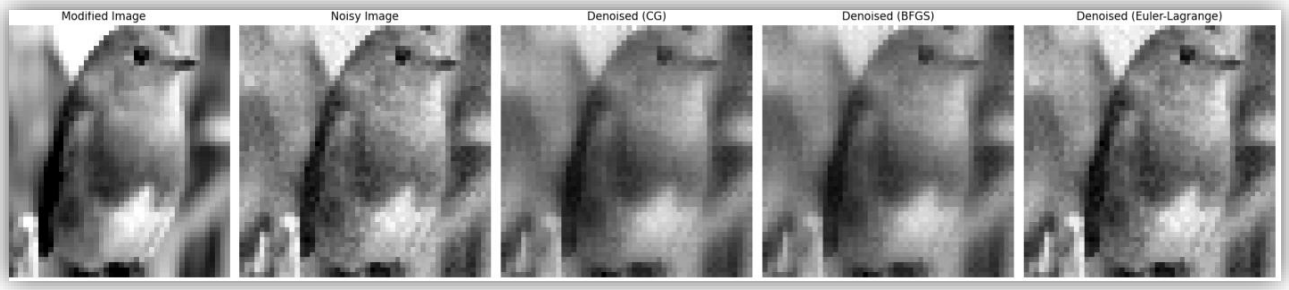
CURE-MOOG Ludovic
PORET Guillaume
DOS ANJOS DOS SANTOS Hugo

02/06/2024

Approches gradientielles et newtoniennes pour le traitement numérique des images



Professeur : M. Peschard

Table des matières

I. Introduction

Dans le domaine en constante évolution de l'optimisation numérique et du traitement d'image, l'efficacité des méthodes algorithmiques joue un grand rôle, notamment dans des applications industrielles et académiques où la précision et la rapidité sont primordiales. Ce rapport se concentre sur l'étude comparative de méthodes d'optimisation avancées, notamment les méthodes de gradient et Newtoniennes, et leur application dans le contexte spécifique du traitement d'images numériques. Notre objectif est d'évaluer et de comparer l'efficacité de ces techniques en résolvant des problèmes complexes de minimisation associés à la qualité et à la clarté des images numériques.

Nous abordons premièrement les méthodes de gradient à pas optimal et le gradient conjugué, en explorant leur capacité à converger vers des solutions optimales dans un cadre théorique et via des simulations numériques. Ces méthodes, bien qu'élégantes dans leur simplicité, présentent des défis particuliers en termes de convergence et de gestion de la complexité, que nous étudions à travers des modélisations détaillées et des analyses de performance.

Ensuite, nous explorons les méthodes Newtoniennes, y compris la méthode de Newton classique et les méthodes de Quasi-Newton, qui promettent une convergence plus rapide sous certaines conditions. L'accent est mis sur leur application à des problèmes d'optimisation non linéaires et leur efficacité dans la manipulation des matrices de grande taille, typiques des images de haute résolution.

La section suivante est consacrée à l'application de ces méthodes d'optimisation au traitement d'image numérique. Nous utiliserons la régularisation de Tikhonov et le modèle de Rudin-Osher-Fatemi pour débruiter un image préalablement bruité, ainsi que des techniques d'inpainting pour la reconstruction des partie manquantes d'une image .

Chaque méthode est évaluée non seulement en termes de capacité de réduction de bruit et de restauration de détails, mais aussi en regard de la complexité algorithmique et de l'efficacité computationnelle, en utilisant des simulations numériques concrètes. Ces analyses sont essentielles pour déterminer les méthodes les plus adaptées à des types spécifiques de problèmes de traitement d'image.

II. Autour des méthodes de gradient et pré conditionnement

Le but de ce premier travail préliminaire est d'étudier la convergence des méthodes d'optimisation vers le minimum d'une fonction quadratique, dans le contexte spécifique du traitement d'image. Nous analyserons leurs performances en termes de vitesse de convergence et de précision, et discuterons également du pré-conditionnement. Commençons par effectuer un rappel des méthodes :

1. Rappel des différentes méthodes de gradient

a) Gradient à pas optimal

Dans notre méthode de gradient à pas optimal, chaque itération ajuste la position actuelle en utilisant la direction du gradient négatif de la fonction objectif afin de s'orienter dans le sens de la pente de la fonction afin de trouver le minimum et le pas α est choisi de manière à minimiser exactement la fonction le long de cette direction.

Mathématiquement, cela se traduit par $\alpha = \frac{r^T r}{r^T A r}$, où r est le gradient de la fonction au point courant x, et A est utilisée ici pour calculer le produit scalaire avec r pour simuler une sorte d'évaluation quadratique, même si dans la méthode de pas optimal général, A ne serait pas nécessairement explicitée. L'implémentation commence avec un point initial x_0 et calcule le gradient en ce point. À chaque itération, le pas α est déterminé pour minimiser la fonction le long de la direction actuelle du gradient. Le point x est alors mis à jour en soustrayant le produit du pas optimal α avec le gradient actuel r.

Cette procédure est répétée jusqu'à ce que la norme du gradient soit inférieure à un seuil de tolérance où jusqu'à un nombre maximal d'itérations, indiquant la convergence vers un minimum local.

b) Gradient conjugué

La méthode du gradient conjugué commence par initialiser x à partir d'un point de départ x_0, calculant ensuite le gradient négatif r comme la direction initiale de descente telle que précédemment. La direction de recherche d est initialement définie égale à -r. Pour chaque itération, on calcule un pas α qui détermine la meilleure avancée le long de d, avec $\alpha = \frac{r^T r}{r^T A d}$.

Après la mise à jour de x, le nouveau gradient $r_{nouveau}$ est calculé, et si sa norme est sous un seuil de tolérance, la méthode s'arrête, signifiant qu'un minimum est potentiellement atteint. Si non, la méthode ajuste d pour la prochaine itération en la combinant avec $r_{nouveau}$ via un facteur de conjugaison $\beta = \frac{r_{nouveau}^T r_{nouveau}}{r^T r}$, permettant ainsi de prendre en compte la nouvelle information tout en conservant une partie de l'ancienne direction.

Cette combinaison assure que les directions de recherche sont conjuguées par rapport à A, optimisant la progression vers le minimum. Ainsi la principale différence réside dans l'ajustement de la direction en combinant le nouveau gradient avec la direction précédent modélisé par le paramètre β.

2. Modélisation des trajectoires

Commençons par implémenter une fonction J ainsi que son gradient ∇J. Nous utiliserons ces éléments pour tester nos différents programmes. Ensuite, nous modéliserons les lignes de niveau et la surface de notre courbe, et nous inclurons les méthodes du gradient à pas optimal et du gradient conjugué.

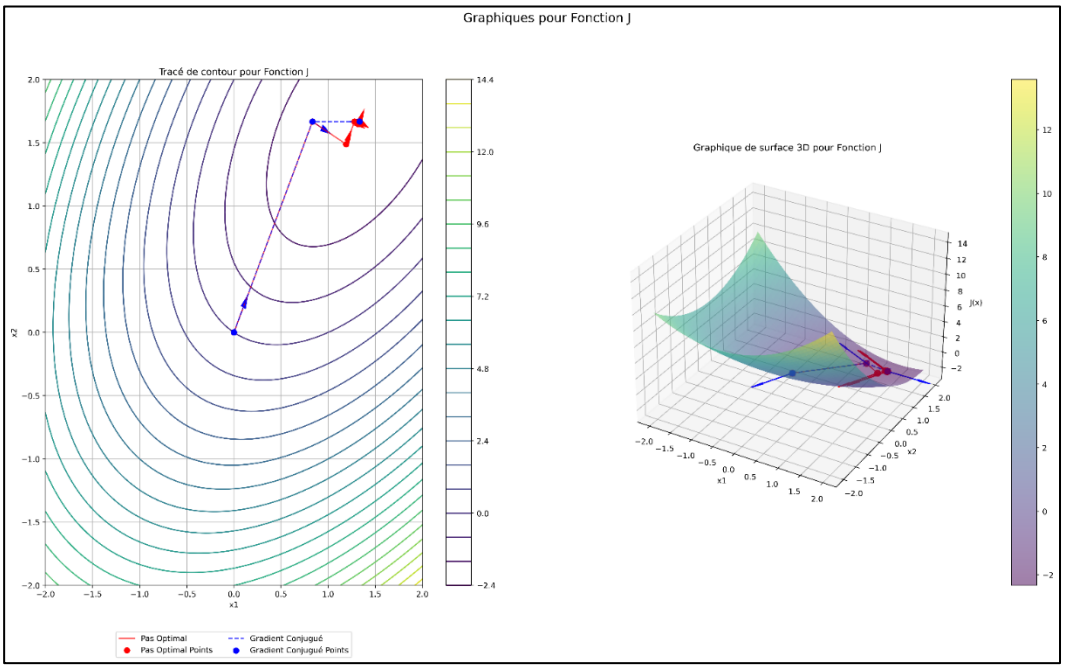


Figure 1 : Tracé du contour et surface 3D de la fonction J(x)

Nous observons que le gradient conjugué se dirige plus rapidement vers la solution que le gradient à pas optimal. De plus, cela est vérifié par la sortie de notre programme Python suivante :

```
Pas Optimal: x* = [1.33333312 1.6666664 ], J(x*) = -2.3333333333332718, Nombre d'itérations = 14, Temps écoulé = 0.0006746000144630671 s
Gradient Conjugué: x* = [1.33333333 1.66666667], J(x*) = -2.3333333333333335, Nombre d'itérations = 2, Temps écoulé = 3.929997910745442e-05 s
```

3. Performances et analyse de la convergence

Analysons la convergence et la performance des méthodes d'optimisation utilisées pour résoudre le problème quadratique. Nous examinerons la convergence de $J(x_n)$ vers son minimum et de $\|x_n\|$ vers la solution du problème. Nous implémenterons la matrice hessienne de J et optimiserons les calculs en exploitant la structure creuse de la matrice A. Enfin, nous comparerons les méthodes en termes de temps de calcul et d'itérations nécessaires pour atteindre une précision donnée, en représentant graphiquement les résultats obtenus.

a) Convergence de $J(x^n)$

Regardons, par le biais du graphique suivant, la convergence de $J(x^n)$ en fonction du nombre d'itérations :

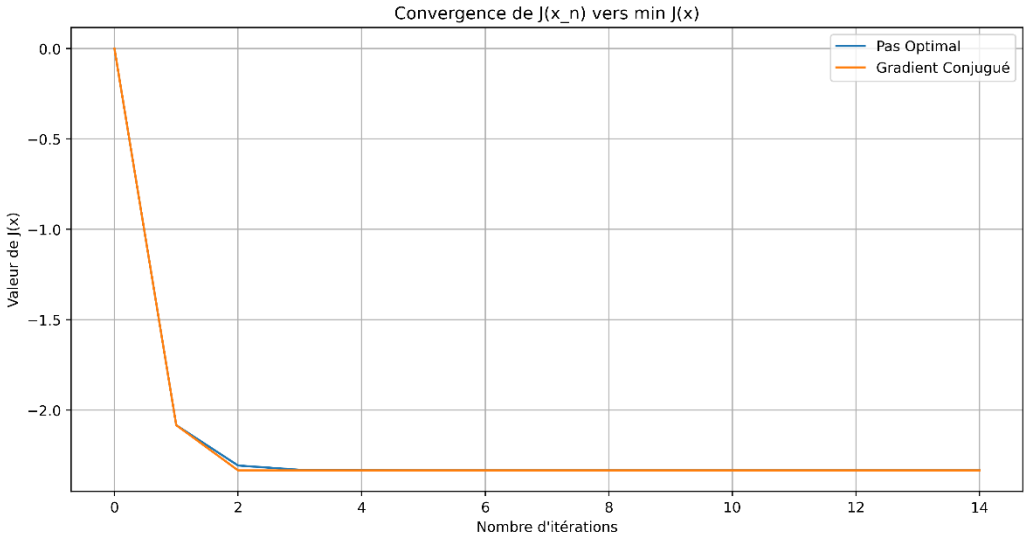


Figure 2 : Convergence de $J(x_n)$ vers $min\ J(x)$

Ce graphique montre la convergence de $J(x^n)$ vers la valeur minimale, nous voyons bien que la valeur est pratiquement atteinte $(\text{à } -2{,}333..)$ au bout de 3 itérations pour les deux méthodes.

Traçons désormais la convergence de la norme de x^n noté $\|x_n\|$

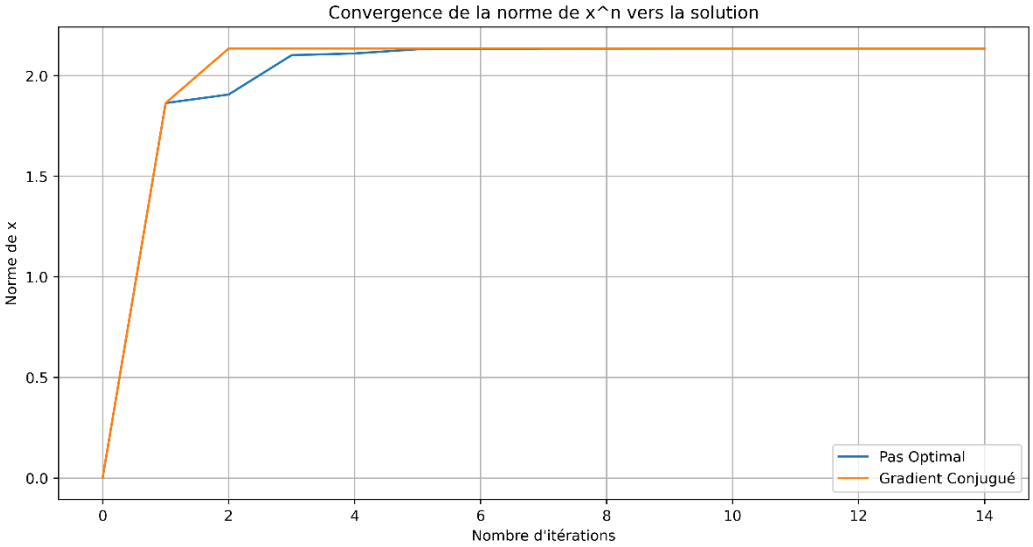


Figure 3 : Convergence de la norme de x_n vers la solution

Le graphique ci-dessus illustre la convergence de la norme de x_n vers la solution pour deux méthodes d'optimisation : le gradient à pas optimal et le gradient conjugué. Nous observons une nouvelle fois que le gradient conjugué atteint rapidement une norme proche de la solution en très peu d'itérations, montrant une convergence rapide. En comparaison, la méthode à pas optimal converge également, mais de manière légèrement plus lente.

b) Performances et temps de calcul

Exploitons la structure creuse de la matrice A afin de réduire le temps de calcul des méthodes d'optimisation. Nous ferons tourner les algorithmes pour différentes tailles de problèmes $N \in \{10, 20, 30, 40, 50\}$, en utilisant des critères d'arrêt basés sur le nombre d'itérations ou la précision souhaitée. Nous représenterons graphiquement la convergence des deux méthodes en traçant $log(\|Ax^n - b\|)$ en fonction du nombre d'itérations pour $N = 50$ et $N = 100$. Enfin, nous comparerons les temps de calcul et le nombre d'itérations nécessaires pour atteindre la précision souhaitée, en résumant les résultats dans un tableau. Cette analyse nous permettra d'évaluer l'efficacité et la performance des méthodes d'optimisation en fonction de la taille du problème.

