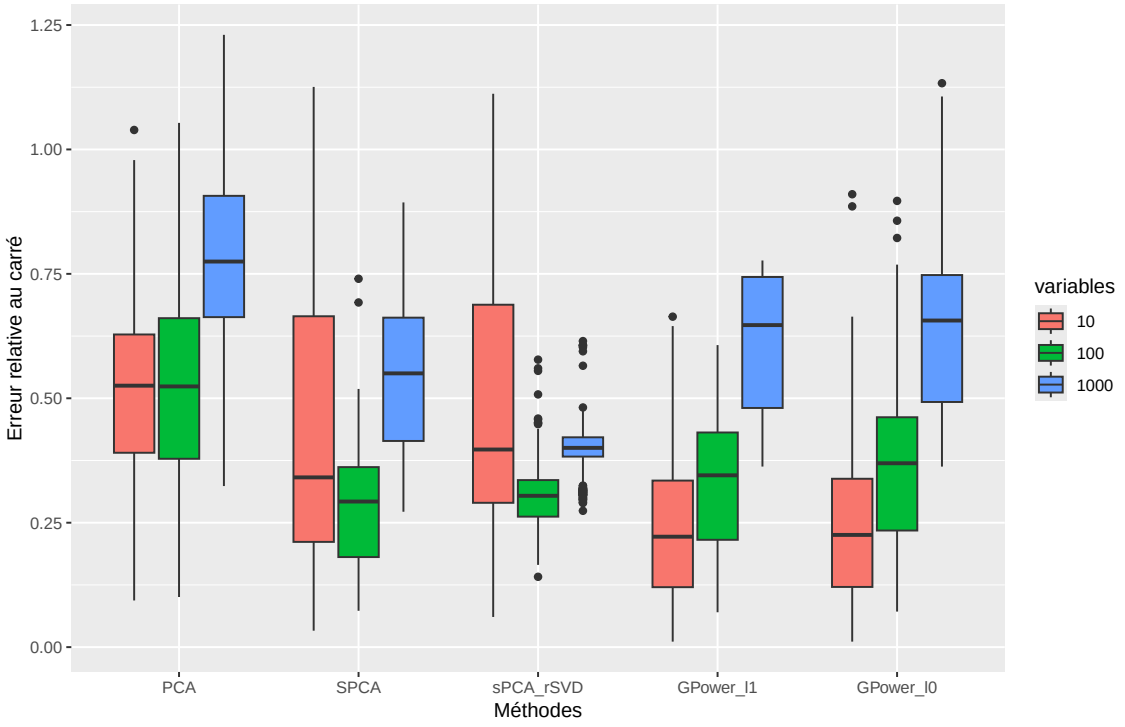

Figure 2: Comparaison de l'erreur relative au carré entre les matrices des composantes principales par méthode et selon le nombre de variables.

meilleure selon les caractéristiques imposées sur nos jeux de données. Voyons d'abord l'impact du nombre de variables des tables de données sur le SRE (figure 2). Nous pouvons observer que les méthodes SPCA et sPCA_rSVD ont relativement le même comportement lorsque le nombre de variables varie. Pour cette dernière méthode, nous comprenons que les valeurs aberrantes observées à la figure précédente 1 viennent des jeux de données avec peu de variables. Sa variance diminue fortement lorsque le nombre de variables est grand, ce qui rend la méthode sPCA_rSVD très intéressante lorsque l'on veut trouver des valeurs stables pour les composantes principales d'un grand jeu de donnée. Les méthodes GPower sont très similaires et obtiennent de bonnes performances pour un jeu de données avec peu de variables, mais l'erreur augmente fortement avec le nombre de variables qui augmente, et les composantes principales risquent de ne pas être correctement calculées.

Une autre caractéristique intéressante à observer est le niveau de parcimonie que l'on a induit dans nos données (figure 3). En général, la SRE a tendance à diminuer lorsque la proportion de parcimonie augmente, c'est-à-dire que les méthodes ont bien tendance à trouver les composantes principales correctes de nos données générées lorsque le nombre de 0 dans la matrice des weights/des loadings augmente. Ce n'est cependant pas forcément le cas pour les méthodes SPCA et sPCA_rSVD, où un grand niveau de parcimonie des coefficients des composantes principales peut augmenter l'erreur. Les valeurs aberrantes de la figure 1 sont aussi observées lorsque l'on a induit beaucoup de 0 dans ces coefficients (une proportion de 90% de 0). Nous pouvons déduire de ce graphique que lorsque l'on

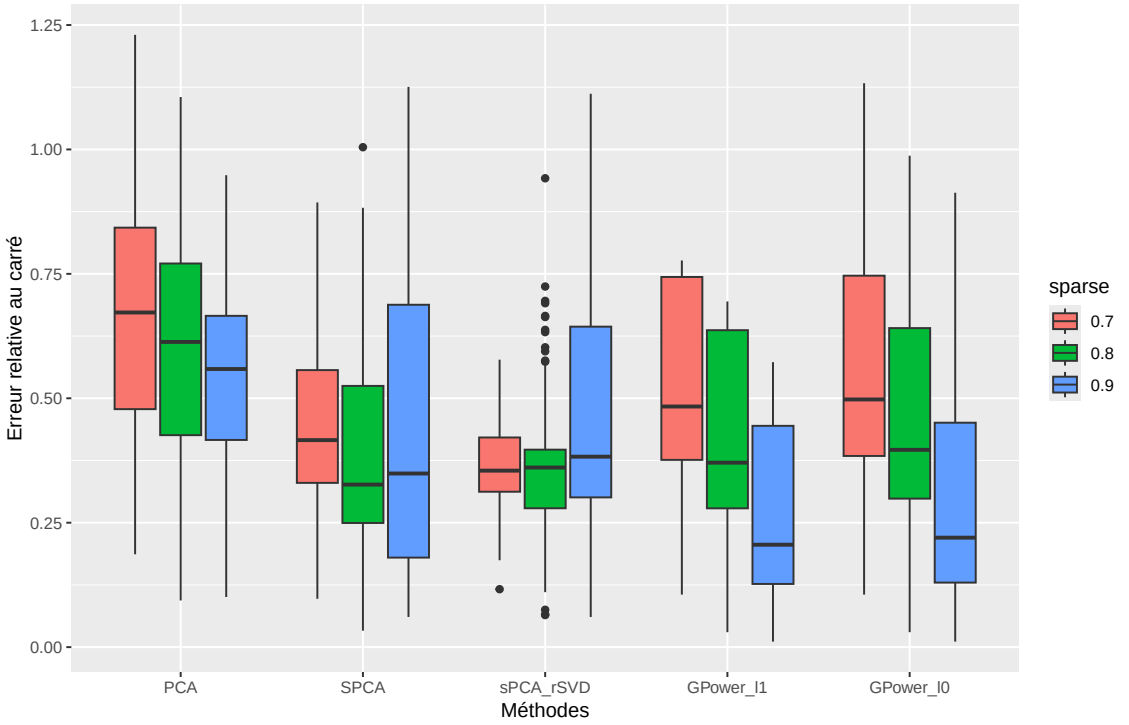

Figure 3: Comparaison de l'erreur relative au carré entre les matrices des composantes principales par méthode et selon la proportion de parcimonie.

désire des weights/loadings fortement parcimonieuses, les méthodes GPower présentent de meilleures performances quant à la bonne reconstitution des composantes principales.

Pour les autres caractéristiques comme la variance expliquée et le nombre de composantes principales, aucune particularité n'a été relevée. Il n'y a pas de différence notable d'erreur si l'on change l'une ou l'autre de ces caractéristiques. Les graphiques ressemblent à la figure 1 et peuvent être retrouvés à l'annexe B.1.

Passons désormais au deuxième critère de performance qui est le taux de mauvaises identifications. Nous pouvons noter que la méthode de l'ACP classique n'a pas été reprise dans la comparaison de l'ensemble des méthodes car le MR est un critère de performance qui va calculer une mesure illustrant le nombre de 0 effectivement reconnu par la méthode comparé à la structure de parcimonie artificielle de notre jeu de données. L'ACP n'est pas une méthode induisant de la parcimonie dans les coefficients des composantes principales et il est extrêmement rare d'y observer un 0. Toutes les valeurs du MR sur un jeu de données où l'ACP a été appliquée sont systématiquement égales à 1.

Regardons maintenant la comparaison des autres méthodes sans prendre en compte une caractéristique particulière des jeux de données (figure 4). Nous pouvons observer que les quatre méthodes reprises ici ont des valeurs du MR proche de 0, ce qui veut dire qu'elles ont tendance a bien retrouver la même disposition des 0 que dans les weights/les loadings des jeux de données générés. Celles montrant une légèrement moins bonne performance

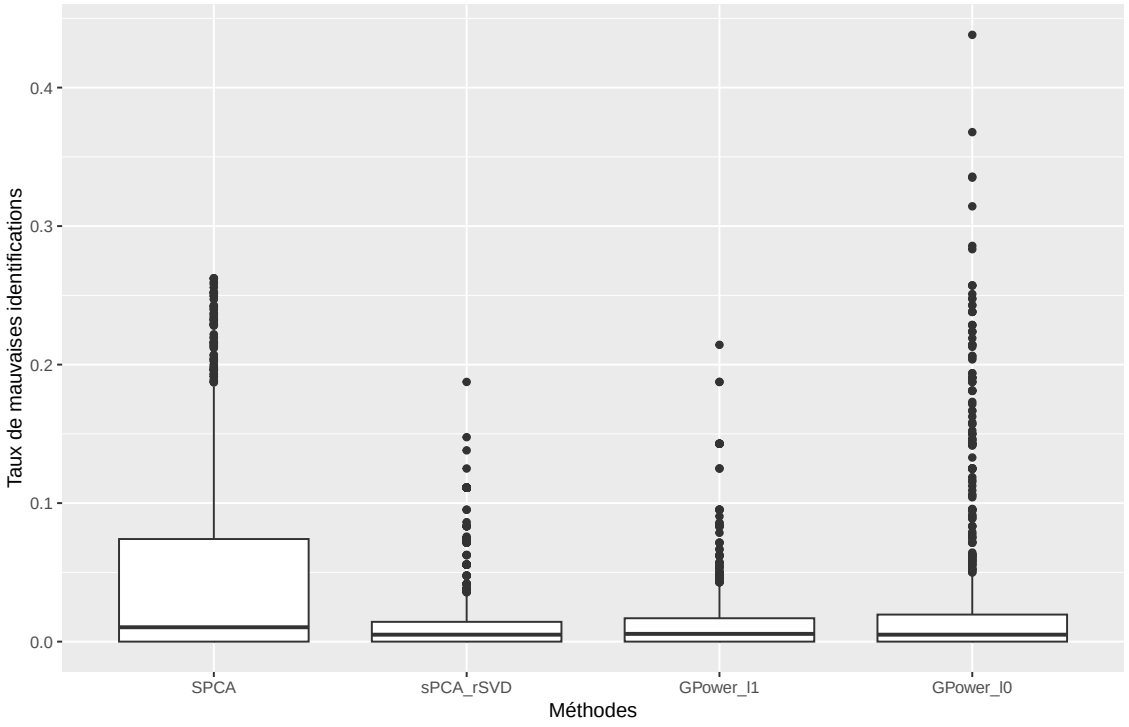

Figure 4: Comparaison du taux de mauvaises identifications par méthode.

sont la SPCA ayant pas mal de valeurs situées dans l'intervalle $[0, 0.08]$ et la GPower avec régularisation l_0 ou de nombreuses valeurs aberrantes sont à observer, montant jusqu'à presque 0.45. Alors, si nous désirons une répartition fidèle de la parcimonie dans les coefficients des composantes principales, Le choix des méthodes sPCA_rSVD et GPower avec régularisation l_1 est avisé.

Des différences de comportement ont été aperçues pour les quatre différentes caractéristiques. Voyons d'abord l'impact du nombre de variables (figure 5). Les méthodes GPower montrent une performance quasi parfaite lorsque le nombre de variables est petit. Elle n'est pas trop mal non plus lorsqu'au contraire ce nombre est grand, même si les valeurs aberrantes de la méthode GPower avec régularisation l_0 sont plutôt élevées. La SPCA montre un MR tout à fait décent pour les deux premiers nombres de variables fixés à 10 et 100. C'est un peu similaire à la sPCA_rSVD sauf que pour elle, lorsque le nombre de variables est grand, les valeurs du MR sont très peu dispersées et proches de 0.

Voyons ensuite l'impact de la proportion de parcimonie (figure 6). Les valeurs du MR ont tendance à se rapprocher 0 avec une proportion de parcimonie plus élevée. Ce qui est débatable pour la méthode SPCA qui ne montre qu'une légère augmentation, et la méthode sPCA_rSVD qui a une meilleure moyenne de performance lorsque la proportion de parcimonie est de 0.9 mais dont les résultats se voient être davantage dispersés. Les méthodes GPower restent les meilleurs dans ce dernier cas.

Pour les deux dernières caractéristiques, nous observons de faibles tendances générales.

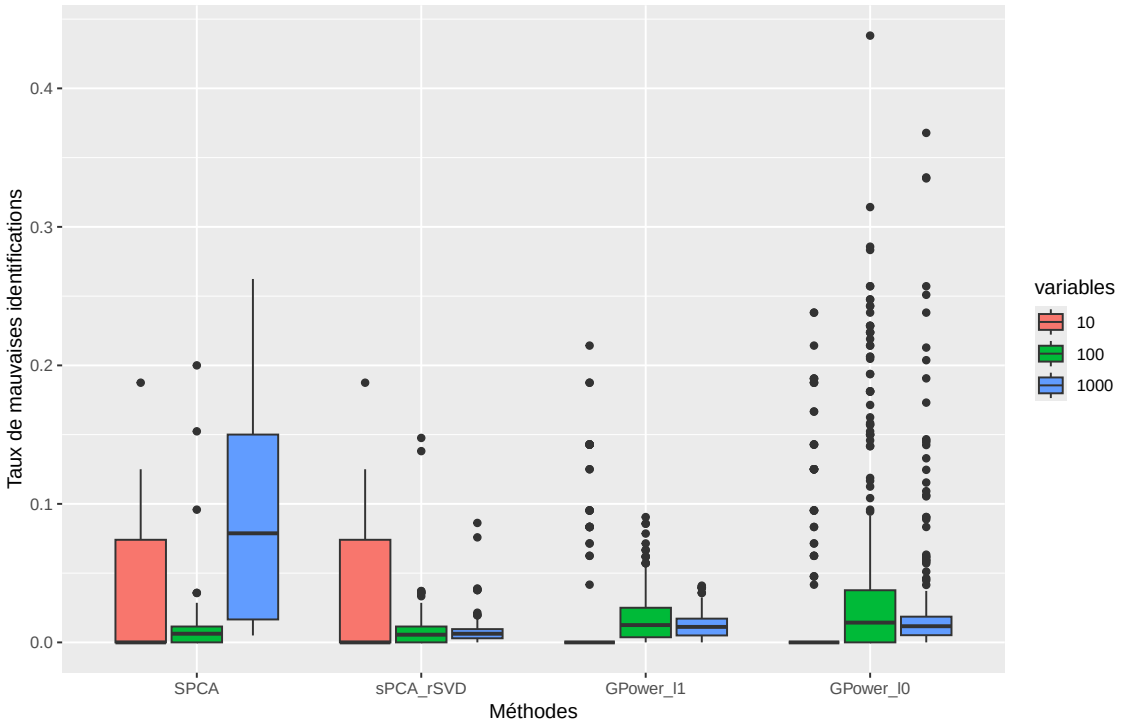

Figure 5: Comparaison du taux de mauvaises identifications par méthode selon le nombre de variables.

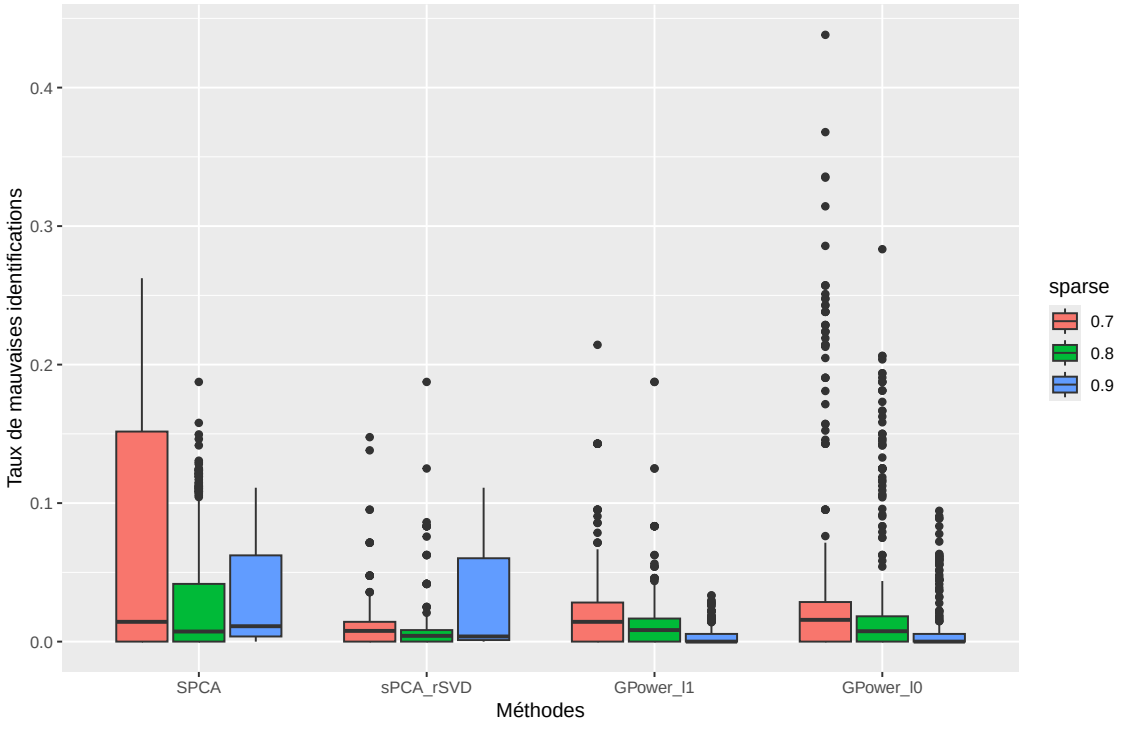

Figure 6: Comparaison du taux de mauvaises identifications par méthode selon la proportion de parcimonie.

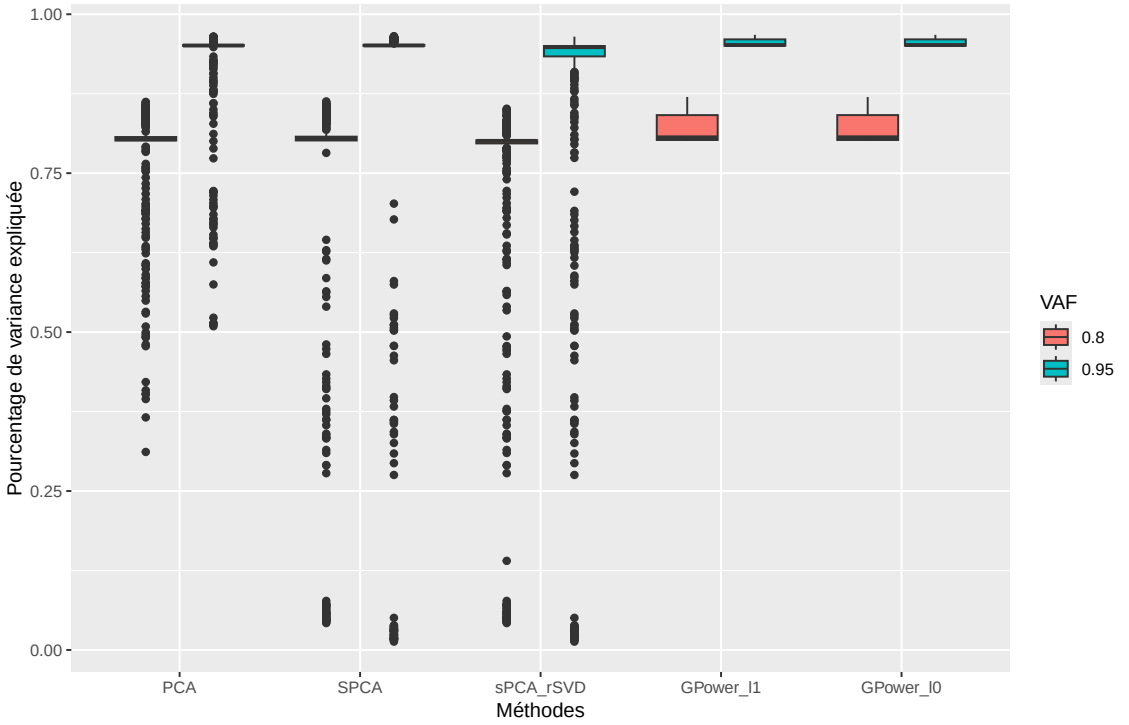

Figure 7: Comparaison du pourcentage de variance expliquée par méthode selon le pourcentage de variance expliquée que l'on a initialement induit dans les données.

