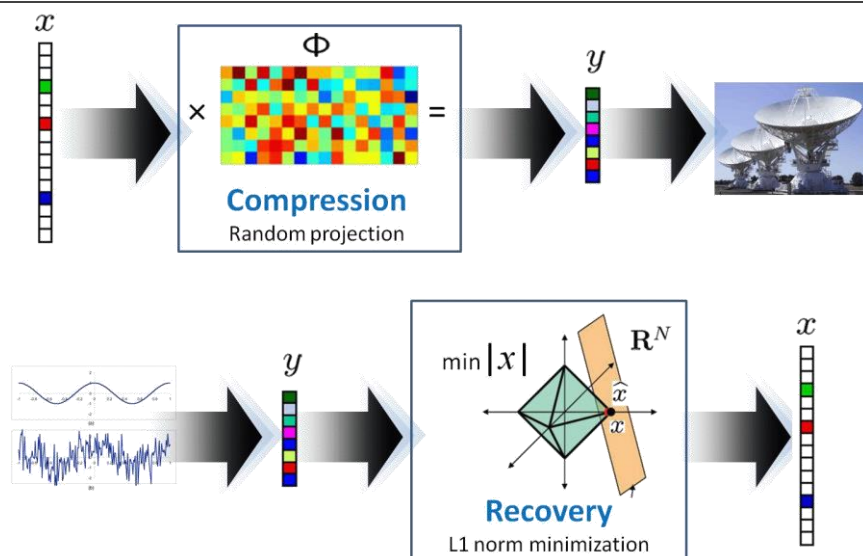


CY-TECH
2^{ème} année d'ingénieur



AUPHAN Gwendal – RICHAUDEAU Alexandre – ELYOUSFI Amal

Rapport de projet Compressive Sensing



2022 – 2023

Introduction

Dans le cadre du module Compressive Sensing (CS) étudié au deuxième semestre de la deuxième année du cycle ingénieur, nous avons été amenés à réaliser un projet.

Ce projet porte principalement sur la notion de codage parcimonieux, élément clé du Compressive Sensing. Il s'agit des techniques qui exploitent la parcimonie des signaux pour réduire le nombre de données nécessaires pour représenter le signal de manière précise.

L'élément clé qui nous permettra de passer d'une compression classique, inconveniente en termes de ressources et de temps, à une méthode de Compressive Sensing est la matrice de mesure. C'est une matrice aléatoire, qui suit certaines règles qui est utilisée pour transformer notre signal d'origine en une représentation parcimonieuse, de dimension bien inférieure à notre signal d'origine. C'est à partir de cette représentation parcimonieuse qu'on pourra reconstruire le signal d'origine.

C'est une méthode très importante vouée à être de plus en plus utilisée, face à l'explosion de la quantité des données d'une part et de ressources de stockages limités d'autre part. Ce processus est principalement utilisé lors de traitement de données couteuse en termes de stockage. Par exemple dans des cadres médicaux, tels que le stockage des IRM, qui nécessitent un espace de stockage important. Ce processus vient en continuité des processus déjà démocratisés aujourd'hui tels que jpeg, mp3...

Ce projet nous a permis d'acquérir de nouvelles connaissances, de découvrir de nouveaux algorithmes, et de comprendre en profondeur le fonctionnement mathématique du codage parcimonieux.

Table des matières

1.	Présentation globale du projet	4
2.	Processus complet	5
a)	Apprentissage du dictionnaire :	5
b)	Matrice de mesure :	6
c)	Représentation parcimonieuse :	7
i.	MP	7
ii.	OMP	7
iii.	StOMP :	8
iv.	CoSaMP:	8
v.	Relaxation l_{pet} optimisation convexe	9
3.	Processus mathématique.....	10
i.	Construction de α	10
ii.	Reconstruction du signal.....	10
4.	Applications / Expérimentations.....	11
	Cas du projet	11
	Préparation	11
	Algorithme	11
	Apprentissage du dictionnaire	13
	Détermination de la matrice de mesure.....	15
	Comparaison des algorithmes	16
	Résultats.....	16
5.	Conclusion.....	20

1. Présentation globale du projet

Notre projet s'articule en plusieurs parties. Le signal acquis est un signal déjà compressé, on a appliqué une matrice de mesure φ à notre signal d'origine qui nous a permis de faire directement l'acquisition comprimée.