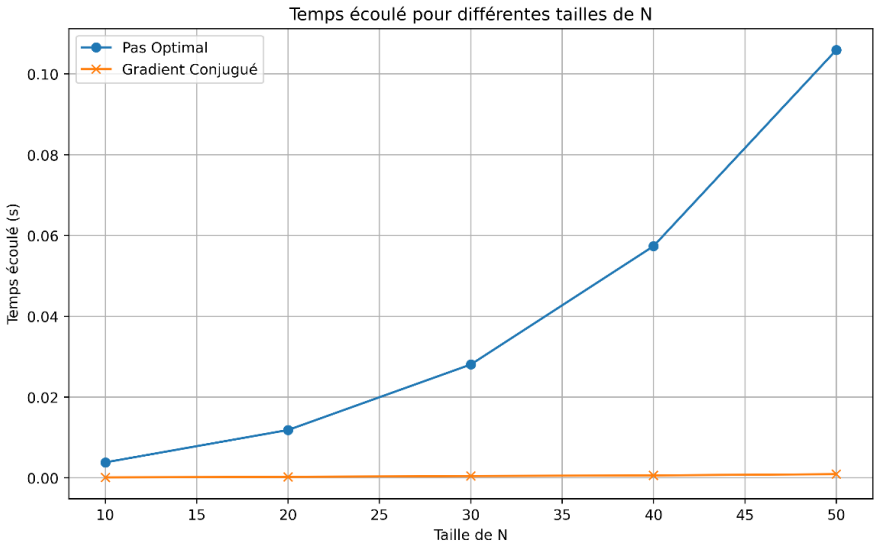


Figure 4 : Temps écoulé pour différentes tailles de N

Sur ce graphique, nous observons une nette différence dans les performances des deux méthodes. Le temps de calcul de la méthode à pas optimal augmente de manière significative avec la taille de N, indiquant une complexité temporelle plus élevée. En revanche, le gradient conjugué montre une stabilité linéairement croissante bien plus basse, avec des temps de calcul presque constants, indépendamment de la valeur des dimensions du problème. Cette différence souligne l'efficacité supérieure du gradient conjugué, surtout pour des problèmes de grande dimension, où sa capacité à maintenir un temps de calcul faible et constant le rend particulièrement avantageux.

Regardons désormais la convergence en utilisant le logarithme avec la fonction $log(\|Ax^n - b\|)$ et avec différents n, on verra $n = 50$ et $n = 100$ en fonction du nombre d'itérations telle que représentée par les courbes suivantes :

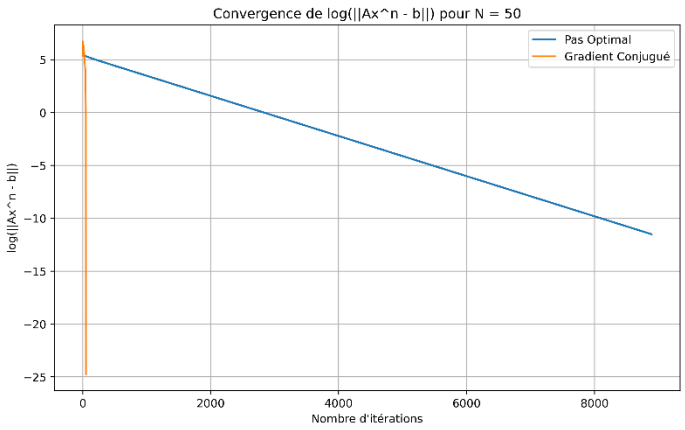


Figure 5 : Convergence du log pour $N = 50$

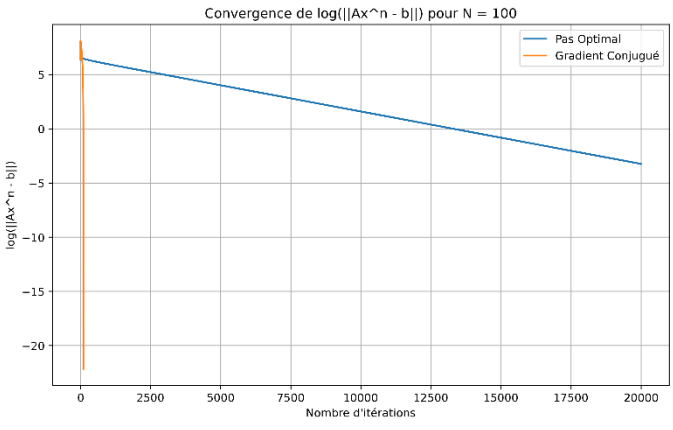


Figure 6 : Convergence du log pour $N = 100$

On observe, sur les 2 courbes précédentes, que le gradient conjugué atteint rapidement des valeurs très basses de log, démontrant une convergence rapide et efficace. En revanche, la méthode à pas optimal montre une diminution beaucoup plus lente du logarithme, nécessitant un nombre significatif d'itérations pour approcher, avec précision désirée, la solution. Cela confirme la capacité du gradient conjugué à gérer efficacement les problèmes de grande taille, réduisant rapidement l'erreur résiduelle avec un nombre d'itérations minimal. Cette performance supérieure du gradient conjugué est particulièrement avantageuse pour les applications où la rapidité et l'efficacité de la convergence sont primordiales.

Nous générons un tableau récapitulatif afin d'avoir une vue sur quelques chiffres :

Taille de N	Méthode	Temps écoulé (s)	Nombre d'itérations
50	Pas Optimal	0.107680	8894
50	Gradient Conjugué	0.000972	50
100	Pas Optimal	0.374724	20000
100	Gradient Conjugué	0.002730	100

Figure 7 : Tableau de résultats pour les différentes méthodes

Le tableau récapitulatif présente une comparaison des temps de calcul et du nombre d'itérations nécessaires pour atteindre la précision souhaitée ($\|Ax^n - b\| \leq 10^{-5}$) entre les méthodes de gradient à pas optimal et de gradient conjugué pour deux tailles de problème ($N = 50$ et $N = 100$). Les résultats montrent que le gradient conjugué nécessite significativement moins de temps et d'itérations pour converger. De plus, on observe que le nombre d'itérations pour le gradient conjugué est égal à la taille de la matrice, ce qui confirme son efficacité et sa rapidité en comparaison avec la méthode à pas optimal, qui nécessite beaucoup plus d'itérations et de temps de calcul.

4. Avec utilisation du pré conditionnement

Le pré conditionnement d'une matrice est une technique visant à améliorer la convergence des méthodes itératives notamment celles du gradient ici. Cette approche consiste à modifier la matrice initiale du problème d'optimisation pour qu'elle soit mieux conditionnée, facilitant ainsi la convergence de nos méthodes. En équilibrant les échelles des différentes variables ou en atténuant les effets indésirables de la géométrie sous-jacente du problème, le pré conditionnement permet d'obtenir des solutions plus efficacement. Dans cette section, nous explorerons le concept de pré conditionnement à travers un problème de minimisation.

Afin de préconditionner la fonction J, que l'on appellera $\tilde{J}$ nous avons utilisé une matrice diagonale D afin de transformer la matrice A et B initiale en une nouvelle matrice $\tilde{A}$ et $\tilde{B}$. Le but de cette transformation est de rendre le problème d'optimisation mieux conditionné.

On définit la matrice diagonale D telle que :

$$D = diag\left(\frac{1}{i}\right) \text{ pour } i = 1, \dots, N$$

On transforme la matrice A que l'on écrira $\tilde{A}$ par la transformation suivante :

$$\tilde{A} = DAD$$

De même pour $\tilde{b}$ qui après démonstration devient :

$$\tilde{b} = D^{-1}b$$

Le problème de minimisation devient bien alors :

$$\min_{x \in \mathbb{R}^N} \tilde{J}(x) = \frac{1}{2} x^T \tilde{A} x - \tilde{b}^T x$$

Calculons maintenant le nombre d'itérations nécessaire pour avoir $\|Ax^n - b\|_2 \leq 10^{-10}$ avec $n \leq 2 \times 10^{-3}$ pour les 2 méthodes de gradient avec la matrice J initiale comparée avec notre nouvelle matrice J préconditionné.

```
 • Pas Optimal Gradient (J): Convergence en 225 itérations, temps écoulé 0.0023 secondes
1: 0.05619061628075346
 • Gradient Conjugué (J): Convergence en 6 itérations, temps écoulé 0.0002 secondes
2: 0.056190555212082396
 • Pas Optimal Gradient (J modifiée): Convergence en 8 itérations, temps écoulé 0.0005 secondes
3: 12.095116580909371
 • Gradient Conjugué (J modifiée): Convergence en 5 itérations, temps écoulé 0.0001 secondes
4: 12.09511658359668
```

Figure 8 : Sortie de l'algorithme python des performances des méthodes

Nous observons que le pré conditionnement améliore notablement la performance de la méthode de gradient à pas optimal, réduisant le nombre d'itérations nécessaires de 225 à 8 et le temps de calcul de $\approx 10^{-3}$ à $\approx 10^{-4}$ secondes. De manière similaire, bien que le gradient conjugué soit déjà performant, le préconditionnement réduit légèrement le nombre d'itérations nécessaires et le temps de calcul, passant de 6 à 5 itérations et divisant par 2 le temps de calcul. Ces résultats démontrent l'efficacité du préconditionnement pour accélérer la convergence, surtout pour la méthode de gradient à pas optimal.

III. Autour des méthodes Newtoniennes

1. Méthode de Newton

Dans notre méthode de Newton, chaque itération ajuste la position actuelle en utilisant à la fois le gradient et la Hessienne de la fonction objectif afin de s'orienter plus efficacement vers le minimum. Cette méthode exploite l'information de la courbure de la fonction pour accélérer la convergence, en particulier près du minimum. Mathématiquement, à chaque itération, le gradient $g = \nabla J(x)$ et la Hessienne $H = \nabla^2 J(x)$ de la fonction au point courant x sont calculés. L'étape de mise à jour est ensuite déterminée en résolvant le système linéaire $H\Delta x = -g$ pour obtenir la direction de mise à jour Δx. Le point x est alors mis à jour en ajoutant Δx. Ce processus est répété jusqu'à ce que la norme du gradient soit inférieure à un seuil de tolérance, ou jusqu'à ce qu'un nombre maximal d'itérations soit atteint. Cette méthode est particulièrement efficace pour les fonctions convexes bien conditionnées, offrant une convergence quadratique rapide près de la solution optimale.

2. Performances et analyse de convergence de la méthode de Newton

Après avoir implémenté la méthode de Newton telle que l'annexe nous le demande, nous testons de tester la méthode pour $N = 2$ et nous traçons les points calculés à chaque itération :

On note :

$$J(x) = \log\left(\sum_{i=1}^{N} e^{x_i}\right)$$

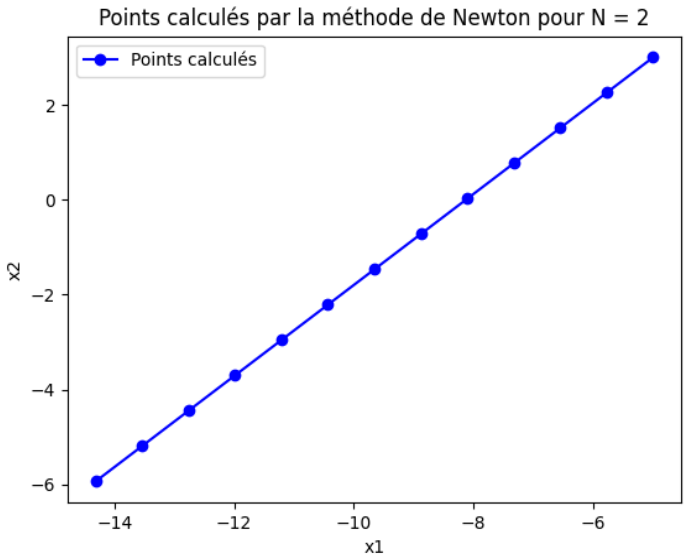


Figure 9 : Points calculés par la méthode de Newton pour la dimension $N = 2$

On observe une trajectoire linéaire, indiquant que les itérations successives suivent une direction bien définie vers la solution optimale. Cette linéarité suggère une convergence régulière et stable de la méthode de Newton dans ce cas particulier, ce qui est cohérent avec les attentes pour une fonction quadratique bien conditionnée.

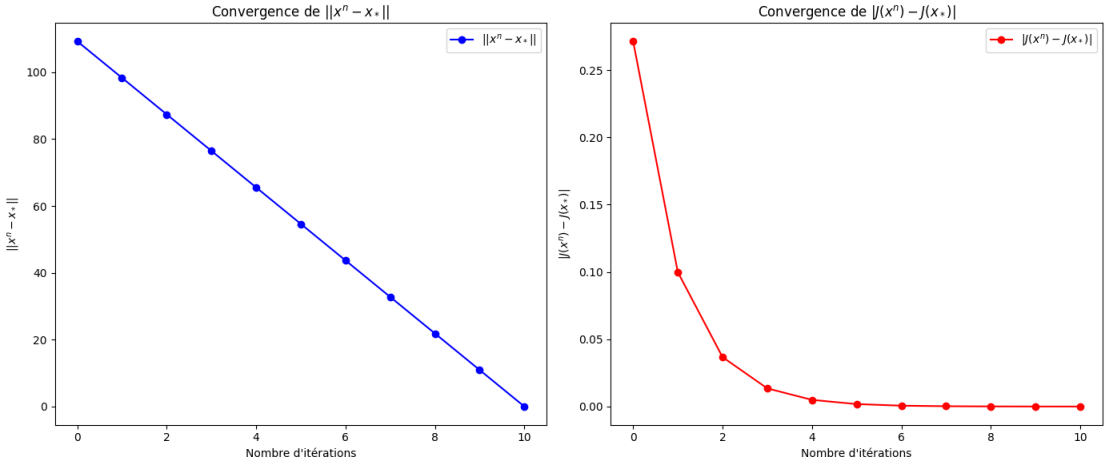


Figure 10 : Convergence des normes de x et $J(x)$

Les graphiques ci-dessus illustrent la convergence de la méthode de Newton pour la fonction objective J. Le premier graphique montre la norme de la différence $\|x^n - x_*\|$ entre les itérations successives et la solution optimale x_*. La décroissance linéaire régulière de cette norme indique une convergence stable et efficace, où chaque itération rapproche systématiquement x de x_*. Le second graphique présente la différence $\|J(x^n) - J(x_*)\|$ entre les valeurs de la fonction objective aux points x^n et x_*. On observe une diminution rapide initiale, suivie par une stabilisation, démontrant que la méthode de Newton améliore rapidement la valeur de J pour s'approcher de celle de la solution optimale. Ces courbes mettent en évidence l'efficacité de la méthode de Newton, avec une phase initiale de descente rapide, essentielle pour la performance globale de l'algorithme, suivie d'une phase de raffinement garantissant la précision de la solution finale.