Dans le cas de la variance expliquée, le MR trouvé par chacune des méthodes se rapproche de 0 lorsque le pourcentage de variance expliquée gardée par la table de données est proche de 100%. Dans le cas du nombre de composantes principales, c'est une diminution des performances que l'on observe pour chaque méthode lorsque le nombre de composantes principales augmente. Ces deux graphiques se trouvent à l'annexe B.2.

Notre dernier critère de performance est la variance expliquée, donc l'information captée du jeu de données d'origine par sa reconstitution grâce aux différentes méthodes. Cette fois, nous allons directement regarder la comparaison des résultats selon la proportion de variance expliquée que l'on a induite artificiellement dans nos tables de données (figure 7). Pour juger d'une bonne performance de la méthode au niveau de la variance expliquée, il faut que le résultat obtenu par l'équation 16 soit proche de la variance expliquée que l'on a fixé à 80% ou 95%. Nous observons que ces valeurs sont en général bel et bien retrouvées par les différentes méthodes, même si en dehors des méthodes GPower, elles obtiennent énormément de valeurs aberrantes. Après analyse des graphiques de comparaison selon le nombre de variables et selon la proportion de parcimonie (figures reprises à l'annexe B.3), nous observons que ces valeurs aberrantes proviennent respectivement des tables de données ayant peu de variables, et celles ayant une proportion plus grande de parcimonie dans leurs weights/loadings. Les résultats des méthodes GPower ont certes une variance légèrement plus élevés, mais elles assurent de ne pas observer des reconstitutions de tables de données n'ayant quasiment pas captées d'information des tables de données d'origine.

Le graphique de la comparaison générale de la variance expliquée est aussi retrouvé à l'annexe B.3, ainsi que la comparaison selon le nombre de composantes principales, où une légère amélioration des résultats peut être observée lorsque ce nombre augmente. Il est en effet logique que l'on va garder plus d'information lorsque le nombre de composantes principales se rapprochent du nombre de variables du jeu de données d'origine.

3.5 Conclusion de la simulation

En conclusion, ce chapitre de simulation nous a permis de comparer les performances des cinq méthodes d'analyse en composantes principales, à savoir l'ACP, la SPCA, la sPCA_rSVD, la GPower avec régularisation l_1 et avec régularisation l_0, à partir de jeux de données générés. Après avoir expliqué en détail chacune des fonctions R permettant d'appliquer ces méthodes, nous avons pu comparer les résultats.

Parmi les méthodes appliquant la parcimonie sur les coefficients des composantes principales, c'est la SPCA qui a montré les moins bonnes performances. Cette conclusion était attendue du fait que c'est la méthode étant apparue avant les autres et qui a servi de base pour les suivantes.

Parmi les méthodes GPower, celle utilisant la régularisation l_1 a constamment montré une légère meilleure performance que lorsque la régularisation l_0 est utilisée.

Les méthodes sPCA_rSVD et GPower avec régularisation l_1 sortent du lot, ce qui se trouve être la même conclusion que notre article de référence de Guerra-Urzola et al. (2021), mais chacune d'elles se voit être plus performante que l'autre selon diverses situations.

D'après les analyses dans ce chapitre, il sera préférable d'utiliser la méthode sPCA_rSVD lorsque le jeu de données possède un grand nombre de variables (environ plus de 100 colonnes) ou que l'on désire un niveau de parcimonie dans les weights/loadings plutôt faible (moins que 80%). Ce constat est applicable que ce soit pour obtenir les meilleures composantes principales ou pour trouver la meilleure structure de parcimonie des coefficients des composantes principales.

La méthode GPower avec régularisation l_1 a donc une conclusion inverse, où ses performances seront de meilleure qualité lorsque le nombre de variables du jeu de données en question est faible, ou que l'on désire un plus grand niveau de parcimonie dans les résultats finaux. Une différence qui pourrait favoriser la méthode GPower comme celle choisie par défaut est la garantie d'une perte minimale d'information dans les composantes principales, ce qui n'est pas assuré par la méthode sPCA_rSVD.

4 Application

Dans cette partie, les deux méthodes ayant montré les meilleures performances dans la section 3.4.2, à savoir la sPCA_rSVD et la GPower avec régularisation l_1, seront illustrées en les appliquant directement sur des jeux de données réels. Deux jeux de données ont été sélectionnés pour cette tâche :

- les données `pitprops` des travaux de Jeffers (1967) qui sont très adaptées à ce genre d'analyse. Elles permettront de montrer le bon fonctionnement des méthodes sans rentrer dans les détails.
- le jeu de données `country-data` disponible sur le site Kaggle (Gohel, 2022), créé dans l'objectif d'y appliquer des exercices concernant l'ACP. C'est un jeu de données n'ayant jamais été analysé par des méthodes de parcimonie. Il sera donc intéressant de voir le comportement des méthodes dans un cas très concret plus détaillé.

Le code utilisé tout le long de cette section se trouve à l'annexe J.

4.1 Données pitprops

Les données `pitprops` sont disponibles dans le package `elasticnet` (Zou and Hastie, 2020) sous forme d'une matrice de corrélation de taille 13×13. Comme expliqué dans l'historique de la partie 2.1, cette matrice de corrélation a été utilisée par Jeffers (1967) afin d'illustrer la difficulté d'interprétation des composantes principales. Elle a été calculée à partir d'un jeu de données de 13 variables et 180 observations auquel nous n'avons pas accès, mais dont les variables ont été décrites. Lorsque l'on applique la fonction `prcomp` sur ces données, nous obtenons la table 3.

Beaucoup de ces coefficients ont des valeurs non-négligeables, il est dès lors difficile de faire correspondre les variables qui apportent le plus d'information à une composante principale. Nous pouvons alors appliquer une des méthodes de parcimonie que nous avons vues jusqu'à présent. Comme les méthodes sPCA_rSVD et GPower avec régularisation l_1 ont montré les meilleures performances dans la partie 3.4 des simulations, ce sont celles que nous allons appliquer sur ces données. Leurs tables de coefficients des composantes principales est le tableau 4. L'utilisation d'une matrice de corrélation ne pose pas de problème et est même favorisée pour la méthode GPower, car elle permet de réduire à p le nombre de lignes lorsque $n > p$. Cela devient plus dérangeant pour la méthode sPCA_rSVD qui a besoin de la matrice des données dans son problème d'optimisation (voir équation 7). Il a été prouvé dans l'article de Shen and Huang (2008) que les loadings dépendent des

Coefficient des composantes principales		
Variables	CP1	CP2
topdiam	-0.2944	-0.3821
length	-0.3061	-0.3716
moist	0.1477	-0.2823
testsg	0.0987	-0.0680
ovensg	-0.0577	0.5444
ringtop	-0.2287	0.3087
ringbut	-0.3712	0.1776
bowmax	-0.3498	-0.0973
bowdist	-0.3464	-0.2363
whorls	-0.3979	-0.0295
clear	0.1684	-0.1528
knots	0.3152	-0.0812
diaknot	0.2573	-0.3350

Table 3: Table des coefficients des deux premières composantes principales de la matrice de corrélation `pitprops` trouvée par l'ACP classique.

Loadings par sPCA_rSVD			Weights par GPower l_1		
Variables	CP1	CP2	Variables	CP1	CP2
topdiam	0	-0.7008	topdiam	0	0
length	0	-0.6885	length	0	0
moist	0	0	moist	0	0
testsg	0	0	testsg	0	0
ovensg	0	0	ovensg	0	0
ringtop	0	0	ringtop	0	-0.6848
ringbut	0	0	ringbut	0.5164	0
bowmax	0	0	bowmax	0.4545	0
bowdist	0	-0.1866	bowdist	0.4074	0
whorls	0.6421	0	whorls	0.6007	0
clear	-0.7429	0	clear	0	0
knots	0	0	knots	0	0
diaknot	-0.1891	0	diaknot	0	0.7287

Table 4: Table des loadings trouvés par sPCA_rSVD à gauche et des weights trouvés par GPower l_1 à droite des deux premières composantes principales de la matrice de corrélation `pitprops`.

données d'origine X uniquement par le biais de la matrice de Gram X^TX, et donc de la matrice de covariance. Nous obtiendrons les mêmes résultats peu importe les données X qui donnent la même matrice de corrélation. Il est donc possible d'utiliser en argument une matrice de corrélation à condition de prendre sa racine carré, ce qui est fait dans le code à l'annexe J.

Les autres arguments permettant de produire ces tables sont évidemment le nombre de composantes principales fixé à 2, et le niveau de parcimonie qui est respectivement de 3 valeurs non-égales à 0 par composantes principales pour la sPCA_rSVD, et de 77% pour la GPower. C'est le pourcentage permettant d'obtenir le même nombre d'éléments non-égaux à 0 dans les deux exemples.

En comparaison avec la table des coefficients des composantes principales trouvés par ACP classique, les résultats sont beaucoup plus parcimonieux, mais nous observons une grande différence entre les deux méthodes quant aux variables choisies pour expliquer ces composantes principales. Nous allons rentrer plus en détail sur la façon de choisir une méthode et comment optimiser le choix des paramètres dans l'exemple suivant.

4.2 Données country-data

Ce jeu de données a été choisi dans le but d'illustrer de manière concrète l'utilisation d'une méthode de parcimonie. Il possède 167 observations pour 10 variables. Chaque ligne correspond à un pays, où des informations économiques et démographiques y sont présentées. Nous avons comme variables :

- `country` : le nom du pays. Cette variable est retirée car elle n'est pas quantitative, elle représente simplement un indice des lignes.
- `child_mort` : morts d'enfants de moins de 5 ans pour 1000 naissances.
- `exports` : exports de biens et services exprimés en pourcentage du PIB total.
- `health` : un indice santé. Cette variable est négligée car aucune information n'est disponible la concernant.
- `imports` : imports de biens et services exprimés en pourcentage du PIB total.
- `income` : revenu net par personne.
- `inflation` : la mesure du taux de croissance annuel du PIB total.
- `life_expec` : le nombre d'années moyen qu'un nouveau-né devrait vivre si le niveau de mortalité actuel resterait constant.
- `total_fer` : le nombre de naissances pour chaque femme si le taux actuel de fertilité par rapport à l'âge resterait constant.

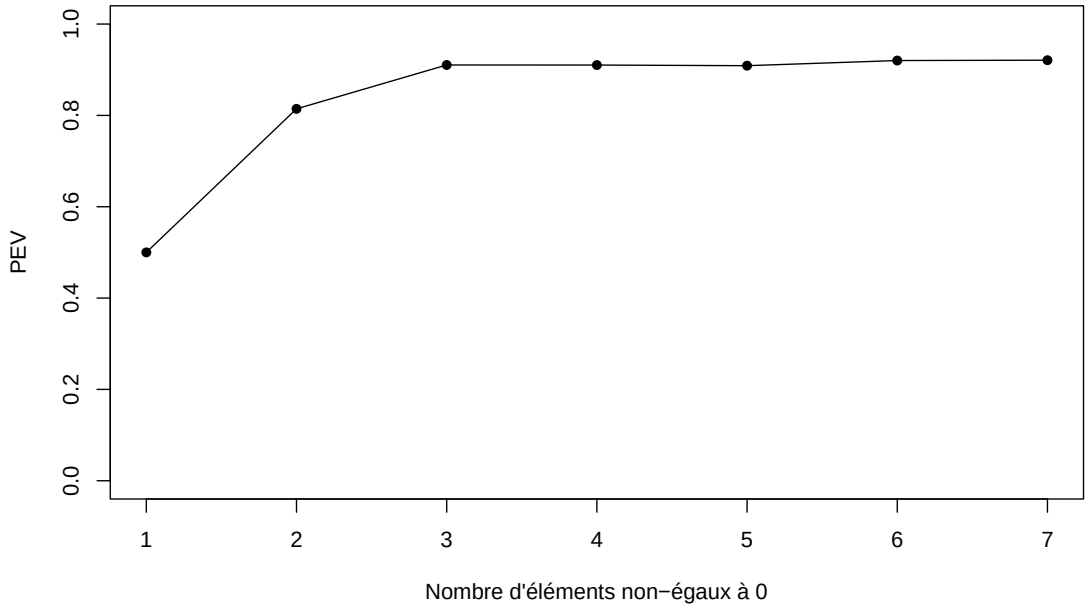

Figure 8: Rapport entre le nombre d'éléments non-égaux à 0 que l'on désire dans chacune des 4 premières composantes principales et la proportion de variance expliquée obtenue par la méthode sPCA_rSVD.

- `gdpp` : le PIB total du pays.

Le jeu de données après son pré-traitement a donc une taille de 167×8. Notre objectif maintenant est d'appliquer nos deux méthodes sur ces données tout en trouvant la combinaison du nombre de composantes principales et de niveau de parcimonie maximisant la variance expliquée, avec un nombre faible de composantes principales. La variance expliquée, comme nous avons vu dans la section 3.4.1, représente une proportion d'information gardée par les nouvelles composantes principales. Pour ce faire, nous allons sélectionner les cinq premières composantes principales et produire un graphique pour chacune d'elles. Ce graphique mettra en jeu le niveau de parcimonie par rapport à la PEV, qui correspond à la PEV vue dans la section 3.4.1.

Les deux graphes les plus intéressants ont été sélectionnés ici, mais les huit autres (10 en tout, 5 pour chacune des méthodes) peuvent être retrouvés à l'annexe C. Considérons un PEV de 90% comme étant la valeur désirée pour juger d'une bonne reconstitution des données. Dans les deux cas, nous pouvons obtenir une telle PEV qu'à partir de 4 composantes principales sélectionnées. La PEV sur l'axe y de chaque graphique a été fixée par un intervalle de 0 à 1 pour avoir une vision claire sur son évolution lorsque l'on augmente le nombre de composantes principales.

Pour la méthode sPCA_rSVD, nous pouvons observer le graphique 8. Les 5 graphiques liés à cette méthode partagent un schéma similaire, à la différence de l'augmentation nette de la

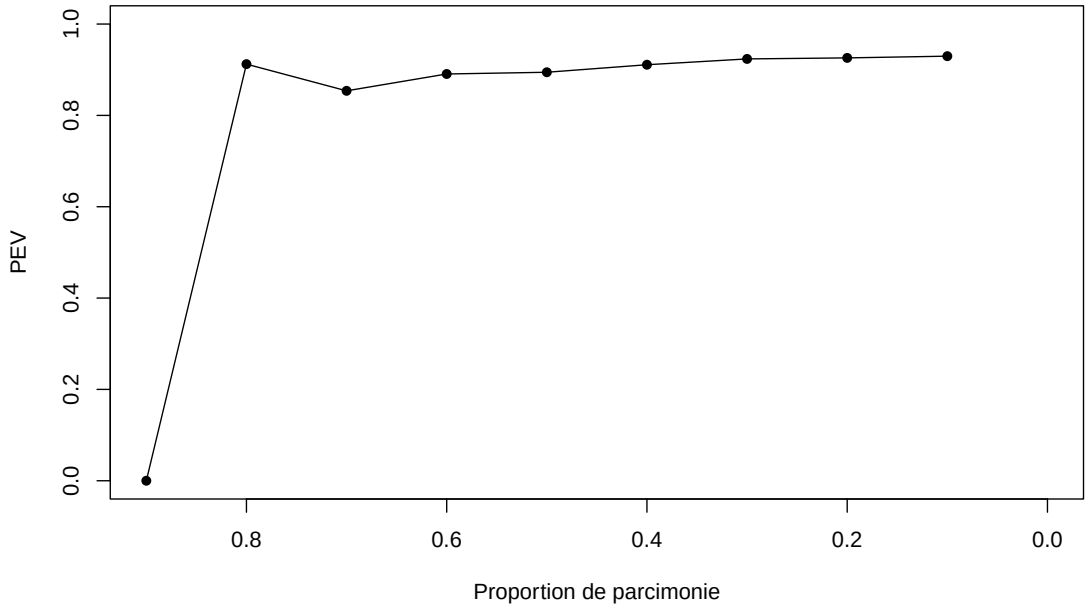

Figure 9: Rapport entre la proportion de parcimonie que l'on désire dans l'ensemble des 4 premières composantes principales et la proportion de variance expliquée obtenue par la méthode GPower l_1.

PEV à chaque ajout d'une composante principale. A partir de la quatrième sélectionnée, nous atteignons effectivement une PEV supérieure à 90% lorsque nous avons 3 loadings non-égaux à 0 pour chaque composantes principales. Dans ce cas, le PEV est exactement de 91.04%.

Concernant la méthode GPower l_1, le graphique en question est la figure 9. En comparaison avec les autres graphes de la même méthode, la différence de PEV entre la prise en compte ou non d'une cinquième composante principale est minime. Il est donc naturel de s'arrêter à 4 composantes principales, d'autant plus que l'on obtient plus de 90% dans cette configuration. Lorsque la proportion de parcimonie dans la matrice des weights est fixée à 0.8, la PEV vaut 91.23%.

Les deux méthodes sont plutôt efficaces sur ce jeu de données. La GPower présente tout de même une variance expliquée légèrement plus élevée que pour la sPCA_rSVD, avec un niveau de parcimonie supérieur. De plus, nous sommes bien dans le cas décris dans la section 3.5 où le nombre de variables est relativement petit et une grande proportion de parcimonie est demandée. La méthode GPower semble plus efficace dans cette situation et nous optons donc pour cette méthode pour la suite de cette section.

Maintenant que nous avons choisi notre méthode, que nous avons le nombre de composantes principales idéal et le niveau de parcimonie optimal, nous pouvons calculer la matrice des coefficients des composantes principales afin qu'elle nous aide à interpréter ces nouvelles

Weights par GPower l_1				
Variables	CP1	CP2	CP3	CP4
child_mort	0	0.5932	0	0
exports	0	0	0.7071	0
imports	0	0	0.7071	0
income	0	0	0	0.7071
inflation	1	0	0	0
life_expec	0	-0.5739	0	0
total_fer	0	0.5645	0	0
gdpp	0	0	0	0.7071

Table 5: Table des weights des 4 premières composantes principales de la matrice des données `country-data` trouvés par la méthode GPower l_1.

composantes principales. Les weights sont contenus dans la table 5. La méthode GPower a tendance à vite converger vers une seule variable explicative lorsque la proportion de parcimonie imposée est grande. C'est le cas ici pour la première composante principale où les coefficients sont nuls à part celui de la variable `inflation`. Cela veut dire que la première composante principale peut être vue comme la variable `inflation` car elle capte le maximum de variance des données d'origine. En regardant le graphique C.13 à l'annexe C, nous remarquons que lorsque la proportion de parcimonie est fixée à 0.8, cette composante principale à elle seule explique 16.46% de la variance totale, ce qui ne représente pas une grande contribution. Les graphiques suivants, a savoir les figures C.14, C.15 et 9 montrent respectivement une PEV pour la deuxième composante principale de 29.35%, la troisième composante principale de 12.5% et la quatrième composante principale de 32.91%, le total faisant bien 91.23%. Nous en concluons donc que la deuxième et la quatrième composantes principales sont les plus importantes en terme de variance expliquée.