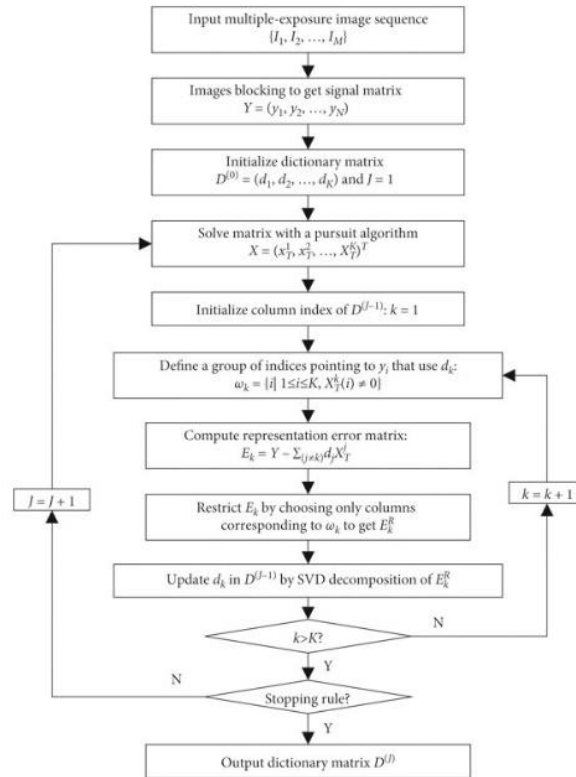
Il nous faudra dans un premier temps réaliser l'apprentissage du dictionnaire à l'aide de la méthode K-SVD (K-Singular Value Decomposition). Ce dictionnaire sera ensuite utilisé pour l'ensemble des signaux de mêmes types. Il faut faire attention à ne pas faire du sur-apprentissage, cas dans lequel notre dictionnaire ne serait adapté qu'à nos signaux d'entraînements.

Nous avons vu plusieurs méthodes de codage parcimonieux tels que MP, OMP, StOMP, CoSaMP et IRLS. Ces méthodes sont utilisées dans l'apprentissage du dictionnaire et dans la représentation parcimonieuse. La représentation parcimonieuse nous permet de déterminer α , vecteur s — *parcimonieux*. Un vecteur s — *parcimonieux* comporte s valeurs non nulles. C'est ce vecteur qu'on va transmettre et qui nous servira à reconstruire le signal d'origine, en le multipliant à notre dictionnaire. Le but principal de ce projet est donc de reconstruire le signal d'origine à l'aide du vecteur α .

2. Processus complet

a) Apprentissage du dictionnaire :

L'apprentissage du dictionnaire à partir des données est une étape très importante et plutôt complexe en Compressive Sensing. Le dictionnaire généré dépend alors des données utilisées. La méthode ici mise en pratique pour l'apprentissage est la méthode KSVD. C'est la méthode de base la plus répandue. Voici le raisonnement logique de l'algorithme.



Il existe aussi d'autres dictionnaires, tels que celui de Fourier et de Wavelet. Ce sont des structures généralisées, qui marchent généralement bien mais qui ne sont pas adaptés à nos données. Le fait que la structure de ces dictionnaires est préétablie peut-être limitante.

b) Matrice de mesure :

Utiliser une matrice de mesure constitue la toute première étape de notre processus. Elle est appliquée directement au signal brut X , de manière à stocker uniquement $Y = \varphi X$. Cela permet de réduire considérablement la taille des données à stocker. D'où le terme « acquisition comprimée ». C'est donc à partir de Y qu'on détermine la représentation parcimonieuse. Étant donné que le volume de données de Y est largement inférieur, cette opération s'avère bien moins onéreuse en termes de coût de calcul que si elle avait été appliquée à X . C'est le dispositif d'observation (celui qui récolte les données) qui effectue cette étape, elle est donc normalement transparente dans le processus. Nous nous demandons alors quelle matrice de mesure utiliser.

Pour la choisir, nous générons une matrice de manière aléatoire répondant à certains critères. Dans notre cas, nous allons comparer les 5 matrices usuelles suivantes, connues pour répondre aux conditions nécessaires (vu en cours). Ensuite, il suffit de calculer la cohérence mutuelle entre le dictionnaire et la matrice de mesure. On choisira la matrice de mesure dont la cohérence avec le dictionnaire est la plus faible.

Voici les 5 matrices mentionnées :

- Φ_1 : matrice aléatoire obtenue par une loi uniforme $\mathcal{U}(0, 1)$
- Φ_2 : matrice aléatoire obtenue par une loi de Bernoulli avec valeurs -1 ou 1 .
- Φ_3 : matrice aléatoire obtenue par une loi de Bernoulli avec valeurs 0 ou 1 .
- Φ_4 : matrice aléatoire obtenue par une loi normale $\mathcal{N}\left(0, \frac{1}{M}\right)$
- Φ_5 : matrice aléatoire et creuse générée par Scilab ou certaines librairies de Python.

c) Représentation parcimonieuse :

i. MP

La méthode MP (Matching Pursuit) est la méthode la plus basique pour minimiser α sous la contrainte $\|x - D\alpha\|_2 \leq \varepsilon$. C'est un algorithme de poursuite gloutonne.

```

 $\alpha \leftarrow 0$ 
 $R \leftarrow x$ 
 $k \leftarrow 0$ 
Tant que  $\|R\| > \varepsilon$  et  $k < N$ 
   $m \leftarrow \underset{1 \leq j \leq n}{\operatorname{argmax}} \frac{|d_j^T R|}{\|d_j\|_2}$ 
   $z \leftarrow \frac{d_m^T R}{\|d_m\|_2^2}$ 
   $\alpha_m \leftarrow \alpha_m + z$ 
   $R \leftarrow R - z d_m$ 
   $k \leftarrow k + 1$ 
Fin Tant que