Plus encore, voici les résultats que nous avons obtenus pour $\epsilon = 10^{-5}$ et $N \in \{10, 20, 30, 40, 50\}$ en comparaison des 3 méthodes (Newton, Gradient à pas optimal et Gradient conjugué) :

```
Resultats pour la methode de Newton :
N    Nombre d iterations Temps de calcul (s) Valeur de x*
2    12                  0.0003907000        [-3.7811171542  3.4088247687]
10   5                   0.0002315000        [-0.0656292869  0.0512069218]
20   8                   0.0003072000        [ 0.0463290577 -0.1086031842]
30   9                   0.0002515000        [-0.0115962654 -0.0440931678]
40   6                   0.0001678000        [-0.0483845621  0.0508810116]
50   8                   0.0002927000        [-0.0077578151 -0.0077266729]

Resultats pour le gradient non optimise :
N    Nombre d iterations Temps de calcul (s) Valeur de x*
2    9                   0.0001256000        [-0.2875770717  0.2724611782]
10   9                   0.0001870000        [-0.0540505902  0.0448798497]
20   6                   0.0000810000        [ 0.0537837517 -0.0913367213]
30   7                   0.0000912000        [-0.0102728899 -0.0431081532]
40   13                  0.0001616000        [-0.0471186976  0.0500768630]
50   8                   0.0001071000        [-0.0080720879 -0.0080135801]

Resultats pour le gradient conjugue :
N    Nombre d iterations Temps de calcul (s) Valeur de x*
2    2                   0.0000451000        [-0.2875650862  0.2724649625]
10   2                   0.0000444000        [-0.0540499069  0.0448804047]
20   2                   0.0000472000        [ 0.0537842230 -0.0913370096]
30   2                   0.0000437000        [-0.0102730199 -0.0431084984]
40   2                   0.0000457000        [-0.0471186936  0.0500765490]
50   2                   0.0000437000        [-0.0080719097 -0.0080136874]
```

Figure 10 : Tableau des résultats pour les 3 méthodes avec leurs solutions

Le tableau des résultats compare la méthode de Newton, le gradient non optimisé, et le gradient conjugué pour différentes valeurs de N, en termes de nombre d'itérations, temps de calcul, et valeur finale de x_* La méthode de Newton converge rapidement avec peu d'itérations (6 à 12) et des temps de calcul moyennement courts, bien que les valeurs de x_*montrent de légères variations. Le gradient non optimisé nécessite entre 6 et 13 itérations avec des temps de calcul plus court et parvient à des valeurs plus cohérentes de x_*. Le gradient conjugué se distingue par une convergence rapide en très peu d'itérations (souvent 2) et des temps de calcul extrêmement courts, avec des valeurs de x_* stables et précises. Ainsi, bien que la méthode de Newton et le gradient conjugué soient toutes deux efficaces, le gradient conjugué présente un léger avantage en termes de stabilité et de cohérence des résultats.

Traçons désormais les graphes de convergence de $J(x^n)$ ainsi que x^n pour ces différentes méthodes de façon graphique et vérifions ce que nous venons de montrer :

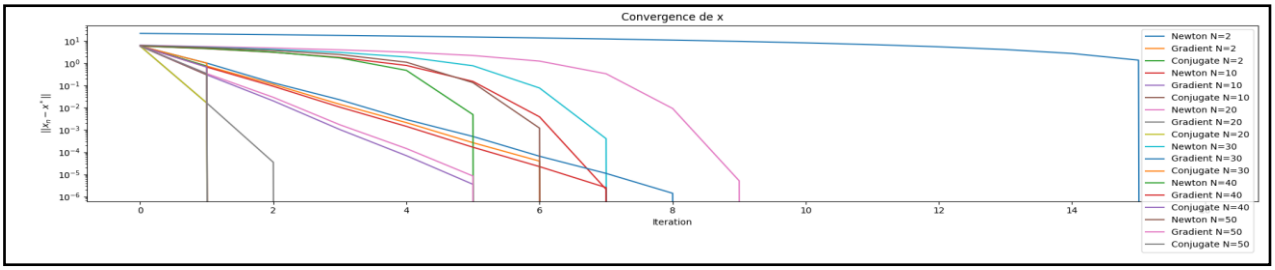


Figure 11 : Graphes des différentes méthodes en fonction de la convergences de x selon N

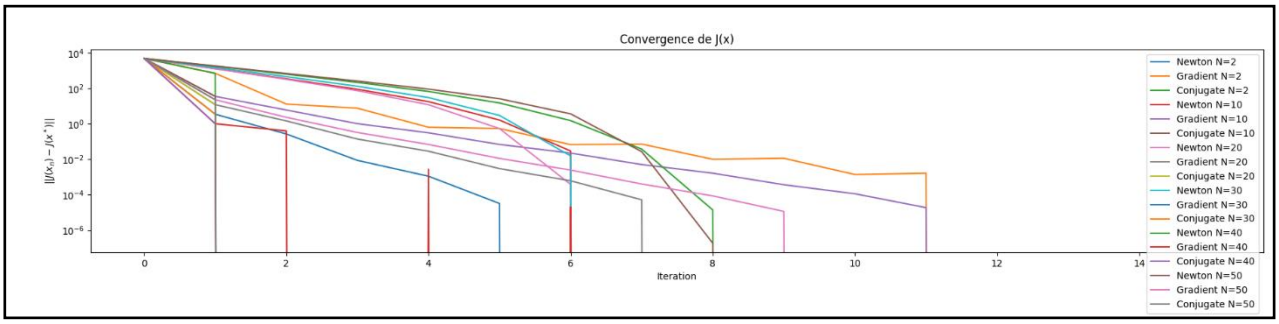


Figure 12 : Graphes des différentes méthodes en fonction de la convergences de J(x) selon N

La Figure 11 illustre la convergence de x pour diverses méthodes et valeurs de N, révélant que les méthodes de Newton convergent généralement plus rapidement que les méthodes de gradient conjugué. Contrairement à ce que l'on pourrait attendre, la méthode de Newton montre une convergence plus lente pour certaines valeurs de N, nécessitant plus d'itérations pour atteindre des valeurs faibles de ‖xk−x∗‖. La méthode du Gradient, quant à elle, converge de manière régulière et relativement rapide comparée à Newton pour certaine valeurs de N, montrant une descente graduelle mais stable. Enfin, la méthode du Gradient Conjugué se distingue par une convergence plus rapide pour la plupart des valeurs de N, atteignant rapidement des valeurs faibles de l'erreur.

La Figure 12 présente la convergence de J(x) pour les mêmes méthodes et valeurs de N. On observe que la méthode de gradient Conjugué est la plus rapide en termes de convergence de J(x). Elle dépasse les méthodes de Newton et du Gradient en atteignant plus rapidement des valeurs basses de J(x) et en maintenant une performance stable même avec l'augmentation de N. La méthode de Newton, bien que rapide au début, se stabilise et fluctue davantage, surtout pour des valeurs élevées de N. La méthode du Gradient est la plus lente, avec une descente plus progressive, nécessitant plus d'itérations pour converger.

3. Méthodes de Quasi-Newton

a) Méthode de Broyden-Fletcher-Goldfarb-Shanno

La méthode BFGS est une technique d'optimisation avancée qui est utilisée pour résoudre des problèmes d'optimisation non linéaires sans contraintes. La méthode BFGS est conçue pour approximer la Hessienne de la fonction objectif au lieu de la calculer directement, ce qui la rend particulièrement utile pour des problèmes où la Hessienne est difficile à déterminer ou coûteuse en calcul. Dans notre code, la méthode BFGS est implémentée via la fonction « minimize » de la bibliothèque « SciPy ».

Le processus commence avec un point initial x_0 et procède à l'optimisation en utilisant le gradient de la fonction, fourni par la fonction « grad », ainsi que le point initial et des paramètres supplémentaires comme le coefficient c. La méthode BFGS ajuste itérativement les approximations de la Hessienne en utilisant des informations tirées des gradients successifs. À chaque itération, elle calcule une direction de descente qui utilise cette Hessienne approximative et met à jour le point courant en conséquence.

b) Méthode Davidon–Fletcher–Powell (DFP)

La méthode Davidon-Fletcher-Powell (DFP) est une technique d'optimisation avancée utilisée pour résoudre des problèmes d'optimisation non linéaires sans contraintes. Comme la méthode BFGS, DFP est conçue pour approximer la Hessienne de la fonction objectif, évitant ainsi le calcul direct de cette matrice qui peut être coûteux ou impraticable. La méthode commence par choisir un point initial x_0 et une matrice d'identité B_0 comme approximation initiale de la Hessienne inverse.

À chaque itération, la direction de descente d est calculée en multipliant l'approximation courante de la Hessienne inverse BB par le gradient de la fonction $\nabla J(x)$. Le pas η est déterminé en minimisant $J(x_0 + \eta d)$ Le point x est ensuite mis à jour à l'aide de ce pas. La différence des points $s = x - x_0$ et la différence des gradients $y = \nabla J(x) - \nabla J(x_0)$ sont utilisées pour mettre à jour B en utilisant une formule de correction spécifique. Ce processus est répété jusqu'à ce que la norme du gradient soit inférieure à un seuil de tolérance, ou jusqu'à un nombre maximal d'itérations.

Dans notre implémentation, ce processus est suivi pour ajuster itérativement l'approximation de la Hessienne inverse, calculer les directions de descente, et mettre à jour les points de manière efficace. La méthode DFP, en combinant les informations des gradients successifs et en mettant à jour l'approximation de la Hessienne, permet une convergence rapide vers le minimum de la fonction objectif.

4. Analyse et résultats

Afin de tester nos 2 méthodes, nous lançons un test en dimension 2 avec la fonction $J(x) = \log\left(\sum_{i=1}^{N} e_i^{a^T x + b_i}\right)$ en la traçant la trajectoire de l'algorithme. Cela est illustrée sur la figure ci-dessous :

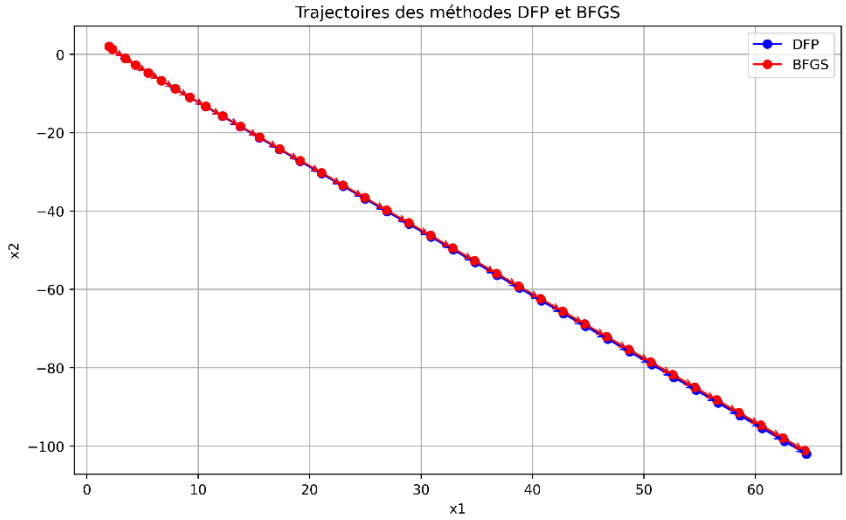


Figure 13 : Trajectoires des méthodes DFP et BFGS

On observe une trajectoire linéaire des 2 méthodes commençant à un point de départ identique qui est ici $[2, 2]$, indiquant que les itérations successives suivent une direction bien définie vers la solution optimale. Cette linéarité suggère une convergence régulière et stable de la méthode de ces 2 méthodes dans ce cas particulier.

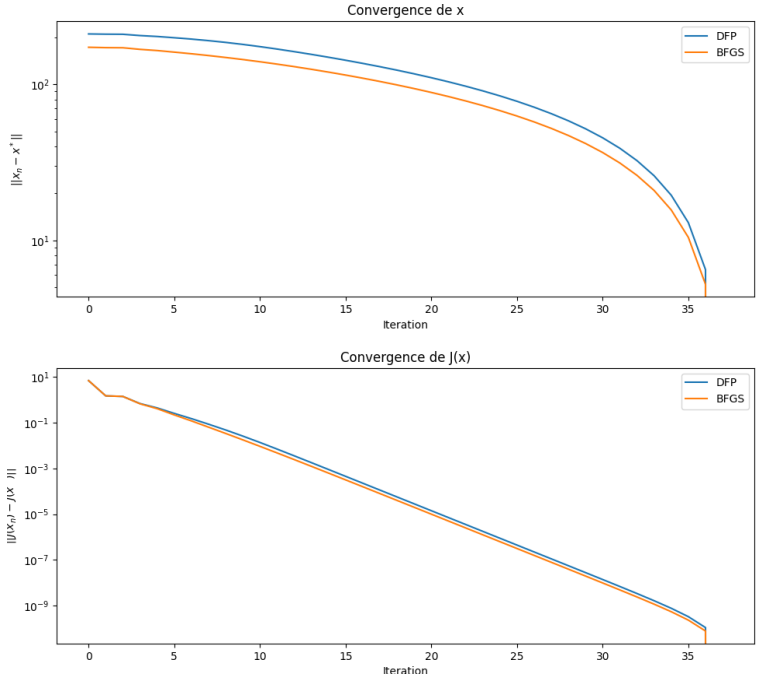


Figure 14 : Convergence de x et $J(x)$ en fonctions des méthodes

Les graphiques ci-dessus illustrent la convergence des 2 méthodes pour la fonction objective J définie précédemment. Le premier graphique montre la norme de la différence $\|x^n - x_*\|$ entre les itérations successives et la solution optimale x_*. La décroissance courbée de cette norme indique une convergence stable et efficace, où chaque itération rapproche systématiquement x de x_*. Le second graphique présente la différence $\|J(x^n) - J(x_*)\|$ entre les valeurs de la fonction objective aux points x^n et x_*. On observe une diminution plutôt linéaire, démontrant que la méthode ces deux méthodes améliorent rapidement la valeur de J pour s'approcher de celle de la solution optimale. La méthode DFP et BFGS paraissent relativement similaire quant aux performances obtenues.

