

$\rho = 1$ II.2251 fonction qui détermine $\rho_{optimal}$ figure.caption.59
II.2351 code d'Implmentation des Courbes de Niveau et du Gradient de J avec ρ_{opt} figure.caption.60
 $\rho = 1$ II.2652 Courbe de ρ_{opt} figure.caption.61
 $\rho = \rho_{opt}$ II.2853 des Courbes de Niveau et du Gradient de J avec ρ_{opt} figure.caption.65
 $\rho = \rho_{opt}$ II.2954 Solution par méthode de Gradient Projecté selon n pour ρ_{opt} figure.caption.66
 $\rho = \rho_{opt}$ II.3054 Courbe de ρ_{opt} figure.caption.67
 η



ÉCOLE NATIONALE D'INGÉNIEURS DE TUNIS

Département Génie Industriel

Rapport

Mini-projet d'Optimisation

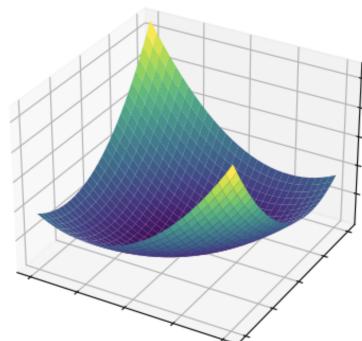
Elaboré par :

Balti Chaima

Lakhzouri Roukaya

Encadré par :

M. Moakher Maher



2^{eme} Année MIIndS

Année universitaire : 2023/2024

Remerciements

Nous tenons à exprimer notre gratitude envers toutes les personnes qui ont apporté leur soutien, leurs conseils ou leur participation, contribuant ainsi à la concrétisation de ce projet.

Nous souhaitons exprimer notre reconnaissance particulière envers M. Maher Moakher pour sa disponibilité et sa patience infaillibles, manifestées au moment où nous en avions le plus besoin.

Table des matières

Table des figures	4
I Optimisation sans contraintes	8
I.1 L'équation d'Euler-Lagrange associée au problème de minimisation	8
I.2 Solution de l'équation différentielle pour le cas $f(x) = 1$	10
I.3 Approximation Quadratique du Problème de Minimisation dans \mathbb{R}^n	10
I.4 Implémentation et Visualisation de la Fonctionnelle J_n	13
I.5 Analyse de la Symétrie de la Matrice A pour $n \geq 2$ et Validation Numérique	15
I.6 Solution théorique	16
I.7 La méthode du gradient à pas fixe	18
I.8 La méthode du gradient à pas optimal	20
I.8.1 Méthode du gradient à pas optimal	21
I.9 La méthode du gradient conjugué	23
I.10 Analyse Comparative des Méthodes de Minimisation Quadratique	25
I.10.1 Erreurs par rapport à ρ et à la dimension n	26
I.10.2 Temps d'Exécution en Fonction de ρ et de la Dimension n .	30
I.10.3 Resultats	33
I.11 Conclusion	33
II Optimisation sous contraintes	34
II.1 Discrétisation et Approche pour le Problème de Minimisation sous Contrainte	34
II.2 Unicité de solution :	37
II.3 Implémentation des fonctionnelles J_n et ∇J_n	37
II.4 Obtention de la Solution Théorique avec <code>scipy.optimize.minimize</code>	38
II.5 Les conditions de KKT	39
II.6 Influence de variation de taille de Matrice A	41
II.7 Détermination de l'ensemble K_n	43
II.8 Généralisation si on a une Projection sur un parallélépipède	44
II.9 Implémentation de code de l'algorithme de Gradient Projecté	44

II.10 Influence de variation de taille de Matrice A	46
II.11 Amélioration de l'algorithme du gradient projeté par la détermination du pas optimal	50
II.12 Changement de fonction $f(x)$	53
II.13 Implimentation de code de methode de pénalisation à pas fixe	55
II.14 Influence de η sur la solution obtenue par methode de pénalité	55
II.15 Implimentation de code de methode de pénalisation à pas optimal	58
II.15.1 Comparaison entre la méthode de pénalisation à pas fixe et à pas optimal	60
II.16 Relation entre méthode d'Uzawa et prbleme Dual	61
II.17 Convergence de méthode d'Uzawa	61
II.18 Influence de variation de ρ	63
Bibliographie	67

Table des figures

I.1	Script Python de Définition des Variables	13
I.2	Script Python de définition de la fonction J	13
I.3	Script Python de Visualisation 3D de la Fonctionnelle $J(u)$ pour $n = 2$ sur le Pavé $[10,10] \times [10,10]$	14
I.4	Visualisation 3D de la Fonctionnelle $J(u)$ pour $n = 2$	14
I.5	Script Python pour la Vérification de la Définie Positivité de la Matrice A pour diverses Dimensions	16
I.6	Script Python pour la resolution théorique	16
I.7	Comparaison entre la Solution Exacte et la Solution Théorique Approchée	17
I.8	Script Python de Définition de la Fonction pour la Méthode du Gradient à Pas Fixe	18
I.9	Solution de (P3) avec la Méthode du Gradient à Pas Fixe	18
I.10	Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas fixe	19
I.11	Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas fixe	19
I.12	Script Python de la fonction de la méthode de la section dorée	20
I.13	Script Python de la fonction de la méthode du gradient à pas optimal .	21
I.14	Script Python de la solution avec la méthode du gradient à pas optimal .	21
I.15	Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas optimal	22
I.16	Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas optimal	22
I.17	Script Python de la fonction de la méthode du gradient conjugué .	23
I.18	Script Python de la solution avec la méthode du gradient conjugué .	23
I.19	Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient conjugué	24
I.20	Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient conjugué	24
I.21	Script Python de comparaison des méthodes	25

I.22	Script Python pour tableaux de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.	26
I.23	Tableaux de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.	27
I.24	Script Python pour Graph de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.	28
I.25	Graph de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.	28
I.26	Script Python pour tableaux de comparaison des Temps d'Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.	30
I.27	tableaux de comparaison des Temps d'Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.	31
I.28	Script Python pour Graph de comparaison des Temps d'Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.	32
I.29	Graph de comparaison des Temps d'Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.	32
II.1	Implémentation des fonctionnelles J_n et ∇J_n	37
II.2	code de la solution theorique	38
II.3	solution de la solution theorique pour $f(x)=1$	38
II.4	Solution théorique pour $f(x) = \pi^2 \sin(\pi x)$	39
II.5	Illustration de Solution théorique pour $f(x) = 1$ et $f(x) = \pi^2 \sin(\pi x)$	39
II.6	Valeur de solution theorique en fonction de n	41
II.7	Graphe de Solution théorique en variant n	42
II.8	Graphe de Solution théorique et fonction obtacle g pour chaque n	42
II.9	Script projK.py	44
II.10	code de l'algorithme de Gradiant Projété pas fixe	45
II.11	code d'implimentation des Courbes de Niveau et du Gradient de J avec la méthode du Gradient Projoté	45
II.12	Les Courbes de Niveau et du Gradient de J	46
II.13	fonction permettant de determiner ρ	46
II.14	fonction de calcule de temps d'exécution et nombre d'itérations	46
II.15	Code permet de visulaiser les courbes de solution en fonction de n	47
II.16	Solution par methode de Gradiant Projoté selon n	48
II.17	Les Courbes de g et solution approché en variant n	48
II.18	Solution par methode de Gradiant Projoté selon n	49
II.19	Les Courbes de g et solution approché en variant n	49
II.20	Solution par methode de Gradiant Projoté selon n pour $\rho = 0.5$	50
II.21	Solution par methode de Gradiant Projoté selon n pour $\rho = 1$	50
II.22	fonction qui determine $\rho_{optimal}$	51

II.23 code d'Implimentation des Courbes de Niveau et du Gradient de J avec ρ_{opt}	51
II.24 des Courbes de Niveau et du Gradient de J avec ρ_{opt}	51
II.25 Solution par méthode de Gradiant Projoté selon n pour ρ_{opt}	52
II.26 Courbe de g et solution approche suivant n rt avec $\rho = \rho_{opt}$	52
II.27 code de $f(x)$	53
II.28 des Courbes de Niveau et du Gradient de J avec ρ_{opt}	53
II.29 Solution par méthode de Gradiant Projoté selon n pour ρ_{opt}	54
II.30 Courbe de g et solution approchée suivant n et avec $\rho = \rho_{opt}$	54
II.31 code de méthode de pénalisation	55
II.32 Code de tracer les courbes de solution en fonction de η	56
II.33 solution approché par méthde de pénalisation en fonction de η	56
II.34 Les courbes de solution en fonction de η	57
II.35 les courbes de solution approché , fonction g et solution theorique en fonction de η	57
II.36 Tableau de variation de l'erreur en fonction de η	58
II.37 Code de calcule de l'erreur pour chaque η	58
II.38 Code de l'algorithme de pénalisation à pas optimal	59
II.39 Solution et erreur en fonction de η	59
II.40 les courbes de solution approchée , fonction g et solution theorique en fonction de η	60
II.41 Tableaux comparatif de méthode de pénalité à pas fixe et à pas optimal	60
II.42 Code de l'algorithme d'Uzawa	63
II.43 Code de l'algorithme d'Uzawa et résultat obtenue	64
II.44 Comparaison entre les méthode d'Uzawa avec des diffrent ρ	65

Introduction : Enoncé du mini-projet

On se propose dans ce mini-projet de résoudre numériquement le problème de minimisation :

$$\min_{u \in K} J(u)$$

où

$$J(u) = \int_0^1 \frac{1}{2} u'(x)^2 - f(x)u(x) dx \quad (1)$$

$f \in L^2(]0, 1[)$ est une fonction donnée et K est un sous ensemble fermé et convexe de $H_0^1(]0, 1[)$ définit par :

$$H_0^1(]0, 1[) := \left\{ u \in H^1(]0, 1[) : u(0) = u(1) = 0 \right\}$$

Ce mini-projet représente ainsi une opportunité stimulante pour approfondir notre compréhension des méthodes numériques liées à la résolution de problèmes de minimisation. En s'attardant sur les deux scénarios *l'optimisation sans contrainte* et *l'optimisation sous contraintes*, nous chercherons à évaluer la performance et la robustesse de diverses méthodes, tout en développant une expertise pratique dans l'application de ces approches aux problèmes réels.

Chapitre I

Optimisation sans contraintes

Dans cette partie, on considère le cas où K est tout l'espace $H_0^1(]0, 1[)$, c'est-à-dire on considère le problème de minimisation sans contrainte :

$$\min_{u \in H_0^1(]0, 1[)} J(u) \quad (P1)$$

I.1 l'équation d'Euler-Lagrange associée au problème de minimisation

$]0, 1[$ est un ouvert borné régulier de \mathbb{R} de frontière assez régulière $\{0, 1\}$. Etant donné $f \in L^2(]0, 1[)$, Soit le problème de Dirichlet homogène suivant :

$$(P2) \quad \begin{cases} -u'' = f, & \text{dans } (]0, 1[) \\ u(0) = u(1) = 0 \end{cases}$$

Soit $v \in H_0^1(]0, 1[)$; On multiplie l'équation par v et on intègre sur $]0, 1[$:

$$-\int_0^1 u'' v dx = \int_0^1 f v dx, \quad \forall v \in H_0^1(]0, 1[)$$

Ensuite, on intègre par parties et on obtient

$$\int_0^1 u' v' dx = \int_0^1 f v dx, \quad \forall v \in H_0^1(]0, 1[) \quad (2)$$

On pose :

$$a(u, v) = \int_0^1 u'v' dx \text{ et } \ell(v) = \int_0^1 fv dx$$

D'où (2) s'écrit sous la forme :

$$a(u, v) = \ell(v), \quad \forall v \in H_0^1(]0, 1[)$$

Vérification des hypothèses de Lax-Milgram

Par linéarité de l'intégrale ; $a(., .)$ est une forme bilinéaire de $H_0^1(]0, 1[)$ et $\ell(.)$ est une forme linéaire de $H_0^1(]0, 1[)$

i) La continuité de $a(., .)$:

D'après Cauchy-Schwartz on a :

$$|a(u, v)| = \left| \int_0^1 u'v' dx \right| \leq \|u'\|_{H_0^1(]0, 1[)} \|v'\|_{H_0^1(]0, 1[)}$$

Et à cause de l'équivalence des normes, on obtient :

$$|a(u, v)| \leq \|u\|_{H_0^1(]0, 1[)} \|v\|_{H_0^1(]0, 1[)}$$

Donc $a(., .)$ est continue.

ii) La coercivité de $a(., .)$:

D'après l'Inégalité de Poincaré on a :

$$\begin{aligned} |a(u, u)| &= \left| \int_0^1 u'u' dx \right| = \left| \int_0^1 u'(x)^2 dx \right| \\ &\geq \int_0^1 |u'(x)|^2 dx = \|u'\|_{H_0^1(]0, 1[)}^2 \\ &\geq \left(\frac{1}{C_p}\right)^2 \|u\|_{H_0^1(]0, 1[)}^2 \end{aligned}$$

Donc $a(., .)$ est coercive.

iii) La continuité de $\ell(.)$:

D'après Cauchy-Schwartz on a :

$$|\ell(v)| = \left| \int_0^1 fv dx \right| \leq C \|v\|_{H_0^1(]0, 1[)}$$

avec $C = \|f\|_{H_0^1(]0, 1[)}$ une constante , on a donc $\ell(.)$ est continue.

D'après *le théorème de Lax-Milgram*, et puisque la forme bilinéaire $a(.,.)$ est clairement symétrique ; On a (P2) admet une unique solution $u \in H_0^1(]0, 1[)$ et il est équivalent au problème de minimisation :

$$\min_{u \in H_0^1(]0, 1[)} \left(\frac{1}{2} a(u, u) - \ell(u) \right)$$

or on a :

$$\frac{1}{2} a(u, u) - \ell(u) = \int_0^1 u'(x)^2 - f(x)u(x) dx = J(u)$$

Conclusion :

$$\min_{u \in H_0^1(]0, 1[)} J(u) \iff \begin{cases} \text{Trouver } u \text{ telle que} \\ -u'' = f, & \text{dans } (]0, 1[) \\ u(0) = u(1) = 0 \end{cases}$$

I.2 Solution de l'équation différentielle pour le cas $f(x) = 1$

On a $f(x) = 1$ d'où :

$$u''(x) = -1 \Rightarrow u(x) = -\frac{1}{2}x^2 + ax + b, \quad x \in]0, 1[$$

Avec a et b deux constantes tel que $u(0) = u(1) = 0$ d'où $\begin{cases} a = \frac{1}{2} \\ b = 0 \end{cases}$

et donc la solution u_e est :

$$u_e = \frac{1}{2}(-x^2 + x)$$

I.3 Approximation Quadratique du Problème de Minimisation dans \mathbb{R}^n

Pour trouver une solution approchée de (P1) on discrétise $[0, 1]$ en $n + 1$ sous intervalles égaux $[x_i, x_{i+1}]$, $i = 0, \dots, n$, où $x_i = ih$ avec $h = 1/(n + 1)$. Puis on approche la solution $u(x)$ par une fonction $u_h(x)$, continue, affine par morceaux ;

$$u_h(x) = \sum_{i=1}^n u_i \phi_i(x)$$

où

$$\phi_i(x) = \begin{cases} \frac{x-x_{i-1}}{h} & \text{si } x \in [x_{i-1}, x_i] \\ \frac{x_{i+1}-x}{h} & \text{si } x \in [x_i, x_{i+1}] \\ 0 & \text{si non} \end{cases}$$

et u_i sont des valeurs approchées de $u_e(x_i)$.