Cela signifie que lorsque l'on rend la structure des composantes principales plus parcimonieuse, l'ordre décroissant de ces composantes en terme de variance expliquée semble ne pas être garantie comme dans l'ACP classique. Ce problème n'est pas mentionné dans les articles de références des méthodes sélectionnées. La première composante reste la plus importante de manière générale, mais à certains niveaux de parcimonie, comme 0.8 ici, l'interprétabilité y est privilégiée au profit de la variance expliquée. Dans notre exemple, ce phénomène est aussi observable à plus petit échelle dans les graphiques où la méthode sPCA_rSVD est appliquée (voir les figure C). Ce n'est donc à priori pas une particularité de la GPower mais bien des méthodes de parcimonie. Il est cependant nécessaire de réaliser davantage de travaux sur ce problème éventuel de l'ACP parcimonieuse avant d'en tirer des conclusions.

Les variables décrivant le mieux la composante principale 2 sont `child_mort`, `life_expec` et `total_fer`. Ce sont les variables plutôt liées à la démographie des pays. La composante principale a une relation positive avec l'augmentation de la mortalité et du nombre de

naissances, et une relation négative lorsque la durée de vie augmente. En d'autres termes, la deuxième composante principale est liée positivement aux pays les moins développées.

La troisième composante principale prend en compte les variables `imports` et `exports`, qui illustrent les transactions commerciales avec les autres pays. Cette composante apporte le moins de variance expliquée, c'est-à-dire que ces variables ont un poids plus faible dans la variance totale captée des données d'origine.

La quatrième composante principale est expliquée par les variables `income` et `gdpp`, toutes deux y contribuant positivement. Lorsque le salaire et le PIB du pays sont élevés, cette composante principale se verra influencée vers le haut.

En somme, aucune variable n'est négligée lorsque l'on sélectionne les 4 premières composantes principales (table 5), mais nous pouvons nous faire un ordre d'importance des variables. Cela nous a au final aidé à interpréter plus facilement les composantes principales que dans le cas où on aurait appliqué l'ACP classique, au prix d'une légère perte d'information que l'on a limité.

5 Conclusion et discussion

En conclusion, nous avons présenté le fonctionnement et les mathématiques derrières l'ACP, ainsi que trois de ses extensions appliquant de la parcimonie dans la structure des composantes principales. La SPCA transforme le problème de l'ACP en un problème de régression et y applique une pénalisation de LASSO (voir section 2.2). La sPCA_rSVD régularise la décomposition en valeurs singulières de la matrice des données afin d'obtenir une approximation à rang plus faible (voir section 2.3). La GPower optimise un critère de parcimonie et trouve un schéma de parcimonie pour les coefficients des composantes principales (voir section 2.4). Une synthèse de la partie théorique est disponible à la section 2.5. Ce mémoire s'est concentré sur les développements mathématiques des méthodes, en délaissant une interprétation plutôt géométrique de leurs comportements. Il peut être intéressant d'explorer cette facette de l'ACP parcimonieuse, comme par exemple vérifier si les méthodes retrouvent bien des composantes principales orthogonales entre elles.

Nous avons ensuite simulé ces méthodes sur des données générées artificiellement afin de les comparer sur divers critères (voir section 3.2). Chaque package R contenant ces fonctions a été présenté, ainsi que la description de leurs codes sources, permettant de faire le lien entre l'outil et les statistiques (voir section 3.3). Après analyse des résultats, les méthodes sPCA_rSVD et GPower avec régularisation l_1 ont montré les meilleures performances (voir section 3.4.2). Il est cependant important de noter que malgré le choix de valeurs les plus intéressantes pour les critères des données générés, le plan expérimental est tout de même très restreint. Une simulation avec d'autres valeurs des critères mérite d'être exploré, notamment au niveau de la proportion de parcimonie où les valeurs choisies sont plutôt hautes ($0.7, 0.8$ et 0.9).

Nous avons enfin appliqué les deux meilleures méthodes sur les données réelles `pitprops` et `country-data`, où ce dernier exemple a été développé afin de montrer le bon fonctionnement des méthodes et la simplification de l'interprétation des composantes principales (voir section 4). Ce choix des données `country-data` a été fait arbitrairement, afin de voir si les méthodes pouvaient s'adapter à un nouveau jeu de données analysé nul part ailleurs par des méthodes d'ACP parcimonieuse, mais il existe bien d'autres applications. Nous avons par exemple les données génomiques qui possèdent un très grand nombre de variables et où des techniques d'ACP sont nécessaires, ou alors des traitements d'images, où les méthodes de parcimonie se sont retrouvés efficaces dans le cadre de sélection de features.

Remerciements

Mon promoteur Professeur Segers Johan.

Références

Baglama, J. and L. Reichel (2005). Augmented implicitly restarted lanczos bidiagonalization methods. *SIAM Journal on Scientific Computing 27*(1), 19–42.

Baglama, J., L. Reichel, and B. W. Lewis (2022). irlba: Fast truncated singular value decomposition and principal components analysis for large dense and sparse matrices. R package version 2.3.5.1, `https://CRAN.R-project.org/package=irlba`.

Cadima, J. and I. T. Jolliffe (1995). Loading and correlations in the interpretation of principle compenents. *Journal of Applied Statistics 22*(2), 203–214. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/757584614.

d'Aspremont, A., F. Bach, and L. El Ghaoui (2008). Optimal solutions for sparse principal component analysis. *Journal of Machine Learning Research 9*(7).

Gohel, V. (2022). Clustering & PCA assignment. Kaggle, `https://www.kaggle.com/datasets/vipulgohel/clustering-pca-assignment`, consulté le 16 mai 2024.

Golub, G. H. and C. F. V. Loan (2013). *Matrix Computations*. JHU Press.

Guan, Y. and J. Dy (2009). Sparse probabilistic principal component analysis. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, pp. 185–192. PMLR. ISSN: 1938-7228.

Guerra-Urzola, R., N. C. de Schipper, A. Tonne, K. Sijtsma, J. C. Vera, and K. Van Deun (2023). Sparsifying the least-squares approach to PCA: comparison of lasso and cardinality constraint. *Advances in Data Analysis and Classification 17*(1), 269–286.

Guerra-Urzola, R., K. Van Deun, J. C. Vera, and K. Sijtsma (2021). A guide for sparse pca: model comparison and applications. *psychometrika 86*(4), 893–919.

Hastie, T., J. Friedman, and R. Tibshirani (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology 24*(6), 417–441. Place: US Publisher: Warwick & York.

Jeffers, J. N. R. (1967). Two case studies in the application of principal component analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics) 16*(3), 225–236. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.2307/2985919.

Jolliffe, I. T., N. T. Trendafilov, and M. Uddin (2003). A modified principal component technique based on the LASSO. *Journal of Computational and Graphical Statistics 12*(3), 531–547. Publisher: Taylor & Francis.

Journée, M., Y. Nesterov, P. Richtárik, and R. Sepulchre (2010). Generalized power method for sparse principal component analysis. *Journal of Machine Learning Research 11*(2).

Kaiser, H. F. (1958). The varimax criterion for analytic rotation in factor analysis. *Psychometrika 23*(3), 187–200.

Kiers, H. A. L. (1994). Simplimax: Oblique rotation to an optimal target with simple structure. *Psychometrika 59*(4), 567–579.

MA, Z. (2013). Sparse principal component analysis and iterative thresholding. *The Annals of Statistics 41*(2), 772–801.

Pearson, K. (1901). Principal components analysis. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 6*(2), 559.

Segers, J. (2021). Lstat2110 – analyse des données. Cours de l'UCL (LSBA) en master statistiques et sciences des données.

Shen, H. and J. Z. Huang (2008). Sparse principal component analysis via regularized low rank matrix approximation. *Journal of Multivariate Analysis 99*(6), 1015–1034.

SurajGupta (2016). prcomp code source R sur Github. accessed 04.04.2024, `"https://github.com/SurajGupta/rsource/blob/master/src/library/stats/R/prcomp.R"`.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological) 58*(1), 267–288.

Verhoef, R. (2021). gpowerr: Implementing the Sparse PCA Generalized Power method in R. MSc thesis, Eindhoven University of Technology, `https://research.tue.nl/nl/studentTheses/gpowerr`.

Vines, S. K. (2000). Simple principal components. *Journal of the Royal Statistical Society: Series C (Applied Statistics) 49*(4), 441–451. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-9876.00204.

Wang, Y. and S. Van Aelst (2019). ltsspca: Sparse principal component based on least trimmed squares. R package version 0.1.0, `https://CRAN.R-project.org/package=ltsspca`.

Whittle, P. (1952). On principal components and least square methods of factor analysis. *Scandinavian Actuarial Journal 1952*(3), 223–239. Publisher: Taylor & Francis.

Yi, S., Z. Lai, Z. He, Y.-m. Cheung, and Y. Liu (2017). Joint sparse principal component analysis. *Pattern Recognition 61*, 524–536. Publisher: Elsevier.

Zou, H. and T. Hastie (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology 67*(2), 301–320.

Zou, H. and T. Hastie (2020). elasticnet: Elastic-net for sparse estimation and sparse pca. R package version 1.3, `https://CRAN.R-project.org/package=elasticnet`.

Zou, H., T. Hastie, and R. Tibshirani (2006). Sparse principal component analysis. *Journal of Computational and Graphical Statistics 15*(2), 265–286.

Zou, H. and L. Xue (2018). A selective overview of sparse principal component analysis. *Proceedings of the IEEE 106*(8), 1311–1320.

Annexe

A GPower méthode en bloc

C'est une méthode GPower présentée dans l'article de Journée et al. (2010) permettant de calculer les coefficients des k premières composantes principales en une seul fois, et non séquentiellement. En partant de la formulation l_1 vue précédement dans la partie 2.4, Le terme λ va devenir un vecteur où les éléments λ_l sont les termes de pénalisation correspondant à chaque vecteur des composantes principales avec $l = 1,\ldots,k$. Chaque élément de ce vecteur devra être inférieur à la plus grande des normes d'une des colonnes de X multipliée par μ_l, pour éviter une matrice de weights remplis de 0. Un vecteur μ va donc aussi rentrer en compte dans le problème d'optimisation, représentant un vecteur de moyenne de taille k qui sera appliquée à chacune des composantes principales. D'abord, pour trouver notre matrice intermédiaire $\hat{Z}$, nous avons :

$$\hat{Z} = \arg\max_{Z} \sum_{l=1}^{k} \sum_{j=1}^{p} S(\mu_l |x_j^T z_l|, \lambda_l)^2$$

Où la fonction S reste la même fonction seuil de type soft vu précédemment. Les éléments de la matrice $\hat{W}$ sont ensuite trouvés par :

$$\hat{w_{jl}} = \frac{sign(x_j^T z_l) S(\mu_l |x_j^T z_l|, \lambda_l)}{\sqrt{\sum_{h=1}^{p} S(\mu_l |x_h^T z_l|, \lambda_l)^2}}, j = 1,\ldots,p, l = 1,\ldots,k$$

Où la normalisation se fait ici par une norme de Frobenius car nous ne sommes plus dans un cas vectoriel mais matriciel. Mis à part le fait de bien prendre en compte tous les éléments de la matrice en question, le calcul reste inchangé. L'ensemble des $\hat{w_{jl}}$ calculés formeront la matrice $\hat{W}$ représentant la matrice des weights parcimonieuse.

Au niveau pratique, la méthode s'applique quasi de la même façon que pour la méthode séquentielle, à part le fait de prendre en compte les différents éléments comme des matrices et non des vecteurs. Il existe cependant une différence au niveau du calcul des weights car P ne suffit plus à trouver les bonnes valeurs. La matrice diagonale N est introduite, avec les éléments du vecteur μ sur la diagonale et des 0 ailleurs. Nous utilisons aussi $\bar{P}$ qui représente le complément de P, c'est-à-dire que chaque 0 devient 1 et vise versa. Nous pouvons à partir de là trouver W par la formule vue plus haut qui va également normaliser les résultats. Nous pouvons ensuite fixer à 0 tous les éléments de W ayant le même indexe à un élément égale à 1 dans la matrice $\bar{P}$. Afin de répéter cet algorithme jusqu'à convergence pour trouver $\hat{W}$, le nouveau Z est calculé par $Z = \frac{XWN}{\|XWN\|}$. Voici l'algorithme pour cette seconde étape :

Algorithm 6 Matrice $\hat{W}$ pour GPower en bloc et régularisation l_1

1. Initialisation :
Matrice de données $X \in R^{n \times p}$
Vecteur des termes contrôlant la parcimonie $[\lambda_1, \ldots, \lambda_k]^T \geq 0$
Matrice $N = diag(\mu_1, \ldots, \mu_k)$
$\hat{Z}$ calculé par l'algorithme 3

2. Inversion de la matrice P pour obtenir $\bar{P}$ où les 1 et les 0 sont inversés.

3. $W = X^T ZN$

4. $W = W(W^T W)^{-\frac{1}{2}}$

5. $W_{\hat{P}} = 0$

6. $Z = \frac{XWN}{\|XWN\|}$

7. Répéter les étapes de 3 à 6 jusqu'à ce que W atteint un seuil de précision attendu ou que le maximum d'itération est atteint.

8. $\hat{W} = W$

B Graphiques de la simulation

B.1 Box-plot de l'erreur relative au carré

Figures B.1 et B.2 illustrant la comparaison des méthodes au niveau de l'erreur relative au carré.

B.2 Box-plot du taux de mauvaises identifications

Figures B.3 et B.4 illustrant la comparaison des méthodes au niveau du taux de mauvaises identifications.

B.3 Box-plot du pourcentage de variance expliquée

Figures B.5, B.6, B.7 et B.8 illustrant la comparaison des méthodes au niveau du pourcentage de variance expliquée désirée.

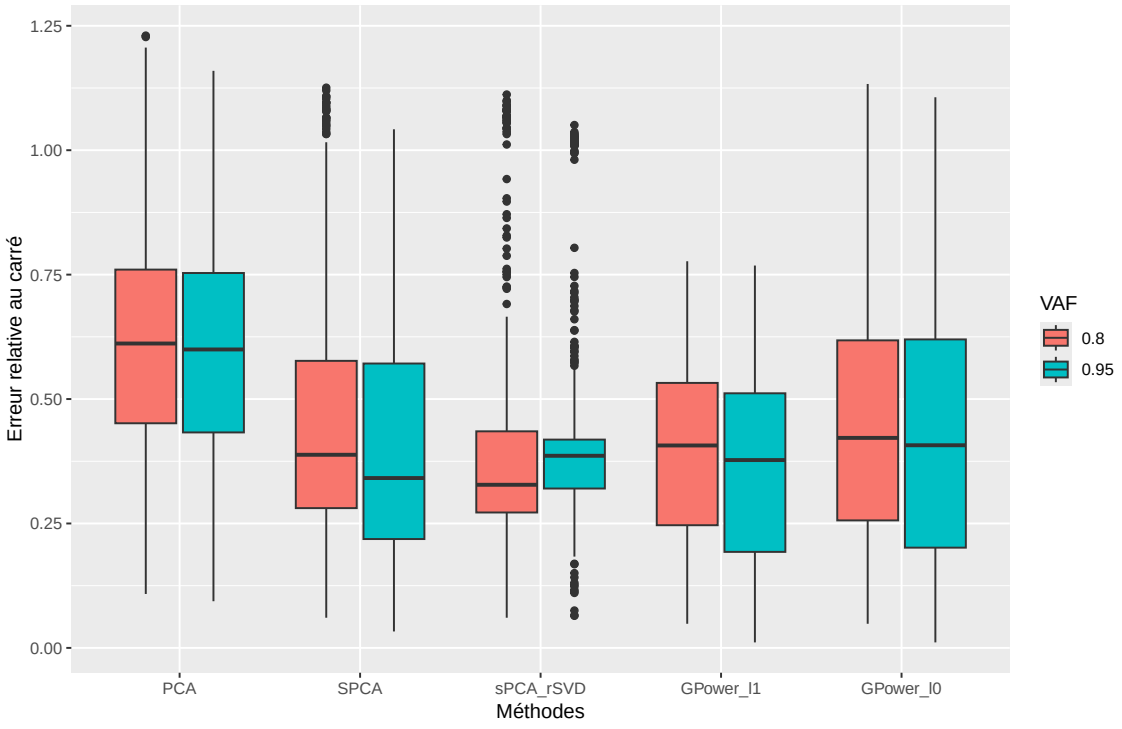

Figure B.1: Comparaison de l'erreur relative au carré entre les matrices des composantes principales par méthodes et selon le pourcentage de variance expliquée.

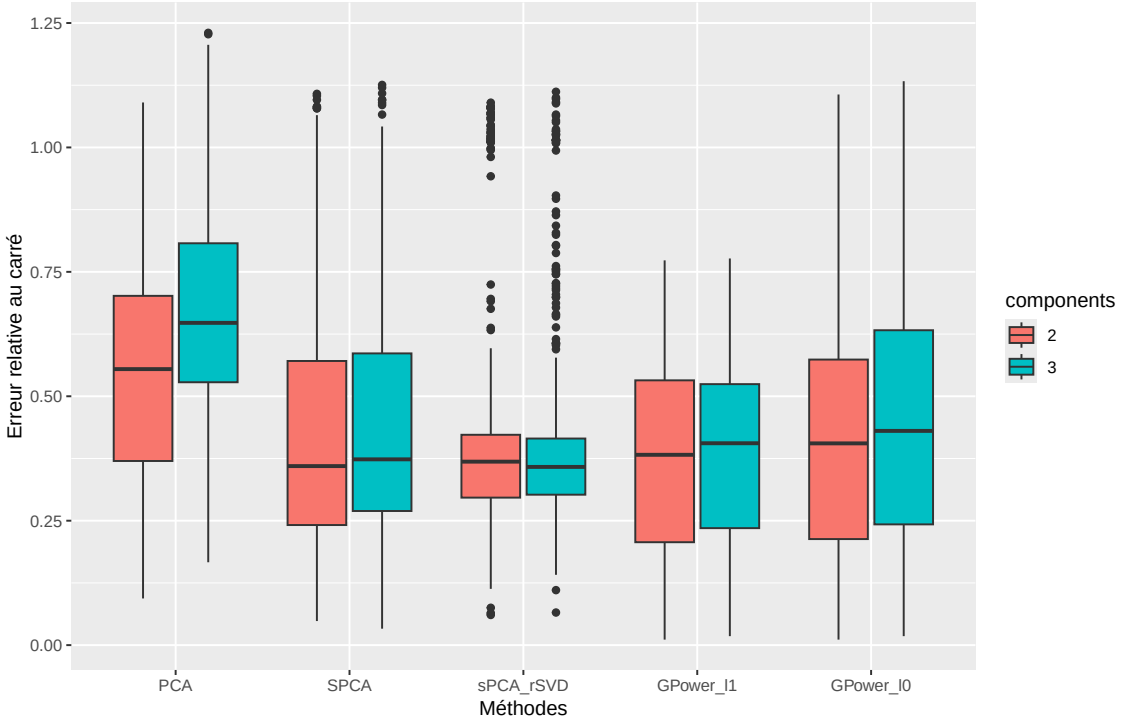

Figure B.2: Comparaison de l'erreur relative au carré entre les matrices des composantes principales par méthodes et selon le nombre de composantes principales.