Finalement, traçons les différentes convergences pour la fonction $J(x)$ pour nos différentes méthodes à des dimensions différentes :

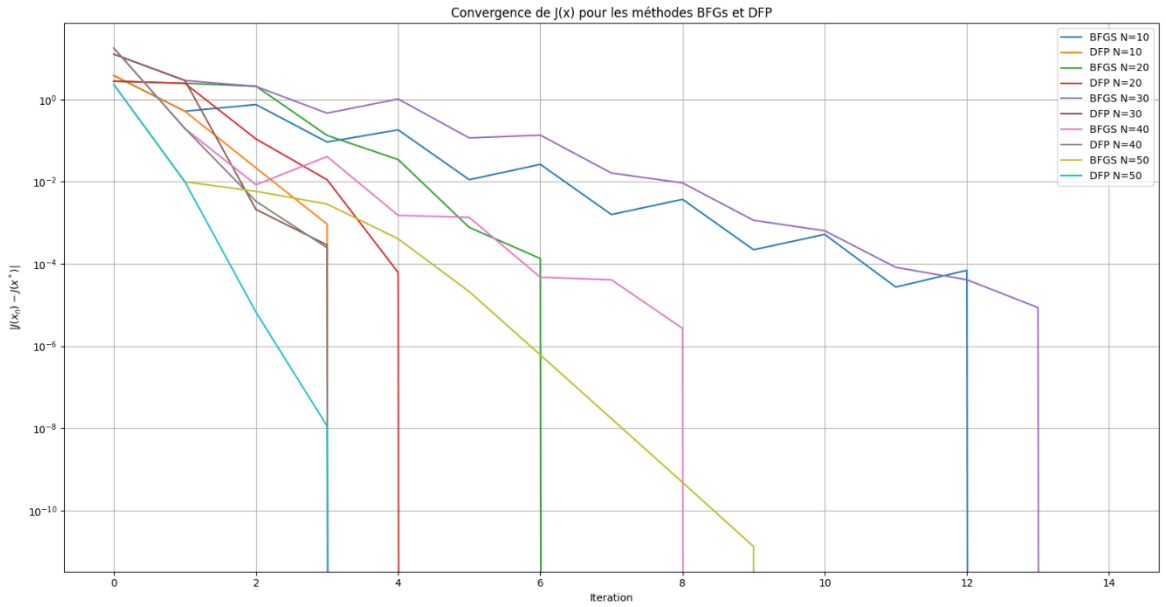


Figure 15 : Convergence de $J(x)$ pour les méthodes BFGS et DFP

Bien que les résultats visibles sur le graphique montrent une performance variable entre les méthodes BFGS et DFP selon les critères de tolérance, en théorie et dans la pratique courante, la méthode BFGS est souvent privilégiée pour sa robustesse et sa convergence rapide. En examinant de plus près les données, on constate que la méthode BFGS tend à stabiliser sa convergence au-delà des premières itérations, indiquant une approche plus consistante et moins sensible aux conditions initiales par rapport à DFP. Cette tendance est soutenue par la capacité de BFGS à mieux approximer la matrice hessienne inverse, fournissant ainsi des mises à jour plus précises et efficaces de la direction de recherche. De plus, la convergence superlinéaire de BFGS, par opposition à la convergence linéaire de DFP, en fait un choix préférable pour des applications exigeant une grande précision et efficacité, même si les résultats expérimentaux peuvent varier en fonction des spécificités du problème et de l'implémentation

IV. Introduction et manipulation des images

Cette partie du rapport présente les différentes étapes de traitement et d'analyse d'images en utilisant des techniques de manipulation d'image et d'optimisation numérique. Les images utilisées dans ce rapport sont traitées à l'aide de bibliothèques Python telles que *PIL* et *numpy*. L'objectif est de démontrer les étapes de conversion d'images, de modification de contraste, de calcul de gradients, de détection de contours et d'analyse de bruit dans les images.

Nous avons choisi d'utiliser l'image de l'oiseau durant tout le TP3. Vous pouvez retrouver la première partie du TP1 dans le fichier « TP3 - P1 ».

Figure 16 : Image originale choisie

Pour commencer, nous avons chargé une image en couleur et l'avons convertie en niveaux de gris. La conversion en niveaux de gris est une étape préliminaire importante dans le traitement d'image car elle simplifie les calculs en réduisant le nombre de canaux de couleur.

```
# Charger l'image
image_path = "oiseau.png"
image = Image.open(image_path)

# Convertir l'image en niveaux de gris
gray_image = image.convert("L")
```

Figure 17 : Procédure python de chargement de l'image avec conversion en niveau de gris

L'image en niveaux de gris a été redimensionnée à une taille de 100×100 pixels pour normaliser les dimensions de l'image. Ensuite, une région de 50×50 pixels a été recadrée à partir du centre de l'image redimensionnée afin de simplifier notre étude.

```
# Redimensionner l'image à 100x100 pixels
resized_image = gray_image.resize((100, 100))

# Recadrer une région de 50x50 pixels
cropped_image = resized_image.crop((25, 25, 75, 75))
```

*Figure 18 : Redimensionnement de l'image en 50*50*

Nous avons écrit une fonction pour modifier l'image en ajustant sa moyenne et son écart-type afin d'augmenter le contraste de l'image. Cette technique permet de rehausser les détails présents dans l'image et simplifier plus tard le dé bruitage.

```python
def modify_image(image, new_mean, new_std):
    # Convertir l'image en tableau numpy
    img_array = np.array(image)
    image_array = img_array/255
    print(image_array)
    print(image_array.shape)

    # Calculer la moyenne et l'écart-type actuels
    current_mean = np.mean(image_array)
    current_std = np.std(image_array)

    # Ajuster les valeurs des pixels
    modified_image_array = (image_array - current_mean) / current_std * new_std + new_mean
    modified_image_array = np.clip(modified_image_array, a_min: 0, a_max: 255)  # Limiter les valeurs entre 0 et 255
    modified_image_array = modified_image_array.astype(np.uint8)  # Convertir en entiers 8 bits

    # Convertir en image PIL
    modified_image = Image.fromarray(modified_image_array)
    return modified_image


# Modifier l'image pour augmenter le contraste
new_mean = 128
new_std = 64
modified_image = modify_image(cropped_image, new_mean, new_std)
```

*Figure 18 : Redimensionnement de l'image en 50*50*

Voici l'ensemble des images que nous avons obtenu à la suite de ces diverses opérations.

Figure 19 : Affichage de toutes les images d'étapes en étapes partant de l'originale jusqu'à celle modifiée pour que l'on utilisera tout le long de l'étude

Dans un second temps, nous avons affiché les différentes étapes de traitement de l'image, y compris l'image originale, l'image en niveaux de gris, l'image redimensionnée, l'image recadrée et l'image avec contraste ajusté. De plus, nous avons tracé l'histogramme de l'image modifiée pour visualiser la distribution des valeurs de pixels.

```
1 usage
def plot_histogram(image):
    # Convertir l'image en tableau numpy
    image_array = np.array(image)

    # Calculer l'histogramme
    hist, bins = np.histogram(image_array.flatten(), bins=256, range=[0, 256])

    # Afficher l'histogramme
    plt.plot(hist)
    plt.title("Histogramme de l'image")
    plt.xlabel("Valeurs de pixel")
    plt.ylabel("Fréquence")
    plt.show()
```

Figure 20 : Algorithme python affichant l'histogramme de l'image

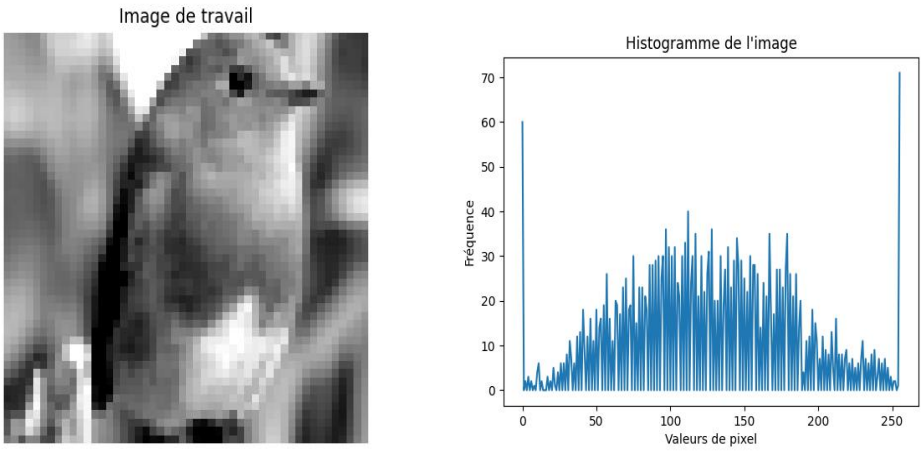


Figure 21 : Histogramme (à droite) du dégradé de gris de l'image de travail (à gauche)

L'histogramme présente une distribution relativement uniforme dans la plage centrale (entre 50 et 200), ce qui suggère que l'image a un bon contraste avec une diversité de niveaux de gris. Les valeurs de pixel sont réparties de manière assez homogène, ce qui peut indiquer que les détails de l'image sont bien répartis. Les pics aux extrémités suggèrent des zones de saturation, probablement dues à l'ajustement du contraste, tandis que la distribution plus uniforme au centre indique une bonne répartition des nuances de gris.

Ensuite, nous avons implémenté une fonction pour calculer le gradient (∇) de l'image en utilisant des différences finies. Ensuite, nous avons calculé la norme du gradient qui aide à détecter les contours dans l'image ainsi que son histogramme.

```python
def grad(image):
    img_array = np.array(image, dtype=float) / 255.0  # Normaliser les valeurs des pixels entre 0 et 1
    m, n = img_array.shape

    grad_x = np.zeros((m, n))
    grad_y = np.zeros((m, n))

    # Calculer les dérivées partielles
    for i in range(m):
        for j in range(n):
            if i < m - 1:
                grad_x[i, j] = img_array[i + 1, j] - img_array[i, j]
            else:
                grad_x[i, j] = 0

            if j < n - 1:
                grad_y[i, j] = img_array[i, j + 1] - img_array[i, j]
            else:
                grad_y[i, j] = 0

    return grad_x, grad_y
```

Figure 22 : Algorithme du gradient implémenté sur python

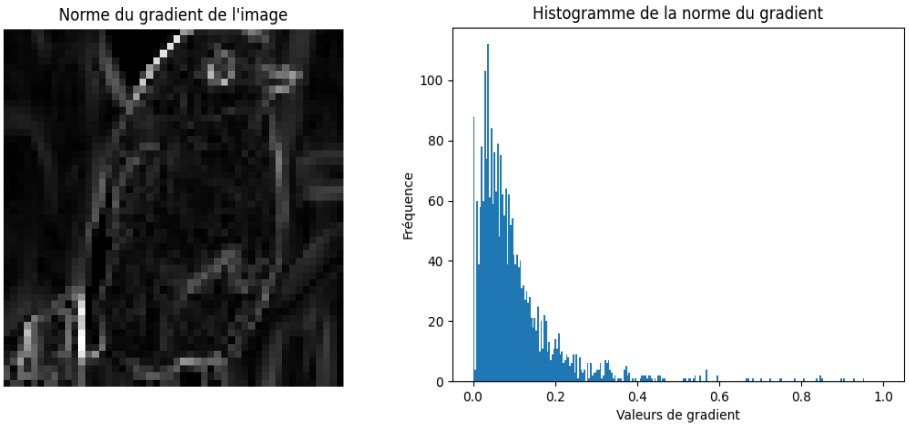


Figure 23 : Histogramme (à droite) de la norme du gradient de l'image (à gauche)

Les zones claires dans l'image de la norme du gradient correspondent aux contours et aux zones de transition rapide entre différentes intensités de gris. Ces zones indiquent les bords des objets présents dans l'image. On peut clairement voir les contours du corps de l'oiseau et certaines structures internes, ce qui indique que le calcul du gradient a été efficace pour extraire les caractéristiques principales de l'image. Les zones sombres correspondent aux régions de l'image où il y a peu ou pas de variation dans l'intensité des pixels. Ces zones sont généralement des surfaces uniformes ou lisses.

En analysant l'histogramme, on constate que la majorité des valeurs de gradient sont proches de 0, comme le montre le pic élevé à gauche de l'histogramme. Cela signifie qu'une grande partie de l'image est relativement uniforme

Nous avons ensuite détecté les contours de l'image en appliquant un seuil de 0.1 sur la norme du gradient et en affichant les points où la norme dépasse ce seuil. En appliquant ce seuil et en affichant les points où la norme dépasse ce seuil, on peut efficacement détecter les contours de notre image. Cette approche permet de simplifier l'image, d'identifier les structures importantes, de faciliter la segmentation et d'améliorer la perception visuelle pour diverses applications de traitement d'image et de vision par ordinateur.

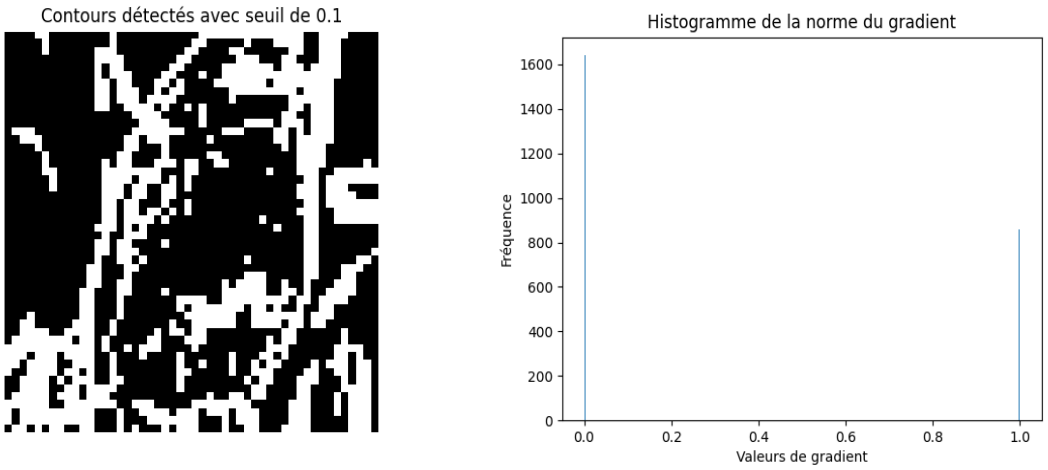


Figure 24 : Histogramme (à droite) de la norme du gradient en noir et blanc avec un seul de 0,1 de l'image (à gauche)

En analysant notre histogramme, nous observons une prédominance des zones noires indiquant que la majeure partie de l'image est relativement uniforme en termes d'intensité des pixels. Les zones blanches, bien que moins nombreuses, représentent les contours et sont essentielles pour l'analyse des structures et des formes présentes dans l'image. Cette approche permet de simplifier l'image en se concentrant sur les caractéristiques les plus importantes tout en réduisant les détails moins pertinents.

Pour conclure, nous avons calculé la divergence du gradient afin d'obtenir le Laplacien (Δ) de l'image, un outil mathématique puissant qui permet d'analyser les variations locales de l'intensité des pixels de manière plus approfondie. Le Laplacien met en évidence les zones de l'image où l'intensité des pixels change rapidement, identifiant ainsi les détails fins et les structures complexes. Il est particulièrement utile pour détecter les bords, les coins et les autres caractéristiques de texture de l'image. En fournissant une mesure de la courbure de la surface de l'image, le Laplacien aide à distinguer les régions plates des régions où l'intensité change de manière significative, facilitant ainsi des tâches avancées de traitement d'image telles que le rehaussement des contours, la détection des objets, et l'analyse des textures.

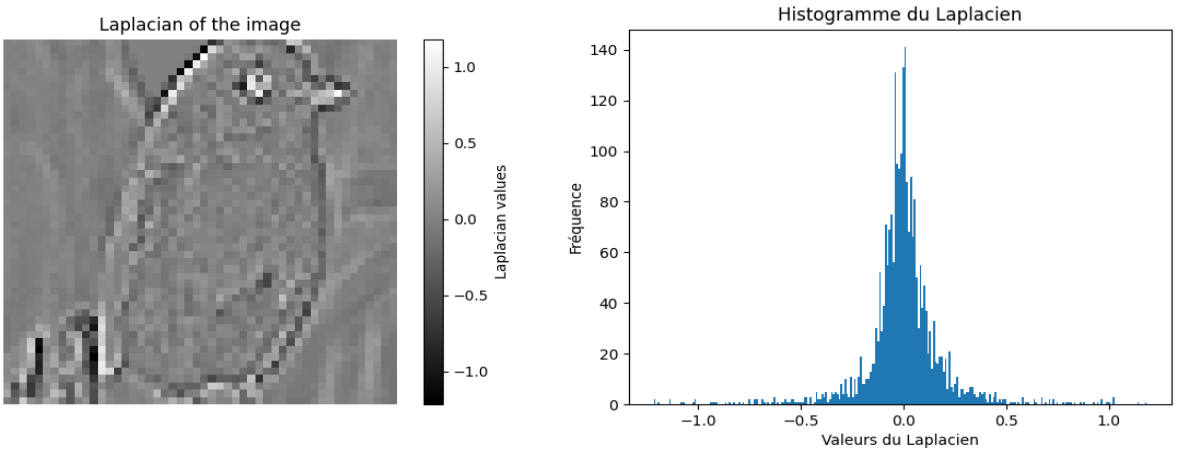


Figure 25 : Histogramme (à droite) du laplacien de l'image (à gauche)

Les zones où la valeur du Laplacien est élevée (claires) ou très basse (sombres) indiquent des transitions rapides dans l'intensité des pixels. Ces transitions sont typiques des contours et des bordures des objets dans l'image. Les régions avec des valeurs de Laplacien proches de zéro (nuances de gris moyen) correspondent aux zones de l'image où l'intensité des pixels est relativement uniforme sans changements brusques.