On peut recourir à la méthode des trapèzes pour estimer l'intégrale $J(u)$ de l'équation (1) par $J_n(u)$:

$$J_n(u) = \int_0^1 \frac{1}{2} \left(\sum_{i=1}^n u_i \phi'_i(x) \right)^2 - f(x) \sum_{i=1}^n u_i \phi_i(x) dx$$

avec

$$\phi'_i(x) = \begin{cases} \frac{1}{h} & \text{si } x \in [x_{i-1}; x_i] \\ \frac{-1}{h} & \text{si } x \in [x_i; x_{i+1}] \\ 0 & \text{sinon} \end{cases}$$

$$\begin{aligned} \int_0^1 \left(\sum_{i=1}^n u_i \phi'_i(x) \right)^2 dx &= \int_0^1 \left(\sum_{i=1}^n u_i \frac{1}{h} \mathbb{1}_{[x_{i-1}(x); x_i]} + u_i \frac{-1}{h} \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx \\ &= \frac{1}{h^2} \int_0^1 \left(\sum_{i=1}^n u_i \mathbb{1}_{[x_{i-1}; x_i]}(x) - u_i \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx \\ &= \frac{1}{h^2} \sum_{k=1}^{n-1} \left(\int_{x_k}^{x_{k+1}} \left(\sum_{i=1}^n u_i \mathbb{1}_{[x_{i-1}; x_i]}(x) - u_i \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx \right) \\ &= \frac{1}{h^2} \left(\int_0^{x_1} \left(\sum_{i=1}^n u_i \mathbb{1}_{[x_{i-1}; x_i]}(x) - u_i \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx \right. \\ &\quad + \int_{x_1}^{x_2} \left(\sum_{i=1}^n u_i \mathbb{1}_{[x_{i-1}; x_i]}(x) - u_i \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx + \dots \\ &\quad \left. + \int_{x_{n-1}}^1 \left(\sum_{i=1}^n u_i \mathbb{1}_{[x_{i-1}; x_i]}(x) - u_i \mathbb{1}_{[x_i; x_{i+1}]}(x) \right)^2 dx \right) \\ &= \frac{1}{h} [(u_1 - u_0)^2 + (u_1 - u_2)^2 + \dots + (u_{n-1} - u_n)^2 + (u_{n+1} - u_n)^2] \end{aligned}$$

or $u_0 = u(0) = 0$ et $u_{n+1} = u(1) = 0$ d'où

$$\int_0^1 \left(\sum_{i=1}^n u_i \phi'_i(x) \right)^2 dx = \frac{1}{h} \left[\sum_{i=1}^n 2u_i^2 - 2u_i u_{i-1} \right]$$

D'autre part :

$$\begin{aligned}
\int_0^1 f(x) \left(\sum_{i=1}^n u_i \phi_i(x) \right) dx &= \sum_{i=1}^n \int_0^1 f(x) u_i \phi_i(x) dx \\
&= \sum_{i=1}^n \int_0^1 \left(f(x) u_i \frac{x - x_{i-1}}{h} \mathbb{1}_{[x_{i-1}; x_i]} + f(x) u_i \frac{x_{i+1} - x}{h} \mathbb{1}_{[x_i; x_{i+1}]} \right) dx \\
&= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x) u_i \frac{x - x_{i-1}}{h} dx + \int_{x_i}^{x_{i+1}} f(x) u_i \frac{x_{i+1} - x}{h} dx
\end{aligned}$$

En utilisant la formule du trapèze, on obtient :

$$\begin{aligned}
\int_0^1 f(x) \left(\sum_{i=1}^n u_i \phi_i(x) \right) dx &= \sum_{i=1}^n (x_i - x_{i-1}) \left[\frac{f(x_i) u_i (x_i - x_{i-1}) + 0}{2h} \right. \\
&\quad \left. + (x_{i+1} - x_i) \frac{0 + f(x_i) u_i (x_{i+1} - x_i)}{2h} \right] dx \\
&= h \sum_{i=1}^n f(x_i) u_i , \quad \text{puisque } h = x_{i+1} - x_i
\end{aligned}$$

Donc

$$J_n(u) = \frac{1}{h} \sum_{i=1}^n 2u_i^2 - 2u_i u_{i-1} - h \sum_{i=1}^n f(x_i) u_i$$

Etant donné que h est une constante, la minimisation de J est équivalente à la minimisation de la fonction $\frac{J(u)}{h}$ d'où on peut réécrire ;

$$J_n(u) = \frac{1}{h^2} \sum_{i=1}^n 2u_i^2 - 2u_i u_{i-1} - \sum_{i=1}^n f(x_i) u_i$$

et par suite le problème de minimisation (1) peut être approximé par le problème de minimisation quadratique dans \mathbb{R}^n :

$$\min_{u \in \mathbb{R}^n} \frac{1}{2} \langle Au, u \rangle - \langle b, u \rangle \quad (P3)$$

Avec la matrice $A \in \mathbb{M}_n(\mathbb{R})$ et le vecteur $b \in \mathbb{R}^n$ sont donnés par :

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{bmatrix}, \quad \text{et} \quad \mathbf{b} = \begin{bmatrix} f(x_1) \\ \vdots \\ \vdots \\ \vdots \\ f(x_n) \end{bmatrix}$$

I.4 Implémentation et Visualisation de la Fonctionnelle J_n

```
#Definition des Variables:
#La fonction f:
def f(x):
    Y=1
    return Y
#Dimension de l'espace:
n=2
#Pas de subdivision:
h = 1/(n + 1)
#La matrice A:
A = (1/(h**2))*diags([-1*np.ones(n-1),2*np.ones(n),-1*np.ones(n-1)],[-1,0,1]).toarray()
#Subdivision de l'intervalle [0,1]:
x=np.ones((n+2,1))
for i in range(0,n+2):
    x[i]=i*h
#Le vecteur b:
b=np.ones((n,1))
for i in range(1,n):
    b[i]=f(x[i])
#Nombre d'or
phi=(1 + np.sqrt(5))/2
```

FIGURE I.1 – Script Python de Définition des Variables

Nous avons défini la fonction J de manière générale, puis spécifiquement pour le cas où $n = 2$, sous le nom J2_graph, afin de faciliter sa représentation graphique en Python.

```
#La fonction J(u):
def J(u):
    Au = np.dot(A, u)
    Au_u = np.dot(np.transpose(Au), u)
    b_u = np.dot(np.transpose(b), u)
    Y=(1/2)*Au_u - b_u
    return Y
#Cas ou n=2;pour faciliter le graph
def J2_graph(x,y):
    Au = np.array([A[0, 0]*x + A[0, 1]*y, A[1, 0]*x + A[1, 1]*y])
    Au_u = Au[0]*x + Au[1]*y
    b_u = b[0]*x + b[1]*y
    Y=(1/2)*Au_u - b_u
    return Y
```

FIGURE I.2 – Script Python de définition de la fonction J

```

x_vals = np.linspace(-10, 10, 30)
y_vals = np.linspace(-10, 10, 30)

x_grid, y_grid = np.meshgrid(x_vals, y_vals)
J_values = J2_graph(x_grid,y_grid)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.plot_surface(x_grid, y_grid, J_values, cmap='viridis')

ax.set_zlabel('J(u)')
ax.set_title('Representation de J(u)')

plt.show()

```

FIGURE I.3 – Script Python de Visualisation 3D de la Fonctionnelle $J(u)$ pour $n = 2$ sur le Pavé $[10,10] \times [10,10]$

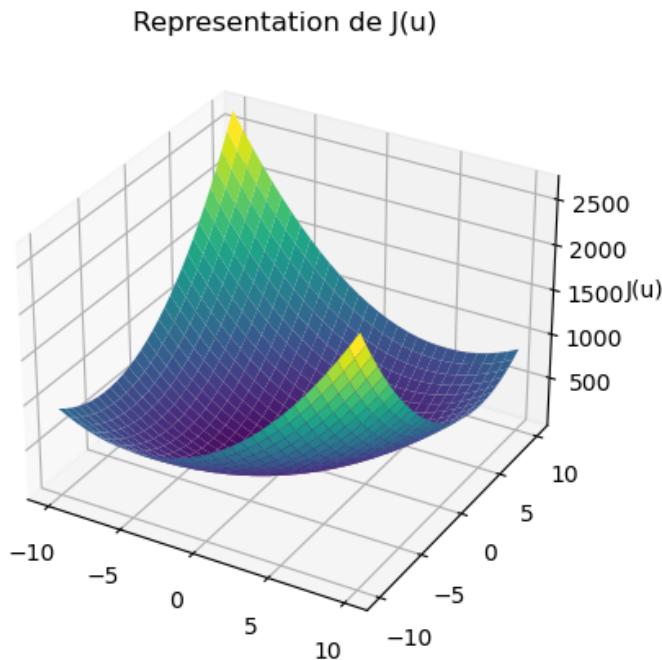


FIGURE I.4 – Visualisation 3D de la Fonctionnelle $J(u)$ pour $n = 2$

I.5 Analyse de la Symétrie de la Matrice A pour $n \geq 2$ et Validation Numérique

La matrice A est clairement symétrique. Il suffit de vérifier que A est définie positive.

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}$$

$$\text{Soient } X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$$

$$\begin{aligned} \langle AX, X \rangle &= \frac{1}{h^2} \sum_{i=0}^n 2x_i^2 - 2x_i x_{i-1} \\ &= \frac{1}{h^2} \left(\sum_{i=0}^n (x_{i+1} - x_i)^2 + x_0^2 + x_n^2 \right) \geq 0 \end{aligned}$$

d'où A est Symétrique

Supposons

$$\begin{aligned} \langle AX, X \rangle &= 0 \\ \iff x_0 &= x_{n+1} = 0 \text{ et } x_i = x_{i+1} \quad \forall i \in 1; n-1 \\ \iff x_i &= 0 \quad \forall i \in 0; n \\ \iff X &= 0 \end{aligned}$$

Donc A est symétrique définie positive $\forall n \geq 2$

Validation Numérique :

```
#Verification numerique : A dans Sn++(IR) (n=2,3..10)
print("{:<15} {:<15}".format("Dimension", "A est definie Positive"))

for k in range(2, 11):
    A = (1 / (1**2)) * diags([-1 * np.ones(k - 1), 2 * np.ones(k), -1 * np.ones(k - 1)], [-1, 0, 1]).toarray()

    # Check if A is positive definite
    is_positive_definite = (np.linalg.eigvals(A) > 0).all()

    # Print the results in a table
    print("{:<15} {:<15}".format(k, "Oui" if is_positive_definite else "Non"))

Dimension      A est definie Positive
2              Oui
3              Oui
4              Oui
5              Oui
6              Oui
7              Oui
8              Oui
9              Oui
10             Oui
```

FIGURE I.5 – Script Python pour la Vérification de la Définie Positivité de la Matrice A pour diverses Dimensions

I.6 Solution théorique

Puisque le problème quadratique (P3) ne comporte aucune contrainte, si u est la solution, alors

$$\nabla J_n(u) = 0$$

Cela implique que $Au - b = 0$. Par conséquent, U est la solution de l'équation

$$Au = b$$

En utilisant la fonction `numpy.linalg.solve()` en Python, nous pouvons résoudre cette équation comme suit :

```
#Calcule de La solution Theorique :
sol_theorique = np.linalg.solve(A, b)
print("Solution Theorique :", sol_theorique.flatten())
```

Solution Theorique : [0.11111111 0.11111111]

FIGURE I.6 – Script Python pour la resolution théorique

Ainsi, la solution théorique à ce problème quadratique (pour $n = 2$) est la suivante :

$$u_t = \begin{bmatrix} 0.11111111 \\ 0.11111111 \end{bmatrix} \approx \begin{bmatrix} \frac{1}{9} \\ \frac{1}{9} \end{bmatrix} \Rightarrow u_t(x) = u_1\phi_1(x) + u_2\phi_2(x)$$

$$\Rightarrow u_t(x) = \begin{cases} \frac{1}{9}\left(\frac{x-x_0}{h}\right) & \text{si } x \in [x_0, x_1] \\ \frac{1}{9}\left(\frac{1}{3h}\right) & \text{si } x \in [x_1, x_2] \\ \frac{1}{9}\left(\frac{x_3-x}{h}\right) & \text{si } x \in [x_2, x_3] \end{cases} \quad \text{Avec ; } X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{3} \\ \frac{2}{3} \\ 1 \end{bmatrix} \quad \text{et } h = \frac{1}{3}$$

$$\Rightarrow u_t(x) = \begin{cases} \frac{x}{3} & \text{si } x \in \left[0, \frac{1}{3}\right] \\ \frac{1}{9} & \text{si } x \in \left[\frac{1}{3}, \frac{2}{3}\right] \\ \frac{1-x}{3} & \text{si } x \in \left[\frac{2}{3}, 1\right] \end{cases}$$

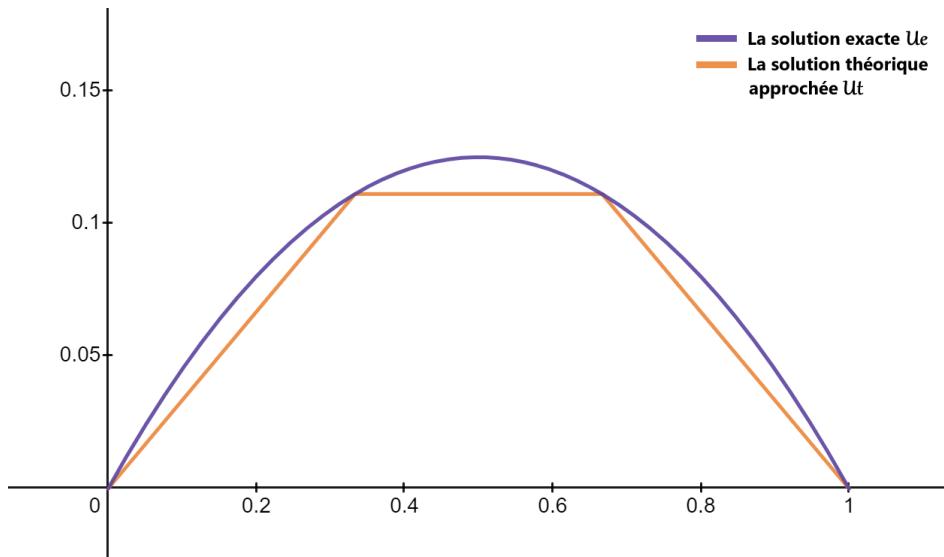


FIGURE I.7 – Comparaison entre la Solution Exacte et la Solution Théorique Approchée