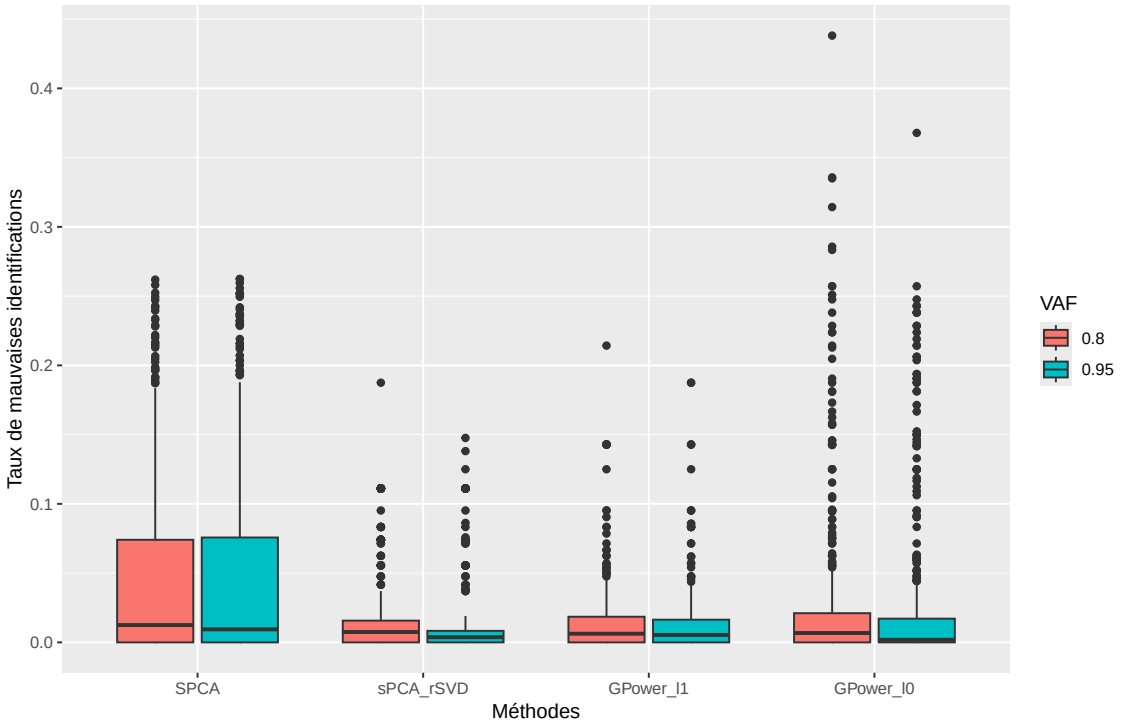

Figure B.3: Comparaison du taux de mauvaises identifications par méthodes selon le pourcentage de variance expliquée.

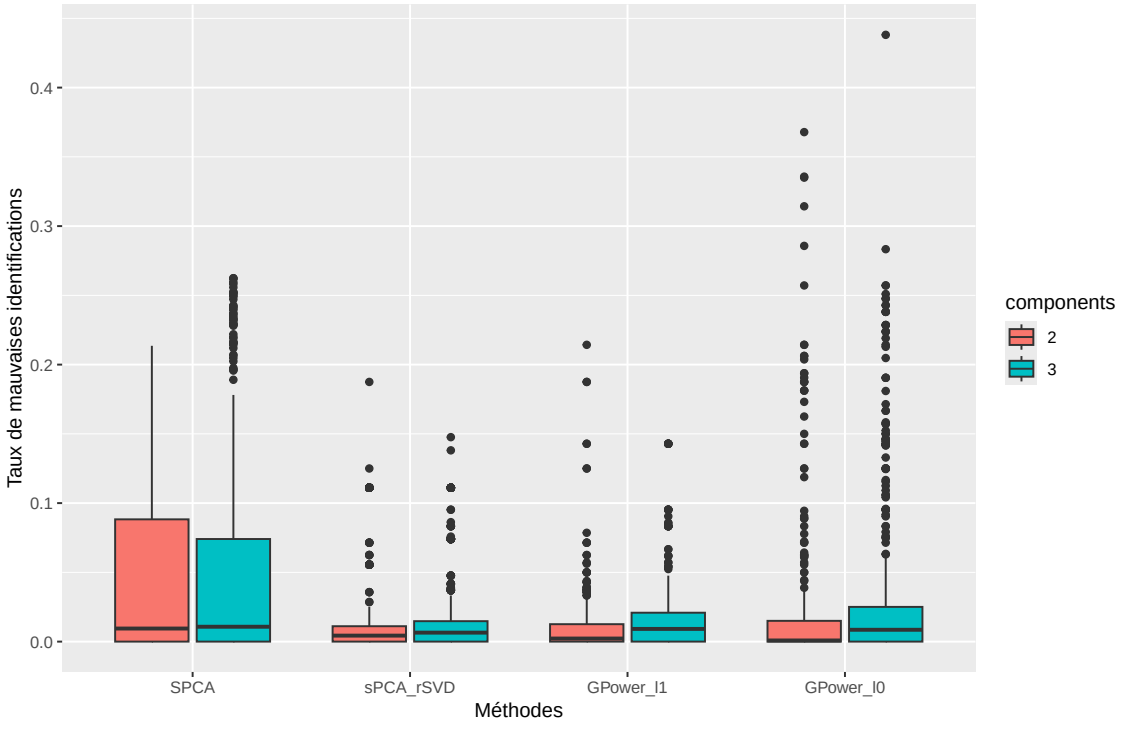

Figure B.4: Comparaison du taux de mauvaises identifications par méthodes selon le nombre de composantes principales.

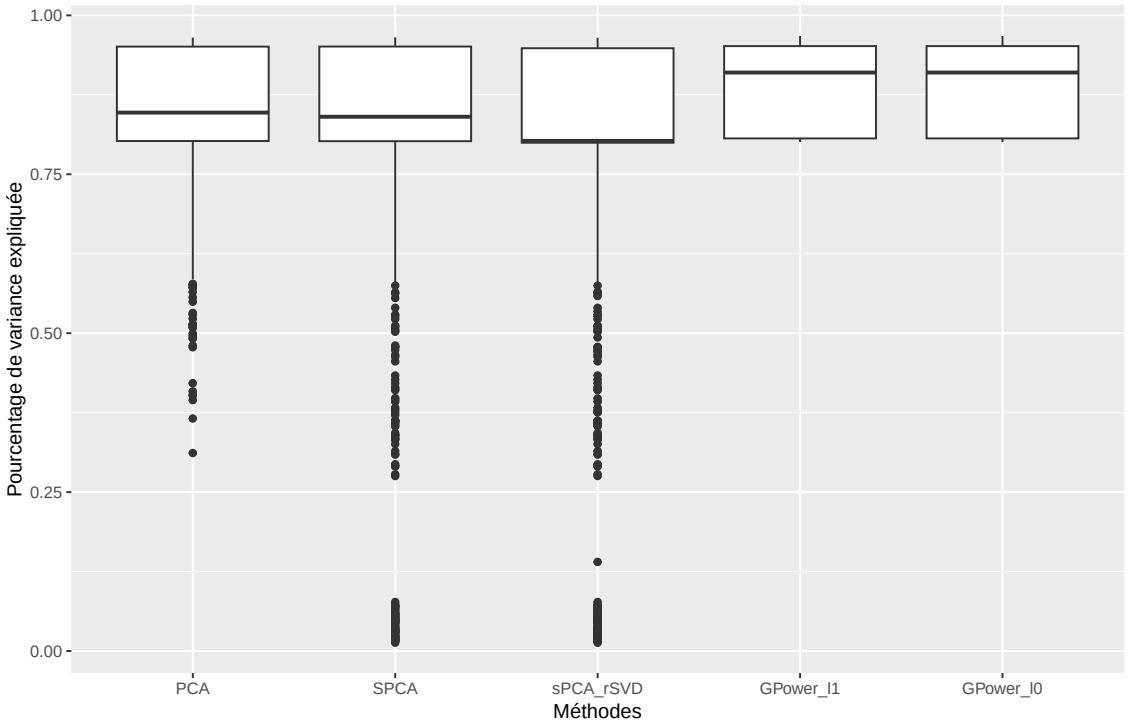

Figure B.5: Comparaison du pourcentage de variance expliquée par méthodes.

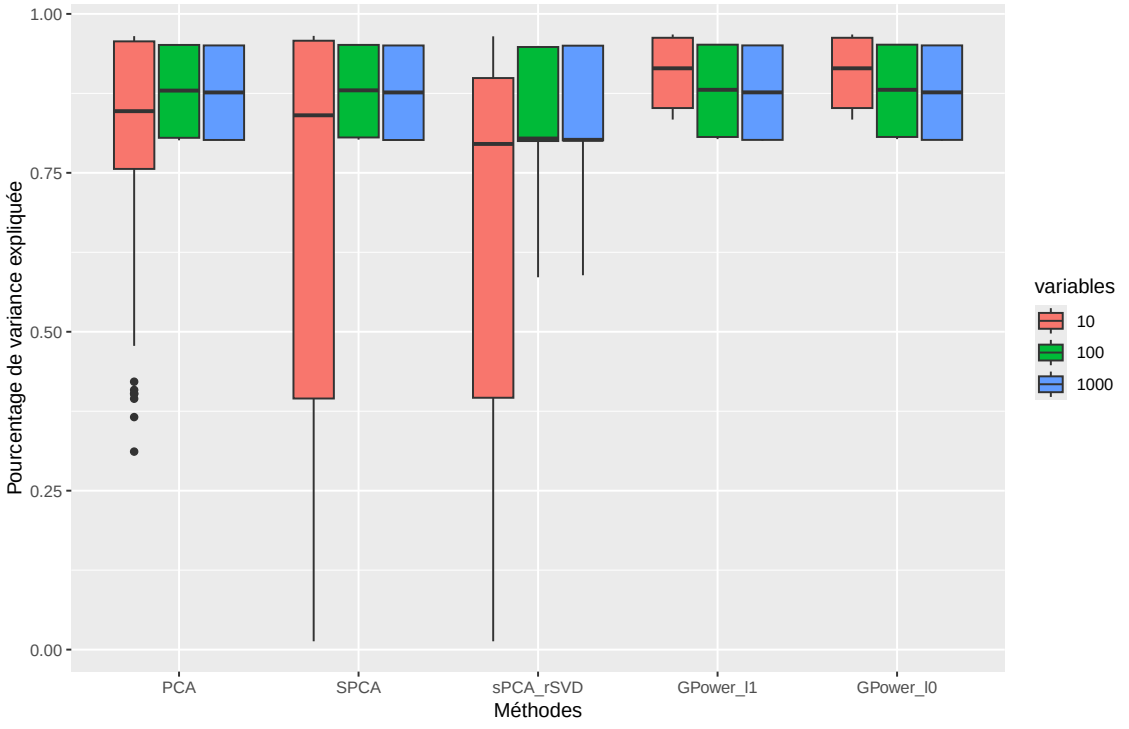

Figure B.6: Comparaison du pourcentage de variance expliquée par méthodes selon le nombre de variables.

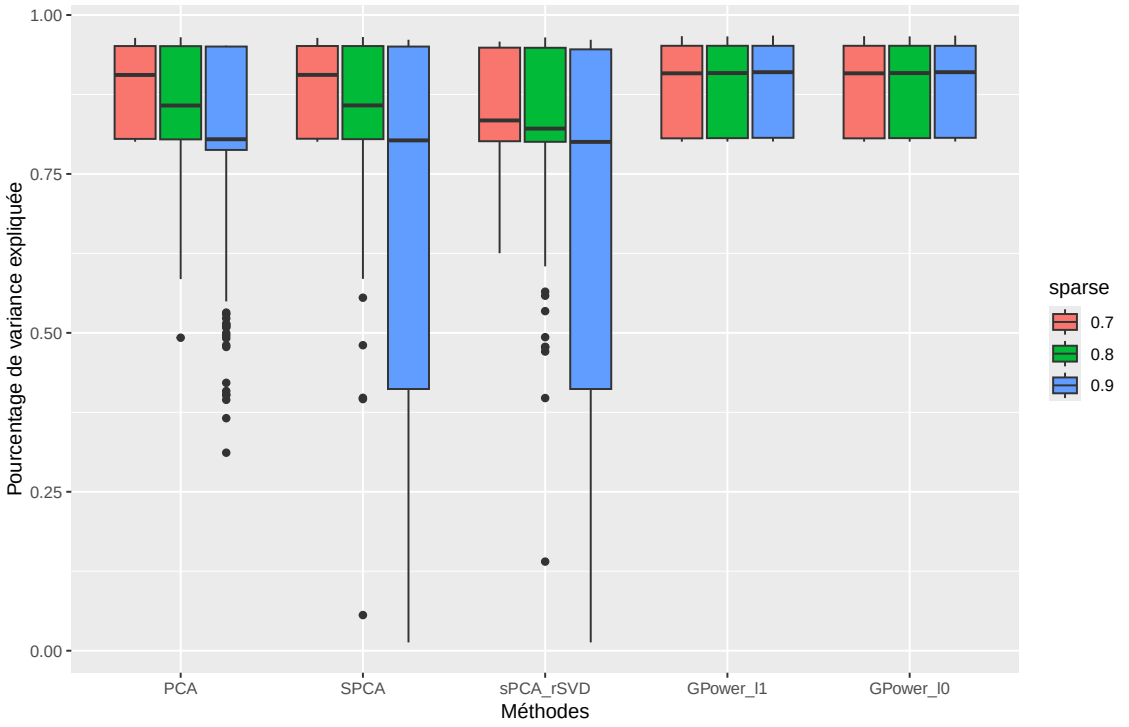

Figure B.7: Comparaison du pourcentage de variance expliquée par méthodes selon la proportion de parcimonie.

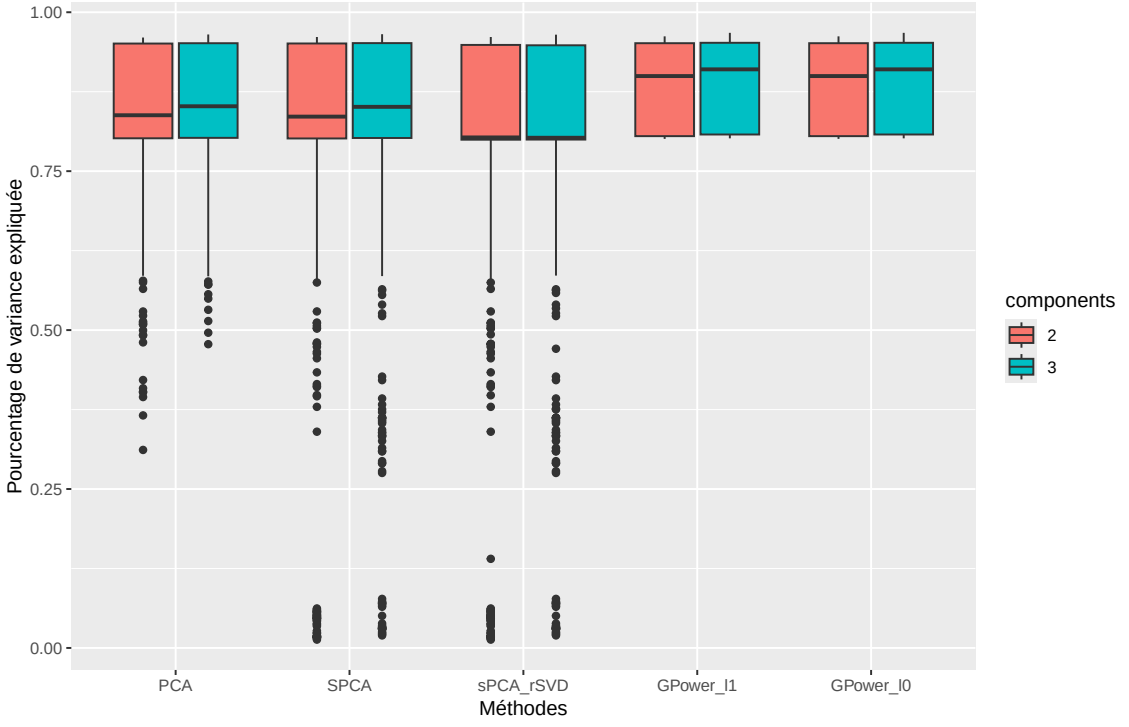

Figure B.8: Comparaison du pourcentage de variance expliquée par méthodes selon le nombre de composantes principales.

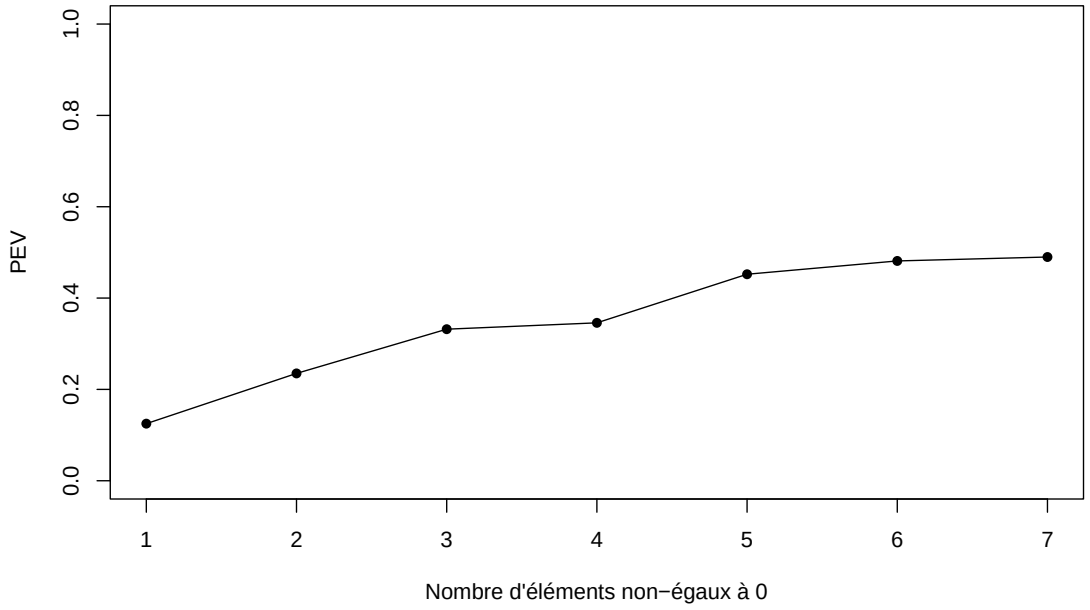

Figure C.9: Rapport entre le nombre d'éléments non-égaux à 0 que l'on désire dans la première composante principale et la proportion de variance expliquée obtenue par la méthode sPCA_rSVD.

C Graphiques de l'application

Figures C.9, C.10, C.11 et C.12 illustrant l'évolution de la PEV selon le niveau de parcimonie avec la méthode sPCA_rSVD.

Figures C.13, C.14, C.15 et C.16 illustrant l'évolution de la PEV selon le niveau de parcimonie avec la méthode GPower l_1.

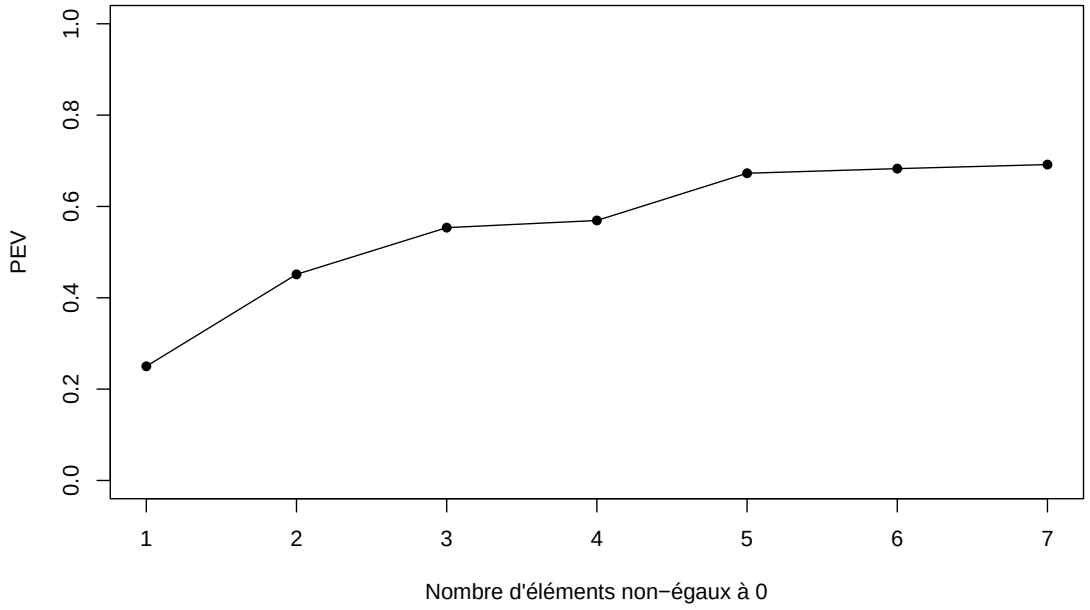

Figure C.10: Rapport entre le nombre d'éléments non-égaux à 0 que l'on désire dans chacune des 2 premières composantes principales et la proportion de variance expliquée obtenue par la méthode sPCA_rSVD.