```

ii. OMP

Un grand défaut de l'algorithme MP est le fait qu'un atome peut être sélectionné plusieurs fois. C'est donc redondant et coûteux en calcul, on intègre alors une nouvelle condition, qui permet de sélectionner uniquement des atomes non déjà étudiés.

```

 $\alpha \leftarrow 0$   $R \leftarrow x$   $Index \leftarrow []$   $k \leftarrow 0$   $A \leftarrow []$ 
Tant que  $\|R\| > \varepsilon$  et  $k < N$ 
   $m \leftarrow \underset{1 \leq j \leq n}{\operatorname{argmax}} \frac{d_j^T R}{\|d_j\|_2}$ 
   $Index \leftarrow [Index, m]$ 
   $A \leftarrow [A, d_m]$ 
   $\alpha_{Index} \leftarrow \operatorname{pinv}(A)x$ 
   $R \leftarrow R - A\alpha_{Index}$ 
   $k \leftarrow k + 1$ 
Fin Tant que

```

iii. StOMP :

La méthode StOMP (Stagewise Orthogonal Matching Pursuit) est une extension de la méthode OMP (Orthogonal Matching Pursuit) qui permet d'améliorer la performance de ce dernier. Le principe est d'établir un seuil k , à partir duquel nous ne nous sélectionnerons que les atomes dont l'implication de ces derniers dépasse ce seuil.

(a) On calcule le seuillage $S^{(k)}$: $S^{(k)} = t \frac{\|R^{(k-1)}\|_2}{\sqrt{K}}$, $2 \leq t \leq 3$.

(b) On sélectionne l'ensemble Λ_k des indices des atomes dont la contribution est supérieure au seuillage $S^{(k)}$

$$\Lambda_k = \left\{ j \in \{1, \dots, K\}, C_j > S^{(k)} \right\}.$$

On applique ensuite le même procédé pour mettre à jour le résiduel jusqu'à la convergence de l'algorithme.

iv. CoSaMP:

L'algorithme CoSaMP (Compressive Sampling Matching Pursuit) est un algorithme itératif comme ses petits frères. La différence se fait encore une fois dans la méthode de sélection des atomes. Dans cet algorithme, nous allons garder à chaque fois les $2s$ atomes les plus importants et les enregistrer au support. Ensuite, on ne gardera que ceux qui résistent au rejet.

2. Mise à jour du support: Fusionner le support de la sélection **supp1** avec le support de la solution de l'itération précédente **supp** :

$$\text{supp} = \text{supp} \cup \text{supp1}$$

On note $\mathbf{AS} = \mathbf{D}(:, \text{supp})$ la matrice des atomes actifs sélectionnés.

4. Rejet: Considérer les s plus grands coefficients de \mathbf{z} .

v. Relaxation l_p et optimisation convexe

La résolution du problème d'optimisation l_0 au sens de la pseudo-norme l_0 est un problème de complexité NP-difficile dans lequel nous cherchons à trouver un vecteur x de dimension n qui a le plus petit nombre de composantes non nulles c'est-à-dire une solution parcimonieuse, sous une contrainte linéaire $x = D\alpha$.

Cependant, la résolution du problème d'optimisation l_0 est difficile car il est non convexe et non linéaire. Donc pour résoudre ce problème, plusieurs méthodes sont proposées, mais celle qui nous intéresse aujourd'hui est la méthode de relaxation convexe. C'est une méthode d'approximation qui remplace la pseudo-norme l_0 par la pseudo-norme l_p , tel que $0 < p < 1$, qui elle est convexe et peut être résolue efficacement à l'aide d'un algorithme d'optimisation convexe.

Ensuite on essaye de transformer le problème P_p en problème de minimisation de moindres carrés avec : $(P_p): \min \|\alpha\|_p$, tel que $x = D\alpha$. L'idée de base est de minimiser la somme des carrés des résidus entre x et $D\alpha$. C'est-à-dire de trouver α qui minimise : $\|x - D\alpha\|^2$. Ceci est équivalent à minimiser la norme euclidienne de la différence entre $D\alpha$ et x . ainsi nous pouvons reformuler le problème P_p comme suit : $(P_p): \min \|x - D\alpha\|_p^2$, tel que α est un vecteur de coefficients.

3. Processus mathématique

i. Construction de α

Le signal qu'on va stocker est $Y = \varphi X$. (X signal initial).

X est de dimension $(1; N)$, Y est de dimension $(1; M)$ et φ est donc de dimension $(M; N)$. Avec $M \ll N$

K-SVD

On sait que $X = D\alpha$

Une fois D déterminé à l'aide de K-SVD, on applique :

$Y = \varphi X = \varphi D\alpha = A\alpha$. C'est ici qu'on va pouvoir déterminer α à l'aide des algorithmes type MP, OMP, IRLS...