I.7 La méthode du gradient à pas fixe

Dans la suite de ce rapport, nous explorerons les détails de l'application pratique de la méthode du gradient à pas fixe et examinerons ses performances dans ce problème d'optimisation.

```
def grad_pas_fixe(ue,rho=0.01,Tol=10**(-6),Max_iter=1000):
    k = 0
    U=[]
    U.append(ue)
    err_grad_pas_fixe = Tol
    #Boucle:
    while ((err_grad_pas_fixe >= Tol)and(k <=Max_iter)) :
        d=-1*grad_J(ue)
        up=ue
        ue=ue+rho*d
        U.append(ue)
        err_grad_pas_fixe=np.linalg.norm(up-ue)
        k = k + 1
    return ue,U
```

FIGURE I.8 – Script Python de Définition de la Fonction pour la Méthode du Gradient à Pas Fixe

```
#Initialisation :
u0= np.reshape(np.array([-1 , 2]),(2,1))
#solution avec La Méthode du gradient à pas fixe:
sol_grad_pas_fixe,U=grad_pas_fixe(u0)
#Erreur de la Méthode du gradient à pas fixe:
err_grad_pas_fixe=np.linalg.norm(sol_grad_pas_fixe-sol_theorique)
print("Solution avec la Méthode du gradient à pas fixe: ",sol_grad_pas_fixe.flatten())
print("Erreur de la Méthode du gradient à pas fixe: ",err_grad_pas_fixe)
```

Solution avec la Méthode du gradient à pas fixe: [0.11111801 0.11111801]
 Erreur de la Méthode du gradient à pas fixe: 9.753085736583595e-06

FIGURE I.9 – Solution de (P3) avec la Méthode du Gradient à Pas Fixe

$$u_{\text{pas_fixe}} = \begin{bmatrix} 0.11111801 \\ 0.11111801 \end{bmatrix} \quad \text{et} \quad \epsilon_{\text{pas_fixe}} \approx 10^{-5}$$

Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas fixe :

```

x_vals = np.arange(-10, 10, 1)
y_vals = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x_vals, y_vals)
Z=J2_graph(X,Y)

X_d,Y_d=grad_J2_graph(X,Y)
plt . contour (X ,Y , Z )
plt . quiver (X ,Y , X_d , Y_d , scale=1000)
gn=[h*i for i in range(1,n+1)]

u0= np.reshape(np.array([-1 , 2]),(2,1))
ue,grad_pas_fixe(u0)
Ux1=[r[0] for r in U]
Uy1=[r[1] for r in U]
plt.plot(Ux1,Uy1,label=" les lignes qui relient uk",color="red")
plt.legend()

plt.show()

```

FIGURE I.10 – Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas fixe

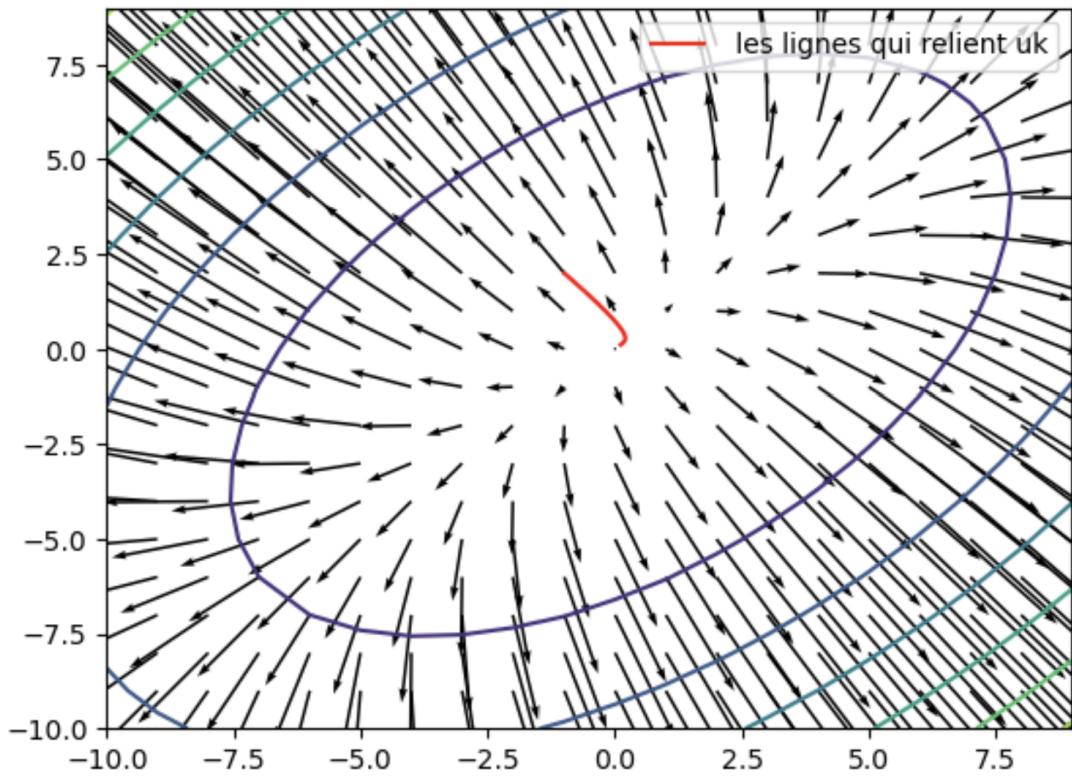


FIGURE I.11 – Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas fixe

I.8 La méthode du gradient à pas optimal

Dans cette section, nous explorons l'utilisation de la méthode de gradient de pas optimal pour minimiser (P3). Une étape clé de cette approche consiste à calculer le coefficient de pas optimal ρ_k , en utilisant la méthode de la section dorée, offrant une stratégie efficace pour converger rapidement vers la solution.

Méthode de la section dorée

```
def sec_doree(ue,de,a,b,Tol=10**(-6)):
    k = 0
    err_sec_doree = b-a
    #Boucle: Tant que err >= Tol :
    while (err_sec_doree >= Tol) :
        #Calculer : a'et b'
        a_prime=a+(b-a)/(phi**2)
        b_prime=a+(b-a)/(phi)
        #Calculer : Im_a_prime=J(u+a'*d) et Im_b_prime=J(u+b'*d)
        Im_a_prime=J(ue+a_prime*de)
        Im_b_prime=J(ue+b_prime*de)
        if Im_a_prime>Im_b_prime :
            a=a_prime
        if Im_b_prime>Im_a_prime :
            b=b_prime
        if Im_a_prime==Im_b_prime :
            a=a_prime
            b=b_prime
        err_sec_doree= b-a
        k = k + 1
    return (a+b)/2
```

FIGURE I.12 – Script Python de la fonction de la méthode de la section dorée

I.8.1 Méthode du gradient à pas optimal

```

def grad_pas_opt(ue,Tol=10**(-6),Max_iter=1000):
    k = 0
    U=[]
    U.append(ue)
    err_grad_pas_opt = Tol
    #Boucle: Tant que rk >=Tol et k <=Max_iter :
    while ((err_grad_pas_opt >= Tol)and(k <=Max_iter)) :
        up=ue #uk-1=up
        #Calculer la direction de descente :
        d=-1*grad_J(ue)
        #Calculer le pas optimal :
        rho=sec_doree(ue,d,-10,10)
        #Mettre à jour :
        ue=ue+rho*d
        U.append(ue)
        #Calculer l'erreur :
        err_grad_pas_opt=np.linalg.norm(up-ue)
        #Incrémenter le compteur :
        k = k + 1
    return ue,U

```

FIGURE I.13 – Script Python de la fonction de la méthode du gradient à pas optimal

```

#Méthode du gradient à pas optimal:
#Initialisation :
u0= np.reshape(np.array([-1 , 2]),(2,1))
sol_grad_pas_opt,L=grad_pas_opt(u0)
err_grad_pas_opt=np.linalg.norm(sol_grad_pas_opt-sol_theorique)
print("Solution avec la Méthode du gradient à pas optimal: ",sol_grad_pas_opt.flatten())
print("Erreur de la Méthode du gradient à pas optimal: ",err_grad_pas_opt)

Solution avec la Méthode du gradient à pas optimal: [0.1111111 0.11111113]
Erreure de la Méthode du gradient à pas optimal: 1.948803300230865e-08

```

FIGURE I.14 – Script Python de la solution avec la méthode du gradient à pas optimal

$$u_{pas_opt} = \begin{bmatrix} 0.1111111 \\ 0.11111113 \end{bmatrix} \quad \text{et} \quad \epsilon_{pas_opt} \approx 10^{-8}$$

Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas optimal :

```

x_vals = np.arange(-10, 10, 1)
y_vals = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x_vals, y_vals)
Z=J2_graph(X,Y)

X_d,Y_d=grad_J2_graph(X,Y)
plt . contour (X ,Y , Z )
plt . quiver (X ,Y , X_d , Y_d , scale=1000)
gn=[h*i for i in range(1,n+1)]

u0= np.reshape(np.array([-10 , -2.5]),(2,1))
ue,U=grad_pas_opt(u0)
Ux1=[r[0] for r in U]
Uy1=[r[1] for r in U]
plt.plot(Ux1,Uy1,label=" les lignes qui relient uk",color="blue")
plt.legend()

plt.show()

```

FIGURE I.15 – Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas optimal

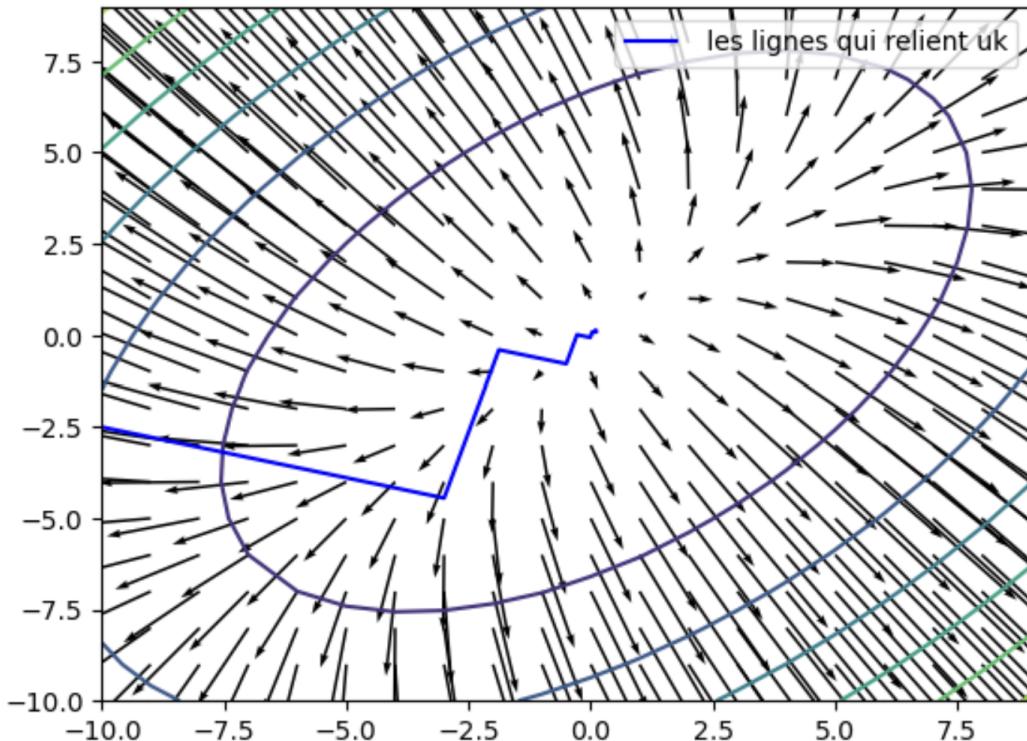


FIGURE I.16 – Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient à pas optimal

I.9 La méthode du gradient conjugué

Dans cette section, nous adopterons la méthode du gradient conjugué pour atteindre le minimum du notre problème quadratique (P3). Cette approche, connue pour sa convergence rapide, offre une solution efficace en exploitant des directions de recherche conjuguées.

```
def grad_conj(ue,Tol=10**(-6),Max_iter=1000):
    k = 0
    r = grad_J(ue)
    d=-1*r
    U=[]
    U.append(ue)
    err_grad_conj=Tol=10**(-6)
    #Boucle:
    while ((err_grad_conj >= Tol)and(k <=Max_iter)) :
        up=ue #uk-1=up
        #Calculer le pas optimal:
        rd=np.dot(np.transpose(d),r)
        Ad = np.dot(A, d)
        Ad_d = np.dot(np.transpose(Ad), d)
        rho=(-1*rd)/Ad_d
        #Mettre à jour:
        ue=ue+rho*d
        U.append(ue)
        #Calculer le résidu et la direction de descente :
        rp=r #rk-1=rp
        r=grad_J(ue)
        beta=(np.linalg.norm(r)**2)/(np.linalg.norm(rp)**2)
        d=-1*r+beta*d
        #Calculer l'erreur :
        err_grad_conj=np.linalg.norm(up-ue)
        #Incrémenter le compteur :
        k = k + 1
    return ue,U
```

FIGURE I.17 – Script Python de la fonction de la méthode du gradient conjugué

```
#Méthode du gradient conjugué:
#Initialisation :
u0= np.reshape(np.array([-1 , 2]),(2,1))
sol_grad_conj,Ugrad_conj=u0
err_grad_conj=np.linalg.norm(sol_grad_conj-sol_theorique)
print("Solution avec la Méthode du gradient conjugué: ",sol_grad_conj.flatten())
print("Erreur de la Méthode du gradient conjugué: ",err_grad_conj)

Solution avec la Méthode du gradient conjugué: [0.11111111 0.11111111]
Erreur de la Méthode du gradient conjugué: 3.1031676915590914e-17
```

FIGURE I.18 – Script Python de la solution avec la méthode du gradient conjugué

$$u_{pas_conj} = \begin{bmatrix} 0.11111111 \\ 0.11111111 \end{bmatrix} \quad \text{et} \quad \epsilon_{pas_conj} \approx 10^{-17}$$

Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient conjugué :

```

x_vals = np.arange(-10, 10, 1)
y_vals = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x_vals, y_vals)
Z=j2_graph(X,Y)

X_d,Y_d=grad_j2_graph(X,Y)
plt . contour (X ,Y , Z )
plt . quiver (X ,Y , X_d , Y_d , scale=1000)
gn=[h*i for i in range(1,n+1)]

u0=np.reshape(np.array([-4, -7.5]),(2,1))
ue,U=grad_conj(u0)
Ux1=[r[0] for r in U]
Uy1=[r[1] for r in U]
plt.plot(Ux1,Uy1,label=" les lignes qui relient uk",color="purple")
plt.legend()

plt.show()

```

FIGURE I.19 – Script python de la Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient conjugué

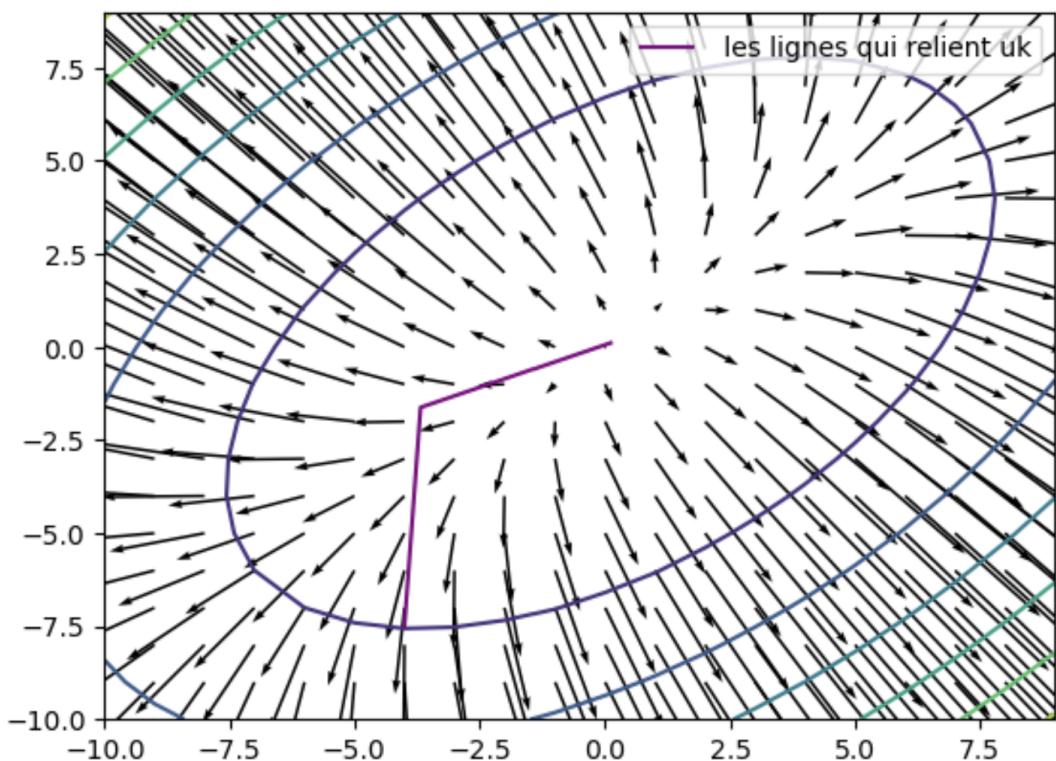


FIGURE I.20 – Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du gradient conjugué

I.10 Analyse Comparative des Méthodes de Minimisation Quadratique

Cette section¹ du rapport se concentre sur l'évaluation comparative des méthodes de minimisation quadratique. Les méthodes examinées comprennent le gradient à pas fixe, le gradient à pas optimal et le gradient conjugué.

En analysant les résultats en termes d'erreurs obtenues et de temps d'exécution associés, cette étude vise à apporter des éclairages significatifs sur la pertinence et l'efficacité de chaque méthode dans différentes configurations. En particulier, nous explorerons comment ces méthodes réagissent à des variations de pas ρ et à des dimensions n du problème .

L'objectif de cette analyse est d'orienter la sélection de la méthode la plus appropriée en fonction des caractéristiques spécifiques du problème de minimisation quadratique étudié.

```
#Comparaison des erreurs pour différentes dimensions et valeurs de rho:
dim=[2, 10, 20, 30, 50, 100]
rhos=[0.1,0.01,0.001,0.0001,0.00001]
r=0
err_grad_pas_fixe=np.zeros((len(dim),5))
err_grad_pas_opt=np.zeros((len(dim),5))
err_grad_conj=np.zeros((len(dim),5))
temps_grad_pas_fixe=np.zeros((len(dim),5))
temps_grad_pas_opt=np.zeros((len(dim),5))
temps_grad_conj=np.zeros((len(dim),5))
for rho in rhos:
    j=0
    for n in dim:
        u= np.reshape(np.random.choice([-2, 1,-1,2], size=n),(n,1))
        #Redefinition des variables:
        h = 1/(n + 1)
        A = (1/(h**2))*diags([-1*np.ones(n-1),2*np.ones(n),-1*np.ones(n-1)],[-1,0,1]).toarray()
        x=np.ones((n+2,1))
        for i in range(0,n+2):
            x[i]=i*h
        b=np.ones((n,1))
        for i in range(1,n):
            b[i]=f(x[i])
        start_time = time.time()
        sol_grad_pas_fixe,no=grad_pas_fixe(u,rho)
        end_time = time.time()
        temps_grad_pas_fixe[j,r]= end_time - start_time
        err_grad_pas_fixe[j,r]=np.linalg.norm(sol_grad_pas_fixe-np.linalg.solve(A, b))
        start_time = time.time()
        sol_grad_pas_opt,no=grad_pas_opt(u)
        end_time = time.time()
        temps_grad_pas_opt[j,r]= end_time - start_time
        err_grad_pas_opt[j,r]=np.linalg.norm(sol_grad_pas_opt-np.linalg.solve(A, b))
        start_time = time.time()
        sol_grad_conj,no=grad_conj(u)
        end_time = time.time()
        temps_grad_conj[j,r]= end_time - start_time
        err_grad_conj[j,r]=np.linalg.norm(sol_grad_conj-np.linalg.solve(A, b))
        j=j+1
    r=r+1
```

FIGURE I.21 – Script Python de comparaison des méthodes

1. C'est la réponse à la 6^{eme} question

I.10.1 Erreurs par rapport à ρ et à la dimension n

Dans toutes les dimensions n , il est notable que la méthode du gradient conjugué se distingue de manière significative en affichant l'erreur la plus faible, suivie de près par la méthode du gradient à pas optimal, et enfin par la méthode du gradient à pas fixe.

En ce qui concerne la méthode du gradient à pas fixe, il est observé que des dimensions plus élevées nécessitent des valeurs plus petites de ρ . Cette nécessité découle du fait que la convergence de la méthode du gradient à pas fixe est conditionnée par la contrainte que le pas ρ doit être $\rho \in \left[0, \frac{2}{\max_{\lambda \in S_P(A)} |\lambda|}\right]$.

En effet à mesure que la dimension n augmente, les valeurs propres augmentent également, restreignant ainsi la plage de valeurs possibles pour le pas ρ . Cette contrainte accrue dans des dimensions plus élevées peut rendre la méthode du gradient à pas fixe moins adaptée et nécessiter des ajustements spécifiques pour maintenir sa convergence.

En revanche, pour la méthode du gradient conjugué et la méthode du gradient à pas optimal, il est observé que l'erreur augmente avec l'augmentation de la dimension. Cette tendance peut être due à des complexités accrues et à des défis inhérents à la gestion de l'optimisation dans des espaces de dimension supérieure.

```
#Tableau de Comparaison des erreurs pour différentes dimensions et valeurs de rho:
# Données
dim = [2, 10, 20, 30, 50, 100]
rhos = [0.1, 0.01, 0.001, 0.0001, 0.00001]

# Initialiser un dictionnaire pour stocker les données
data = {'Dimension': dim}

# Boucle pour chaque valeur de rho
for r in range(len(rhos)):
    rho = rhos[r]

    print ("\n\n*****rho=",rho,"*****")
    data[f'Err_grad_pas_fixe'] = err_grad_pas_fixe[:, r]
    data[f'Err_grad_pas_opt'] = err_grad_pas_opt[:, r]
    data[f'Err_grad_conj'] = err_grad_conj[:, r]

# Créer un DataFrame à partir du dictionnaire
df = pd.DataFrame(data)

# Afficher Le tableau
print(df)
```

FIGURE I.22 – Script Python pour tableaux de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.

*****rho= 0.1 *****				
Dimension	Err_grad_pas_fixe	Err_grad_pas_opt	Err_grad_conj	
0	2	inf	7.425999e-09	1.387779e-17
1	10	NaN	1.599493e-05	4.649059e-16
2	20	NaN	6.576298e-05	5.551766e-16
3	30	NaN	1.223610e-02	2.296725e-15
4	50	NaN	2.408393e-01	2.629631e-15
5	100	NaN	2.628400e-01	2.910112e-14
*****rho= 0.01 *****				
Dimension	Err_grad_pas_fixe	Err_grad_pas_opt	Err_grad_conj	
0	2	0.00001	1.980104e-10	0.000000e+00
1	10	NaN	1.616431e-05	1.458819e-16
2	20	NaN	6.647372e-05	3.297983e-16
3	30	NaN	7.159256e-03	2.055442e-15
4	50	NaN	2.014583e-02	1.003538e-14
5	100	NaN	1.419145e-01	2.643921e-14
*****rho= 0.001 *****				
Dimension	Err_grad_pas_fixe	Err_grad_pas_opt	Err_grad_conj	
0	2	0.00011	1.646599e-07	5.003708e-17
1	10	0.00010	1.795146e-05	4.843825e-16
2	20	0.00010	6.657994e-05	4.746427e-16
3	30	NaN	4.061835e-03	2.686351e-15
4	50	NaN	1.923925e-01	3.766075e-15
5	100	NaN	2.639226e+00	1.109946e-14
*****rho= 0.0001 *****				
Dimension	Err_grad_pas_fixe	Err_grad_pas_opt	Err_grad_conj	
0	2	1.084649	1.439499e-10	0.000000e+00
1	10	0.456531	1.808919e-05	2.335636e-16
2	20	0.420778	6.393554e-05	4.082759e-16
3	30	0.044489	6.531377e-04	3.962882e-15
4	50	0.506199	2.000179e-01	1.181749e-15
5	100	NaN	1.344083e-01	7.368270e-14
*****rho= 1e-05 *****				
Dimension	Err_grad_pas_fixe	Err_grad_pas_opt	Err_grad_conj	
0	2	1.874322	4.740587e-07	6.938894e-17
1	10	2.092433	1.694212e-05	3.207691e-16
2	20	0.931264	6.608352e-05	1.134758e-15
3	30	0.450090	2.474422e-03	1.715645e-15
4	50	1.821027	2.781946e-01	3.152273e-15
5	100	2.482480	1.612631e+00	1.527163e-14

FIGURE I.23 – Tableaux de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.

```

#Graphs de Comparaison des erreurs pour différentes dimensions et valeurs de rho:
r=0
for rho in rhos:
    print("rho=",rho)
    # Tracé des courbes pour chaque valeur de rho
    plt.figure(figsize=(12, 8))
    plt.plot(dim, err_grad_pas_fixe[:,r], marker='o', label=f'Gradient à pas fixe')
    plt.plot(dim, err_grad_pas_opt[:,r], marker='o', label=f'Gradient à pas optimal')
    plt.plot(dim, err_grad_conj[:,r], marker='o', label=f'Gradient conjugué')

    r=r+1

    plt.yscale('log') # Échelle logarithmique pour une meilleure visualisation
    plt.xlabel('Dimension n')
    plt.ylabel('Erreur')
    plt.title(f'Comparaison des erreurs pour différentes dimensions avec,Rho={rho}')
    plt.legend()
    plt.grid(True)
    plt.show()

```

FIGURE I.24 – Script Python pour Graph de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.

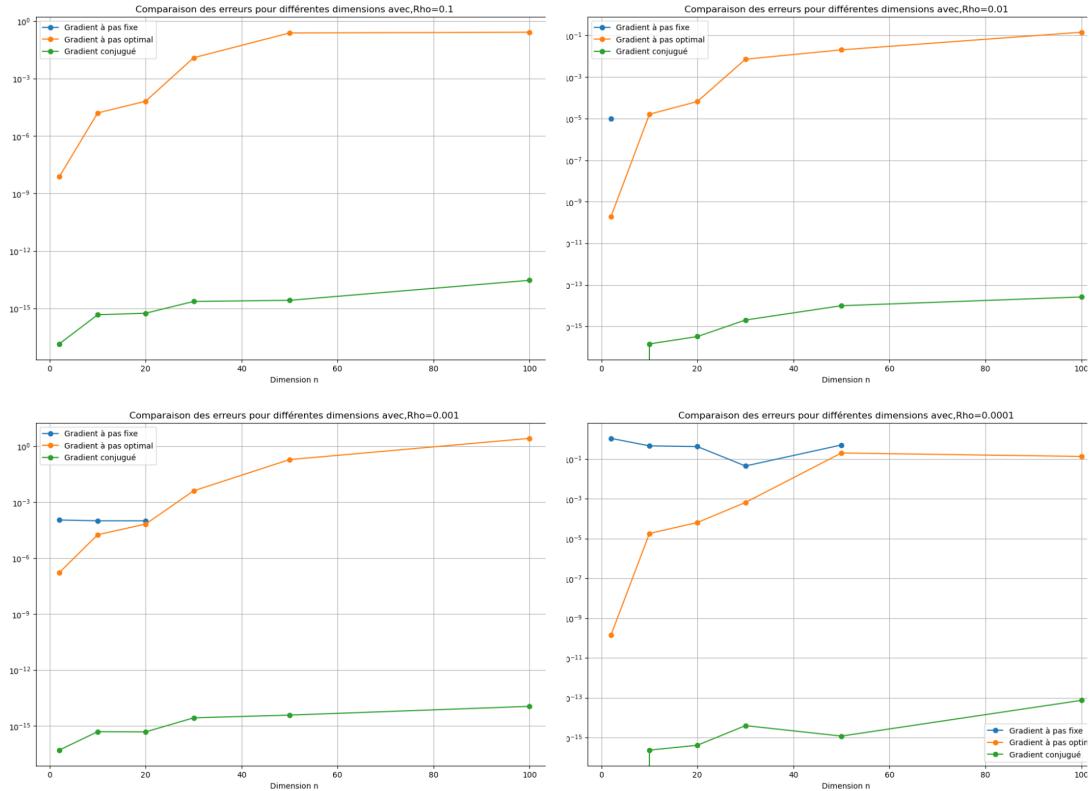
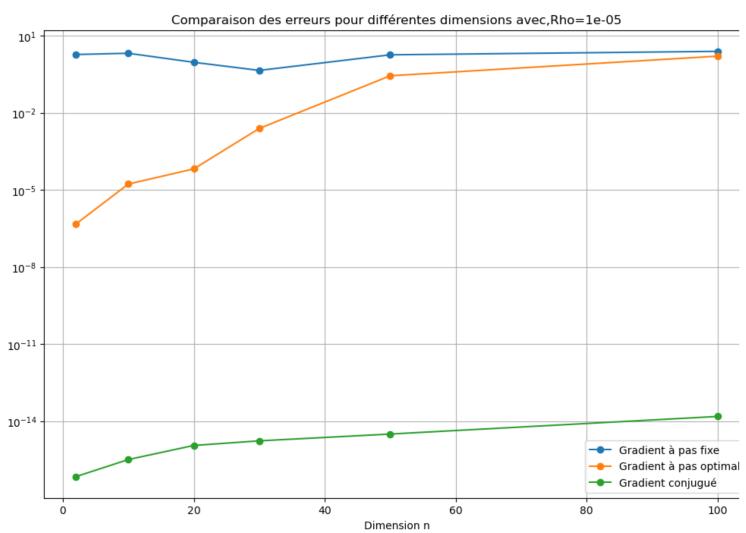


FIGURE I.25 – Graph de comparaison des erreurs en fonction des pas ρ et de la dimension n pour les trois méthodes.