L'histogramme à droite représente la distribution des valeurs du Laplacien dans l'image. Le pic central très marqué à zéro signifie que les zones uniformes sont prédominantes dans l'image. Les valeurs de Laplacien s'étendent des deux côtés de zéro, avec des pics qui diminuent lentement. Ces petits pics représentent les pixels où il y a des variations intenses d'intensité, correspondant aux contours et aux bordures des objets. La présence de valeurs négatives et positives du Laplacien montre les différents types de transitions de gradient, soit d'une intensité élevée à faible (valeurs négatives) ou faible à élever (valeurs positives).

V. Débruitage d'une image numérique

1. Débruitage d'image par la régularisation de Thikonov

Le débruitage d'image est un problème fondamental dans le traitement des images numériques. La méthode de régularisation de Tikhonov est une approche couramment utilisée pour traiter ce problème. L'idée principale est de réduire les variations entre les pixels voisins dans une image débruitée. Pour ce faire, nous posons :

$$r(\nabla u) = \frac{1}{2} \| \nabla u \|^2$$

Le problème d'optimisation peut être formulé comme suit :

$$J(u) = \frac{1}{2} \| v - u \|^2 + \frac{\lambda}{2} \| \nabla u \|^2$$

Où J est une fonction convexe, fortement convexe de module λ et différentiable telle que :

$$\nabla J(u) = (u - v) - \lambda div(\partial x u, \partial y u)$$

Dans un premier temps nous allons reprendre notre image d'oiseau de taille 50×50 est lui appliquons un bruit blanc gaussien de déviation standard 10 à l'aide la fonction python suivante :

```
def bruit(x):
    im = np.array(x, dtype=float)
    v = im.copy()
    for i in range(np.shape(im)[0]):
        for j in range(np.shape(im)[1]):
            v[i, j] = im[i, j] + np.random.normal( loc: 0,  scale: 10)
    return v
```

Figure 26 : Fonction python de la définition du bruit de l'image

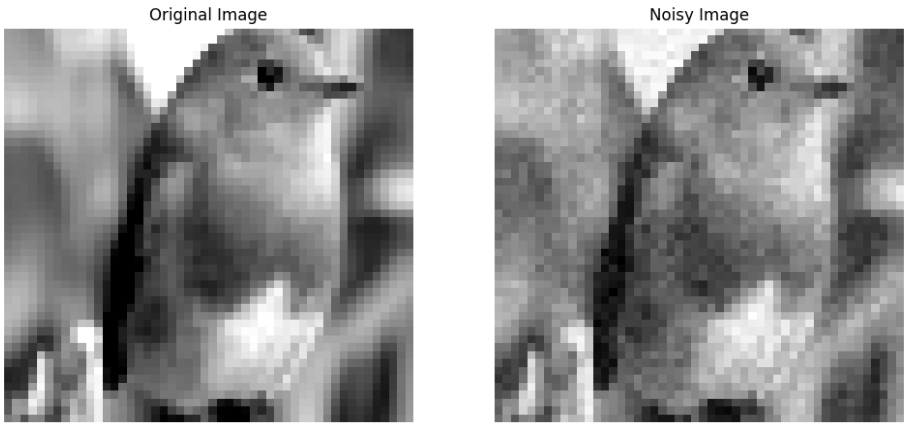


Figure 27 : Image originale à gauche et bruitée à droite

Nous avons calculé le Laplacien de notre image bruitée. Contrairement au Laplacien de l'image non bruitée, où les contours sont bien définis, le Laplacien de l'image bruitée montre que les contours sont considérablement atténués et deviennent difficilement identifiables en raison du bruit.

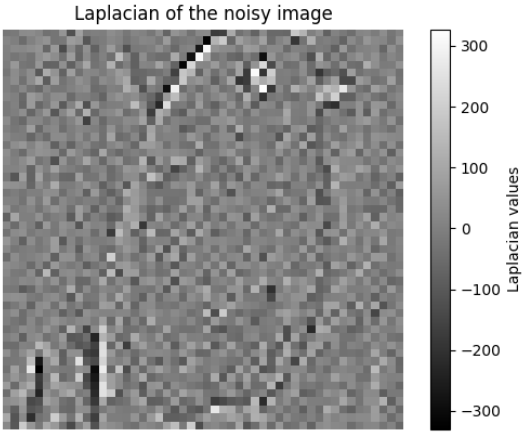


Figure 28 : Laplacien de l'image bruitée

Les fonctions grad et div de la bibliothèque numpy ne répondaient pas entièrement à la définition donnée dans le sujet et causaient des problèmes de résolution avec diverses méthodes. Pour résoudre ces problèmes, nous avons réécrit nos propres fonctions pour calculer le gradient et la divergence :

```python
def grad(x):
    m, n = x.shape
    dx = np.zeros((m, n))
    dy = np.zeros((m, n))
    dx[:m-1, :] = x[1:m, :] - x[:m-1, :]
    dy[:, :n-1] = x[:, 1:n] - x[:, :n-1]
    return dx, dy


3 usages
def div(gx, gy):
    m, n = gx.shape
    dxgx = np.zeros((m, n))
    dxgx[0, :] = gx[0, :]
    dxgx[1:m-1, :] = gx[1:m-1, :] - gx[0:m-2, :]
    dxgx[m-1, :] = -gx[m-2, :]
    dygy = np.zeros((m, n))
    dygy[:, 0] = gy[:, 0]
    dygy[:, 1:n-1] = gy[:, 1:n-1] - gy[:, 0:n-2]
    dygy[:, n-1] = -gy[:, n-2]
    return dxgx + dygy
```

Figure 29 : Image originale à gauche et bruitée à droite

Pour résoudre le problème d'optimisation, nous avons défini la fonction objectif J(u) et son gradient $\nabla J(u)$ conformément aux définitions données dans le sujet.

```python
# Define the objective function J(u)
def J(u, v, lambda_reg):
    u = u.reshape(v.shape)  # Assurez que u a la même forme que v
    grad_x, grad_y = grad(u)  # Calcul des gradients en x et y
    norm_squared = np.sum(grad_x**2 + grad_y**2)  # Somme des carrés des gradients à chaque point
    return 0.5 * np.sum((v - u)**2) + 0.5 * lambda_reg * norm_squared

# Define the gradient of the objective function
4 usages
def grad_J(u, v, lambda_reg):
    u = u.reshape(v.shape)  # Assurez que u a la même forme que v
    grad_x, grad_y = grad(u)  # Calcul des gradients en x et y
    div_grad_u = div(grad_x, grad_y)  # Calcul de la divergence des gradients
    return (u - v - lambda_reg * div_grad_u).flatten()
```

Figure 30 : Définition sur python de la fonction $J(x)$ et de son gradient

Enfin, nous avons récupéré les fonctions BFGS et Gradient Conjugué que nous avions utilisées lors des TP1 et TP2. Ces algorithmes sont particulièrement efficaces pour les problèmes d'optimisation de grande dimension et nous permettrons de débruité notre image.

Maintenant, nous passons à l'analyse des résultats de cette partie :

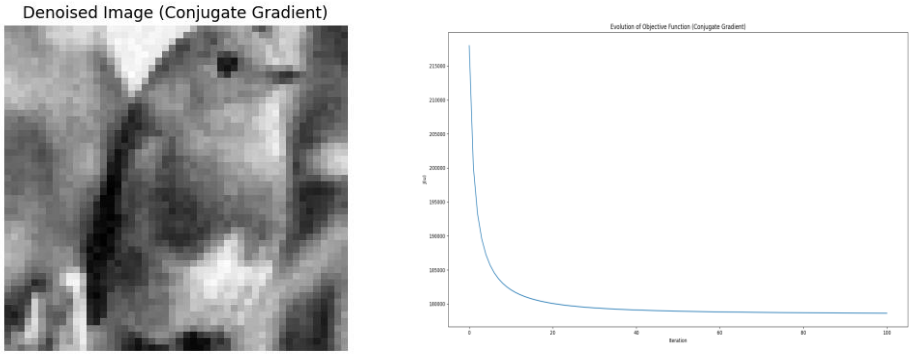


Figure 31 : Image débruitée par la méthode du gradient conjugué accompagnée de l'évolution de la fonction objective J en fonction du nombre d'itérations

L'image obtenue par la méthode du Gradient Conjugué montre une amélioration significative par rapport à l'image bruitée initiale. Les détails de l'oiseau sont plus clairs, bien que des traces de bruit soient encore visibles. Le graphique de convergence associé révèle que l'erreur objective diminue rapidement au cours des premières itérations pour ensuite se stabiliser. Cela indique que l'algorithme a une bonne performance globale, parvenant à réduire efficacement l'erreur initiale.

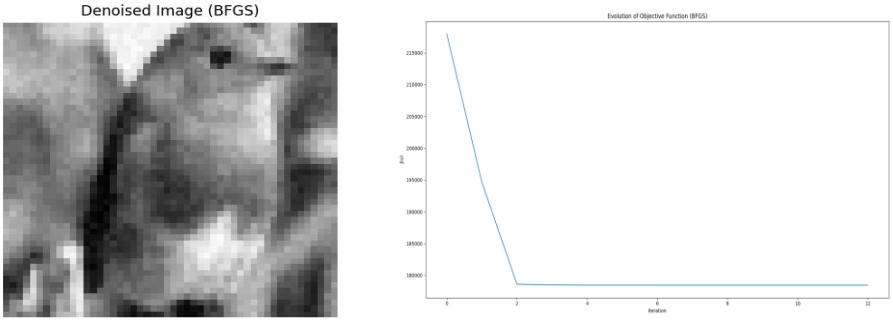


Figure 32 : Image débruitée par la méthode BFGS de l'évolution de la fonction objective J en fonction du nombre d'itérations

En comparaison, l'image obtenue à l'aide de la méthode BFGS présente également une nette amélioration. Le graphique de convergence pour BFGS montre une diminution rapide de l'erreur, se stabilisant plus tôt que pour la méthode du Gradient Conjugué. Cela suggère que BFGS est plus efficace en termes de nombre d'itérations nécessaires pour atteindre une solution optimale.

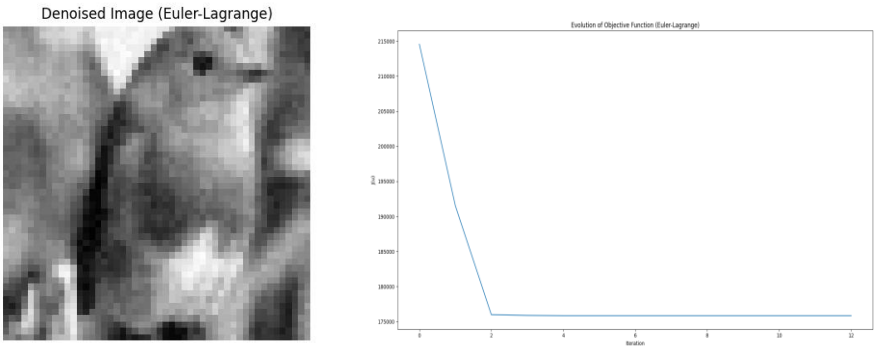


Figure 32 : Image débruitée par la méthode BFGS de l'évolution de la fonction objective J en fonction du nombre d'itérations

Enfin, l'image débruitée obtenue par l'équation d'Euler-Lagrange montre également une amélioration notable. Le graphique de convergence associé montre une diminution rapide de l'erreur dès les premières itérations, se stabilisant très rapidement. Cela indique que cette méthode est aussi très efficace pour atteindre une solution optimale rapidement.

Pour discuter de l'influence du paramètre λ sur la solution û, nous pouvons analyser les graphiques de l'évolution de l'erreur quadratique moyenne (RMSE) par rapport à λ pour les méthodes BFGS et Gradient Conjugué. Ces graphiques montrent comment la valeur de λ affecte la qualité de l'image débruitée et nous permet de déterminer de façon simple le lambda optimal.

```
def calculate_rmse(denoised, original):
    return np.sqrt(np.mean((denoised - original) ** 2))
```

Figure 33 : Définition de la fonction python de calcul de l'indice de qualité RMSE

La fonction calculate_rmse calcule l'erreur quadratique moyenne entre une image débruitée et l'image originale. Elle prend deux paramètres : l'image débruitée et l'image originale. La fonction commence par calculer la différence entre les deux images pour obtenir les erreurs de chaque pixel, puis élève ces erreurs au carré pour s'assurer qu'elles sont positives et amplifie les erreurs plus importantes. Ensuite, elle calcule la moyenne de toutes les erreurs quadratiques, et enfin, prend la racine carrée de cette moyenne pour obtenir la RMSE.

Une faible RMSE signifie que l'image débruitée est très proche de l'image originale, indiquant une bonne performance de la méthode de débruitage, tandis qu'une haute RMSE indique une grande différence entre les deux images, signalant une efficacité moindre du débruitage. Cette mesure est essentielle pour évaluer la qualité des algorithmes de débruitage, car elle prend en compte toutes les erreurs de pixel et les ramène à l'échelle des valeurs de pixel de l'image.

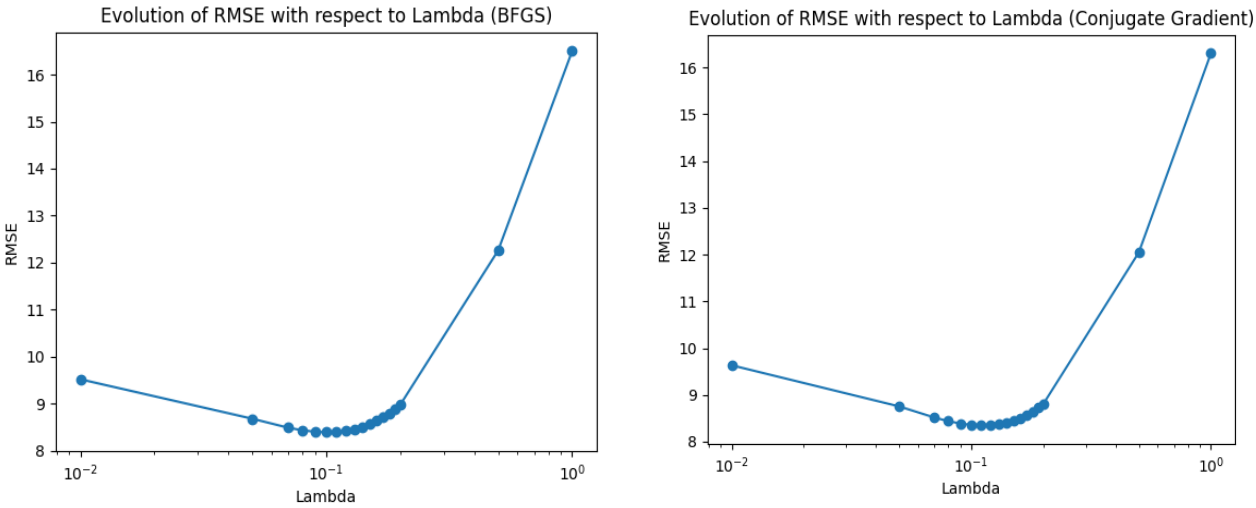


Figure 34 : Evolution du RMSE en fonction de la valeur de λ pour la méthode du gradient conjugué et BFGS

Le graphique de gauche illustre l'évolution du RMSE en fonction de λ pour la méthode BFGS. On observe une courbe en forme de parabole, où le RMSE diminue initialement avec l'augmentation de λ, atteignant un minimum autour de $\lambda \approx 0{,}11$, puis augmente rapidement. Cette tendance indique qu'il existe une valeur optimale de λ pour laquelle l'image débruitée est de la meilleure qualité possible (avec le plus faible RMSE). Lorsque λ est trop faible, la régularisation est insuffisante et le bruit n'est pas suffisamment réduit. À l'inverse, lorsque λ est trop élevé, la régularisation devient excessive, lissant trop l'image et supprimant des détails importants, ce qui conduit à une augmentation du RMSE. Concernant la courbe avec le gradient conjugue nous arrivons à un résultat similaire $\lambda \approx 0{,}12$.