D est de dimension $(N; K)$, A est de dimension (M, K) et α est de dimension $(1; K)$

ii. Reconstruction du signal

Le récepteur reçoit α et lui applique D pour reconstruire le signal. On a alors $\hat{X} = D\alpha$

4. Applications / Expérimentations

Cas du projet

Le projet consiste à effectuer le processus du Compressive Sensing avec les données de fuites de gaz. Nous devons dans l'ordre, implémenter l'algorithme k-svd afin de générer un dictionnaire approprié à nos données. Ensuite, pour chaque pourcentage de P , nous calculerons les cohérences mutuelles de plusieurs matrices de mesure pour ne garder que ϕ_{opt} .

Dans la suite, il ne suffit plus que d'appliquer ϕ_{opt} à au signal brut et d'en tirer sa représentation parcimonieuse. Nous pourrions à la fin reconstruire le signal et comparer les performances des algorithmes ainsi que l'influence du tuning des paramètres sur les résultats.

Préparation

i. Dataset

Le Dataset est constitué de signaux permettant l'entraînement de notre dictionnaire. Ainsi, notre base de données est constituée de 150 signaux de dimension 98, ce qui permettra la création d'un dictionnaire de taille (98, 150). Les données se trouvent dans le fichier « data.xlsx ».

ii. Signaux de test

Afin de connaître l'efficacité du processus, nous devons tester nos algorithmes sur une base de test. Pour cela, nous disposons de 13 autres signaux de test afin valider le processus de compressive sensing. Celui-ci nous a été fournis en format Excel également dans le fichier « data.xlsx ».

Algorithme

Pour exécuter l'algorithme, il faut se rendre, dans le dossier et lancer le fichier main.py. Si un dictionnaire (dico.xlsx) figure déjà dans le dossier, il sera utilisé pour l'algorithme. Si aucun fichier ne figure, le dictionnaire sera calculé.

Pour modifier les paramètres du code, il faut se rendre dans le fichier set_parameters.py du dossier Init_config. On y retrouve les fonctions des 3 captures d'écrans suivantes (def config_run, def config_phi et def config_dico)

```
def config_dico():
    x_train= X_train,          # Dataset d'entrainement
    num_atoms_dico=100,        # Nombre d'atomes du dictionnaire
    func_dico="IRLS",          # Fonction utilisée pour calculer alpha
    maxiter_dico=10,           # Nombre d'itération de l'algo K-svd
    iteration_algo_dico=25,     # Nombre d'itération de l'algo utilisé
    max_alpha_dico = None,      # Nombre maximum d'alpha = None (par défaut)
    arret_stop_algo_dico=0.000001, # Critère d'arrêt de l'algo (Convergence)
    t_st_omp_dico=2.4,          # Paramètre "t" utilisé pour St_omp
    epsilon_irls_dico = 0.1,    # Paramètre "epsilon" pour IRLS
    p_irls_dico = 0.5,          # Paramètre "p" pour IRLS
    approx=False,              # Approximation du K-svd (Si Alpha_nonzero = 0)
    verbose_dico = False,       # Mode Verbose (Affiche d)
    method_dico = "alea",       # Méthode d'initialisation du dictionnaire
    initial_D=None              # Check d'un dictionnaire déjà existant
```

Configuration des paramètres pour la construction du dictionnaire

Configuration des paramètres pour l'implémentation de Phi.
Nous indiquons :

- Le dictionnaire en cours
- low_phi pour les lois aléatoires
- high_phi pour les lois aléatoires
- threshold_phi pour les lois aléatoires

```
def config_phi(dico,liste_P):
    dico_phi = dico,
    low_phi = 0
    high_phi = 1
    p_phi=0.5
    threshold_phi = 0.75
    liste_P_phi = liste_P
```

```
def config_run(dico,liste_phi,liste_P):
    X_origin = X_test,
    dico_run = dico,
    liste_phi_P = liste_phi,
    num_atoms_run = 100,
    liste_P_run = liste_P,
    func_run="MP",
    affichage = True,
    nb_graphs=4,
    iteration_algo_run=30,
    max_alpha_run = None,
    arret_stop_algo_run=0.0000001,
    t_st_omp_run=2.5,
    epsilon_irls_run = 0.1,
    p_irls_run = 0.5,
    verbose_run = False
```

Configuration des paramètres pour la représentation parcimonieuse, puis la reconstruction avec l'affiche des résultats et des calculs d'erreurs

(Les paramètres sont très similaire à la configuration dico car ils utilisent les mêmes algorithmes)

Apprentissage du dictionnaire

i. Initialisation

L'initialisation du dictionnaire peut se faire de plusieurs manières, en théorie nous devrions tirer au hasard des vecteurs dans le dataset avec $K < P$. Si $K = P$, on peut prendre comme initialisation le dataset en entier.

K-SVD utilise un algorithme de matching pursuit, nous avons le choix entre « MP, OMP, StOMP, CaSaOMP, IRLS ». La qualité du dictionnaire pourra ainsi dépendre du choix d'algorithme. De plus, le nombre d'itération et le critère de convergence sont des paramètres déterminants de l'efficacité de K-SVD.