I.10.2 Temps d'Exécution en Fonction de ρ et de la Dimension n

```

dim = [2,10, 20, 30, 50, 100]
rhos = [0.1, 0.01, 0.001, 0.0001]

temps_grad_pas_fixe = np.array(temps_grad_pas_fixe)
temps_grad_pas_opt = np.array(temps_grad_pas_opt)
temps_grad_conj = np.array(temps_grad_conj)

data = {'Dimension': dim}

for r in range(len(rhos)):
    rho = rhos[r]

    print("\n\n*****rho=", rho, "*****")
    data[f'Temps_grad_pas_fixe'] = temps_grad_pas_fixe[:, r]
    data[f'Temps_grad_pas_opt'] = temps_grad_pas_opt[:, r]
    data[f'Temps_grad_conj'] = temps_grad_conj[:, r]

df = pd.DataFrame(data)

print(df)

```

FIGURE I.26 – Script Python pour tableaux de comparaison des Temps d'Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.

D'après les données observées, il est notable que la méthode du gradient pas optimal présente le temps d'exécution le plus élevé, en particulier lorsque la dimension est plus élevée. Ensuite, vient la méthode du pas fixe, qui affiche également des temps d'exécution non négligeables. Cependant, la méthode la plus optimale en termes de temps d'exécution semble être la méthode du gradient conjugué.

D'un autre côté, la méthode du gradient conjugué se distingue par sa capacité à exploiter la structure du problème d'optimisation, ce qui la rend plus efficace, en particulier dans les espaces de grande dimension. Cette méthode s'adapte mieux aux caractéristiques du paysage de la fonction objectif, ce qui se traduit par des temps d'exécution plus courts.

*****rho= 0.1 *****				
Dimension	Temps_grad_pas_fixe	Temps_grad_pas_opt	Temps_grad_conj	
0	2	0.022228	0.015026	0.000000
1	10	0.004501	0.545840	0.001000
2	20	0.003999	1.530455	0.002016
3	30	0.004029	2.008089	0.002068
4	50	0.003053	1.911536	0.003007
5	100	0.002524	1.968176	0.008147

*****rho= 0.01 *****				
Dimension	Temps_grad_pas_fixe	Temps_grad_pas_opt	Temps_grad_conj	
0	2	0.005334	0.004598	0.001013
1	10	0.014744	0.589512	0.000999
2	20	0.007521	1.672577	0.001999
3	30	0.006158	2.078109	0.003000
4	50	0.005067	2.129132	0.004609
5	100	0.003934	2.303905	0.010088

*****rho= 0.001 *****				
Dimension	Temps_grad_pas_fixe	Temps_grad_pas_opt	Temps_grad_conj	
0	2	0.028496	0.019464	0.001209
1	10	0.024344	0.467285	0.002930
2	20	0.024559	1.861634	0.003000
3	30	0.021590	1.950840	0.003067
4	50	0.007387	1.903248	0.002999
5	100	0.005999	2.155258	0.007971

*****rho= 0.0001 *****				
Dimension	Temps_grad_pas_fixe	Temps_grad_pas_opt	Temps_grad_conj	
0	2	0.025931	0.004000	0.000000
1	10	0.026068	0.532023	0.000999
2	20	0.027150	1.817883	0.002270
3	30	0.026372	1.913124	0.003013
4	50	0.027767	2.197543	0.004115
5	100	0.014050	2.208733	0.008996

*****rho= 1e-05 *****				
Dimension	Temps_grad_pas_fixe	Temps_grad_pas_opt	Temps_grad_conj	
0	2	0.024998	0.044072	0.000000
1	10	0.025002	0.538281	0.001000
2	20	0.026842	1.602306	0.002000
3	30	0.026000	1.786734	0.002001
4	50	0.022010	1.918905	0.004430
5	100	0.028515	2.255432	0.009443

FIGURE I.27 – tableaux de comparaison des Temps d’Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.

```

# Créer des DataFrames pour chaque méthode
df_temps_grad_pas_fixe = pd.DataFrame(temps_grad_pas_fixe, index=dim, columns=rhos)
df_temps_grad_pas_opt = pd.DataFrame(temps_grad_pas_opt, index=dim, columns=rhos)
df_temps_grad_conj = pd.DataFrame(temps_grad_conj, index=dim, columns=rhos)

# Afficher un graphique pour chaque valeur de rho
for rho in rhos:
    # Créer une figure
    plt.figure(figsize=(12, 8))

    # Graphique pour la méthode du gradient à pas fixe
    plt.plot(dim, df_temps_grad_pas_fixe[rho], marker='o', label='Gradient à pas fixe')

    # Graphique pour la méthode du gradient à pas optimal
    plt.plot(dim, df_temps_grad_pas_opt[rho], marker='o', label='Gradient à pas optimal')

    # Graphique pour la méthode du gradient conjugué
    plt.plot(dim, df_temps_grad_conj[rho], marker='o', label='Gradient conjugué')

    # Ajouter des détails au graphique
    plt.yscale('log')
    plt.xlabel('Dimension n')
    plt.ylabel('Temps de calcul (s)')
    plt.title(f'Temps de calcul pour différentes méthodes (Rho={rho})')
    plt.legend()
    plt.grid(True)
    plt.show()

```

FIGURE I.28 – Script Python pour Graph de comparaison des Temps d’Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.

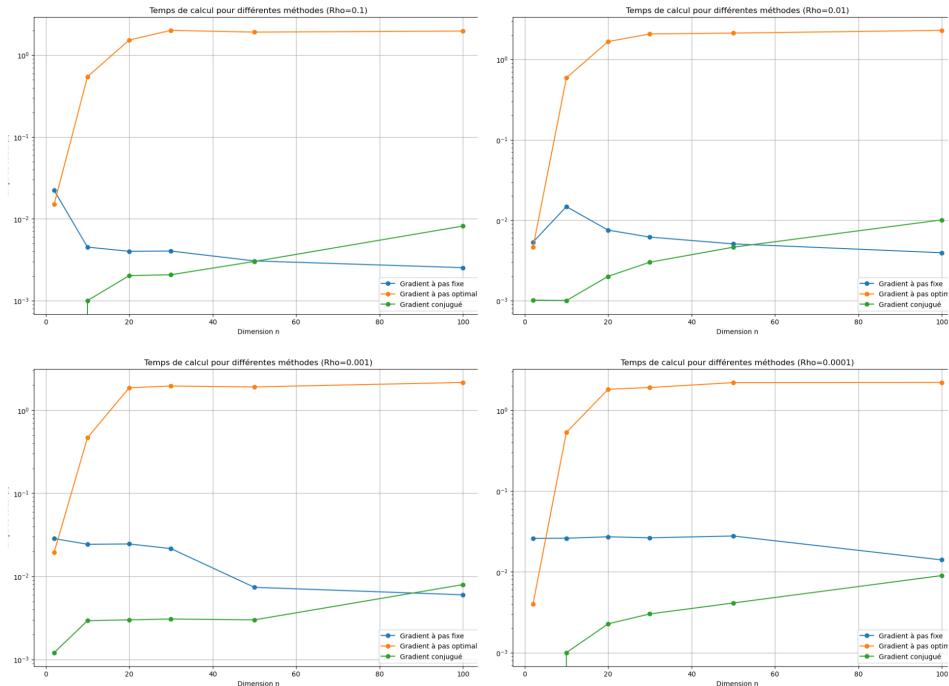
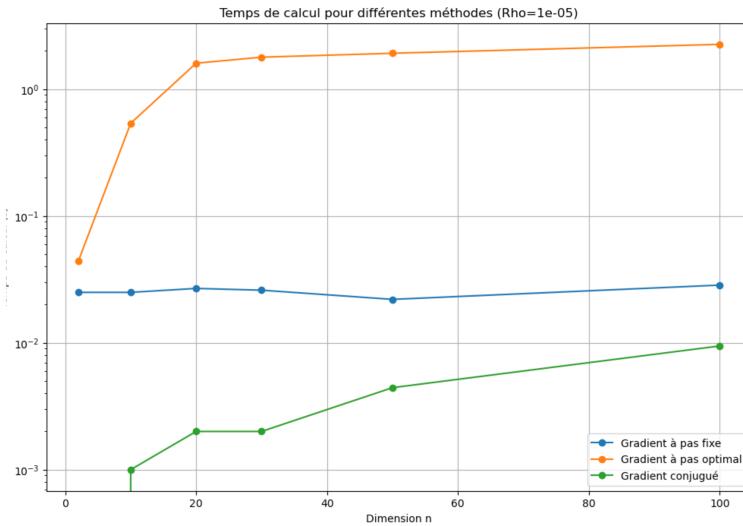


FIGURE I.29 – Graph de comparaison des Temps d’Exécution en fonction des pas ρ et de la dimension n pour les trois méthodes.



I.10.3 Résultats

En conclusion, les observations basées sur les données révèlent des tendances significatives quant à la performance des différentes méthodes d'optimisation dans des espaces de dimensions variées. La méthode du gradient conjugué se démarque comme la plus efficace, affichant des temps d'exécution plus courts et une erreur minimale, ce qui en fait un choix préférentiel, surtout dans des espaces de grande dimension.

En résumé, le choix de la méthode d'optimisation doit être judicieusement adapté à la dimension de l'espace dans lequel elle est appliquée. La méthode du gradient conjugué se positionne comme une option privilégiée, offrant une combinaison optimale de performances et d'efficacité.

I.11 Conclusion

Après avoir exploré en détail les mécanismes de l'optimisation sans contrainte, nous orientons désormais notre attention vers un domaine tout aussi crucial, mais caractérisé par des défis supplémentaires : l'optimisation sous contrainte. Cette transition marque une progression naturelle dans notre exploration des méthodes d'optimisation, élargissant notre champ d'investigation pour aborder des situations où des contraintes spécifiques pèsent sur la variable à optimiser.

Chapitre II

Optimisation sous contraintes

Nous envisageons maintenant la résolution numérique du problème de minimisation sous contrainte suivant :

$$\min_{u \in K} J(u)$$

où

$$K = \{u \in H_0^1(]0, 1[) \mid u(x) \geq g(x), \forall x \in [0, 1]\}$$

Discrétisation et résolution directe

II.1 Discrétisation et Approche pour le Problème de Minimisation sous Contrainte

on considère le problème de Dirichlet homogène suivant :

$$(P1) \quad \begin{cases} -u''(x) = f(x), & x \text{ dans } (]0, 1[) \\ u(0) = u(1) = 0 \end{cases}$$

On a la formulation variationnelle du problème :

$$(PV) \quad \begin{cases} \text{Trouver } u \text{ dans } K \\ a(u, v) = l(v) \quad \text{pour tout } v \in K \end{cases} \quad (\text{II.1.1})$$

avec

$$a(u, v) = \int_0^1 u'(x)v'(x) dx \quad l(v) = \int_0^1 f(x)v(x) dx$$

et comme on a $K \subset H_0^1(]0, 1[)$ Les hypothèses du lemme de Lax-Milgram restent valables, d'où le problème 2 est équivalent à un problème de minimisation :

$$(PM) \quad \begin{cases} \text{Trouver } u \text{ dans } K \\ J_1(u) < J_1(v) \quad \text{pour tout } v \in K \end{cases} \quad (\text{II.1.2})$$

avec

$$\begin{aligned} J_1(u) &= \frac{1}{2}a(u, u) - l(u) \\ &= \int_0^1 u'(x)^2 dx - \int_0^1 f(x)u(x) dx \\ &= \int_0^1 (u'(x)^2 - f(x)u(x)) dx \\ &= J(u) \end{aligned}$$

$$\min_{u \in K} J(u) \iff \begin{cases} \text{Trouver } u \text{ dans } K \\ a(u, v) = l(v) \quad \text{pour tout } v \in K \end{cases} \quad (\text{II.1.3})$$

Soit u une solution du problème de minimisation sous contrainte. Alors, u est également une solution du problème variationnel. Ainsi, $\forall v \in K$ on a

$$\int_0^1 u'(x)v'(x) dx = \int_0^1 f(x)v(x) dx$$

où $u(x) = u_h(x) = \sum_{i=1}^n u_i \phi_i(x)$ avec $u_i > g(x_i) \forall 1 \leq i \leq n$. En posant $v(x) = \phi_j(x) \forall 1 \leq j \leq n$, on obtient donc

$$\begin{aligned} &\Rightarrow \int_0^1 \sum_{i=1}^n u_i \phi'_i(x) \phi'_j(x) dx = \int_0^1 f(x) \phi_j(x) dx \quad \forall 1 \leq j \leq n \\ &\Rightarrow \sum_{i=1}^n \left(\int_0^1 u_i \phi'_i(x) \phi'_j(x) dx \right) = \int_0^1 f(x) \phi_j(x) dx \\ &\Rightarrow \sum_{i=1}^n A_{ij} u_i = b_j \end{aligned}$$

$\Rightarrow Au = b \quad A \in M_n(\mathbb{R}) \quad \text{et} \quad b \in M_{n,1}(\mathbb{R}) \quad \text{telle que}$

$$\begin{cases} A_{ij} = \int_0^1 \phi'_i(x) \phi'_j(x) dx \\ b_j = \int_0^1 f(x) \phi_j(x) dx \end{cases}$$

On détermine maintenant les coefficients A_{ij} et b_j de la manière suivante :

$$A_{ij} = \int_0^1 \phi'_i(x) \phi'_j(x) dx$$

$$\phi_i(x) = \begin{cases} \frac{x-x_{i-1}}{h} & \text{si } x \in [x_{i-1}, x_i], \\ \frac{x_{i+1}-x}{h} & \text{si } x \in [x_i, x_{i+1}], \\ 0 & \text{si non.} \end{cases} \Rightarrow \phi'_i(x) = \begin{cases} \frac{1}{h} & \text{si } x \in [x_{i-1}, x_i], \\ \frac{-1}{h} & \text{si } x \in [x_i, x_{i+1}], \\ 0 & \text{si non.} \end{cases}$$

$$A_{ii} = \int_0^1 (\phi'_i(x))^2 dx = \int_{x_{i-1}}^{x_{i+1}} \frac{1}{h^2} dx = \frac{1}{h^2} \int_{x_{i-1}}^{x_{i+1}} dx = \frac{1}{h^2} (x_{i+1} - x_{i-1}) = \frac{1}{h^2} 2h = \frac{2}{h}$$