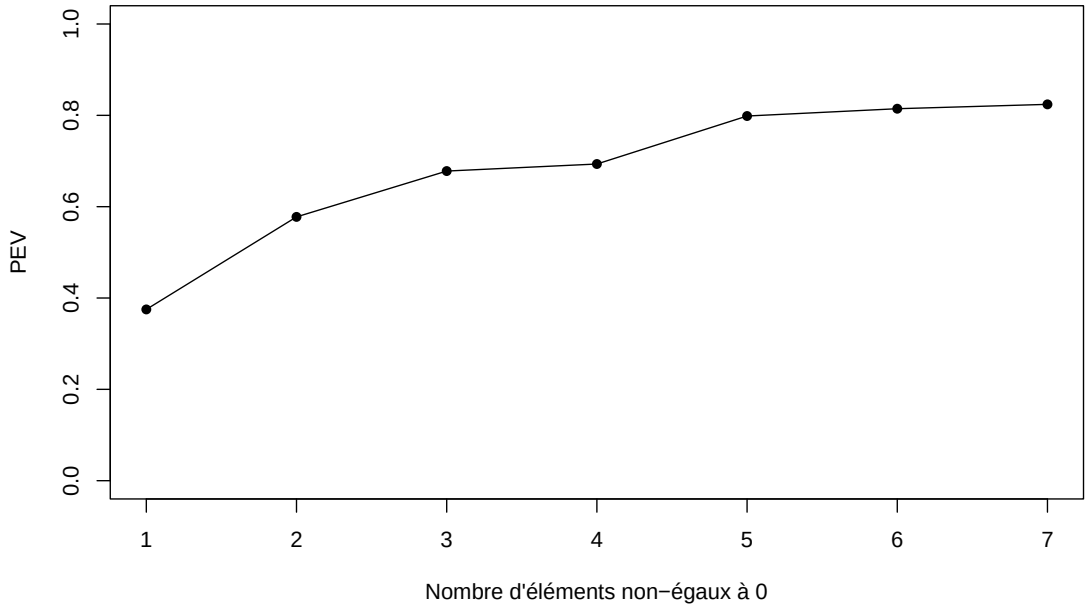

Figure C.11: Rapport entre le nombre d'éléments non-égaux à 0 que l'on désire dans chacune des 3 premières composantes principales et la proportion de variance expliquée obtenue par la méthode sPCA_rSVD.

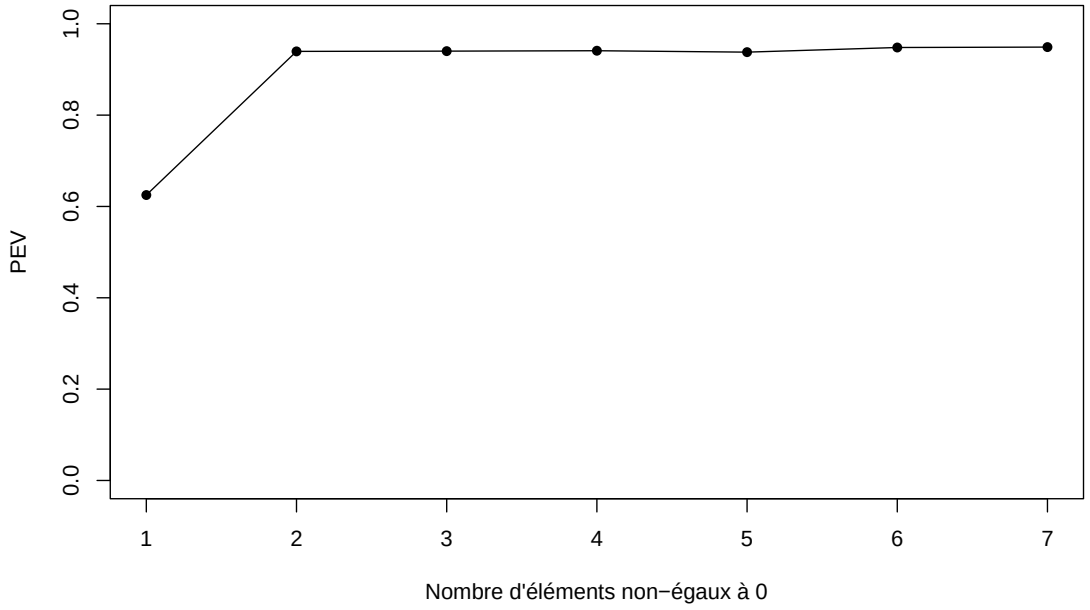

Figure C.12: Rapport entre le nombre d'éléments non-égaux à 0 que l'on désire dans chacune des 5 premières composantes principales et la proportion de variance expliquée obtenue par la méthode sPCA_rSVD.

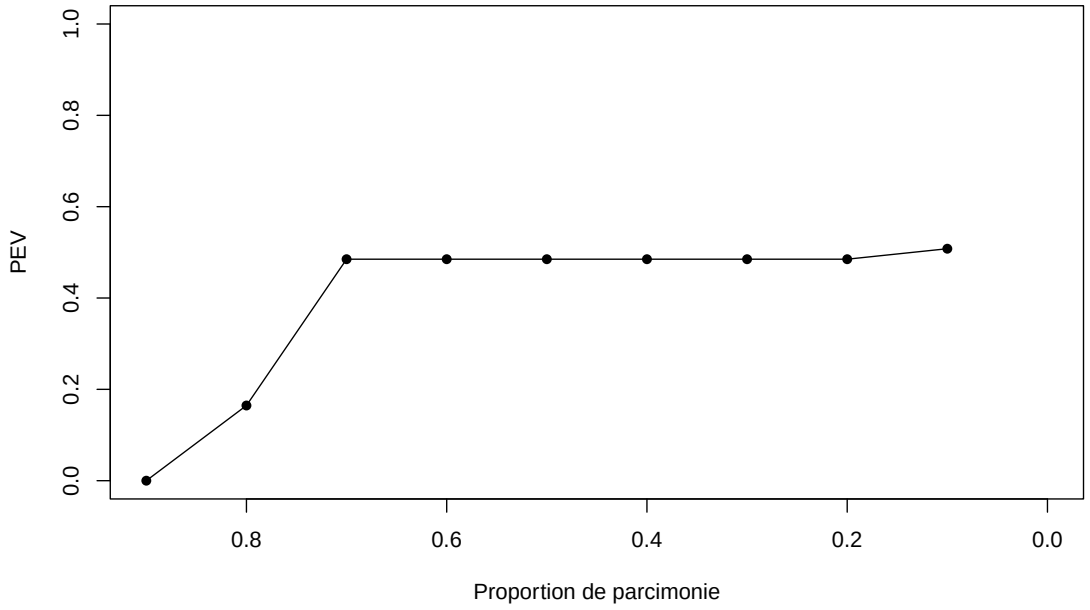

Figure C.13: Rapport entre la proportion de parcimonie que l'on désire dans la première composante principale et la proportion de variance expliquée obtenue par la méthode GPower l_1.

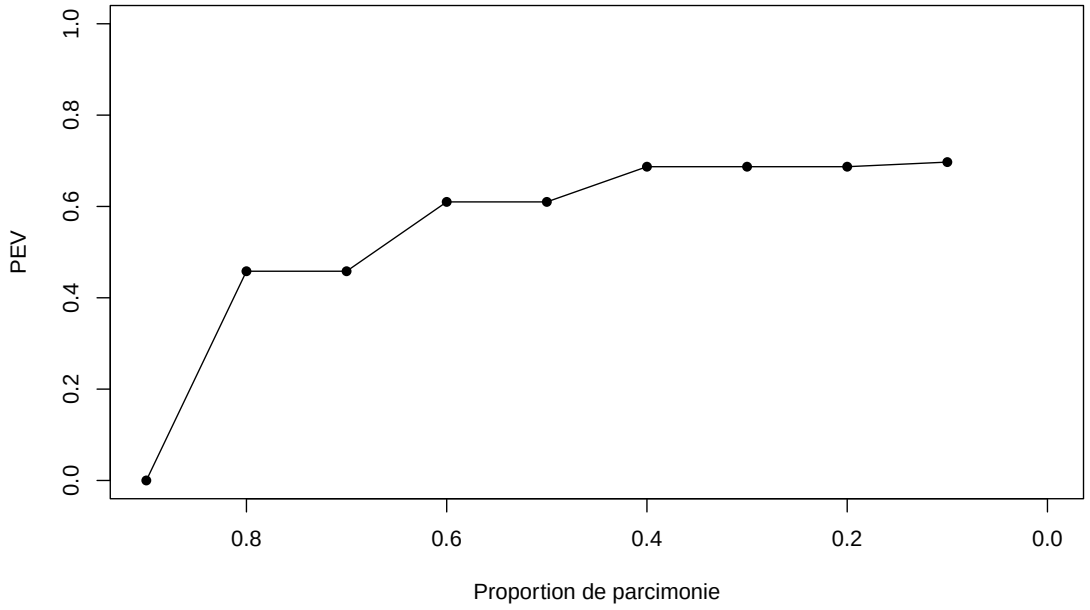

Figure C.14: Rapport entre la proportion de parcimonie que l'on désire dans l'ensemble des 2 premières composantes principales et la proportion de variance expliquée obtenue par la méthode GPower l_1.

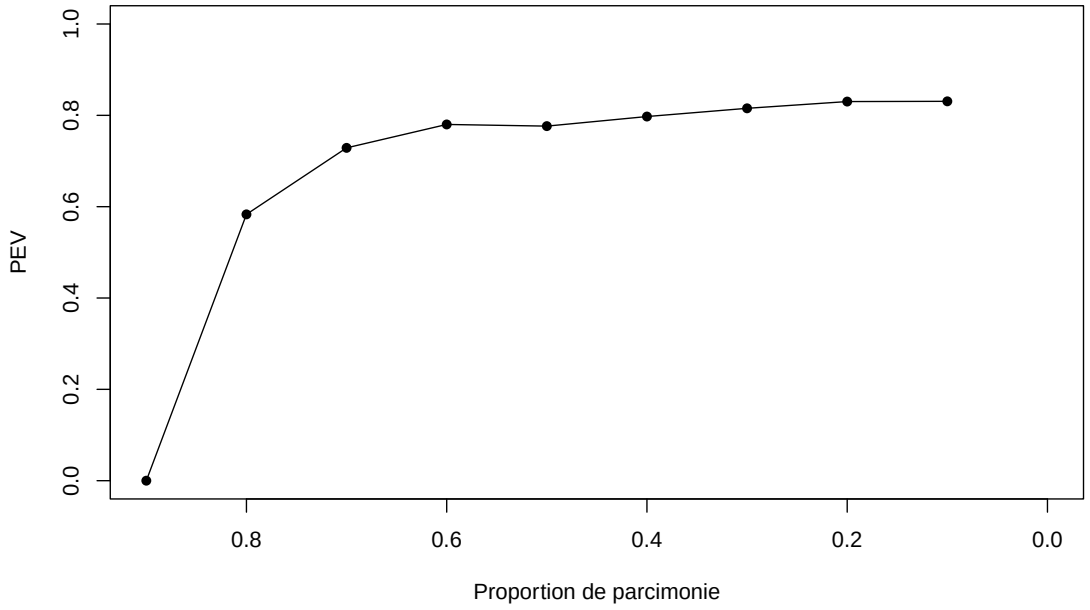

Figure C.15: Rapport entre la proportion de parcimonie que l'on désire dans l'ensemble des 3 premières composantes principales et la proportion de variance expliquée obtenue par la méthode GPower l_1.

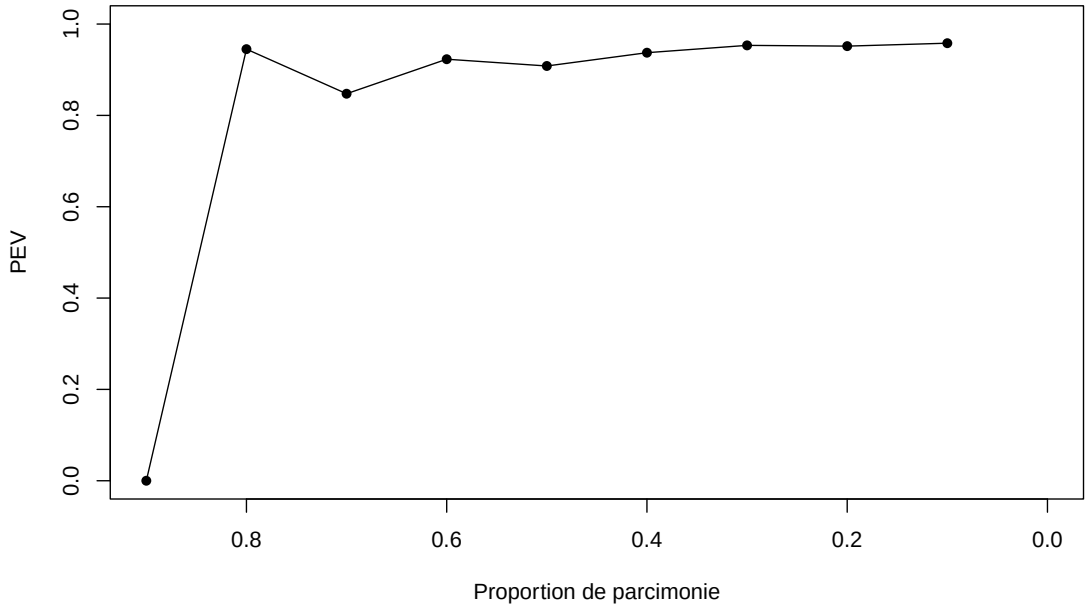

Figure C.16: Rapport entre la proportion de parcimonie que l'on désire dans l'ensemble des 5 premières composantes principales et la proportion de variance expliquée obtenue par la méthode GPower l_1.

D Fonction prcomp

Code source de la fonction `prcomp` disponible dans la base R (voir section 3.3.1).

```
prcomp.default <-
function(x, retx = TRUE, center = TRUE, scale. = FALSE, tol = NULL, ...)
{
    chkDots(...)
    x <- as.matrix(x)
    x <- scale(x, center = center, scale = scale.)
    cen <- attr(x, "scaled:center")
    sc <- attr(x, "scaled:scale")
    if(any(sc == 0))
        stop("cannot rescale a constant/zero column to unit variance")
    s <- svd(x, nu = 0)
    s$d <- s$d / sqrt(max(1, nrow(x) - 1))
    if (!is.null(tol)) {
        ## we get rank at least one even for a 0 matrix.
        rank <- sum(s$d > (s$d[1L]*tol))
        if (rank < ncol(x)) {
            s$v <- s$v[, 1L:rank, drop = FALSE]
            s$d <- s$d[1L:rank]
        }
    }
```

```
    dimnames(s$v) <-
        list(colnames(x), paste0("PC", seq_len(ncol(s$v))))
    r <- list(sdev = s$d, rotation = s$v,
              center = if(is.null(cen)) FALSE else cen,
              scale = if(is.null(sc)) FALSE else sc)
    if (retx) r$x <- x %*% s$v
    class(r) <- "prcomp"
    r
}
```

E Fonction spca

Code source de la fonction `spca` disponible dans le package `elasticnet` (voir section 3.3.2).

```
spca <-
function(x, K, para, type = c("predictor", "Gram"),
    sparse = c("penalty", "varnum"), use.corr = FALSE,
    lambda = 1e-06, max.iter = 200, trace = FALSE, eps.conv = 0.001)
{
    call <- match.call()
    type <- match.arg(type)
    sparse <- match.arg(sparse)
    vn <- dimnames(x)[[2]]
    x <- switch(type, predictor = {
        n <- dim(x)[1]
        p <- dim(x)[2]
        if (n/p >= 100) {
            cat("You may wish to restart and
            use a more efficient way \n")
            cat("let the argument x be the sample
            covariance/correlation matrix and set type=Gram \n")
        }
        x <- scale(x, center = TRUE, scale = use.corr)
    }, Gram = {
        x <- rootmatrix(x)
    })
    svdobj <- svd(x)
    v <- svdobj$v
    totalvariance <- sum((svdobj$d)^2)
    alpha <- as.matrix(v[, 1:K, drop = FALSE])
    beta <- alpha
    for (i in 1:K) {
        y <- drop(x %*% alpha[, i])
        beta[, i] <- solvebeta(x, y, paras = c(lambda, para[i]),
            sparse = sparse)
    }
    xtx <- t(x) %*% x
    temp <- beta
    normtemp <- sqrt(apply(temp^2, 2, sum))
    normtemp[normtemp == 0] <- 1
    temp <- t(t(temp)/normtemp)
    k <- 0
    diff <- 1
    while ((k < max.iter) & (diff > eps.conv)) {
        k <- k + 1
        alpha <- xtx %*% beta
        z <- svd(alpha)
        alpha <- (z$u) %*% t(z$v)
```

```
        for (i in 1:K) {
            y <- drop(x %*% alpha[, i])
            beta[, i] <- solvebeta(x, y,
            paras = c(lambda, para[i]), sparse = sparse)
        }
        normbeta <- sqrt(apply(beta^2, 2, sum))
        normbeta[normbeta == 0] <- 1
        beta2 <- t(t(beta)/normbeta)
        diff <- convcheck(beta2, temp)
        temp <- beta2
        if (trace) {
            if (k%%10 == 0) {
                cat("iterations", k, fill = TRUE)
            }
        }
    }
    normbeta <- sqrt(apply(beta^2, 2, sum))
    normbeta[normbeta == 0] <- 1
    beta <- t(t(beta)/normbeta)
    dimnames(beta) <- list(vn, paste("PC", 1:K, sep = ""))
    u <- x %*% beta
    R <- qr.R(qr(u))
    pev <- diag(R^2)/totalvariance
    obj <- list(call = call, type = type, K = K,
        loadings = beta, pev = pev, var.all = totalvariance,
        vn = vn, para = para, lambda = lambda)
    class(obj) <- "spca"
    obj
}
```