Paramètres de la fonction `create_dico` :

```
def create_dico(x_train,          # Dataset d'entrainement
               num_atoms,        # Nombre d'atomes du dictionnaire
               maxiter=15,       # Nombre d'itération de l'algo K-svd
               func="OMP",       # Fonction utilisée pour calculer alpha
               iteration_algo=25, # Nombre d'itération de l'algo utilisé
               max_alpha = None, # Nombre maximum d'alpha = None (par défaut)
               arret_stop_algo=0.00001, # Critère d'arrêt de l'algo (Convergence)
               t_st_omp=3,       # Paramètre "t" utilisé pour St_omp
               epsilon_irls = 0.1, # Paramètre "epsilon" pour IRLS
               p_irls = 0.5,     # Paramètre "p" pour IRLS
               approx=False,     # Approximation du K-svd (Si Alpha_nonzero = 0)
               verbose = False,  # Mode Verbose (Affiche d)
               method_dico = "no_alea", # Méthode d'initialisation du dictionnaire
               initial_D=None):   # Check d'un dictionnaire déjà existant
```

Nous avons réalisé un code très modulable, et nous pouvons agir directement dans les paramètres afin de choisir l'algorithme pour calcul alpha. Nous pouvons mettre par exemple, dans le paramètre **func** « MP », « OMP », « STOMP », « COSAMP » et « IRLS ». Dans le paramètre **method_dico**, nous pouvons aussi mettre « alea1 », « alea2 » et « alea3 ». Le paramètre **Verbose** permet d'afficher plus ou moins de détails quand les algorithmes tournent, il prend en entrée « True » ou « False »

Apprentissage du dictionnaire avec la méthode OMP avec la configuration ci-dessus.

On observe que la méthode a convergé assez vite mais stagne autour des 24000 d'erreur de norme. `erreur update = norm(Dataset – Dico. α)`

```
erreur update 115900.76412827484
erreur update 23727.190758986788
erreur update 26124.847426793734
erreur update 27807.73727789411
erreur update 27441.48301463596
erreur update 25204.32061189373
erreur update 28961.27775781012
erreur update 29804.359651607476
erreur update 26758.328465886454
erreur update 25887.953743499358
erreur update 25264.098714683474
erreur update 24533.180892483208
erreur update 21515.552201770628
erreur update 26254.031480681377
erreur update 24004.100884935284
```

Comparaison des algorithmes dans la construction du dictionnaire

	MP	OMP	STOMP	COSAMP	IRLS
Erreur de reconstruction durant l'apprentissage du dictionnaire	195826.703184485	260988.21411391534	2480236.502578039	170152.1806520829	9731394.094677078
	214740.80509574054	173248.88158007516	1019704.0671029888	44774.35943432193	336249.74358087394
	215616.84565646545	213533.13076667843	1018741.885446382	38569.06787283571	0.01107422103569934
	245944.97295800238	177718.62586284525	58831.310274616	37562.92581212679	0.016534220750643785
	209462.16320026293	186772.84992233792	156614.53582630103	35812.502061249055	0.016585414550694427
	118174.34374703269	242099.67716548927	45012.85222717663	35209.307364336135	0.021313236788076048
	231423.21797266544	238112.94914131818	13.93273213142779	38164.9609546413	0.043719830899473455
	126313.23730789233	210693.9402093232	11.955807253394083	36070.81827247119	0.013163785223493159
	256084.71106045457	178174.03427965855	10.259401153514878	35158.84518769757	0.010684579345431405
	129690.87120998668	213324.47519806714	8.803709961475322	33645.24578477189	0.009185028620199789

Détermination de la matrice de mesure

Tableau : Cohérence mutuelle

	15	20	25	30	50	75
phi1	8.862144	8.873059	8.923493	8.838995	8.855659	8.957984
phi2	3.814656	3.615533	3.328483	3.516988	3.498748	3.856367
phi3	7.474046	7.701876	7.505432	7.838211	7.662179	7.814215
phi4	3.251384	3.426923	4.097772	4.04303	3.747409	3.91645
phi5	7.976836	8.050896	8.029222	7.968912	8.183703	8.043426

Cette cohérence mutuelle est toujours encadrée par 1 et \sqrt{N} , soit ici entre 1 et $\sim 12,25$. On sélectionnera alors la matrice de mesure dont la cohérence de mesure est la plus faible. Nous remarquons que phi2 et phi4 sont toujours les matrices de cohérences les plus adaptées. Il faut cependant noter que la cohérence mutuelle entre le dictionnaire D et la matrice de mesure phi varie à chaque fois qu'une nouvelle matrice de mesure est implémenté.