Lorsque $j = i + 1$, les coefficients $A_{i,i+1}$ sont définis comme suit :

$$A_{i,i+1} = \int_0^1 (\phi'_i(x))(\phi'_{i+1}(x)) dx = \int_{x_{i-1}}^{x_{i+1}} \frac{-1}{h^2} dx = \frac{-1}{h^2} (x_{i+1} - x_i) = \frac{-1}{h}$$

$$A_{i,i-1} = \int_0^1 (\phi'_i(x))(\phi'_{i-1}(x)) dx = \int_{x_{i-1}}^{x_i} \frac{-1}{h^2} dx = \frac{-1}{h^2} (x_i - x_{i-1}) = \frac{-1}{h}$$

$$b_j = \int_0^1 f(x)\phi_j(x) dx = \int_{x_{j-1}}^{x_{j+1}} f(x)\phi_j(x) dx = \int_{x_{j-1}}^{x_j} f(x) \frac{(x - x_{j-1})}{h} dx + \int_{x_j}^{x_{j+1}} f(x) \frac{(x_{j+1} - x)}{h} dx$$

Par la formule du trapèze, cela devient :

$$\begin{aligned} b_j &= (x_i - x_{i-1}) \frac{f(x_i)(x_i - x_{i-1}) + 0}{2h} + (x_{i+1} - x_i) \frac{0 + f(x_i)(x_{i+1} - x_i)}{2h} \\ &= h \frac{f(x_i)}{2} + h \frac{f(x_i)}{2} \\ &= hf(x_i). \end{aligned}$$

d'où

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{bmatrix}, \quad \text{et} \quad \mathbf{b} = \begin{bmatrix} f(x_1) \\ \vdots \\ \vdots \\ \vdots \\ f(x_n) \end{bmatrix}$$

on a donc

$$\begin{cases} a(u, \phi_j) = \sum_{i=1}^n A_{ij} u_i \\ l(\phi_j) = \int_0^1 f(x)\phi_j(x) dx = b_j \end{cases}$$

$$\begin{aligned}
\text{On a } J_n(u) &= \frac{1}{2}a(u, u) - l(u) = \frac{1}{2}a\left(u, \sum_{j=1}^n u_j \phi_j\right) - l\left(\sum_{i=1}^n u_i \phi_i\right) \\
&= \frac{1}{2} \sum_{j=1}^n u_j a(u, \phi_j) - \sum_{i=1}^n u_i \varphi(\phi_j) \\
&= \frac{1}{2} \sum_{j=1}^n \left(\sum_{i=1}^n A_j u_j \right) u_j - \sum_{i=1}^n u_i b_i \\
&= \frac{1}{2} \langle Au, u \rangle - \langle b, u \rangle
\end{aligned}$$

et par suite le problème de minimisation (PM) peut être approximé par le problème de minimisation quadratique dans K_n :

$$(PM') \quad \min_{u \in K_n} J_n(u)$$

avec $K_n = \{v \in \mathbb{R}^n \mid v(x) \geq g(x), \forall x \in [0, 1]\}$ et $J_n(u) \frac{1}{2} \langle Au, u \rangle - \langle b, u \rangle$

II.2 Unicité de solution :

K_n est un ensemble convexe, fermé et non vide. De plus, A est symétrique définie positive, ce qui rend J_n coercive et strictement convexe. La fonction J_n est continue en tout point de K_n . Ainsi, le problème (PM') admet une unique solution.

II.3 Implémentation des fonctionnelles J_n et ∇J_n

```

def J(u, A, fx):
    Au = np.dot(A, u)
    Au_u = np.dot(np.transpose(Au), u)
    fx_u = np.dot(np.transpose(fx), u)
    Y=(1/2)*Au_u - fx_u
    return Y

def DJ(u, A, fx):
    Au = np.dot(A, u)
    Y=Au - fx
    return (Y)

```

FIGURE II.1 – Implémentation des fonctionnelles J_n et ∇J_n

II.4 Obtention de la Solution Théorique avec `scipy.optimize.minimize`

`scipy.optimize.minimize` est une fonction de la bibliothèque SciPy qui permet de trouver le minimum d'une fonction scalaire ou vectorielle d'une ou plusieurs variables. Dans cette partie, nous souhaitons résoudre le problème de minimisation (PM') à l'aide de cette fonction pour

$$n = 100, f(x) = 1, g(x) = \max\{1.5 - 20(x - 0.6)^2, 0\}.$$

```
# Definir les fonctions f, g et solution_minimisation_sous_contraintes
n = 100

f = lambda x : np.ones(np.shape(x))
g = lambda x : np.maximum(1.5-20*(x-0.6)**2, 0)

#Pas de subdivision:
h = 1/(n + 1)
#La matrice A:
A = (1/(h**2))*diags([-1*np.ones(n-1), 2*np.ones(n), -1*np.ones(n-1)], [-1, 0, 1]).toarray()

def solution_minimisation_sous_contraintes(f,g,J,DJ,n):
    u = np.zeros((n,))
    x = np.linspace(0,1,n+2)
    xv = x[1:-1]
    fv = np.array ([ f(e) for e in xv ])
    gv = np.array ([ g(k) for k in xv ])

    Jf = lambda u : J(u, A, fv)
    DJf = lambda u : DJ(u, A, fv)

    const = ({'type':'ineq', 'fun':(lambda u : (u-gv.reshape((n,)))) ,
              'jac':(lambda u : np.eye(np.size(u))) })

    res = minimize(Jf , u, method = 'SLSQP', jac = DJf , constraints = const ,
                  tol = 10**(-8) , options = {'disp': True , 'maxiter': 5000})
    return(res.x)
```

FIGURE II.2 – code de la solution théorique

```
Positive directional derivative for linesearch (Exit mode 8)
Current function value: 425.0036617613788
Iterations: 214
Function evaluations: 998
Gradient evaluations: 210
```

FIGURE II.3 – solution de la solution théorique pour $f(x)=1$

```

Optimization terminated successfully      (Exit mode 0)
Current function value: -177.60589779142458
Iterations: 115
Function evaluations: 513
Gradient evaluations: 115

```

FIGURE II.4 – Solution théorique pour $f(x) = \pi^2 \sin(\pi x)$

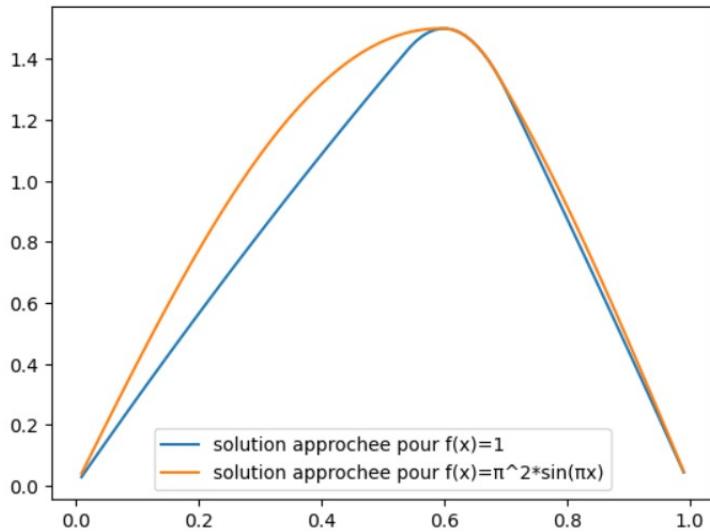


FIGURE II.5 – Illustration de Solution théorique pour $f(x) = 1$ et $f(x) = \pi^2 \sin(\pi x)$

II.5 Les conditions de KKT

$$\begin{aligned}
J(u) &= \int_0^1 \left(\frac{1}{2} u'(x)^2 - f(x)u(x) \right) dx \\
&= \int_0^1 \frac{-1}{2} u'(x)^2 dx + \int_0^1 u'(x)^2 dx - \int_0^1 f(x)u(x) dx
\end{aligned}$$

Par une intégration par partie on a

$$J(u) = \int_0^1 \frac{-1}{2} u'(x)^2 dx - \int_0^1 u''(x)u(x) dx - \int_0^1 f(x)u(x) dx$$

K est un ensemble d'inégalités affines. On suppose que u est un minimiseur de J sur K , et que J est différentiable en $u \in K$. On peut donc appliquer KKT :

$\exists \lambda \in \mathbb{R}$ tel que :

$$\begin{cases} \nabla J(u) + \lambda \nabla(g - u)(u) = 0 \\ \lambda \geq 0, g(x) - u(x) \leq 0 \\ \langle g(x) - u(x), \lambda \rangle = 0 \end{cases}$$

$$\begin{aligned}
h(\varepsilon) &= \int_0^1 \frac{1}{2} [(u + \varepsilon w)']^2 - f(u + \varepsilon w) dx \\
h'(0) &= \left. \frac{g(\varepsilon) - g(0)}{\varepsilon} \right|_{\varepsilon=0} \\
&= \int_0^1 \left(\frac{1}{2} (u')^2 + 2\varepsilon w' u' + (w')^2 - f \cdot u - \varepsilon f \cdot w \right) dx - \frac{1}{2} ((u')^2 - fu \\
&= \int_0^1 \left(u' w' + \frac{1}{2} \varepsilon (w')^2 - fw \right) dz \\
&= \int_0^1 (u' w' - fw) dx \\
&= \int_0^1 (u' w'' - fw) dx \\
&= \int_0^1 (u' - f) w dx
\end{aligned}$$

On a de plus $\nabla(g - u) = -1$

$$\text{d'où } \nabla J(u) + \lambda \nabla(g - u) = 0 \Leftrightarrow - \int_0^1 (u'' + f) w dx = \lambda \quad \forall w$$

on a d'après la troisième équation de KKT $(g(x) - f(x))\lambda = 0$

$$\begin{aligned}
1^{\text{ère}} \text{ cas : } \lambda &= 0 \Rightarrow - \int_0^1 (u'' + f) w dx = \lambda \quad \forall w \\
&\Rightarrow u'' + f = 0 \\
2^{\text{ème}} \text{ cas : } g(x) - f(x) &= 0
\end{aligned}$$

donc les conditions de KKT peut etre ecrite comme suit :

$$\left\{
\begin{array}{l}
(g(x) - f(x))(u'' + f) = 0 \\
g(x) - u(x) \leq 0 \\
\lambda \geq 0
\end{array}
\right. \Leftrightarrow \left\{
\begin{array}{l}
(g(x) - f(x))(u'' + f) = 0 \\
g(x) - u(x) \leq 0 \\
u''(x) + f(x) \leq 0
\end{array}
\right.$$

car $\lambda = - \int_0^1 (u'' + f) w dx \quad \forall w$ don si $\lambda \geq 0$ on a $u''(x) + f(x) \leq 0$ d'où on obtient le système :

$$\begin{cases} (g(x) - f(x))(u''(x) + f(x)) = 0 \\ g(x) \leq u(x) \\ f(x) \leq -u''(x) \end{cases}$$

II.6 Influence de variation de taille de Matrice A

Dans cette partie on a varie la taille de matice n et on a obtenue les resulta suivante :

```
*****n= 2 *****
Optimization terminated successfully      (Exit mode 0)
    Current function value: 11.29638888888895
    Iterations: 7
    Function evaluations: 10
    Gradient evaluations: 7
*****
*****n= 5 *****
Optimization terminated successfully      (Exit mode 0)
    Current function value: 23.531944444777686
    Iterations: 9
    Function evaluations: 16
    Gradient evaluations: 9
*****
*****n= 50 *****
Optimization terminated successfully      (Exit mode 0)
    Current function value: 214.45508063186318
    Iterations: 64
    Function evaluations: 241
    Gradient evaluations: 63
*****
*****n= 100 *****
Optimization terminated successfully      (Exit mode 0)
    Current function value: 425.0037041411946
    Iterations: 118
    Function evaluations: 511
    Gradient evaluations: 117
```

FIGURE II.6 – Valeur de solution theorique en fonction de n

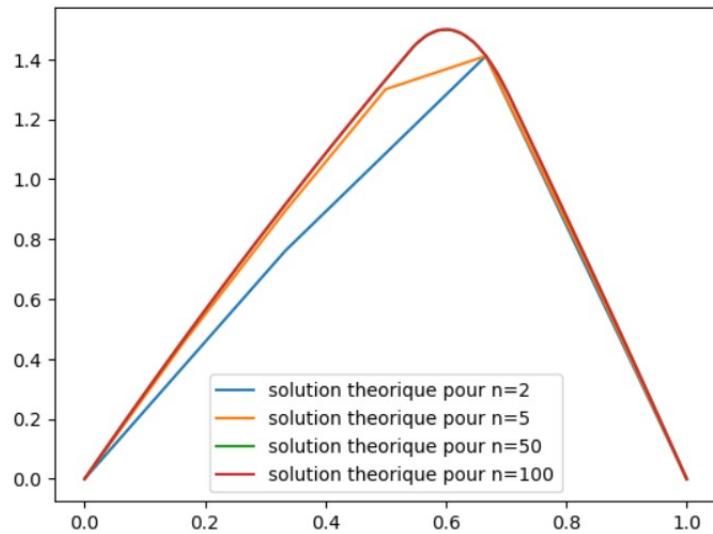


FIGURE II.7 – Graphe de Solution théorique en variant n

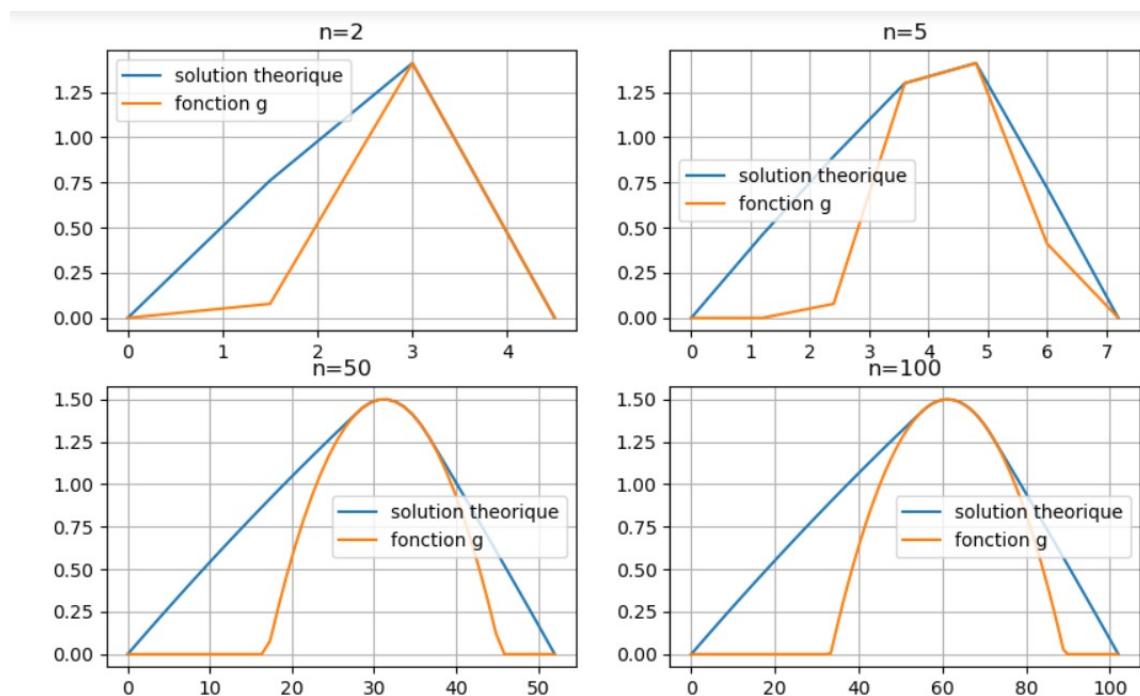


FIGURE II.8 – Graphe de Solution théorique et fonction obtacle g pour chaque n

Après avoir générée des figures pour la fonction obstacle $g(x)$ et la solution théorique, nous pouvons examiner visuellement si les propriétés attendues sont satisfaites.