Voici le bilan de toutes les images obtenues. En synthèse, bien que la méthode du Gradient Conjugué obtienne un RMSE légèrement inférieur, elle rend l'image plus floue. Par conséquent, la méthode d'Euler-Lagrange semble être la plus efficace pour débruiter l'image tout en préservant les détails, faisant d'elle la méthode privilégiée pour cette tâche.

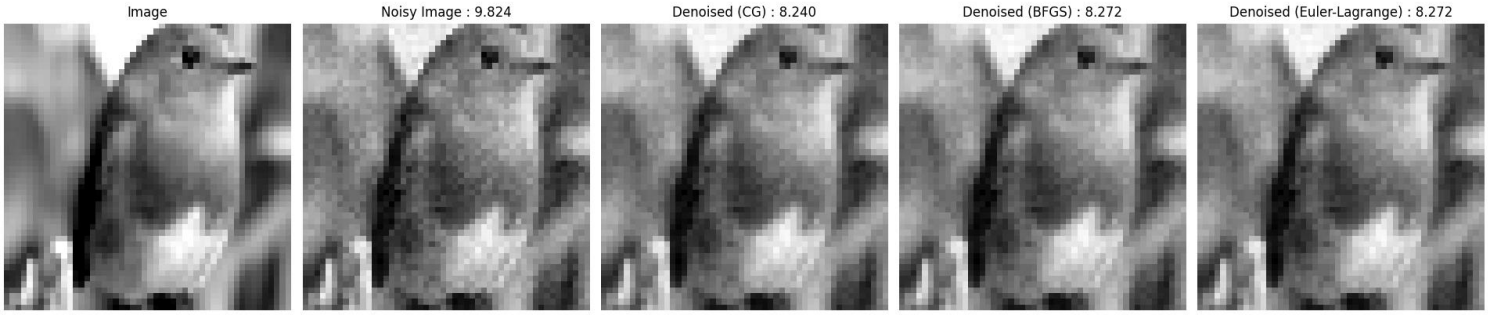


Figure 35 : Images par les différentes méthodes accompagnées de leur indice RMSE

```
RMSE avant traitement: 9.824425674590964
RMSE Gradient Conjugué: 8.239875038863975
RMSE BFGS: 8.271971021856643
RMSE Euler-Lagrange: 8.271966084354458
```

Figure 36 : Résultat RMSE de nos différentes méthodes

Nous avons eu l'idée d'entreprendre de tester de débruité une image déjà débruitée pour voir si une deuxième application de cette méthode pourrait améliorer encore la qualité de

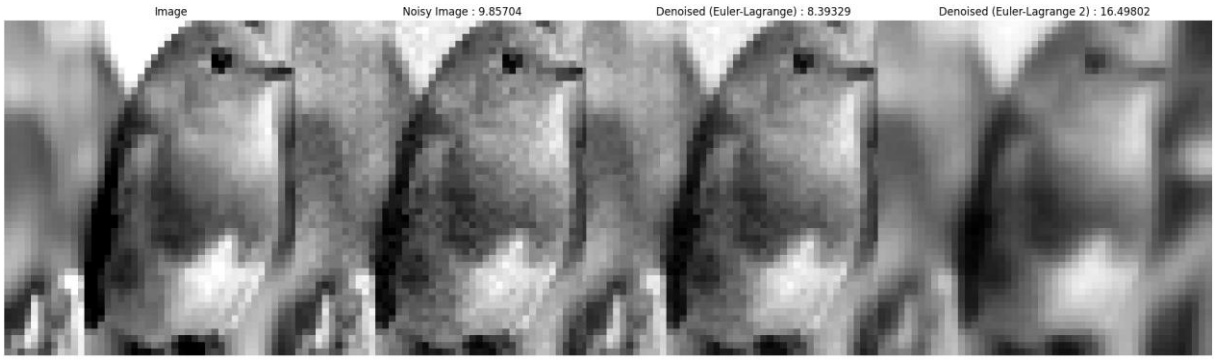


Figure 37 : test de débruitage d'une image débruité

En observant ces images, on peut constater que la deuxième application de la méthode d'Euler-Lagrange n'améliore pas la qualité de l'image, mais au contraire, augmente significativement la RMSE et rend flou notre image comme si l'on lui donnait une valeur de λ plus élevée.

En conclusion, repasser une image déjà débruitée par un processus de débruitage ne sert à rien et peut même nuire à la qualité de l'image. Une seule application de la méthode de débruitage, lorsqu'elle est bien choisie, est suffisante pour obtenir des résultats optimaux.

2. Débruitage pour le modèle de Rudin-Osher-Fatemi

Dans le cadre de notre projet de traitement d'images, nous nous concentrons sur l'amélioration du modèle de débruitage proposé par Rudin, Osher et Fatemi en 1992, basé sur la minimisation de la variation totale (TV). Cependant, la fonction TV n'est pas différentiable, ce qui complique l'utilisation des méthodes de descente de gradient. Pour remédier à ce problème, nous proposons une approximation différentiable de la norme TV en utilisant une fonction d'activation ϕ_α .

La fonction ϕ_α est définie par : $\phi_\alpha := \begin{cases} \mathbb{R} & \longrightarrow \mathbb{R}^+ \\ s & \longmapsto |s| - \alpha \ln\left(\frac{\alpha+|s|}{\alpha}\right) \end{cases} \quad \alpha > 0.$

Afin d'utiliser ϕ_α comme fonction d'activation, nous devons répondre aux questions suivantes :

1. Montrer que ϕ_α est de classe C^2 pour $\alpha > 0$

Pour montrer que ϕ_α est de classe C^2, nous devons vérifier que ϕ_α est deux fois continûment différentiable.

- Calcul de la première dérivée ϕ'_α :

Pour s > 0, $\phi'\alpha(s) = \frac{d}{ds}(s - \alpha\, ln(1 + \frac{s}{\alpha})) = 1 - \alpha \cdot \frac{1}{1+\frac{s}{\alpha}} \cdot \frac{1}{\alpha} = 1 - \frac{\alpha}{\alpha + s}$

Pour s < 0, $\phi'\alpha(s) = \frac{d}{ds}\left(-s - \alpha\, ln\left(1 + \frac{-s}{\alpha}\right)\right) = -1 + \alpha \cdot \frac{1}{1+\frac{-s}{\alpha}} \cdot \frac{1}{\alpha} = -1 + \frac{\alpha}{\alpha - s}$

Pour s = 0, par continuité, $\phi'\alpha(0) = \lim_{s\to 0}\left(1 - \frac{\alpha}{\alpha+|s|}\right) = 0$

Donc en combinant nos résultats, nous obtenons :

$\phi'\alpha(s) = sgn(s)\ (\ 1 - \frac{\alpha}{\alpha + |s|}\ )$ avec $sgn(s)$ le signe de la fonction s.

- Calcul de la seconde dérivée ϕ''_α :

Pour s > 0, $\phi''\alpha(s) = \frac{d}{ds}(1 - \frac{\alpha}{\alpha + s}) = 0 + \frac{\alpha}{(\alpha + s)^2}$

Pour s < 0, $\phi''\alpha(s) = \frac{d}{ds}(-1 + \frac{\alpha}{\alpha - s}) = 0 - \frac{\alpha}{(\alpha - s)^2}$

Pour s = 0, par continuité, $\phi''\alpha(0) = \lim_{s\to 0}\left(\frac{\alpha}{(\alpha + |s|)^2}\right) = \frac{\alpha}{\alpha^2} = \frac{1}{\alpha}$

La continuité de $\phi''\alpha(s)$ peut être vérifiée par inspection de ses valeurs limites, ce qui montre que ϕ_α est bien de classe C^2 pour $\alpha > 0$.

2. *En déduire que $\phi'\alpha$ est une fonction lipschitzienne de constante k ≤ $\frac{1}{\alpha}$*

Une fonction est dite lipschitzienne s'il existe une constante L telle que, pour tous x, y,

$$| f(x) - f(y) | \leq L | x - y |$$

Pour prouver que $\phi'\alpha$ est lipschitzienne avec une constante k ≤ $\frac{1}{\alpha}$, nous devons montrer que la dérivée $\phi'\alpha$ a une variation bornée.

Nous avons trouvé en question 1 que la dérivée seconde est :

$$\phi''\alpha(s) = \begin{cases} \dfrac{\alpha}{(\alpha + s)^2} & si\ s > 0 \\ -\dfrac{\alpha}{(\alpha - s)^2} & si\ s > 0 \\ \dfrac{1}{\alpha} & si\ s = 0 \end{cases}$$

Il est clair que pour tout $s \neq 0$, $\phi''\alpha(s) = \frac{\alpha}{(\alpha + |s|)^2}$ est maximal lorsque $|s|$ est minimal, soit pour $|s| = 0$.

Donc, $|\phi''\alpha(s)| \leq \frac{1}{\alpha}$. Cela signifie que $\phi'\alpha$ est une fonction lipschitzienne avec une constante k ≤ $\frac{1}{\alpha}$, car la dérivée seconde $\phi''\alpha(s)$ est toujours inférieure ou égale à $\frac{1}{\alpha}$ en valeur absolue.

3. *Vérifier que $\phi\alpha$ est convexe*

Pour tout $s \neq 0$ et $\alpha > 0$, $\phi''\alpha(s) = \frac{\alpha}{(\alpha + |s|)^2} > 0$ comme quotient de deux termes positifs.

Pour tout $s = 0$ et $\alpha > 0$, $\phi''\alpha(s) = \frac{1}{\alpha} > 0$

Puisque pour $\phi''\alpha(s) > 0$ pour tous $s \in R$, alors on en déduit que ϕ_α est convexe sur R.

Dans un second temps, nous avons réalisé une fonction Python « computePhi » qui calcule ϕα et trace la fonction ϕ_α pour les valeurs de $\alpha \in \{0.25, 0.5, 1, 1.25, 1.5, 2\}$, afin de vérifier également que $\phi_\alpha(s)$ approche de manière différentiable $|s|$ plus α est petit.

```
1 usage
def computePhi(s, alpha):
    s = np.asarray(s)
    abs_s = np.abs(s)
    phi_alpha = abs_s - alpha * np.log((alpha + abs_s) / alpha)
    return phi_alpha, abs_s


# Définir les valeurs de s et alpha
s = np.linspace(-5, stop: 5, num: 100)
alphas = [0.05, 0.1, 0.25, 0.5, 1, 1.25, 1.5, 2]

# Tracer les courbes pour les valeurs de alpha
plt.figure(figsize=(12, 6))
for alpha in alphas:
    phi_alpha1, abs_s = computePhi(s, alpha)
    plt.plot( *args: s, phi_alpha1, label=f'φ_{alpha}(s)')

plt.plot( *args: s, abs_s, label=f'|s|', linestyle='--')

plt.title('Fonctions φ(s) pour différentes valeurs de α')
plt.xlabel('s')
plt.ylabel('Valeur de φ_α(s) et |s|')
plt.axhline( y: 0, color='grey', linewidth=0.5)
plt.axvline( x: 0, color='grey', linewidth=0.5)
plt.legend()
plt.grid(True)
plt.savefig("fonctions_phi_alpha.png")
plt.show()
```

Figure 38 : Fonction python de calcul de ϕ

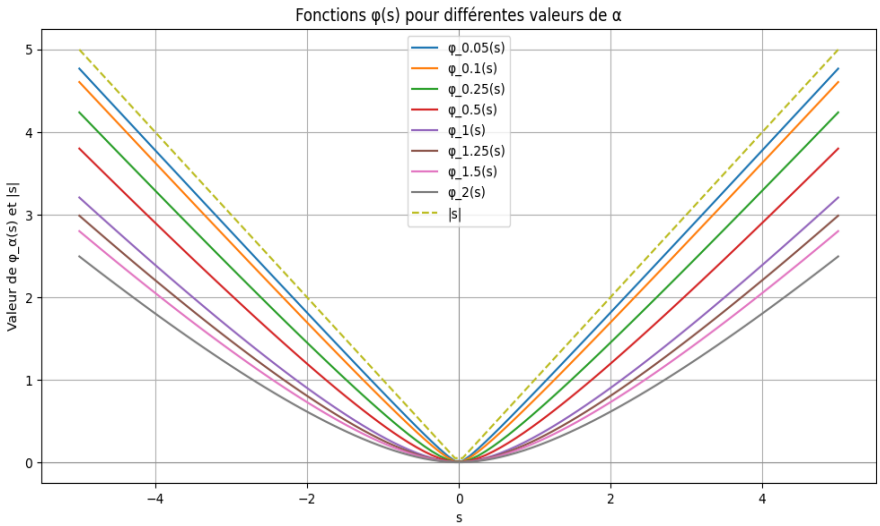


Figure 39 : Graphique des résultats de nos $\phi_\alpha(s)$

Le graphique obtenu montre les différentes courbes de ϕα(s) pour plusieurs valeurs de α comparées à la fonction |s|. On peut observer que pour des valeurs plus petites de α, la courbe ϕα(s) se rapproche davantage de la fonction |s|, confirmant que l'approximation est meilleure lorsque α est petit.

Nous allons maintenant utiliser ces fonctions dans la suite de notre code pour résoudre le problème d'optimisation et appliquer les techniques de débruitage sur des images :

```python
def phi_alpha(s, alpha):
    return np.abs(s) - alpha * np.log((alpha + np.abs(s)) / alpha)

2 usages
def phi_prime(grad_u, alpha):
    return np.sign(grad_u) * (1 - alpha / (alpha + np.abs(grad_u)))
```

Figure 40 : Nos fonctions de calcul de $\phi_\alpha(s)$ et $\phi'_\alpha(s)$

Pour résoudre le problème de débruitage d'image en utilisant le modèle de Rudin-Osher-Fatemi (ROF) modifié avec la fonction d'activation différentiable ϕα, nous reformulons le problème d'optimisation en utilisant une nouvelle fonctionnelle en remplaçant le terme non différentiable par une variante approchée différentiable :

$$J(u) = \frac{1}{2} \parallel v - u \parallel^2 + \lambda \sum_{j=1}^{n} \sum_{i=1}^{n} (\phi_\alpha(\partial_x u_{i,j}) + \phi_\alpha(\partial_y u_{i,j}))$$

Où :

- v est l'image restaurée.
- u_0 est l'image bruitée initiale.
- λ est un paramètre de régularisation.
- $\partial_x u_{i,j}$ et $\partial_y u_{i,j}$ sont les dérivées partielles de u par rapport aux coordonnées x et y.
- ϕ_α est la fonction d'activation différentiable.