Comparaison des algorithmes

Tableau : Erreur moyenne sur les 13 signaux de test (Apprentissage avec IRLS)

	15	20	25	30	50	75
MP	21923.261816	28963.009164	32769.44556	14958.697581	27458.534231	19751.549699
OMP	11003.989884	10982.074957	11250.029831	11706.367556	8700.299979	6644.354526
STOMP	8093.275449	8404.433116	8357.05861	8329.479878	8817.558937	18520.55607
COSAMP	11361.961632	14757.794782	12283.08427	13024.596936	11745.892454	8295.976196
IRLS	7655.451611	7945.828155	7600.681322	7989.590816	7483.747475	6309.072355

Tableau : Erreur moyenne sur les 13 signaux de test (Apprentissage avec OMP)

	15	20	25	30	50	75
MP	21207.744476	19688.284511	25397.912049	28785.775091	23319.431795	21778.837864
OMP	10387.244176	11358.192641	11975.782845	11295.427068	9685.93324	6855.189507
STOMP	8576.91007	8870.093543	8605.553069	10091.125072	18451.749814	8127.348861
COSAMP	10367.382069	11789.742813	11843.746921	11827.081256	11884.51352	7767.822984
IRLS	8739.719129	8525.288316	8549.861634	8698.882958	7960.762084	7126.901805

Résultats

La partie résultats nous permet de voir la bonne réalisation du processus de compressive sensing.

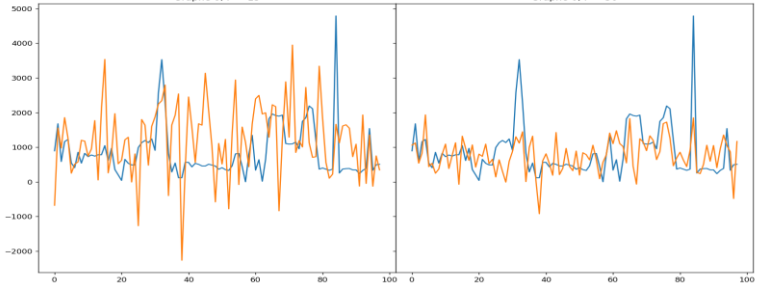
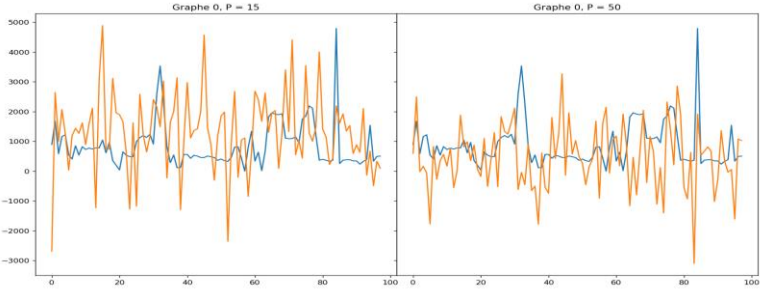
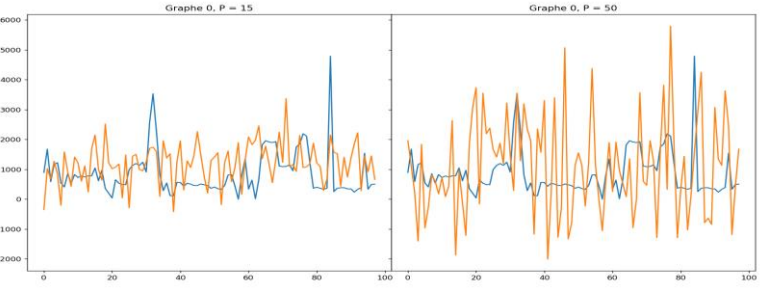
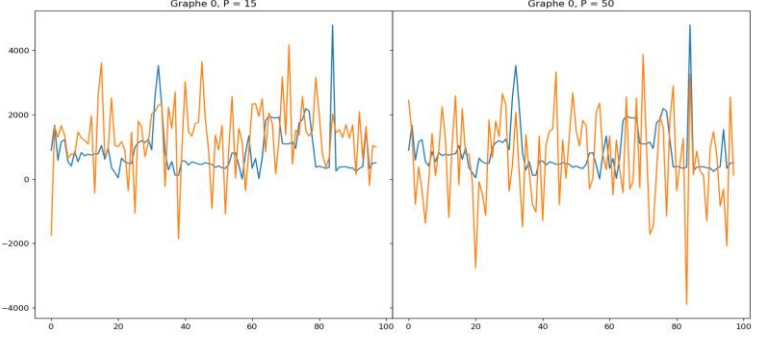
Pourcentage de parcimonie de alpha pour les vecteurs d'entraînement après l'entraînement du dictionnaire. Pour chaque atome :