E.1 solvebeta

Code source de la fonction `solvebeta` utilisée dans la fonction `spca` .

```
solvebeta <-
function (x, y, paras, max.steps, sparse = c("penalty", "varnum"),
    eps = .Machine$double.eps)
{
    sparse <- match.arg(sparse)
    if (missing(sparse))
        sparse <- "penalty"
    nm <- dim(x)
    n <- nm[1]
    m <- nm[2]
    im <- seq(m)
    one <- rep(1, n)
    vn <- dimnames(x)[[2]]
    lambda <- paras[1]
    if (lambda > 0) {
```

```
    maxvars <- m
}
if (lambda == 0) {
    maxvars <- min(m, n - 1)
    if (m == n) {
        maxvars <- m
    }
}
d1 <- sqrt(lambda)
d2 <- 1/sqrt(1 + lambda)
Cvec <- drop(t(y) %*% x) * d2
ssy <- sum(y^2)
residuals <- c(y, rep(0, m))
if (missing(max.steps)) {
    max.steps <- 50 * maxvars
}
penalty <- max(abs(Cvec))
if (sparse == "penalty" && penalty * 2/d2 <= paras[2]) {
    beta <- rep(0, m)
}
else {
    beta <- rep(0, m)
    first.in <- integer(m)
    active <- NULL
    ignores <- NULL
    actions <- as.list(seq(max.steps))
    drops <- FALSE
    Sign <- NULL
    R <- NULL
    k <- 0
    while ((k < max.steps) & (length(active) < maxvars -
        length(ignores))) {
        action <- NULL
        k <- k + 1
        inactive <- if (k == 1)
            im
        else im[-c(active, ignores)]
        C <- Cvec[inactive]
        Cmax <- max(abs(C))
        if (!any(drops)) {
            new <- abs(C) == Cmax
            C <- C[!new]
            new <- inactive[new]
            for (inew in new) {
              R <- updateRR(x[, inew], R, x[, active], lambda)
              if (attr(R, "rank") == length(active)) {
                nR <- seq(length(active))
                R <- R[nR, nR, drop = FALSE]
                attr(R, "rank") <- length(active)
```

```
        ignores <- c(ignores, inew)
        action <- c(action, -inew)
      }
      else {
        if (first.in[inew] == 0)
          first.in[inew] <- k
        active <- c(active, inew)
        Sign <- c(Sign, sign(Cvec[inew]))
        action <- c(action, inew)
      }
    }
}
else action <- -dropid
Gi1 <- backsolve(R, backsolvet(R, Sign))
A <- 1/sqrt(sum(Gi1 * Sign))
w <- A * Gi1
u1 <- drop(x[, active, drop = FALSE] %*% w * d2)
u2 <- rep(0, m)
u2[active] <- d1 * d2 * w
u <- c(u1, u2)
if (lambda > 0) {
    maxvars <- m - length(ignores)
}
if (lambda == 0) {
    maxvars <- min(m - length(ignores), n - 1)
}
if (length(active) == maxvars - length(ignores)) {
    gamhat <- Cmax/A
}
else {
    a <- (drop(u1 %*% x[, -c(active, ignores)]) +
      d1 * u2[-c(active, ignores)]) * d2
    gam <- c((Cmax - C)/(A - a), (Cmax + C)/(A +
      a))
    gamhat <- min(gam[gam > eps], Cmax/A)
    Cdrop <- c(C - gamhat * a, -C + gamhat * a) -
      (Cmax - gamhat * A)
}
dropid <- NULL
b1 <- beta[active]
z1 <- -b1/w
zmin <- min(z1[z1 > eps], gamhat)
if (zmin < gamhat) {
    gamhat <- zmin
    drops <- z1 == zmin
}
else drops <- FALSE
beta2 <- beta
beta[active] <- beta[active] + gamhat * w
```

```
            residuals <- residuals - (gamhat * u)
            Cvec <- (drop(t(residuals[1:n]) %*% x) +
                d1 * residuals[-(1:n)]) * d2
            penalty <- c(penalty, penalty[k] - abs(gamhat * A))
            if (sparse == "penalty" && rev(penalty)[1] * 2/d2 <=
                paras[2]) {
                s1 <- rev(penalty)[1] * 2/d2
                s2 <- rev(penalty)[2] * 2/d2
                beta <- (s2 - paras[2])/(s2 - s1) * beta + (paras[2] -
                  s1)/(s2 - s1) * beta2
                beta <- beta * d2
                break
            }
            if (any(drops)) {
                dropid <- seq(drops)[drops]
                for (id in rev(dropid)) {
                  R <- downdateR(R, id)
                }
                dropid <- active[drops]
                beta[dropid] <- 0
                active <- active[!drops]
                Sign <- Sign[!drops]
            }
            if (sparse == "varnum" && length(active) >= paras[2]) {
                break
            }
            if (!is.null(vn))
                names(action) <- vn[abs(action)]
            actions[[k]] <- action
        }
    }
    return(beta)
}
```

E.2 convcheck

Code source de la fonction `convcheck` utilisée dans la fonction `spca`.

```
convcheck <-
function (beta1, beta2)
{
    a <- apply(abs(beta1 + beta2), 2, max)
    b <- apply(abs(beta1 - beta2), 2, max)
    d <- length(a)
    x <- rep(1, d)
    for (i in 1:d) {
        x[i] <- min(a[i], b[i])
    }
    max(x) }
```

F Fonction ssvd

Code source de la fonction `ssvd` disponible dans le package `irlba` (voir section 3.3.3).

```
ssvd <-
function (x, k = 1, n = 2, maxit = 500, tol = 0.001, center = FALSE,
    scale. = FALSE, alpha = 0, tsvd = NULL, ...)
{
    if (alpha < 0 || alpha >= 1)
        stop("0 <= alpha < 1")
    if (is.logical(center) && center)
        center <- colMeans(x)
    if (is.logical(scale.)) {
        if (scale.) {
            if (is.numeric(center)) {
                f <- function(i) sqrt(sum((x[, i]
                - center[i])^2)/(nrow(x) - 1L))
                scale. <- vapply(seq(ncol(x)), f, pi,
                USE.NAMES = FALSE)
            }
            else scale. <- apply(x, 2L,
            function(v) sqrt(sum(v^2)/max(1, length(v) - 1L)))
        }
    }
    if (all(n > ncol(x) - 1)) {
        warning("no sparsity constraints specified")
        return(irlba(x, k, ...))
    }
    n <- ncol(x) - n
    if (length(n) != k)
        n <- rep(n, length.out = k)
    s <- tsvd
    if (is.null(tsvd))
        s <- irlba(x, k, scale = scale., center = center, ...)
    lambda <- c()
    soft <- function(x, u, p) {
        y <- crossprod(x, u)
        if (is.numeric(center))
            y <- y - sum(u) * center
        if (is.numeric(scale.))
            y <- y/scale.
        a <- abs(y)
        z <- apply(a, 2, sort)
        lambda <<- vapply(seq(length(p)), function(j) (1 - alpha) *
            z[p[j], j] + alpha * z[p[j] + 1, j], pi, USE.NAMES = FALSE)
        sign(y) * pmax(sweep(a, 2, lambda, `-`), 0)
    }
    s$v <- s$d * s$v
    iter <- 0
```

```
    delta_u <- Inf
    while (delta_u > tol && iter < maxit) {
        u <- s$u
        s$v <- soft(x, s$u, n)
        if (is.numeric(scale.))
            s$v <- s$v/scale.
        if (is.numeric(center)) {
            xsv <- x %*% s$v - drop(crossprod(center, s$v))
            s$u <- qr.Q(qr(xsv))
        }
        else {
            xsv <- x %*% s$v
            s$u <- qr.Q(qr(xsv))
        }
        s$u <- sweep(s$u, 2, apply(xsv, 2, function(x) sign(head(x[x !=
            0], 1)))/apply(s$u, 2, function(x) sign(head(x[x !=
            0], 1))), `*`)
        delta_u <- max(1 - diag(abs(crossprod(u, s$u))))
        iter <- iter + 1
    }
    if (iter >= maxit)
        warning("Maximum number of iterations reached
        before convergence: solution may not be optimal.
        Consider increasing 'maxit'.")
    s$v <- s$v %*% diag(1/sqrt(apply(s$v, 2, crossprod)),
        ncol(s$v), ncol(s$v))
    d <- s$v
    if (is.numeric(scale.))
        d <- d/scale.
    d1 <- x %*% d
    if (is.numeric(center))
        d1 <- d1 - drop(crossprod(center, d))
    d <- crossprod(s$u, d1)
    list(u = s$u, v = s$v, d = d, iter = iter, lambda = lambda,
        center = center, scale = scale., n = n, alpha = alpha)
}
```

F.1 irlba

Code source de la fonction `irlba` utilisée dans la fonction `ssvd`.

```
irlba <-
function (A, nv = 5, nu = nv, maxit = 1000, work = nv + 7, reorth = TRUE,
    tol = 1e-05, v = NULL, right_only = FALSE, verbose = FALSE,
    scale = NULL, center = NULL, shift = NULL, mult = NULL, fastpath = TRUE,
    svtol = tol, smallest = FALSE, ...)
{
    ropts <- options(warn = 1)
    mflag <- new.env()
```

```
mflag$flag <- FALSE
on.exit(options(ropts))
interchange <- FALSE
eps <- .Machine$double.eps
mcall <- as.list(match.call())
random <- eval(mcall[["rng"]])
if (is.null(random))
    random <- rnorm
maxritz <- eval(mcall[["maxritz"]])
if (is.null(maxritz))
    maxritz <- 3
eps2 <- eval(mcall[["invariant_subspace_tolerance"]])
if (is.null(eps2))
    eps2 <- eps^(4/5)
du <- eval(mcall[["du"]])
dv <- eval(mcall[["dv"]])
ds <- eval(mcall[["ds"]])
deflate <- is.null(du) + is.null(ds) + is.null(dv)
if (is.logical(scale) && !scale)
    scale <- NULL
if (is.logical(shift) && !shift)
    shift <- NULL
if (is.logical(center) && !center)
    center <- NULL
if (smallest)
    fastpath <- FALSE
if (any(dim(A) > 2^32 - 1))
    fastpath <- FALSE
if (deflate == 3) {
    deflate <- FALSE
}
else if (deflate == 0) {
    deflate <- TRUE
    warning("The deflation options have been deprecated.
        Please modify your code to not use them.")
    if (length(ds) > 1)
        stop("deflation limited to one dimension")
    if (!is.null(dim(du)))
        du <- du[, 1]
    if (!is.null(dim(dv)))
        dv <- dv[, 1]
}
else stop("all three du ds dv parameters must be specified for deflation")
if (!is.null(center)) {
    if (is.logical(center) && center)
        center <- colMeans(A)
    if (deflate)
        stop("the center parameter can't be specified together
            with deflation parameters")
```

```
    if (length(center) != ncol(A))
        stop("center must be a vector of length ncol(A)")
    if (fastpath && !right_only)
        du <- NULL
    else du <- 1
    ds <- 1
    dv <- center
    deflate <- TRUE
}
if ("integer" == typeof(A))
    A <- A + 0
iscomplex <- is.complex(A)
m <- nrow(A)
n <- ncol(A)
if (is.null(nu))
    nu <- nv
if (!is.null(mult) && deflate)
    stop("the mult parameter can't be specified together
        with deflation parameters")
missingmult <- FALSE
if (is.null(mult)) {
    missingmult <- TRUE
    mult <- `%*%`
}
k <- max(nu, nv)
if (k <= 0)
    stop("max(nu, nv) must be positive")
if (k > min(m - 1, n - 1))
    stop("max(nu, nv) must be strictly less than min(nrow(A), ncol(A))")
if (k >= 0.5 * min(m, n)) {
    warning("You're computing too large a percentage of
        total singular values, use a standard svd instead.")
}
if (work <= 1)
    stop("work must be greater than 1")
if (tol < 0)
    stop("tol must be non-negative")
if (maxit <= 0)
    stop("maxit must be positive")
if (work <= k && !right_only)
    work <- k + 1
if (work >= min(n, m)) {
    work <- min(n, m)
    if (work <= k) {
        k <- work - 1
        warning("Requested subspace dimension too large! Reduced to ",
            k)
    }
}
```

```
k_org <- k
w_dim <- work
if (right_only) {
    w_dim <- 1
    fastpath <- FALSE
}
if (n > m && smallest) {
    interchange <- TRUE
    temp <- m
    m <- n
    n <- temp
}
if (verbose) {
    message("Working dimension size ", work)
}
if (min(m, n) < 6) {
    A <- as.matrix(A)
    if (verbose)
        message("Tiny problem detected, using standard `svd` function.")
    if (!is.null(scale)) {
        A <- sweep(A, 2, scale, "/")
        dv <- dv/scale
    }
    if (!is.null(shift))
        A <- A + diag(shift, nrow(A), ncol(A))
    if (deflate) {
        if (is.null(du))
            du <- rep(1, nrow(A))
        A <- A - (ds * du) %*% t(dv)
    }
    s <- svd(A)
    if (smallest) {
        return(list(d = tail(s$d, k), u = s$u[, tail(seq(ncol(s$u)),
            k), drop = FALSE], v = s$v[, tail(seq(ncol(s$v),
            k)), drop = FALSE], iter = 0, mprod = 0))
    }
    return(list(d = s$d[1:k], u = s$u[, 1:nu, drop = FALSE],
        v = s$v[, 1:nv, drop = FALSE], iter = 0, mprod = 0))
}
if (deflate)
    fastpath <- fastpath && is.null(du)
fastpath <- fastpath && (("Matrix" %in% attributes(class(A)) &&
    ("dgCMatrix" %in% class(A))) || "matrix" %in% class(A))
if (fastpath && missingmult && !iscomplex && !right_only) {
    RESTART <- 0L
    RV <- RW <- RS <- NULL
    if (is.null(v)) {
        v <- random(n)
        if (verbose)
```

```
            message("Initializing starting vector v with samples
                from standard normal distribution.
                \nUse ‘set.seed‘ first for reproducibility.")
    }
    else if (is.list(v)) {
        if (is.null(v$v) || is.null(v$d) || is.null(v$u))
            stop("restart requires left and right singular vectors")
        if (max(nu, nv) <= min(ncol(v$u), ncol(v$v)))
            return(v)
        RESTART <- as.integer(length(v$d))
        RND <- random(n)
        RND <- orthog(RND, v$v)
        RV <- cbind(v$v, RND/norm2(RND))
        RW <- v$u
        RS <- v$d
        v <- NULL
    }
    SP <- ifelse(is.matrix(A), 0L, 1L)
    if (verbose)
        message("irlba: using fast C implementation")
    SCALE <- NULL
    SHIFT <- NULL
    CENTER <- NULL
    if (!is.null(scale)) {
        if (length(scale) != ncol(A))
            stop("scale length must match number of matrix columns")
        SCALE <- as.double(scale)
    }
    if (!is.null(shift)) {
        if (length(shift) != 1)
            stop("shift length must be 1")
        SHIFT <- as.double(shift)
    }
    if (deflate) {
        if (length(center) != ncol(A))
            stop("the centering vector length must match
                the number of matrix columns")
        CENTER <- as.double(center)
    }
    ans <- .Call(C_IRLB, A, as.integer(k), as.double(v),
        as.integer(work), as.integer(maxit), as.double(tol),
        as.double(eps2), as.integer(SP), as.integer(RESTART),
        RV, RW, RS, SCALE, SHIFT, CENTER, as.double(svtol))
    if (ans[[6]] == 0 || ans[[6]] == -2) {
        names(ans) <- c("d", "u", "v", "iter", "mprod", "err")
        ans$u <- matrix(head(ans$u, m * nu), nrow = m, ncol = nu)
        ans$v <- matrix(head(ans$v, n * nv), nrow = n, ncol = nv)
        if (tol * ans$d[1] < eps)
            warning("convergence criterion below machine epsilon")
```

```
        if (ans[[6]] == -2)
            warning("did not converge--results might be invalid!;
                try increasing work or maxit")
        return(ans[-6])
    }
    errors <- c("invalid dimensions", "didn't converge",
        "out of memory", "starting vector near the null space",
        "linear dependency encountered")
    erridx <- abs(ans[[6]])
    if (erridx > 1)
        warning("fast code path error ", errors[erridx],
            "; re-trying with fastpath=FALSE.", immediate. = TRUE)
}
W <- matrix(0, m, w_dim)
F <- matrix(0, n, 1)
restart <- FALSE
if (is.list(v)) {
    if (is.null(v$v) || is.null(v$d) || is.null(v$u))
        stop("restart requires left and right singular vectors")
    if (max(nu, nv) <= min(ncol(v$u), ncol(v$v)))
        return(v)
    right_only <- FALSE
    W[, 1:ncol(v$u)] <- v$u
    d <- v$d
    V <- matrix(0, n, work)
    V[, 1:ncol(v$v)] <- v$v
    restart <- TRUE
}
else if (is.null(v)) {
    V <- matrix(0, n, work)
    V[, 1] <- random(n)
}
else {
    V <- matrix(0, n, work)
    V[1:length(v)] <- v
}
B <- NULL
Bsz <- NULL
eps23 <- eps^(2/3)
iter <- 1
mprod <- 0
R_F <- NULL
sqrteps <- sqrt(eps)
Smax <- 1
Smin <- NULL
lastsv <- c()
if (restart) {
    B <- cbind(diag(d), 0)
    k <- length(d)
```

```
        F <- random(n)
        F <- orthog(F, V[, 1:k])
        V[, k + 1] <- F/norm2(F)
    }
    if (deflate && is.null(du))
        du <- 1
    while (iter <= maxit) {
        j <- 1
        if (iter == 1 && !restart) {
            V[, 1] <- V[, 1]/norm2(V[, 1])
        }
        else j <- k + 1
        j_w <- ifelse(w_dim > 1, j, 1)
        VJ <- V[, j]
        if (!is.null(scale)) {
            VJ <- VJ/scale
        }
        if (interchange)
            avj <- mult(VJ, A)
        else avj <- mult(A, VJ)
        W[, j_w] <- as.vector(avj)
        mprod <- mprod + 1
        if (!is.null(shift)) {
            W[, j_w] <- W[, j_w] + shift * VJ
        }
        if (deflate) {
            W[, j_w] <- W[, j_w] - ds * drop(cross(dv, VJ)) *
                du
        }
        if (iter != 1 && w_dim > 1 && reorth) {
            W[, j] <- orthog(W[, j, drop = FALSE], W[, 1:(j -
                1), drop = FALSE])
        }
        S <- norm2(W[, j_w, drop = FALSE])
        if (is.na(S) || S < eps2 && j == 1)
            stop("starting vector near the null space")
        if (is.na(S) || S < eps2) {
            if (verbose)
                message_once("invariant subspace found", flag = mflag)
            W[, j_w] <- random(nrow(W))
            if (w_dim > 1)
                W[, j] <- orthog(W[, j], W[, 1:(j - 1)])
            W[, j_w] <- W[, j_w]/norm2(W[, j_w])
            S <- 0
        }
        else W[, j_w] <- W[, j_w]/S
        while (j <= work) {
            j_w <- ifelse(w_dim > 1, j, 1)
            if (iscomplex) {
```

```
    if (interchange)
      F <- Conj(t(drop(mult(A, Conj(drop(W[, j_w])))))
    else F <- Conj(t(drop(mult(Conj(drop(W[, j_w])),
      A))))
}
else {
    if (interchange)
      F <- t(drop(mult(A, drop(W[, j_w]))))
    else F <- t(drop(mult(drop(W[, j_w]), A)))
}
if (!is.null(shift))
    F <- F + shift * W[, j_w]
if (!is.null(scale))
    F <- F/scale
if (deflate) {
    sub <- sum(W[, j_w]) * dv
    if (!is.null(scale))
      sub <- sub/scale
    F <- F - sub
}
mprod <- mprod + 1
F <- drop(F - S * V[, j])
F <- orthog(F, V[, 1:j, drop = FALSE])
if (j + 1 <= work) {
    R <- norm2(F)
    if (R < eps2) {
      if (verbose)
        message_once("invariant subspace found",
          flag = mflag)
      F <- matrix(random(dim(V)[1]), dim(V)[1], 1)
      F <- orthog(F, V[, 1:j, drop = FALSE])
      V[, j + 1] <- F/norm2(F)
      R <- 0
    }
    else V[, j + 1] <- F/R
    if (is.null(B))
      B <- cbind(S, R)
    else B <- rbind(cbind(B, 0), c(rep(0, ncol(B) -
      1), S, R))
    jp1_w <- ifelse(w_dim > 1, j + 1, 1)
    w_old <- W[, j_w]
    VJP1 <- V[, j + 1]
    if (!is.null(scale)) {
      VJP1 <- VJP1/scale
    }
    if (interchange)
      W[, jp1_w] <- drop(mult(drop(VJP1), A))
    else W[, jp1_w] <- drop(mult(A, drop(VJP1)))
    mprod <- mprod + 1
```

```
            if (!is.null(shift)) {
              W[, jp1_w] <- W[, jp1_w] + shift * VJP1
            }
            if (deflate) {
              W[, jp1_w] <- W[, jp1_w] - ds * drop(cross(dv,
                VJP1)) * du
            }
            W[, jp1_w] <- W[, jp1_w] - R * w_old
            if (reorth && w_dim > 1)
              W[, j + 1] <- orthog(W[, j + 1], W[, 1:j])
            S <- norm2(W[, jp1_w])
            if (S < eps2) {
              if (verbose)
                message_once("invariant subspace found",
                  flag = mflag)
              W[, jp1_w] <- random(nrow(W))
              if (w_dim > 1)
                W[, j + 1] <- orthog(W[, j + 1], W[, 1:j])
              W[, jp1_w] <- W[, jp1_w]/norm2(W[, jp1_w])
              S <- 0
            }
            else W[, jp1_w] <- W[, jp1_w]/S
        }
        else {
            B <- rbind(B, c(rep(0, j - 1), S))
        }
        j <- j + 1
    }
    Bsz <- nrow(B)
    R_F <- norm2(F)
    F <- F/R_F
    Bsvd <- svd(B)
    if (iter == 1) {
        Smax <- Bsvd$d[1]
        Smin <- Bsvd$d[Bsz]
    }
    else {
        Smax <- max(Smax, Bsvd$d[1])
        Smin <- min(Smin, Bsvd$d[Bsz])
    }
    Smax <- max(eps23, Smax)
    if (!reorth && Smin/Smax < sqrteps) {
        warning("The matrix is ill-conditioned.
            Basis will be reorthogonalized.")
        reorth <- TRUE
    }
    if (smallest) {
        jj <- seq(ncol(Bsvd$u), 1, by = -1)
        Bsvd$u <- Bsvd$u[, jj]
```

```
    Bsvd$d <- Bsvd$d[jj]
    Bsvd$v <- Bsvd$v[, jj]
}
R <- R_F * Bsvd$u[Bsz, , drop = FALSE]
ct <- convtests(Bsz, tol, k_org, Bsvd, abs(R), k, Smax,
    lastsv, svtol, maxritz, work, S)
if (verbose) {
    message("iter= ", iter, ", mprod= ", mprod, ", sv[",
        k_org, "]=", sprintf("%.2e", Bsvd$d[k_org]),
        ", %change=", sprintf("%.2e", (Bsvd$d[k_org] -
          lastsv[k_org])/Bsvd$d[k_org]), ", k=", ct$k)
}
lastsv <- Bsvd$d
k <- ct$k
if (ct$converged)
    break
if (iter >= maxit)
    break
if (smallest && (Smin/Smax > sqrteps)) {
    Bsvd2.d <- Bsvd$d
    Bsvd2.d <- diag(Bsvd2.d, nrow = length(Bsvd2.d))
    Bsvd2 <- svd(cbind(Bsvd2.d, t(R)))
    jj <- seq(ncol(Bsvd2$u), 1, by = -1)
    Bsvd2$u <- Bsvd2$u[, jj]
    Bsvd2$d <- Bsvd2$d[jj]
    Bsvd2$v <- Bsvd2$v[, jj]
    Bsvd$d <- Bsvd2$d
    Bsvd$u <- Bsvd$u %*% Bsvd2$u
    Bsvd$v <- cbind(rbind(Bsvd$v, rep(0, Bsz)), c(rep(0,
        Bsz), 1)) %*% Bsvd2$v
    V_B_last <- Bsvd$v[Bsz + 1, 1:k]
    s <- R_F * solve(B, cbind(c(rep(0, Bsz - 1), 1)))
    Bsvd$v <- Bsvd$v[1:Bsz, , drop = FALSE] + s %*% Bsvd$v[Bsz +
        1, ]
    qrv <- qr(cbind(rbind(Bsvd$v[, 1:k], 0), rbind(-s,
        1)))
    Bsvd$v <- qr.Q(qrv)
    R <- qr.R(qrv)
    V[, 1:(k + 1)] <- cbind(V, F) %*% Bsvd$v
    UT <- t(R[1:(k + 1), 1:k] + R[, k + 1] %*% rbind(V_B_last))
    B <- diag(Bsvd$d[1:k], nrow = k) %*% (UT * upper.tri(UT,
        diag = TRUE))[1:k, 1:(k + 1)]
}
else {
    V[, 1:(k + dim(F)[2])] <- cbind(V[, 1:(dim(Bsvd$v)[1]),
        drop = FALSE] %*% Bsvd$v[, 1:k], F)
    B <- cbind(diag(Bsvd$d[1:k], nrow = k), R[1:k])
}
if (w_dim > 1) {
```

```
            W[, 1:k] <- W[, 1:(dim(Bsvd$u)[1]), drop = FALSE] %*%
                Bsvd$u[, 1:k]
        }
        iter <- iter + 1
    }
    if (!ct$converged)
        warning("did not converge--results might be invalid!;
            try increasing maxit or work")
    d <- Bsvd$d[1:k_org]
    if (!right_only) {
        u <- W[, 1:(dim(Bsvd$u)[1]), drop = FALSE] %*% Bsvd$u[,
            1:k_org, drop = FALSE]
    }
    v <- V[, 1:(dim(Bsvd$v)[1]), drop = FALSE] %*% Bsvd$v[, 1:k_org,
        drop = FALSE]
    if (smallest) {
        reverse <- seq(length(d), 1)
        d <- d[reverse]
        if (!right_only)
            u <- u[, reverse]
        v <- v[, reverse]
    }
    if (tol * d[1] < eps)
        warning("convergence criterion below machine epsilon")
    if (right_only)
        return(list(d = d, v = v[, 1:nv, drop = FALSE], iter = iter,
            mprod = mprod))
    return(list(d = d, u = u[, 1:nu, drop = FALSE], v = v[, 1:nv,
        drop = FALSE], iter = iter, mprod = mprod))
}
```