Le gradient est donné par :

$$\nabla J(u) = (u - v) - \lambda \, div(\phi'_\alpha(\partial_x u_{i,j})\partial x;\ \phi'_\alpha(\partial_y u_{i,j})\partial y)$$

```python
# Function to compute the objective (J)
1 usage
def compute_J(u_flat, v, lam, alpha):
    u2d = u_flat.reshape(v.shape)
    grad_u_x, grad_u_y = grad(u2d)
    regularization_term = np.sum(phi_alpha(grad_u_x, alpha)) + np.sum(phi_alpha(grad_u_y, alpha))
    return 0.5 * np.linalg.norm(v - u2d) ** 2 + lam * regularization_term

# Function to compute the gradient of J
7 usages
def compute_gradient_J(u_flat, v, lam, alpha):
    u2d = u_flat.reshape(v.shape)
    grad_u_x, grad_u_y = grad(u2d)
    phi_prime_x = phi_prime(grad_u_x, alpha)
    phi_prime_y = phi_prime(grad_u_y, alpha)
    div_phi_prime = div(phi_prime_x, phi_prime_y)
    return (u2d - v - lam * div_phi_prime).flatten()
```

Figure 41 : Nos nouvelles fonction de J et du gradient de J

Conformément aux instructions du sujet, nous avons implémenté la méthode de gradient à pas fixe pour résoudre notre problème. Les résultats obtenus seront comparés aux méthodes DFP et BFGS pour évaluer l'efficacité de chaque approche dans le débruitage d'images.

```python
def gradient_pas_fixe(u0, v, lam, alpha, tol=1e-6, max_iter=100):
    u = u0.copy()
    niter = 0

    for _ in range(max_iter):
        grad = compute_gradient_J(u, v, lam, alpha).flatten()
        grad_norm = np.linalg.norm(grad)
        if grad_norm < tol:
            break

        # Compute step size based on the norm of gradient
        step_size = 1e-3  # This could be a predefined constant or adaptive
        u -= step_size * grad
        niter += 1

    return u.reshape(v.shape), niter
```

Figure 42 : Notre fonction du gradient à pas fixe

Premièrement, nous allons tracer l'évolution de l'erreur quadratique moyenne entre l'image originale v et l'image débruité en fonction du paramètre λ et observé si nous trouvons un résultat de λ optimal similaire à celui trouvé en partie 1.

```python
lambdas = [0.01, 0.05, 0.1, 0.15, 0.18, 0.2, 0.3, 0.5, 0.6, 0.8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 18, 20]
rmse_values = []
for lambda_reg in lambdas:
    print(f"Lambda: {lambda_reg}")
    u_bfgs1, iter_bfgs = BFGS_optimization(u0, epsilon, nitermax, lam=lambda_reg)
    u_bfgs_image1 = u_bfgs1.reshape(v.shape)
    rmse = calculate_rmse(u_bfgs_image1, image_originale)
    rmse_values.append(rmse)

# Tracé de l'évolution du RMSE en fonction de λ
plt.plot( *args: lambdas, rmse_values, marker='o')
plt.xscale('log')
plt.xlabel('Lambda')
plt.ylabel('RMSE')
plt.title('Évolution du RMSE en fonction de Lambda (BFGS)')
plt.savefig("rmse_lambda_BFGS_2.png")
plt.show()
```

Figure 43 : Notre code de comparaison des RMSE en fonction de lambda

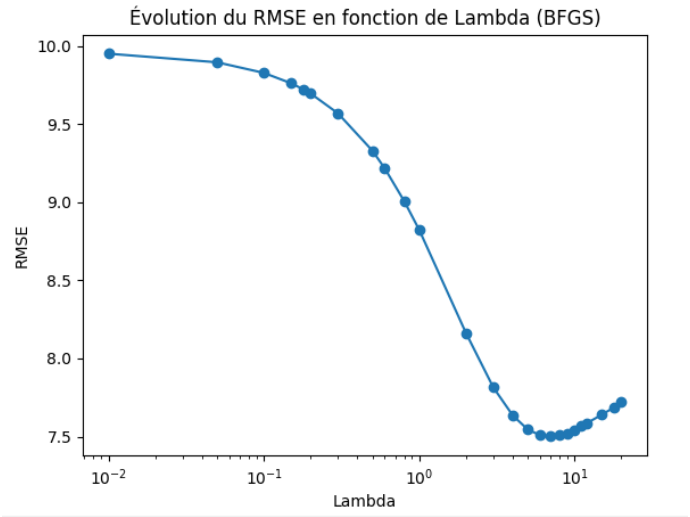


Figure 44 : Evolution du RMSE en fonction de la valeur de λ pour la méthode BFGS

L'analyse du graphique révèle que la valeur optimale de λ pour la méthode BFGS est proche de 6.5, ce qui est significativement différent de la valeur optimale observée précédemment dans la partie sur la méthode de régularisation de Tikhonov (0.1). Cette différence est probablement due aux modifications apportées à la fonctionnelle d'énergie avec l'utilisation de ϕα. Cette augmentation possible de λ apporte plusieurs avantages, tels qu'une meilleure efficacité de réduction du bruit, une réduction de l'overfitting et une stabilisation accrue.

Enfin, nous avons débruité notre images trois méthodes différentes qui sont GFGS, DFP et gradient à pas fixe avec un nombre d'itération max de 100, un alpha de 0.1 ou 0.01, $\lambda = 4$ ainsi qu'une condition d'arrêt $\varepsilon = 10^{-6}$.

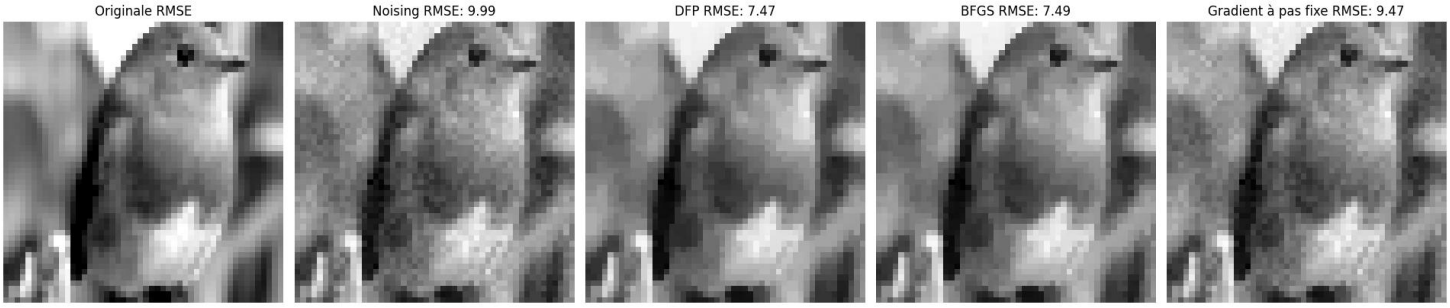


Figure 45 : Images par les différentes méthodes accompagnées de leur indice RMSE pour alpha = 0.1

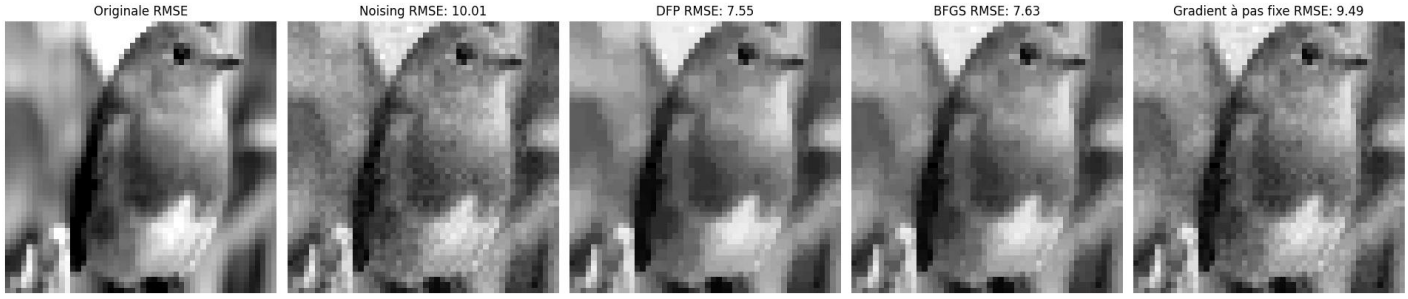


Figure 46 : Images par les différentes méthodes accompagnées de leur indice RMSE pour alpha = 0.01

Les images débruitées présentées ont été obtenues en utilisant deux valeurs différentes de α : 0.1 et 0.01. Voici une analyse de l'influence de α sur les résultats de débruitage, en tenant compte du fait que le bruit ajouté aux images est différent dans chaque cas, ce qui peut biaiser la comparaison directe.

Les méthodes DFP et BFGS montrent des performances similaires pour les deux valeurs de α, avec des RMSE légèrement plus bas pour $\alpha = 0{,}1$. La méthode de gradient à pas fixe est moins efficace que DFP et BFGS pour les deux valeurs de α, avec des RMSE significativement plus élevés.

Ainsi bien que théoriquement une valeur de α plus petite devrait mieux approcher |s|, dans la pratique, une valeur de $\alpha = 0.1$ s'avère être meilleure en termes de stabilité numérique et de qualité de débruitage. Cette valeur offre une régularisation efficace tout en assurant que les algorithmes d'optimisation convergent de manière stable, ce qui se traduit par un RMSE plus bas et une meilleure qualité visuelle des images débruitées.

VI. Inpainting ou désocclusion

L'inpainting, également connu sous le nom de désocclusion, est une technique de restauration d'image qui vise à reconstruire les parties manquantes ou endommagées d'une image. Cette technique est largement utilisée en traitement d'image pour réparer des zones dégradées, effacer des objets indésirables ou restaurer des parties d'une image. L'objectif est de remplir ces lacunes de manière cohérente avec le reste de l'image, en préservant les textures, les couleurs et les motifs environnants pour un résultat visuellement plausible.

Dans le contexte de l'inpainting, l'opérateur de masquage R joue un rôle crucial. Il définit les régions de l'image qui nécessitent une restauration. Le masque est appliqué sur l'image de manière que les valeurs des pixels dans les régions à restaurer soient isolées. Dans les régions non masquées, l'image reste inchangée.

$$R(u)(s) = \begin{cases} 0 & si\ s \in D_{mask} \\ U(s) & sinon \end{cases}$$

Voici comment nous pouvons modéliser notre nouveau problème :

$$J(u) = \frac{1}{2} \parallel Ru - u_0 \parallel^2 + \lambda \sum_{j=1}^{n} \sum_{i=1}^{n} (\phi_\alpha(\partial_x u_{i,j}) + \phi_\alpha(\partial_y u_{i,j}))$$

Cette fonction évalue la différence entre l'image restaurée et l'image originale dans les zones non masquées, en pénalisant les écarts par rapport à l'image originale tout en intégrant des termes de régularisation.

Le gradient est donné par :

$$\nabla J(u) = 2R^* v - 2R^* Ru + \lambda\, div(\frac{\phi_\alpha(\|\nabla u\|)}{\|\nabla u\|} \nabla u)$$

Avec R^* l'opérateur adjoint (inverse) qui sert à garantir que les valeurs des pixels restaurés sont cohérentes avec les valeurs connues de l'image originale tout en permettant la reconstruction des pixels manquants de manière harmonieuse avec le reste de l'image.

En effectuant des recherches sur internet, car n'étant pas clairement définit dans le sujet, j'en ai déduit que R^* pouvait s'écrire comme :

$$R^* = (R^{-1})^t$$

```python
# Fonction pour calculer J(u)
1 usage
def compute_J(u_flat, v, lam, alpha, mask):
    u2d = u_flat.reshape(v.shape)
    R_u = mask * u2d
    grad_u_x, grad_u_y = grad(u2d)
    reg_term = np.sum(phi_alpha(grad_u_x, alpha)) + np.sum(phi_alpha(grad_u_y, alpha))
    fit_term = 0.5 * np.linalg.norm(R_u - v) ** 2
    return fit_term + lam * reg_term

# Fonction pour calculer le gradient de J(u)
7 usages
def compute_gradient_J(u_flat, v, lam, alpha, mask):
    u2d = u_flat.reshape(v.shape)
    dx, dy = grad(u2d)
    norm_grad = np.sqrt(dx ** 2 + dy ** 2)
    R_u = mask * u2d

    lambda_ = 0.00001  # Choisissez un petit lambda
    regularized_mask = mask + lambda_ * np.eye(mask.shape[0])  # Ajoute un lambda sur la diagonale
    mask_inv = np.linalg.inv(regularized_mask)

    R_star_Ru = np.dot(mask_inv.T, R_u)
    R_star_v = np.dot(mask_inv.T, v)

    phi = grad_phi_alpha(norm_grad, alpha)
    phi_x = phi * dx
    phi_y = phi * dy
    div_phi = div(phi_x, phi_y)
    gradient = (2 * R_star_v - 2 * R_star_Ru + lam * div_phi).flatten()
    return gradient
```

Figure 47 : Nos nouvelles fonction de J et du gradient de J

Pour rendre le masque inversible et assurer la stabilité de l'algorithme d'optimisation, j'ai intégré un terme de régularisation λ = 0.00001 au masque, ce qui a permis de modifier efficacement la matrice du masque et de garantir son inversibilité.

Afin de tester nos divers méthodes adapter à l'impaiting, nous avons créée un masque noir de 4×4 pixel que nous avons appliquer à notre image d'oiseau.

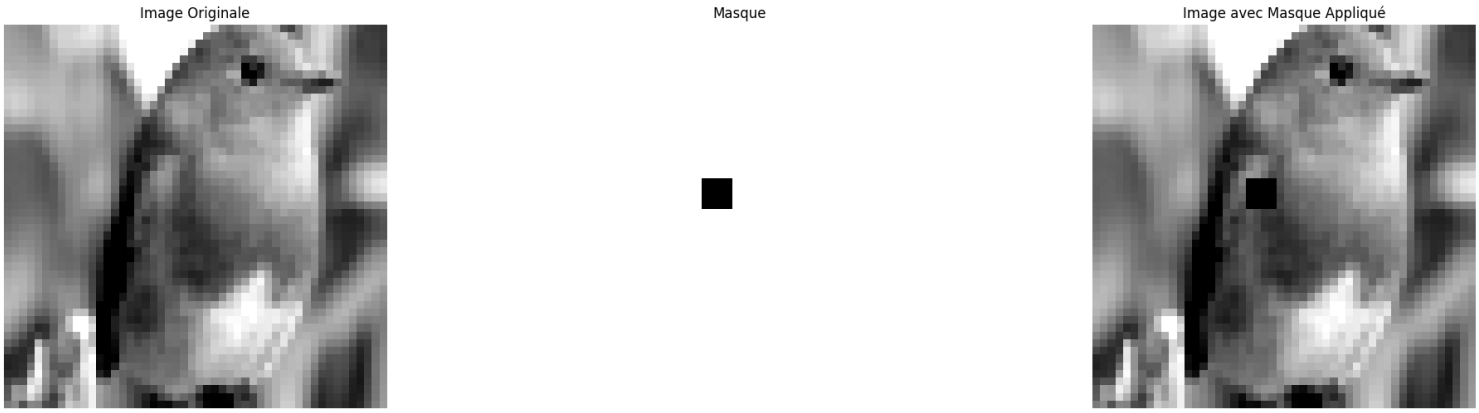


Figure 48 : Application du masque sur notre image d'oiseau

Enfin, nous avons lancer la reconstruction de notre images à l'aide des trois méthodes différentes de la partie précédentes qui sont GFGS (Broyden-Fletcher-Goldfarb-Shanno), DFP (Davidon-Fletcher-Powell) et gradient à pas fixe, que nous avons dû adapté, avec comme paramètres initiaux un nombre d'itération max de 100, un alpha de 0.1, $\lambda = 4$ ainsi qu'une condition d'arrêt $\varepsilon = 10^{-6}$.