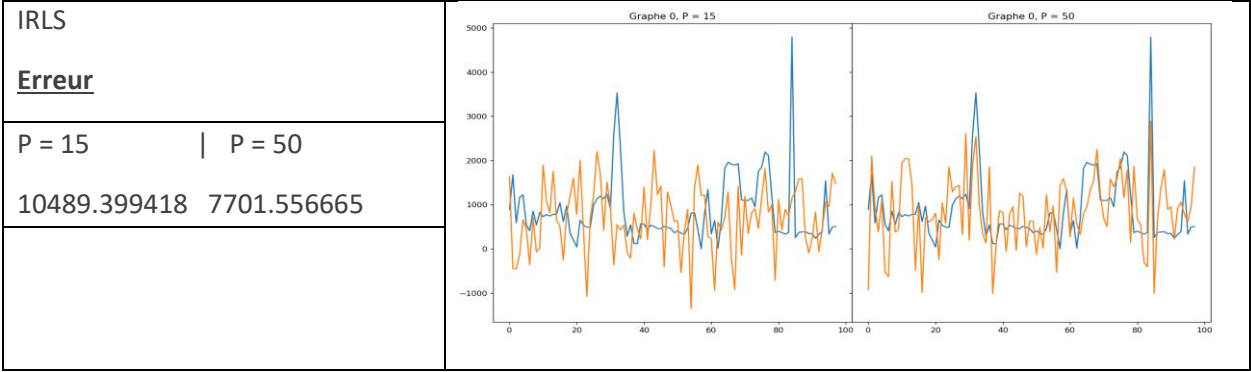
α_1	α_2	α_3										α_n
0,744898	0,153061	0,428571	0,765306	0,214286	0,744898	0,306122	0,663265	0,540816	0,459184	0,744898	0,153061	0,571429	0,

On peut observer que l'algorithme a réussi à trouver une meilleure représentation parcimonieuse pour certains vecteurs d'entraînement plutôt que d'autres. Cela peut peut-être venir de l'initialisation du dictionnaire.

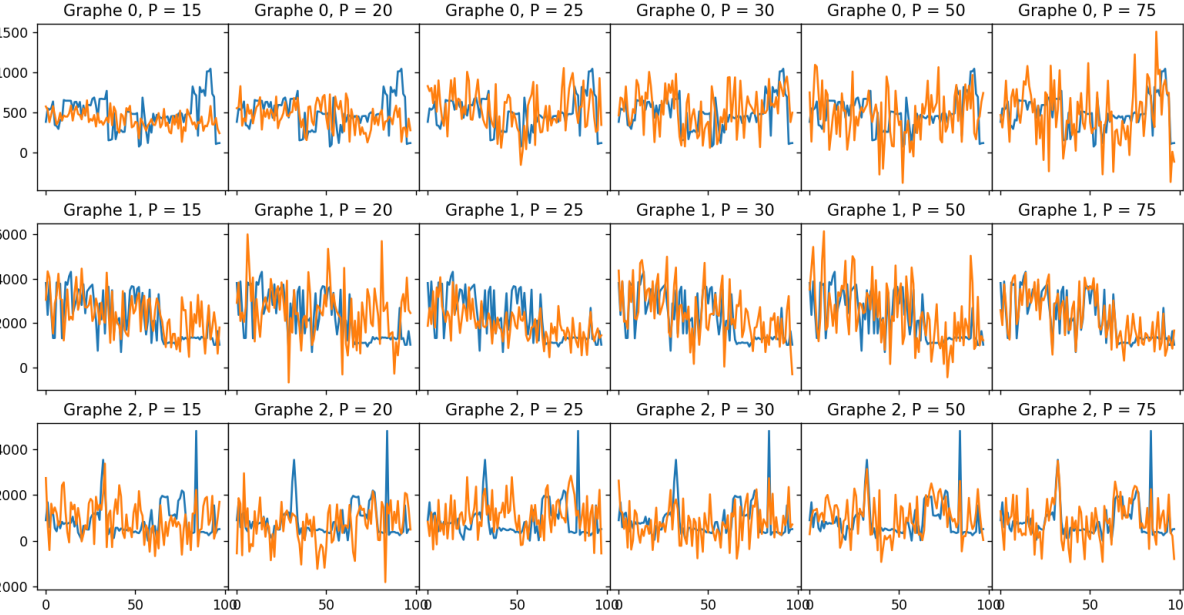
Représentation parcimonieuse pour deux vecteurs de test avec P =15 ; 25 ; 60 ;75 (en ligne)

% de parcimonie	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
85%	0	0	0	1509,651	0	-27715,5	0	0	0	0	0	0	0	0	0
85%	0	0	0	-1586,14	0	0	0	0	0	0	0	0	0	0	0
75%	0	0	0	990,2955	0	-3836,47	0	0	-7092,51	0	0	0	0	0	0
75%	0	0	0	756,9135	0	320,9388	0	0	2249,693	0	0	0	0	0	0
70%	0	0	0	1000,643	0	0	0	0	-1002,62	5492,684	0	11450,07	0	0	0
70%	0	0	0	458,9752	-2819,04	-1996,38	0	0	515,5165	0	0	0	0	0	0
70%	0	5156,902	-12701,2	654,6486	0	2150,95	0	-8319,89	194,7538	0	0	0	0	-7709,43	0
70%	0	0	0	-558,871	0	562,017	0	0	2007,601	3562,512	0	0	0	-1354,79	0