G Fonction gpower

Code source de la fonction `gpower` disponible dans le package `gpowerr` (voir section 3.3.4).

```
gpower.default <-
  function(data,
           k,
           rho,
           reg = "l1",
           center = TRUE,
           block = FALSE,
           mu = 1,
           iter_max = 1000,
           epsilon = 1e-04) {
    data <- as.matrix(data)

    sparsity_error <-
      "Sparcity is set too high, all entries of loading vector are zero"
    reg_method <- "Regularisation method not recognized"
    iter_warning <- "Maximum number of iterations was reached"

    # Checks ------------------------------------------------------------------

    if (any(is.na(data))) {
      warning("Missing values are omitted: na.omit(X).")
      X <- stats::na.omit(X)
    }

    if (any(is.na(rho))) {
      warning("NA found in rho")
    }

    if (block == 1 & any(is.na(mu))) {
      warning("Mu needs to be defined for block algoritm")
    }

    picked <- FALSE

    # Initialize gpower object ------------------------------------------------
    gpowerObj <- list(weights = NULL, scores = NULL)

    p <- nrow(data)
    n <- ncol(data)

    iter_max <- iter_max
    epsilon <- epsilon

    Z <- matrix(0, n, k, dimnames = list(colnames(data)))
```

```
W <- matrix(0, n, k, dimnames = list(colnames(data)))

if (length(rho) == 1) {
  rho <- rep(rho, k)
}

if (length(mu) == 1 & !any(is.na(mu))) {
  mu <- rep(mu, k)
}
# Add centering

if (center) {
  data <- scale(data, scale = TRUE)
  gpowerObj$centers <- attr(data, "scaled:center")
}

X <- data

# Single unit algorithm ----------------------------------------------------

if (!picked & (k == 1 | (k > 1 & !block))) {
  if (reg == "l1") {
    for (c in 1:k) {
      # Loop on the components
      norm_a_i <- rep(0, n)

      for (i in 1:n) {
        norm_a_i[i] <- norm(X[, i], "2")
      }

      rho_max <- max(norm_a_i)
      i_max <- which.max(norm_a_i)
      rho_c <- rho[c] * rho_max

      # Initialisation Allow for warm start
      x <- X[, i_max] / norm_a_i[i_max]
      f <- rep(0, iter_max)
      iter <- 1

      while (TRUE) {
        X_x <- t(X) %*% x
        t_resh <-
          sign(X_x) * pmax(abs(X_x) - rho_c, 0)

        # Cost function
        f[iter] <- sum(t_resh ^ 2)

        if (f[iter] == 0) {
          # Sparsity is too high
```

```
        warning(sparsity_error)
        break
      } else {
        gradient <- X %*% t_resh
        x <- gradient / norm(gradient, "2")
      }

      if (iter > 2) {
        # Stopping criteria
        if ((f[iter] - f[iter - 1]) / f[iter - 1] < epsilon |
            iter > iter_max) {
          if (iter > iter_max) {
            print(iter_warning)
            break
          }
          break
        }
      }

      iter <- iter + 1
    }

    X_x <- t(X) %*% x
    pattern <-
      (abs(X_x) - rho_c) > 0  # Pattern of sparsity
    z <- sign(X_x) * max(abs(X_x) - rho_c, 0)

    if (max(abs(z > 0))) {
      z <- z / norm(z, "2")
    }

    z <- pattern_filling(X, pattern, z)

    # Deflate
    y <- X %*% z
    X <- X - y %*% z

    W[, c] <- z
  }
} else if (reg == "l0") {
  for (c in 1:k) {
    # Loop on the components
    rho_c <- rho[c]
    norm_a_i <- rep(0, n)

    for (i in 1:n) {
      norm_a_i[i] <- norm(X[, i], "2")
    }
```

```
rho_max <- max(norm_a_i)
i_max <- which.max(norm_a_i)
rho_c <- rho_c * rho_max ^ 2

# Initialisation
x <- X[, i_max] / norm_a_i[i_max]
f <- rep(0, iter_max)
iter <- 1

while (TRUE) {
  X_x <- t(X) %*% x
  t_resh <- pmax(X_x ^ 2 - rho_c, 0)

  # Cost function
  f[iter] <- sum(t_resh)

  if (f[iter] == 0) {
    # Sparsity is too high
    warning(sparsity_error)
    break
  } else {
    gradient <- X %*% ((t_resh > 0) * X_x)
    x <- gradient / norm(gradient, "2")
  }

  if (iter > 2) {
    # Stopping criteria
    if ((f[iter] - f[iter - 1]) / f[iter - 1] < epsilon |
        iter > iter_max) {
      if (iter > iter_max) {
        print(iter_warning)
        break
      }
      break
    }
  }

  iter <- iter + 1
}

pattern <-
  ((t(X) %*% x) ^ 2 - rho_c > 0)  # Pattern of sparsity
y <- x
pattern_inv <- pattern == 0

z <- t(X) %*% y
z[pattern_inv] <- 0
norm_z <- norm(z, "2")
z <- z / norm_z
```

```
        y <- y * norm_z

        # Deflate
        X <- X - y %*% t(z)

        W[, c] <- z
      }
    } else {
      warning(reg_method)
    }
    picked <- TRUE
  }

  # Update gpower object ------------------------------------------------------
  scores <- data %*% W
  P <- t(data) %*% MASS::ginv(t(scores))
  data_approx <- scores %*% t(P)
  colnames(W) <- as.character(1:k)

  gpowerObj$weights <- W
  gpowerObj$scores <- scores
  gpowerObj$a_approx <- data_approx
  gpowerObj$prop_sparse <- sum(rowSums(W == 0)) / (n * k)

  # Add component variance (p22)
  AdjVar <- qr.R(qr(scores)) ^ 2

  ### Extract values on the diagonal
  comp_var <- rep(NA, k)
  for (i in 1:k) {
    comp_var[i] <- AdjVar[i, i]
  }

  # Explained variance ratio
  gpowerObj$comp_var <- sort(comp_var, decreasing = TRUE)
  gpowerObj$exp_var <-
    1 - (norm((data - data_approx), type = "F") / norm(data, type = "F")) ^
    2

  class(gpowerObj) <- "gpower"

  gpowerObj
}
```

G.1 pattern_filling

Code source de la fonction `pattern_filling` utilisée dans la fonction `gpower`.

```
pattern_filling <- function(A, pattern, Z = NA, mu = NA) {
```

```r
# Compute a local maximizer of max_{X,Z} trace(X^T A Z N)
# s.t. X^T X=I_m and Z(P)=0 and Diag(Z^T Z)=I_m

p <- nrow(A)
n <- ncol(A)
m <- ncol(pattern)

# Single unit case ------------------------------------------------------------
if (m == 1) {
  support <- pattern != 0

  if (sum(support) == 0) {
    z_red <- rep(0, n)
    support <- 1:n
  } else if (sum(support) == 1) {
    z_red <- 1
  } else {
    u <- Z[support]
    epsilon <- 1e-06
    iter_max <- 1000
    f <- rep(0, iter_max)
    iter <- 1
    A_red <- A[, support]

    while (TRUE) {
      temp <- t(A_red) %*% (A_red %*% u)
      u <- temp / norm(temp, "2")
      f[iter] <- -2 * t(u) %*% temp

      if (iter > 2) {
        # Stopping criteria
        if (abs(f[iter] - f[iter - 1]) / abs(f[iter - 1]) < epsilon |
            iter > iter_max) {
          if (iter > iter_max) {
            print("Max iterations reached")
            break
          }
          break
        }
      }

      iter <- iter + 1
    }

    z_red <- u
  }

  z <- rep(0, n)
  z[support] <- z_red
```

```
    return(z)
  }
}
```

G.2 auto_gpower

Code source de la fonction `auto_gpower` utilisée pour trouver le `roh` optimal pour un certain niveau de parcimonie désiré.

```
auto_gpower.default <-
  function(data,
           k,
           prop_sparse,
           accuracy = 2,
           reg = "l1",
           center = TRUE,
           block = FALSE,
           mu = 1,
           iter_max = 1000,
           epsilon = 1e-04) {
    # Tunes rho to get desired proportion of sparsity

    n <- ncol(data)
    prop_zeros_needed <- round(prop_sparse, accuracy)
    prop_zeros_current <- 0

    if (prop_zeros_needed == 0) {
      # No sparsity
      gpower(data, k, 0, reg, center, block, mu, iter_max, epsilon)
    }

    if (block) {
      max_iterations <- 250
    } else {
      max_iterations <- 100
    }

    # Starting bounds of binary search
    lower <- 0
    upper <- 1
    i <- 0

    while (prop_zeros_needed != prop_zeros_current &
           max_iterations > i) {
      middle <- (lower + upper) / 2

      Z <- tryCatch(
        gpower(
          data = data,
```

```
        k = k,
        rho = middle,
        reg = reg,
        center = center,
        block = block,
        mu = mu,
        iter_max = iter_max,
        epsilon = epsilon
      ),
      error = function(e) {
        NULL
      }
    )

    if (is.list(Z)) {
      prop_zeros_current <-
        round(sum(rowSums(Z$weights == 0)) / (n * k), accuracy)

      if (prop_zeros_needed > prop_zeros_current) {
        lower <- middle
      }
      if (prop_zeros_needed < prop_zeros_current) {
        upper <- middle
      }
      if (prop_zeros_needed == prop_zeros_current) {
        break
      }
    } else {
      upper <- middle
    }
    i <- i + 1
  }

  if (is.list(Z)) {
    return(round(middle, 5))
  } else {
    warning("Unable to find a suitable rho for specified prop_sparse")
  }
}
```

H Code génération des données

Code de Guerra-Urzola et al. (2021) modifié permettant de générer les données. Le code est expliqué dans la section 3.2

```
library("doParallel")
library("mvtnorm")
library("MASS")
library("dplyr")

setwd("C:/UCL/MEMOIRE")
getwd()

dir.create("data_generated") # Directory to save the data

# sparse PCA data simulation #
VAFx = c(.80, .95)           # Proportion of explained variance
p_sparse = c(.7,.8,.9)        # Proportion of sparsity
n_components = c(2,3)           # Number of components
s_size = c(200)                 # Sample size
n_variables = c(10,100,1000)    # Number of variables
n_replications = c(30)           # Number of repetitions

design_matrix <- expand.grid(n_variables=n_variables,s_size=s_size,
                         p_sparse=p_sparse, n_components=n_components,VAFx =VAFx)

design_matrix_replication <- design_matrix[rep(1:nrow(design_matrix),
                                    times = n_replications), ]

Infor_simulation = list(n_data_sets = nrow(design_matrix_replication),
                     n_replications  =n_replications,
                     design_matrix_replication = design_matrix_replication)

save(Infor_simulation, file = "data_generated/Information.RData")

# start simulating the data

sim_data <- foreach(i=1:nrow(design_matrix_replication),
             .options.RNG = 2018,
             .packages = c("MASS","mvtnorm"),
             .combine=rbind)%dopar%{

              set.seed(i) # added to keep the same results

              # set the specific values of the parameters
              R = design_matrix_replication$n_components[i]
              I =  design_matrix_replication$s_size[i]
              J = design_matrix_replication$n_variables[i]
              vafx = design_matrix_replication$VAFx[i]
```

```
propsparse = design_matrix_replication$p_sparse[i]

n_zeros = floor(J*propsparse)
n_non_zero = J-n_zeros

# generating the data
X = mvrnorm(n =I,mu=rep(0,J),Sigma = diag(1,J))

# SVD on Xinit
svd1 <- svd(X)

# P matrix is the right singular vectors
P <- svd1$v[,1:R]
indxnonzero = matrix(sample(1:J,
   size = (n_non_zero*R)),n_non_zero,R )
for (z in 1:R) {
   P[-indxnonzero[,z],z] = 0
}

# Normalizing the columns of P
P <- P%*%diag(svd1$d[1:R])
tmat <- svd1$u[,1:R]
Xtrue <- tmat %*% t(P)

W2  <- P %*% diag(1/sqrt(diag(t(P) %*% P)))
tmat2 = X%*%W2
Xtrue2 = tmat2%*%t(W2)

# Adding noice 1 %
SSqXtrue =  sum(Xtrue^2)        # sum squares of the data set
EX = mvrnorm(I,mu= rep(0,J),Sigma = diag(1,J))  # EX = Error of X
SSqEX = sum(EX^2)               # Sum squares fo the EX
fx = sqrt(SSqXtrue*(1-vafx)/(vafx * SSqEX))
Xnew = Xtrue + fx*EX            # Data with noise

# Adding noice 2 %
SSqXtrue =  sum(Xtrue2^2)       # sum squares of the data set
EX = mvrnorm(I,mu= rep(0,J),Sigma = diag(1,J))  # EX = Error of X
SSqEX = sum(EX^2)               # Sum squares fo the EX
fx = sqrt(SSqXtrue*(1-vafx)/(vafx * SSqEX))
Xnew2 = Xtrue2 + fx*EX          # Data with noise

Xtot = as.matrix(bind_rows(data.frame(Xnew),
   data.frame(Xnew2)))  #joining both X
Ztot = as.matrix(bind_rows(data.frame(tmat),
   data.frame(tmat2)))  #joining both Z
```

```
            # Saving the data and structure
            out = list(X = Xtot, W2 = W2 ,P = P, Z = Ztot ,k = R,
                       Propsparse = propsparse, nzeros =  n_zeros, teller = i)
            save(out, file=paste0("data_generated/PWsparse",i, ".RData"))
            }

# stop Cluster
stopCluster(c1)
```