Ayant dû faire un masque noir sur mon image orignal, je dois maintenant inverser mon filtre à blanc de sorte qu'il devienne un filtre sur fond noir et que je puisse le réintégrer sans problème sur mon image original masqué.

```python
# Créer un masque inversé pour la zone [20:24, 20:24]
inverse_mask_dfp = np.full_like(denoised_dfp, fill_value: 255)
inverse_mask_dfp[20:24, 20:24] = denoised_dfp[20:24, 20:24]
inverse_mask_dfp = inverse_mask_dfp%255

inverse_mask_bfgs = np.full_like(denoised_bfgs, fill_value: 255)
inverse_mask_bfgs[20:24, 20:24] = denoised_bfgs[20:24, 20:24]
inverse_mask_bfgs = inverse_mask_bfgs%255

inverse_mask_fixed_step = np.full_like(denoised_fixed_step, fill_value: 255)
inverse_mask_fixed_step[20:24, 20:24] = denoised_fixed_step[20:24, 20:24]
inverse_mask_fixed_step = inverse_mask_fixed_step%255

v_dfp = inverse_mask_dfp + v
v_bfgs = inverse_mask_bfgs + v
v_fixed_step = inverse_mask_fixed_step + v
```

Figure 49 : Mon processus d'inversion du masque pour le mettre sur fond noir

Voici les zones estimées à l'aide de nos trois méthodes :

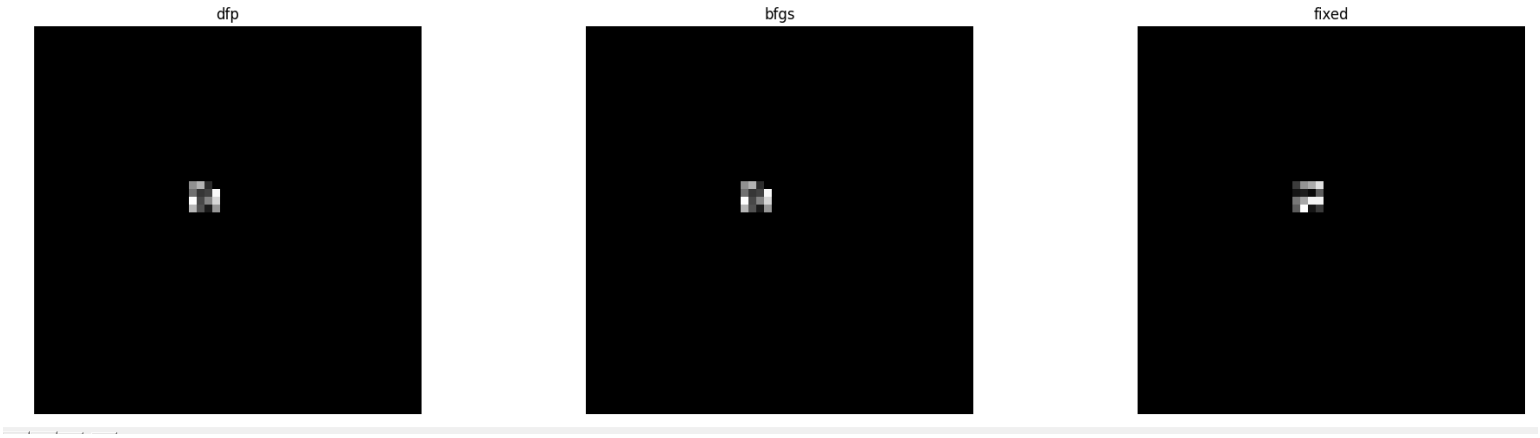


Figure 50 : Les zones masquées supposées par mes 3 méthodes

En superposant ces masque inversé sur l'image originale où les zones non traitées sont masquées en noir, il est possible d'intégrer harmonieusement les pixels restaurés dans leur contexte original sans affecter les autres parties de l'image. Voici ainsi mes images reconstitué pour chacune des méthodes :

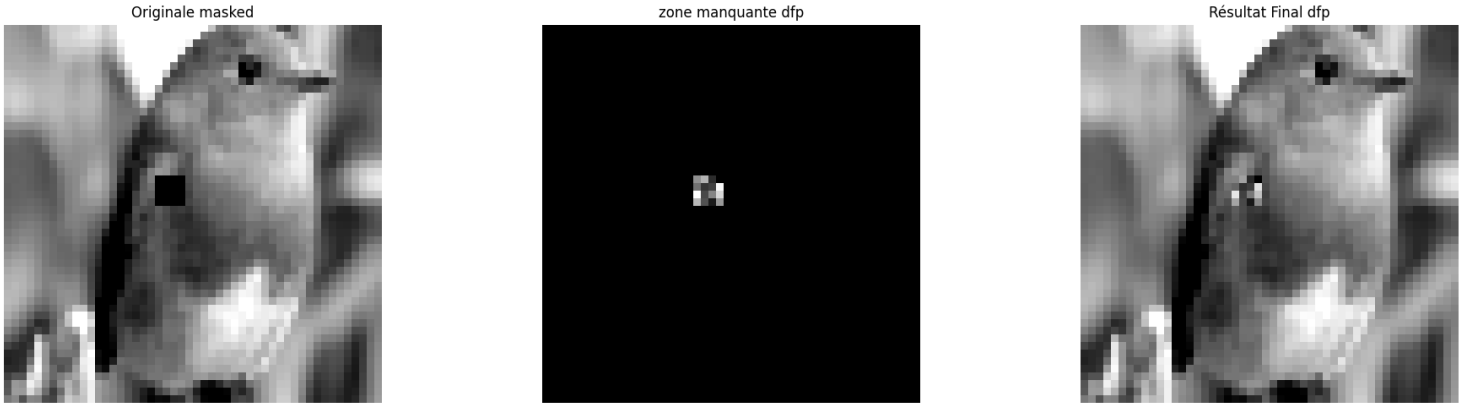


Figure 51 : Mon images finale reconstruite avec la méthode DFP

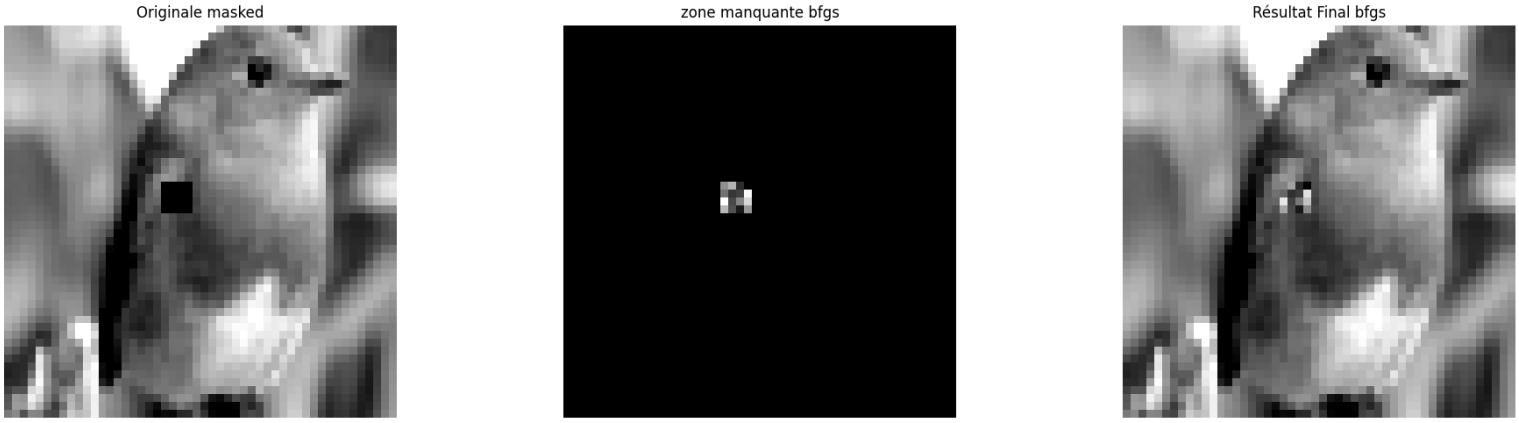


Figure 52 : Mon images finale reconstruite avec la méthode BFGS

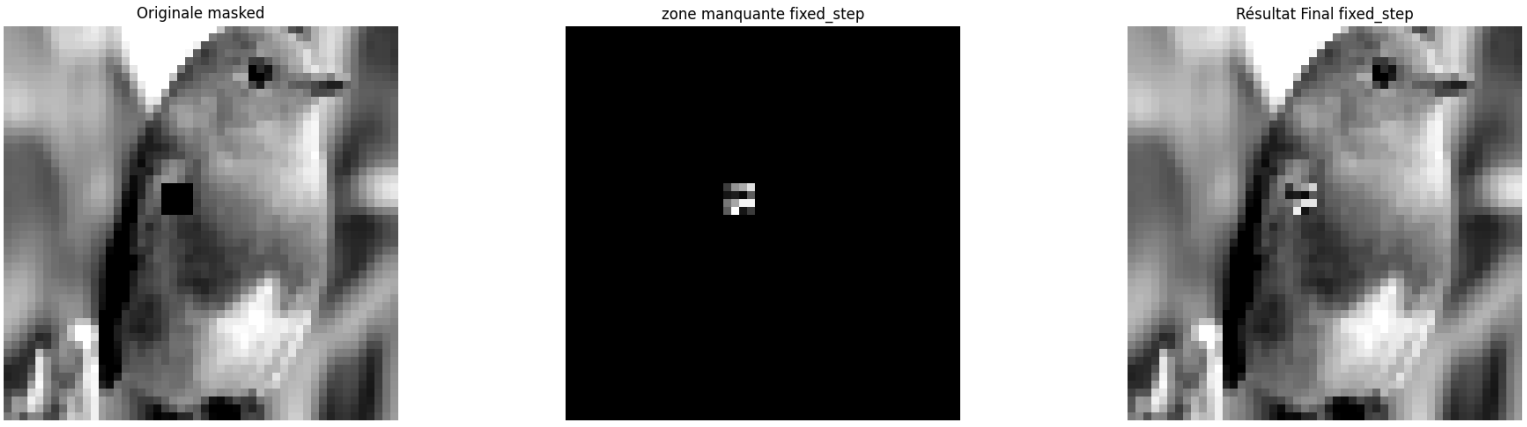


Figure 53 : Mon images finale reconstruite avec la méthode Gradient à pas fixe

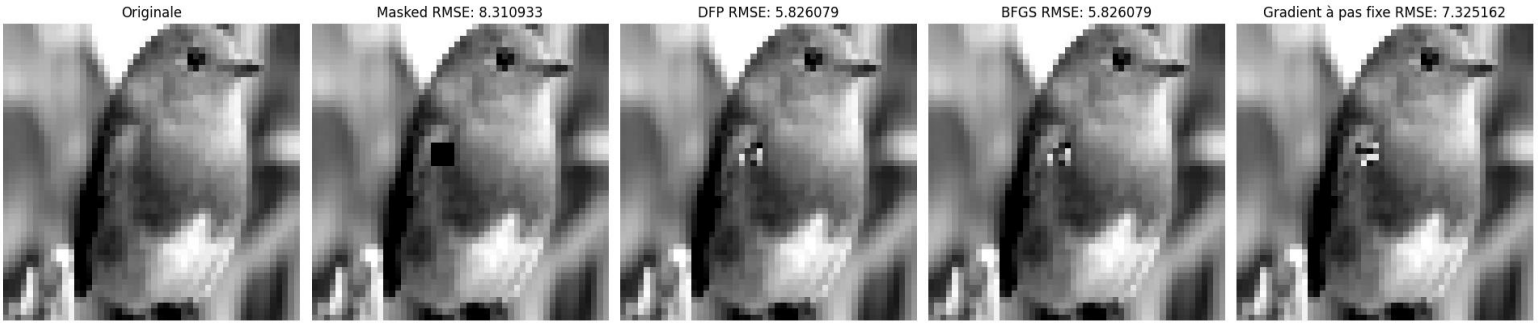


Figure 54 : Images Bilan des méthodes accompagnées de leur indice RMSE

L'image originale montre l'oiseau sans aucune altération, servant de base pour la comparaison. L'introduction du masque a significativement augmenté le RMSE à 8.310933, indiquant l'ampleur de la perte d'information due au masquage.

Les techniques DFP et BFGS ont abouti à des valeurs de RMSE identiques de 5.826079, montrant une restauration significative mais imparfaite de l'image. Bien que ces méthodes aient réussi à atténuer visiblement les effets du masquage, elles n'ont pas complètement rétabli l'image à son état d'origine, comme en témoignent les artefacts et les légères distorsions observées dans la zone restaurée.

Le gradient à pas fixe quant à lui a enregistré un RMSE plus élevé de 7.325162, indiquant une moins bonne performance par rapport aux méthodes DFP et BFGS. Les résultats montrent que, bien que le gradient à pas fixe ait partiellement restauré la zone affectée, la qualité de la restauration était inférieure, résultant en une intégration moins harmonieuse avec les zones non affectées.

VII. Conclusion

En conclusion, nous avons minutieusement examiné diverses méthodes d'optimisation pour la restauration d'images numériques en suivant un plan structuré. Nous avons commencé par rappeler les différentes méthodes de gradient, notamment le gradient à pas optimal et le gradient conjugué, en modélisant les trajectoires et en analysant les performances et la convergence, avec et sans pré-conditionnement.

Ensuite, nous avons exploré les méthodes Newtoniennes, y compris les méthodes de Newton classiques et quasi-Newtoniennes, telles que Broyden-Fletcher-Goldfarb-Shanno et Davidon-Fletcher-Powell, en évaluant leurs performances et leurs résultats. Par la suite, nous avons introduit les concepts de manipulation d' images, abordé les techniques de débruitage avec la régularisation de Thikonov et le modèle de Rudin-Osher-Fatemi, et fait de l'inpainting pour reconstructruires les partie manquantes d'une image.

Ces explorations ont mis en lumière les forces et les limites des approches actuelles dans la restauration d'images. Pour répondre aux objectifs de précision et d'efficacité, l'intégration des techniques d'apprentissage profond, comme les réseaux de neurones convolutifs (CNNs) et les réseaux antagonistes génératifs (GANs) est une piste d'amélioration futur à explorer. Ces technologies permettent d'apprendre des modèles complexes et d'améliorer significativement la qualité des images restaurées. De plus, une approche hybride, combinant différentes techniques d'optimisation, pourrait optimiser la convergence initiale et le raffinement final, offrant ainsi des résultats plus robustes et naturels.