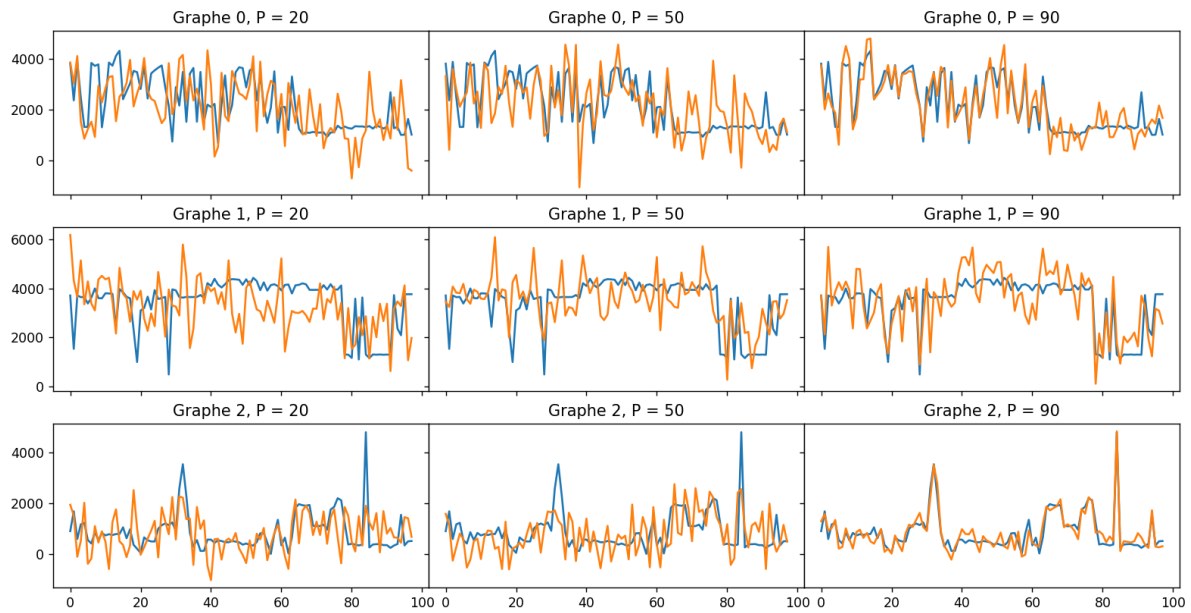
<div>MP</div> <div>Erreur</div> <div> <div>P = 15 P = 50</div> <div>11742.3873 7939.728918</div> </div>	<div> <div>Graphe 0, P = 15</div> <div>Graphe 0, P = 50</div> </div> 
<div>OMP</div> <div>Erreur</div> <div> <div>P = 15 P = 50</div> <div>13783.7511 8997.606599</div> </div>	<div> <div>Graphe 0, P = 15</div> <div>Graphe 0, P = 50</div> </div> 
<div>STOMP</div> <div>Erreur</div> <div> <div>P = 15 P = 50</div> <div>9928.149395 15300.758086</div> </div>	<div> <div>Graphe 0, P = 15</div> <div>Graphe 0, P = 50</div> </div> 
<div>COSAMP</div> <div>Erreur</div> <div> <div>P = 15 P = 50</div> <div>13570.467432 11347.628525</div> </div>	<div> <div>Graphe 0, P = 15</div> <div>Graphe 0, P = 50</div> </div> 



Exemple de reconstruction de signal avec **OMP**



Exemple de reconstruction de signal avec IRLS



5. Conclusion

En conclusion au cours de ce projet, nous avons abordé le sujet du Compressive Sensing et des techniques utilisées pour résoudre des problèmes de reconstruction de signaux à partir d'échantillonnage incomplets ou bruités.

Nous avons commencé par aborder la notion de parcimonie et de sa relation avec le compressive sensing. Nous avons ensuite exploré les différentes étapes du processus de reconstruction en utilisant le CS, en commençant par la génération de matrices de mesure en passant par l'apprentissage du dictionnaire en utilisant l'algorithme K-SVD, ainsi que l'utilisation d'algorithmes de matching poursuit tel que MP, OMP et autres.

Nous avons également discuté de la relaxation l_p , et de la différence avec la pseudo-norme l_0 dans la résolution de problèmes de parcimonie. Ainsi que l'algorithme IRLS qu'on a utilisé dans le contexte de résolution de problèmes d'optimisation convexe afin d'obtenir des solutions parcimonieuses et stables.

De plus, nous avons pu comparer l'impact de la variable P (pourcentage de mesures) sur l'efficacité de la reconstruction des signaux, ainsi que l'utilisation des différents algorithmes et enfin l'utilisation de la meilleure matrice de mesure ϕ .

Nous savons que pas mal de paramètres peuvent être mieux ajustés ainsi que l'efficacité des algorithmes en eux-mêmes, ou bien l'utilisation d'extensions d'algorithmes comme CK-SVD ou DK-SVD etc... Des algorithmes plus performants qui convergent plus rapidement.

De plus, le choix de la matrice de mesure peut être un axe d'amélioration car nous utilisons des matrices de mesures qui ne sont pas personnalisés à la donnée d'entraînement. Il existe aujourd'hui plusieurs algorithmes permettant l'affinage de meilleures matrices de mesures offrant moins de perte d'informations pour un même pourcentage.