I Code simulation

Code de Guerra-Urzola et al. (2021) modifié permettant de sortir les performances des différentes méthodes, et code original produisant des graphiques de comparaison. Le code est expliqué et les résultats analysés dans la section 3.4.

```
library("corrr")
library("ggcorrplot")
library("ggplot2")
library("ggpubr")
library("FactoMineR")
library("factoextra")
library("elasticnet")
library("doParallel")
library("MASS")
library("Matrix")
library("Rcpp")
library("superpc")
library('ltsspca')
library("dplyr")
library("irlba")
library("forcats")
library("expm")
devtools::install_github("plofknaapje/gpowerr")

setwd("C:/Users/wimil_a1d1oc1/OneDrive/Documents/UCL/MEMOIRE/Code")

#Simulation

load("C:/UCL/MEMOIRE/data_generated/information.Rdata")
source("GPower.R")

Ndatasets = Infor_simulation$n_data_sets
MatrixInfo = Infor_simulation$design_matrix_replication

Simulation = foreach(i=1:Ndatasets,.packages = c("elasticnet","combinat","MASS"),
             .combine=rbind)%dopar%{

             # Loading data

             load(paste0("C:/UCL/MEMOIRE/data_generated/PWsparse",i,".RData"))
             Out2 = list(X = out$X, P =out$P, W = out$W, Z= out$Z,
             nzeros = out$nzeros, k = out$k,Propsparse= out$Propsparse )

             # Estimation of the models #

             n_non_zeros2 = dim(Out2$X)[2]-Out2$nzeros

             Wmatrix1 = prcomp(Out2$X,center = TRUE, scale. = TRUE)
```

```
Wmatrix2 = spca(x=Out2$X,K = Out2$k, sparse = "varnum",
para =rep(n_non_zeros2,Out2$k) ,
type = "predictor", use.corr = TRUE)
Wmatrix3 = ssvd(Out2$X, k=Out2$k, n=n_non_zeros2,
center = TRUE, scale. = TRUE)
Wmatrix4 = gpower(data=Out2$X, k=Out2$k, rho=auto_gpower(Out2$X,
Out2$k, prop_sparse = Out2$Propsparse, reg = "l1",
center = TRUE), reg="l1", center=TRUE)
Wmatrix5 = gpower(data=Out2$X, k=Out2$k, rho=auto_gpower(Out2$X,
Out2$k, prop_sparse = Out2$Propsparse, reg = "l0",
center = TRUE), reg="l0", center=TRUE)

W1 = Wmatrix1$rotation[,1:Out2$k] # CP coefficients
W2 = Wmatrix2$loadings # Weights
W3 = Wmatrix3$v # Loadings
W4 = Wmatrix4$weights # Weights
W5 = Wmatrix5$weights # Weights

Z1 = Out2$X%*%W1 # principal component
Z2 = Out2$X%*%W2
Z3 = Out2$X%*%W3
Z4 = Out2$X%*%W4
Z5 = Out2$X%*%W5

out_error1 =  RelativeError(Out2$W, W1) # P and W sparse
out_error2 =  RelativeError(Out2$W, W2)
out_error3 =  RelativeError(Out2$W, W3)
out_error4 =  RelativeError(Out2$W, W4)
out_error5 =  RelativeError(Out2$W, W5)

W1 = W1[,out_error1$permutation]%*%diag(out_error1$s)
W2 = W2[,out_error2$permutation]%*%diag(out_error2$s)
W3 = W3[,out_error3$permutation]%*%diag(out_error3$s)
W4 = W4[,out_error4$permutation]%*%diag(out_error4$s)
W5 = W5[,out_error5$permutation]%*%diag(out_error5$s)

Z1 = Z1[,out_error1$permutation]%*%diag(out_error1$s)
Z2 = Z2[,out_error2$permutation]%*%diag(out_error2$s)
Z3 = Z3[,out_error3$permutation]%*%diag(out_error3$s)
Z4 = Z4[,out_error4$permutation]%*%diag(out_error4$s)
Z5 = Z5[,out_error5$permutation]%*%diag(out_error5$s)

ErrorLW1 = out_error1$Error
ErrorLW2 = out_error2$Error
ErrorLW3 = out_error3$Error
ErrorLW4 = out_error4$Error
ErrorLW5 = out_error5$Error

ErrorScores1 = (norm((Out2$Z-Z1),type = "F")/
```

```
    norm(Out2$Z,type = "F"))^2
ErrorScores2 = (norm((Out2$Z-Z2),type = "F")/
    norm(Out2$Z,type = "F"))^2
ErrorScores3 = (norm((Out2$Z-Z3),type = "F")/
    norm(Out2$Z,type = "F"))^2
ErrorScores4 = (norm((Out2$Z-Z4),type = "F")/
    norm(Out2$Z,type = "F"))^2
ErrorScores5 = (norm((Out2$Z-Z5),type = "F")/
    norm(Out2$Z,type = "F"))^2

ZeroInd = which(as.vector(Out2$W)==0)

W1_vec = as.vector(W1)
W2_vec = as.vector(W2)
W3_vec = as.vector(W3)
W4_vec = as.vector(W4)
W5_vec = as.vector(W5)

if (Out2$Propsparse == 0){
    MR01 = 0
    MR02 = 0
    MR03 = 0
    MR04 = 0
    MR05 = 0
}else{
    wz1 = W1_vec[ZeroInd]
    MR01 = 1-sum(wz1 == 0)/length(ZeroInd)

    wz2 = W2_vec[ZeroInd]
    MR02 = 1-sum(wz2 == 0)/length(ZeroInd)

    wz3 = W3_vec[ZeroInd]
    MR03 = 1-sum(wz3 == 0)/length(ZeroInd)

    wz4 = W4_vec[ZeroInd]
    MR04 = 1-sum(wz4 == 0)/length(ZeroInd)

    wz5 = W5_vec[ZeroInd]
    MR05 = 1-sum(wz5 == 0)/length(ZeroInd)
}


# Sparse Variance #
P1 = t(Out2$X)%*%ginv(t(Z1))
P2 = t(Out2$X)%*%ginv(t(Z2))
P3 = t(Out2$X)%*%ginv(t(Z3))
P4 = t(Out2$X)%*%ginv(t(Z4))
P5 = t(Out2$X)%*%ginv(t(Z5))
```

```
            Xs1 = Z1%*%t(P1)
            Xs2 = Z2%*%t(P2)
            Xs3 = Z3%*%t(P3)
            Xs4 = Z4%*%t(P4)
            Xs5 = Z5%*%t(P5)

            Svar1 = 1 - (norm((Out2$X-Xs1),type = "F")/
                norm(Out2$X,type = "F"))^2
            Svar2 = 1 - (norm((Out2$X-Xs2),type = "F")/
                norm(Out2$X,type = "F"))^2
            Svar3 = 1 - (norm((Out2$X-Xs3),type = "F")/
                norm(Out2$X,type = "F"))^2
            Svar4 = 1 - (norm((Out2$X-Xs4),type = "F")/
                norm(Out2$X,type = "F"))^2
            Svar5 = 1 - (norm((Out2$X-Xs5),type = "F")/
                norm(Out2$X,type = "F"))^2


            # Saving performance #
            return(list(ErrorScores1 = ErrorScores1, MR01 = MR01,Svar1 = Svar1,
                    ErrorScores2 = ErrorScores2, MR02 = MR02,Svar2 = Svar2,
                    ErrorScores3 = ErrorScores3, MR03 = MR03,Svar3 = Svar3,
                    ErrorScores4 = ErrorScores4, MR04 = MR04,Svar4 = Svar4,
                    ErrorScores5 = ErrorScores5, MR05 = MR05,Svar5 = Svar5))
            }

## Results in a table ##
ErrorScores = as.numeric(unlist(Simulation[,"ErrorScores1"]))

MR0 = as.numeric(unlist(Simulation[,"MR01"]))

Svar = as.numeric(unlist(Simulation[,"Svar1"]))

Methodology = rep('PCA',Ndatasets)

p_sparse = Infor_simulation$design_matrix_replication[,"p_sparse"]

DataPCA = cbind(Methodology,p_sparse, MatrixInfo,
                ErrorScores,MR0, Svar)

ErrorScores = as.numeric(unlist(Simulation[,"ErrorScores2"]))

MR0 = as.numeric(unlist(Simulation[,"MR02"]))

Svar = as.numeric(unlist(Simulation[,"Svar2"]))

Methodology = rep('SPCA',Ndatasets)

p_sparse = Infor_simulation$design_matrix_replication[,"p_sparse"]
```

```
DataSPCA = cbind(Methodology,p_sparse, MatrixInfo,
                 ErrorScores,MR0, Svar)

ErrorScores = as.numeric(unlist(Simulation[,"ErrorScores3"]))

MR0 = as.numeric(unlist(Simulation[,"MR03"]))

Svar = as.numeric(unlist(Simulation[,"Svar3"]))

Methodology = rep('sPCA_rSVD',Ndatasets)

p_sparse = Infor_simulation$design_matrix_replication[,"p_sparse"]

DatasPCArSVD = cbind(Methodology,p_sparse, MatrixInfo,
                 ErrorScores,MR0, Svar)

ErrorScores = as.numeric(unlist(Simulation[,"ErrorScores4"]))

MR0 = as.numeric(unlist(Simulation[,"MR04"]))

Svar = as.numeric(unlist(Simulation[,"Svar4"]))

Methodology = rep('GPower_l1',Ndatasets)

p_sparse = Infor_simulation$design_matrix_replication[,"p_sparse"]

DataGPower_l1 = cbind(Methodology,p_sparse, MatrixInfo,
                 ErrorScores,MR0, Svar)

ErrorScores = as.numeric(unlist(Simulation[,"ErrorScores5"]))

MR0 = as.numeric(unlist(Simulation[,"MR05"]))

Svar = as.numeric(unlist(Simulation[,"Svar5"]))

Methodology = rep('GPower_l0',Ndatasets)

p_sparse = Infor_simulation$design_matrix_replication[,"p_sparse"]

DataGPower_l0 = cbind(Methodology,p_sparse, MatrixInfo,
                 ErrorScores,MR0, Svar)

res = bind_rows(DataPCA, DataSPCA, DatasPCArSVD, DataGPower_l1, DataGPower_l0)
res = res%>% select(-p_sparse...5)
names(res)[names(res) == 'p_sparse...2'] <- 'p_sparse'

# Results in graphs
```

```
VAF = as.factor(res$VAFx)
sparse = as.factor(res$p_sparse)
components = as.factor(res$n_components)
size = as.factor(res$s_size)
variables = as.factor(res$n_variables)

res$Methodology <- fct_relevel(res$Methodology, c("PCA", "SPCA", "sPCA_rSVD",
                               "GPower_l1", "GPower_l0"))

p <- ggplot(res, aes(Methodology, ErrorScores))
p + geom_boxplot() +
  xlab("methodes") +
  ylab("Erreur relative au carre")

p1 <- ggplot(res, aes(Methodology, ErrorScores)) +
  geom_boxplot(aes(fill=variables)) +
  xlab("Methodes") +
  ylab("Erreur relative au carre")
p1
p2 <- ggplot(res, aes(Methodology, ErrorScores)) +
  geom_boxplot(aes(fill=VAF)) +
  xlab("Methodes") +
  ylab("Erreur relative au carre")
p2
p3 <- ggplot(res, aes(Methodology, ErrorScores)) +
  geom_boxplot(aes(fill=sparse)) +
  xlab("Methodes") +
  ylab("Erreur relative au carre")
p3
p4 <- ggplot(res, aes(Methodology, ErrorScores)) +
  geom_boxplot(aes(fill=components)) +
  xlab("Methodes") +
  ylab("Erreur relative au carre")
p4

p <- ggplot(res, aes(Methodology, Svar)) +
geom_boxplot() +
  xlab("Methodes") +
  ylab("Pourcentage de variance expliquee")
p
p1 <- ggplot(res, aes(Methodology, Svar)) +
geom_boxplot(aes(fill=variables)) +
  xlab("Methodes") +
  ylab("Pourcentage de variance expliquee")
p1
p2 <- ggplot(res, aes(Methodology, Svar)) +
geom_boxplot(aes(fill=VAF)) +
  xlab("Methodes") +
  ylab("Pourcentage de variance expliquee")
```

```
p2
p3 <- ggplot(res, aes(Methodology, Svar)) +
geom_boxplot(aes(fill=sparse)) +
  xlab("Methodes") +
  ylab("Pourcentage de variance expliquee")
p3
p4 <- ggplot(res, aes(Methodology, Svar)) +
geom_boxplot(aes(fill=components)) +
  xlab("Methodes") +
  ylab("Pourcentage de variance expliquee")
p4

res_noPCA = subset(res, res$Methodology != "PCA")
VAF = as.factor(res_noPCA$VAFx)
sparse = as.factor(res_noPCA$p_sparse)
components = as.factor(res_noPCA$n_components)
size = as.factor(res_noPCA$s_size)
variables = as.factor(res_noPCA$n_variables)

p <- ggplot(res_noPCA, aes(Methodology, MR0)) +
  geom_boxplot() +
  xlab("Methodes") +
  ylab("Taux de mauvaises identifications")
p
p1 <- ggplot(res_noPCA, aes(Methodology, MR0)) +
geom_boxplot(aes(fill=variables)) +
  xlab("Methodes") +
  ylab("Taux de mauvaises identifications")
p1
p2 <- ggplot(res_noPCA, aes(Methodology, MR0)) +
geom_boxplot(aes(fill=VAF)) +
  xlab("Methodes") +
  ylab("Taux de mauvaises identifications")
p2
p3 <- ggplot(res_noPCA, aes(Methodology, MR0)) +
geom_boxplot(aes(fill=sparse)) +
  xlab("Methodes") +
  ylab("Taux de mauvaises identifications")
p3
p4 <- ggplot(res_noPCA, aes(Methodology, MR0)) +
geom_boxplot(aes(fill=components)) +
  xlab("Methodes") +
  ylab("Taux de mauvaises identifications")
p4
```

I.1 relativeError

Code source de la fonction `relativeError` utilisée dans la production des résultats de la simulation (Guerra-Urzola et al., 2021).

```
RelativeError = function(A,B){
  #  This function calculate the relative error between the columns of the
  #  matrix A and B.
  #  Rosember Guerra
  #  12-09-2019
  ##  Inputs
  #  A is the population matrix
  #  B is the matrix to be compared
  ##  Output
  #  er relative error
  #  p is the permutation that gives the max tc
  #  s is the sign of the

  # Number of components
  k=dim(A)[2]

  # Sign combination matrix
  Sign = c(1,-1)
  if (k==2){
    Sign_matrix = expand.grid(Sign,Sign)
  }else{
    Sign_matrix = expand.grid(Sign,Sign,Sign)
  }

  # Columns permutations
  Perms = combinat::permn(1:k)
  NPerms = length(Perms)
  Error_sign = matrix(rep(0,2*NPerms),ncol = 2)

  for (i in 1:NPerms) {
    PB = B[,Perms[[i]]]
    EP = rep(0,dim(Sign_matrix)[1])
    for (j in 1:dim(Sign_matrix)[1]) {
      PBS = PB%*%diag(Sign_matrix[j,])
      EP[j] = (norm((A-PBS),type = 'F')/norm(A,type = "F"))^2
    }
    Error_sign[i,1] = min(EP)
    Error_sign[i,2] = which(EP == Error_sign[i,1])[1]
  }
  R_error = min(Error_sign[,1])
  R_error_indx = which(Error_sign[,1] == R_error)[1]
  p = Perms[[R_error_indx]]
  s = Sign_matrix[Error_sign[R_error_indx,2],]
  return(list(Error = R_error, permutation = p, s= s))
}
```

I.2 correlation

Code source de la fonction `correlation` utilisée dans la production des résultats de la simulation (Guerra-Urzola et al., 2021).

```
correlation = function(A,B){

  k = dim(A)[2]
  Numerator = abs(colSums(A*B));
  Denominator = sqrt(colSums(A*A)*colSums(B*B))
  rv = sum(Numerator/Denominator)/k;
}
```

J Code application

Code original appliquant les différentes méthodes de parcimonie sur des données réelles du chapitre 4. Les packages utilisés sont les mêmes que pour le code de la simulation à l'annexe I.

```
# Application

data("pitprops")

pca <- prcomp(pitprops, center = TRUE, scale. = TRUE)
pca$rotation
pca$x
pca$sdev

SPCA <- spca(x=pitprops, K = 2, sparse = "varnum", para =rep(3,2),
             type = "predictor", use.corr = TRUE)
SPCA$loadings

X = sqrtm(pitprops)
sPCA_rSVD <- ssvd(x=X, k=2, n=3, center = TRUE, scale. = TRUE)

GPower_l1 = gpower(data=pitprops, k=2, rho=auto_gpower(pitprops, 2,
             prop_sparse = 0.77, reg = "l1", center = TRUE),
             reg="l1", center=TRUE)
GPower_l1$weights

GPower_l0 = gpower(data=pitprops, k=2, rho=auto_gpower(pitprops, 2,
             prop_sparse = 0.7, reg = "l0", center = TRUE),
             reg="l0", center=TRUE)
GPower_l0$weights

country <- read.csv("Country-data.csv")
str(country)
colSums(is.na(country))
data <- country[2:10]
head(data)
data_centered <- scale(data, center = TRUE, scale = TRUE)
head(data_centered)
cor_matrix <- cor(data_centered)
ggcorrplot(cor_matrix)
pca <- prcomp(cor_matrix)
summary(pca)
pca$loadings[,1:3]
fviz_eig(pca, addlabels = TRUE)
fviz_pca_biplot(pca)
fviz_cos2(pca, choice = "var", axes = 1:2)

X = data_centered
```

```
X = X[,-3]
X

vec_sPCA_rSVD_sparse <- c(7, 6, 5, 4, 3, 2, 1)
vec_GPower_sparse <- c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9)
K_CP <- c(1, 2, 3, 4, 5)

for (i in K_CP) {

  vec_var_sPCA_rSVD <- c()
  vec_var_GPower <- c()

  for (j in vec_sPCA_rSVD_sparse) {
    sPCA_rSVD <- ssvd(x=X, k=K_CP[i], n=vec_sPCA_rSVD_sparse[j],
                      center = TRUE, scale. = TRUE)
    CP <- sPCA_rSVD$u %*% sPCA_rSVD$d
    X_rec <- CP %*% t(sPCA_rSVD$v)
    vec_var_sPCA_rSVD[j] <-
            1 - (norm((X-X_rec),type = "F")/norm(X,type = "F"))^2
  }

  for (l in 1:length(vec_GPower_sparse)) {
    GPower_l1 = gpower(data=X, k=K_CP[i], rho=auto_gpower(X, K_CP[i],
                prop_sparse = vec_GPower_sparse[l], reg = "l1", center = TRUE),
                reg="l1", center=TRUE)
    CP <- GPower_l1$scores
    X_rec <- CP %*% t(GPower_l1$weights)
    vec_var_GPower[l] <-
            1 - (norm((X-X_rec),type = "F")/norm(X,type = "F"))^2
  }

  assign(paste0("vecSRvar_", K_CP[i]), vec_var_sPCA_rSVD)
  assign(paste0("vecGPvar_", K_CP[i]), vec_var_GPower)
}

#vec_GPower_sparse  = factor(vec_GPower_sparse,
    #levels=c("0.1", "0.2", "0.3", "0.4", "0.5", "0.6", "0.7", "0.8", "0.9"))
#vec_sPCA_rSVD_sparse  = factor(vec_sPCA_rSVD_sparse,
    #levels=c("7", "6", "5", "4", "3", "2", "1"))

plot(vec_sPCA_rSVD_sparse, vecSRvar_1, type="o",
    xlab = "Nombre d'elements non-egaux a 0", ylab = "PEV",
    pch = 16, ylim = c(0,1))
plot(vec_sPCA_rSVD_sparse, vecSRvar_2, type="o",
    xlab = "Nombre d'elements non-egaux a 0", ylab = "PEV",
    pch = 16, ylim = c(0,1))
plot(vec_sPCA_rSVD_sparse, vecSRvar_3, type="o",
    xlab = "Nombre d'elements non-egaux a 0", ylab = "PEV",
    pch = 16, ylim = c(0,1))
```

```
plot(vec_sPCA_rSVD_sparse, vecSRvar_4, type="o",
    xlab = "Nombre d'elements non-egaux a 0", ylab = "PEV",
    pch = 16, ylim = c(0,1))
plot(vec_sPCA_rSVD_sparse, vecSRvar_5, type="o",
    xlab = "Nombre d'elements non-egaux a 0", ylab = "PEV",
    pch = 16, ylim = c(0,1))
vecSRvar_4

plot(vec_GPower_sparse, vecGPvar_1, type="o",
    xlab = "Proportion de parcimonie", ylab = "PEV", pch = 16,
    ylim = c(0,1), xlim = c(0.9,0))
plot(vec_GPower_sparse, vecGPvar_2, type="o",
    xlab = "Proportion de parcimonie", ylab = "PEV", pch = 16,
    ylim = c(0,1), xlim = c(0.9,0))
plot(vec_GPower_sparse, vecGPvar_3, type="o",
    xlab = "Proportion de parcimonie", ylab = "PEV", pch = 16,
    ylim = c(0,1), xlim = c(0.9,0))
plot(vec_GPower_sparse, vecGPvar_4, type="o",
    xlab = "Proportion de parcimonie", ylab = "PEV", pch = 16,
    ylim = c(0,1), xlim = c(0.9,0))
plot(vec_GPower_sparse, vecGPvar_5, type="o",
    xlab = "Proportion de parcimonie", ylab = "PEV", pch = 16,
    ylim = c(0,1), xlim = c(0.9,0))
vecGPvar_4

sPCA_rSVD <- ssvd(x=X, k=4, n=3, center = TRUE, scale. = TRUE)
sPCA_rSVD$v
GPower_l1 = gpower(data=X, k=4, rho=auto_gpower(X, 4, prop_sparse = 0.8,
            reg = "l1", center = TRUE), reg="l1", center=TRUE)
GPower_l1$weights
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté des sciences
Place des sciences, 2 bte L6.06.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/sc**