1. Concavité maximale (Inéquation 4.a) :

- En fait, la solution u est concave, comme le montre visuellement une courbe qui ne s'élève pas au-dessus de ses tangentes.

2. Au-dessus de l'obstacle (Inéquation 4.b) :

- En comparant les courbes, si la courbe de la solution reste constamment au-dessus de la courbe de $g(x)$, cela signifie que la solution est effectivement au-dessus de l'obstacle.

3. Égalité dans une des deux équations (Équation 4.c) :

- L'équation (4.c) traduit le fait que l'on a au moins égalité dans l'une des deux inéquations, c'est-à-dire soit $-u(x) = f(x)$, soit $u(x) = g(x)$, et dans les deux cas, la solution u est au-dessus de l'obstacle. Cela a été vérifié.

Méthode de Gradient Projeté

II.7 Détermination de l'ensemble K_n

$$\begin{aligned} \text{On a } K_n &= \bigcap_{i=1}^n \{v_i \in \mathbb{R} \mid v_i \geq g(x_i)\} \\ &= \bigcap_{i=1}^n \{v_i \in \mathbb{R} \mid h(v_i) \geq 0\}. \end{aligned}$$

avec

$$\begin{aligned} h : \mathbb{R} &\rightarrow \mathbb{R} \\ y &\mapsto h(y) = y - g(x_i) \end{aligned}$$

Donc

$$K_n = \bigcap_{i=1}^n \left(h_i^{-1} ([0, +\infty]) \right)$$

Avec h_i continue sur \mathbb{R} et $[0, +\infty[$ un ensemble fermé, où $(h_i^{-1} ([0, +\infty]))$ est fermé, et K_n est un fermé.

K_n étant un ensemble de contraintes affines, il est convexe.

Le problème de minimisation (PM') a une solution unique u_n dans K_n , confirmant ainsi que K_n est non vide.

→ Ainsi, K_n est un ensemble non vide, fermé et convexe.

Soit $v \in \mathbb{R}^n$

$$\|P_{K_n}(v) - v\| = \min_{u \in K_n} \|u - v\|$$

Si $v \in K_n \implies v_i \geq g(x_i) \forall 1 \leq i \leq n$ et $\min_{u \in K_n} \|u - v\| = 0$

$$\implies P_{K_n}(v) = v \implies P_{K_n}(v)_i = v_i \forall 1 \leq i \leq n$$

$$\implies P_{K_n}(v)_i = v_i = \max(g_i, v_i) \forall 1 \leq i \leq n$$

Si $v \notin K_n \implies v_i < g(x_i)$ et $\min_{u \in K_n} \|u - v\| = \|g - v\|$

$$\implies P_{K_n}(v) = g \implies P_{K_n}(v)_i = g_i \forall 1 \leq i \leq n$$

$$\implies P_{K_n}(v)_i = g_i = \max(g_i, v_i) \forall 1 \leq i \leq n$$

$$\implies P_{K_n}(v)_i = \max(g_i, v_i) \forall 1 \leq i \leq n$$

donc on a $\forall v \in \mathbb{R}^n P_{K_n}(v)_i = \max(g_i, v_i) \forall 1 \leq i \leq n$

II.8 Genaralisation si on a une Projection sur un parallelepipède

Soit $K_B = \{v \in \mathbb{R}^n, v_i \leq b_i \Leftrightarrow -v_i \geq -b_i\}$ et $K_n = \{v \in \mathbb{R}^n, a_i \leq v_i \leq b_i\} = \{v \in K_B, a_i \leq v_i\}$ D'après 2.1 $(P_{K_B}(-v))_i = -(P_{K_B}(v))_i = -\max(-v_i, -b_i) = \min(v_i, b_i)$ Or $(P_{K_n}(v))_i = (P_{K_n}(P_{K_B}(v)))_i = \max(a_i, (P_{K_B}(v))_i)$ car $P_{K_B}(v) \in K_B$ D'où $(P_{K_n}(v))_i = \max(a_i, \min(v_i, b_i))$ (même si $a_i = -\infty$ ou $b_i = +\infty$)

II.9 Implimentation de code de l'algorithme de Gradiant Projété

```
#question 2.3
def projK(u,gn):
    return [max(u[i],gn[i]) for i in range(np.shape(u)[0])]
```

FIGURE II.9 – Script projK.py

```

def gradient_projete_pas_fixe (J, DJ , gn , u0 , rho , Tol , iterMax , store ) :
    uk=u0
    U=[]
    U.append(uk)
    k=0
    rk=Tol
    uk_preced=uk
    while(rk>=Tol)and(k<=iterMax):
        wk=D(J,uk,A,b)
        uk_preced=uk
        uk=projK(uk+rho*wk,gn)
        #uk=[max((uk+rho*wk)[i],gn[i]) for i in range(np.shape(uk)[0])]
        U.append(uk)
        rk=np.linalg.norm([uk[i]-uk_preced[i] for i in range(np.shape(uk)[0])])
        k=k+1
    if store==0 :
        return [uk,k]
    elif store==1 :
        return [U,k]

```

FIGURE II.10 – code de l’algorithme de Gradiant Projété pas fixe

Visualisation des Courbes de Niveau et du Gradient de J avec la méthode du Gradient Projoté :

```

# données sont bidimensionnelles et on veut représenter dans un espace 3D
# cette fonction prend en entrée des vecteur x , y pour faire la simulation
def J_2(x,y,f):
    return (2*x**2-2*y*x+2*y**2)**4.5-x*f(1/3)-y*f(2/3)
#derivee de J_2
def grad_J_2(x,y,f):
    return np.array([(4*x-2*y)*4.5-f(1/3),(4*y-2*x)*4.5-f(2/3)])
x_vals = np.arange(-10, 10, 1)
y_vals = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x_vals, y_vals)
Z=J_2(X,Y,f)

X_d,Y_d=grad_J_2(X,Y,f)
plt . contour (X ,Y ,Z,cmap="jet" )
plt.colorbar()
plt . quiver (X ,Y , X_d , Y_d , scale=1000)
x=np.linspace(0,1,4)
xv=[x[i] for i in range(1,np.shape(x)[0])]

#gn=[h*i for i in range(1,n+2)]
gn=gV(xv)
U1,k1=gradient_projete_pas_fixe (J, DJ , gn , u0 , rho , Tol , iterMax , store )
Ux1=[r[0] for r in U1]
Uy1=[r[1] for r in U1]
plt.plot(Ux1,Uy1,label=" les lignes qui relient uk",color="green")
plt.legend()
print(" le nombre d'iteration : ",k1,U1[-1] )
print(J_n(U1[-1],f))
plt.show()

```

FIGURE II.11 – code d’implimentation des Courbes de Niveau et du Gradient de J avec la méthode du Gradient Projoté

```

le nombre d'iteration : 4
la solution de probleme est: 11.296388888888892

```

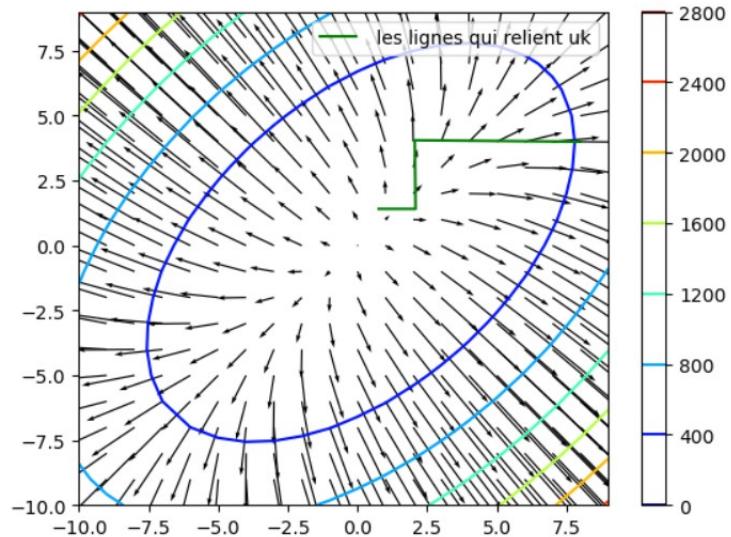


FIGURE II.12 – Les Courbes de Niveau et du Gradient de J

II.10 Influence de variation de taille de Matrice A

pour $\rho = 0.1$:

```

#Pour rho fixe = 0.1
def determiner_rho (n):
    return 0.1

```

FIGURE II.13 – fonction permettant de determiner ρ

```

def calcul_temp_execution_nbr_iter(J, DJ , gn , u0 , rho , Tol , iterMax , store ):
    start_time = time.time()
    uk,k1=gradient_projete_pas_fixe (J, DJ , gn , u0 , rho , Tol , iterMax , store )
    end_time = time.time()
    return [end_time - start_time,k1,uk]

```

FIGURE II.14 – fonction de calcule de temps d'exécution et nombre d'itérations

```

def afficher_sol_approchee(determiner_rho):
    R=np.zeros((4,102))
    L=np.zeros((4,102))
    def g(x):
        return max(1.5-20*(x-0.6)**2,0)
    store=0
    iterMax=100000
    Tol=10**(-5)
    k=0
    for n in [2, 5,50,100]:
        rho=determiner_rho(n)
        #rho=0.1
        gv=lambda x : [max(1.5-20*(x[i]-0.6)**2,0) for i in range(np.shape(x)[0]) ]
        u0=[]
        h=1/(n+1)
        x=np.linspace(0,1,n+2)
        xv=[x[i] for i in range(1,np.shape(x)[0])]

        #gn=[h*i for i in range(1,n+2)]
        gn=gv(xv)
        #gn=[h*i for i in range(1,n+1)]
        for i in range(n):
            if i % 2 == 0:
                u0.append(8)
            else:
                u0.append(4)
        u0=np.zeros(n)
        t_fixe,k1,uk=calcul_temp_execution_nbr_iter(j, D, gn, u0, rho, Tol, iterMax, store )
        print("\n*****\n pour n= ",n," \n le nombre d'itérations : ",k1," \n temps de calcul : ", t_fixe)

```

```

print("\n solution approcée : ",J_n(uk,t), "\n*****")
X=[i*h for i in range(0,n+2)]
Y=np.zeros(n+2)
Y[0]=0
Y[-1]=0
Y[1:n+1]=uk
G=[g(x) for x in X]
for i in range(n+2):
    R[k,i]=Y[i]
    L[k,i]=G[i]
k=k+1
fig, axs = plt.subplots(2, 2, figsize=(10, 6))
h=[2,5,50,100]
for i, ax in enumerate(axs.flatten()):
    n=h[i]
    X=[j*(1/n+1) for j in range(0,n+2)]
    ax.plot(X, [R[i,k] for k in range(n+2)],label="solution approchée")
    ax.plot(X, [L[i,k] for k in range(n+2)],label="fonction g")
    ax.set_title('courbe de g et solution approchée pour n=' +str(n))
    ax.legend()
    ax.grid(True)

```

FIGURE II.15 – Code permet de visualiser les courbes de solution en fonction de n

```
*****
pour n= 2
le nombre d'itérations : 55
temps de calcul : 0.003242969512939453
solution approchée : 11.296388889023326
*****
*****
```

```
*****
pour n= 5
le nombre d'itérations : 390
temps de calcul : 0.014075279235839844
solution approchée : nan
*****
*****
```

```
*****
pour n= 50
le nombre d'itérations : 116
temps de calcul : 0.012015581130981445
solution approchée : nan
*****
*****
```

```
*****
pour n= 100
le nombre d'itérations : 96
temps de calcul : 0.0070056915283203125
solution approchée : nan
*****
```

FIGURE II.16 – Solution par méthode de Gradiant Projété selon n

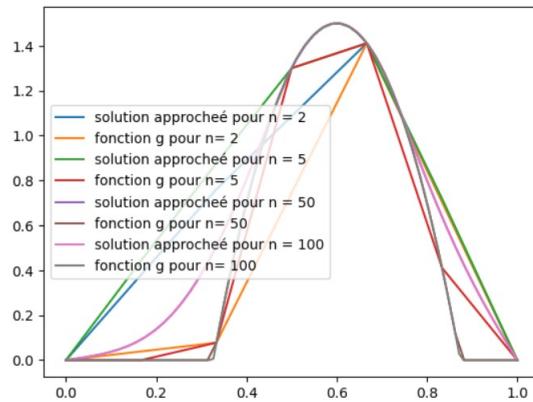


FIGURE II.17 – Les Courbes de g et solution approché en variant n

On constate que pour $\rho = 0.1$, l'algorithme du Gradiant Projété ne fonctionne bien que pour de petites valeurs de n , ce qui est logique étant donné que les valeurs propres de la matrice A sont de l'ordre de 10^{-5} , ce qui traduit ces erreurs pour de grandes valeurs de n .

pour $\rho = 10^{-5}$:

On répète le code énoncé précédemment en essayant de déterminer, mais cette fois on prend $\rho = 10^{-5}$, et voici les résultats observés :

```
*****
pour n= 2
le nombre d'itérations : 13943
temps de calcul : 0.4743943214416504
solution approchée : 11.32415630053867
*****
pour n= 5
le nombre d'itérations : 9674
temps de calcul : 0.35227012634277344
solution approchée : 23.545812868509387
*****
*****
```

```
*****
pour n= 50
le nombre d'itérations : 12436
temps de calcul : 1.1603035926818848
solution approchée : 214.4703452976973
*****
pour n= 100
le nombre d'itérations : 13495
temps de calcul : 2.6040713787078857
solution approchée : 425.01871318481227
*****
```

FIGURE II.18 – Solution par méthode de Gradient Projecté selon n

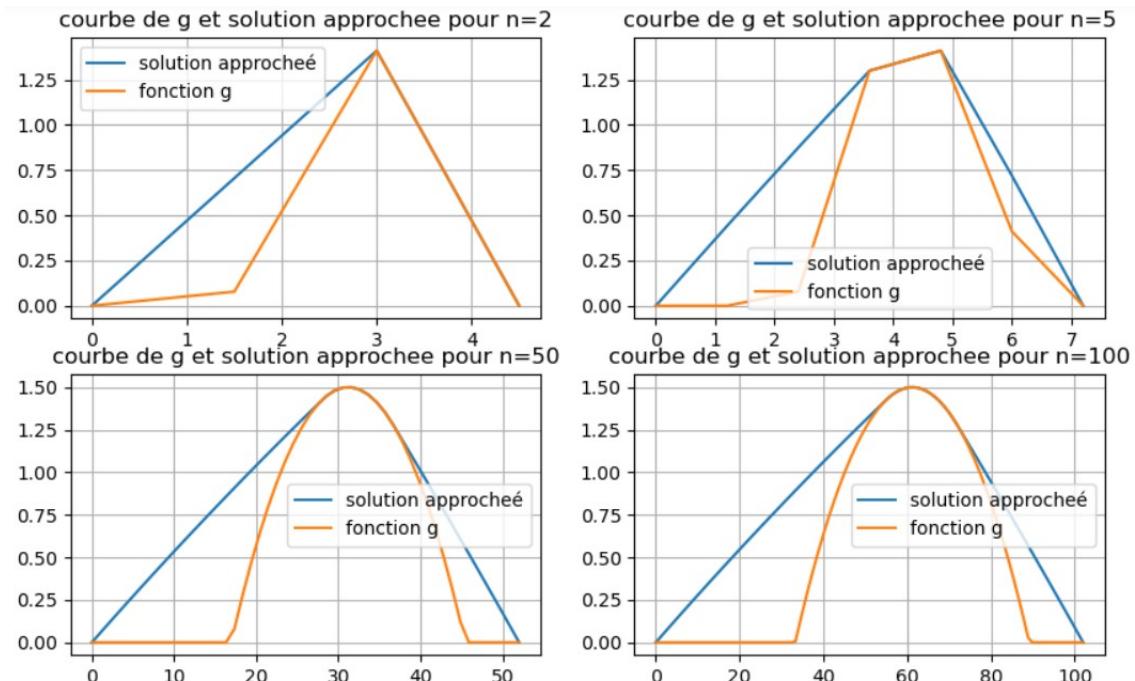


FIGURE II.19 – Les Courbes de g et solution approché en variant n

pour $\rho = 0.5$ et $\rho = 1$

```
*****
pour n= 2
le nombre d'itérations : 476
temps de calcul : 0.01555013656616211

solution approchée : nan
*****  

*****  

pour n= 5
le nombre d'itérations : 209
temps de calcul : 0.010170221328735352

solution approchée : nan
*****  

*****  

pour n= 50
le nombre d'itérations : 93
temps de calcul : 0.0069735050201416016

solution approchée : nan
*****  

*****  

pour n= 100
le nombre d'itérations : 79
temps de calcul : 0.008637189865112305

solution approchée : nan
*****
```

FIGURE II.20 – Solution par méthode de Gradiant Projoté selon n pour $\rho = 0.5$

```
*****
pour n= 2
le nombre d'itérations : 476
temps de calcul : 0.01555013656616211

solution approchée : nan
*****  

*****  

pour n= 5
le nombre d'itérations : 209
temps de calcul : 0.010170221328735352

solution approchée : nan
*****  

*****  

pour n= 50
le nombre d'itérations : 93
temps de calcul : 0.0069735050201416016

solution approchée : nan
*****  

*****  

pour n= 100
le nombre d'itérations : 79
temps de calcul : 0.008637189865112305

solution approchée : nan
*****
```

FIGURE II.21 – Solution par méthode de Gradiant Projoté selon n pour $\rho = 1$

On constate que, tout comme pour $\rho = 0.1$, la méthode du gradient projeté diverge pour $\rho = 0.5$ et $\rho = 1$. Ainsi, il est nécessaire de développer un algorithme permettant de déterminer une valeur appropriée pour ρ .

II.11 Amélioration de l'algorithme du gradient projeté par la détermination du pas optimal

Dans cette section, nous reprenons le travail effectué dans la section précédente en utilisant le pas optimal $\rho_{\text{opt}} = \frac{2}{\lambda_1 + \lambda_n}$, où λ_1 et λ_n sont respectivement les plus petites et plus grandes valeurs propres.

```

#question 2.8 reprendre question 2.5 pour rho optimale
def val_prop_min_max(n):
    A=generate_A(n)
    eigenvalues, eigenvectors = np.linalg.eig(A)
    l_min=min(eigenvalues)
    l_max=max(eigenvalues)
    return [l_min,l_max]
def determiner_rho_(n):
    l_min,l_max=val_prop_min_max(n)
    return(2/(l_min+l_max))

```

FIGURE II.22 – fonction qui determine $\rho_{optimal}$

```

#question 2.8 reprendre question 2.5
# données sont bidimensionnelles et on veut représenter dans un espace 3d
x_vals = np.arange(-10, 10, 1)
y_vals = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x_vals, y_vals)
Z=j_2(X,Y,f)
X_d,Y_d=grad_j_2(X,Y,f)
plt . contour (X ,Y ,Z,cmap="jet" )
plt.colorbar()
plt . quiver (X ,Y , X_d , Y_d , scale=1000)
x=np.linspace(0,1,4)
xv=[x[i] for i in range(1,np.shape(x)[0])]
gn=gv(xv)
U1,k1=gradient_projete_pas_fixe (J, DJ , gn , u0 , rho , Tol , iterMax , store )
Ux1=[r[0] for r in U1]
Uy1=[r[1] for r in U1]
plt.plot(Ux1,Uy1,label=" les lignes qui relient uk",color="green")
plt.legend()
print(" le nombre d'iteration : ",k1,U1[-1])
print(j_n(U1[-1],f))
plt.show()

```

FIGURE II.23 – code d’Implmentation des Courbes de Niveau et du Gradient de J avec ρ_{opt}

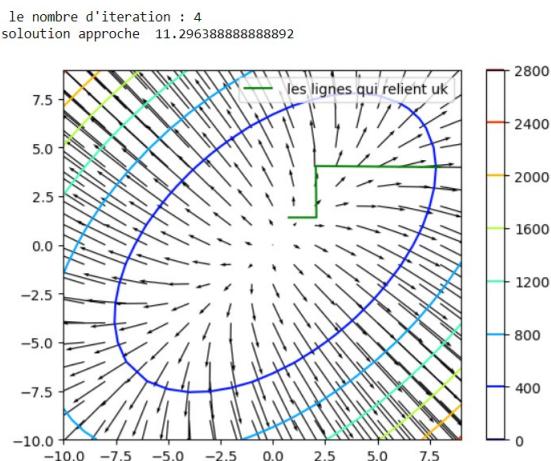


FIGURE II.24 – des Courbes de Niveau et du Gradient de J avec ρ_{opt}

Maintenant, à l'instar du paragraphe précédent, nous allons étudier l'influence de la variation de la matrice A sur la solution. En appliquant le même code que précédemment, mais en modifiant uniquement la fonction *determiner_rho()*, nous obtenons le résultat suivant :

```
*****
pour n= 2
le nombre d'itérations : 3
temps de calcul : 0.0011723041534423828
solution approchée : 11.296388888888892
*****
*****
```

```
pour n= 5
le nombre d'itérations : 18
temps de calcul : 0.0010013580322265625
solution approchée : 23.53194444549579
*****
```

```
*****
pour n= 50
le nombre d'itérations : 1116
temps de calcul : 0.09526276588439941
solution approchée : 214.45512093600993
*****
*****
```

```
pour n= 100
le nombre d'itérations : 3886
temps de calcul : 0.587794303894043
solution approchée : 425.004074832961
*****
```

FIGURE II.25 – Solution par méthode de Gradiant Projoté selon n pour ρ_{opt}

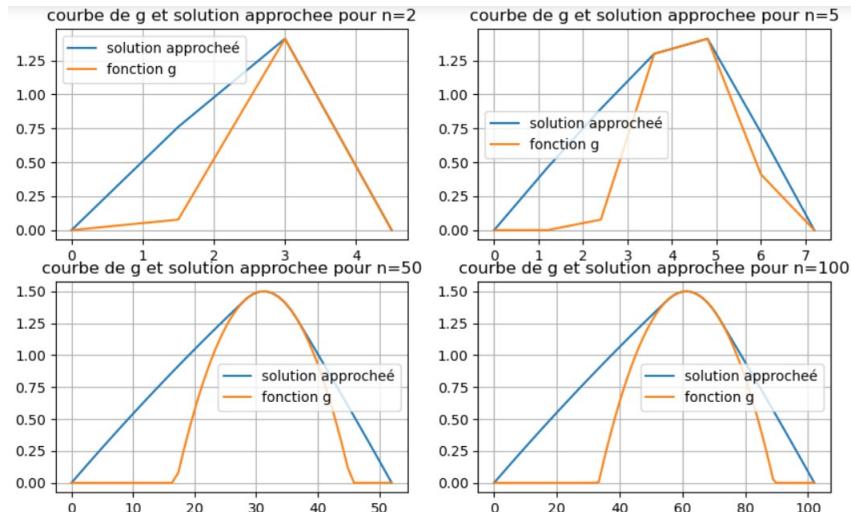


FIGURE II.26 – Courbe de g et solution approchée suivant n rt avec $\rho = \rho_{opt}$

On constate que la méthode avec pas optimal est plus efficace que la méthode du Gradient Projoté à pas fixe, puisqu'elle présente un temps d'exécution et un nombre d'itérations plus réduits.

II.12 Changement de fonction $f(x)$

Dans cette section, nous examinons la fonction $f(x) = \pi^2 * \sin(\pi x)$, et nous allons modifier cette fonction en implémentant le code suivant :

```
import math as m
def f(x):
    return (m.pi**(2))*m.sin((m.pi)*x)
```

FIGURE II.27 – code de $f(x)$

On obtient les résultats suivants :

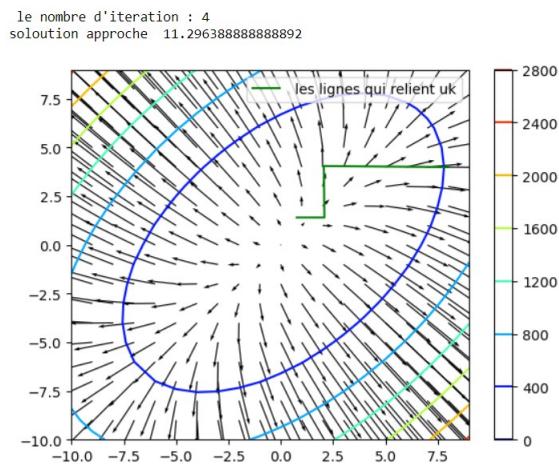


FIGURE II.28 – des Courbes de Niveau et du Gradient de J avec ρ_{opt}

```
*****
pour n= 2
le nombre d'itérations : 3
temps de calcul : 0.0009980201721191406
solution approchée : -6.680367285775928
*****
*****
pour n= 5
le nombre d'itérations : 32
temps de calcul : 0.001001119613647461
solution approchée : -11.426013364396447
*****
*****
pour n= 50
le nombre d'itérations : 1395
temps de calcul : 0.13845181465148926
solution approchée : -89.7873168951399
*****
*****
pour n= 100
le nombre d'itérations : 4828
temps de calcul : 0.7869892120361328
solution approchée : -177.6054006786028
*****
```

FIGURE II.29 – Solution par méthode de Gradiant Projoté selon n pour ρ_{opt}

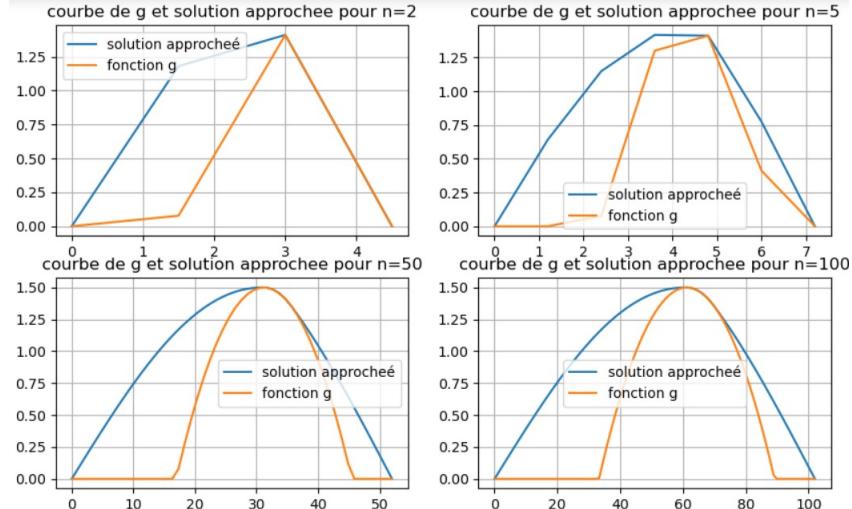


FIGURE II.30 – Courbe de g et solution approchée suivant n et avec $\rho = \rho_{opt}$

Méthode de Pénalisation :

II.13 Implementation de code de methode de pénalisation à pas fixe

```

def penalite(gn,eta,u):
    s=0
    for e in (gn-u):
        s=s+max(e,0)**2
    return (1/eta)*s

def sign(x):
    if x==0:
        return 0
    elif x>0:
        return 1
    else:
        return -1

def grad_penalite(gn,eta,u):
    L=np.zeros(np.shape(u)[0])
    for i in range(np.shape(u)[0]):
        L[i]=(2/eta)*(sign(gn[i]-u[i])*max(gn[i]-u[i],0))
    return L

def grad_pas_fixe(Jf,DJf,u0,rho , epsilon , iterMax , store ):
    n=np.shape(u0)[0]
    k=0
    rk=1
    uk=u0
    U=[]
    U.append(uk)
    while (rk>=epsilon) & (k<= iterMax ):
        d=Df(uk)
        uk=uk+rho*d
        rk=np.linalg.norm(rho*d)
        U.append(uk)
        k=k+1
        #print("rk=",rk)
    if store==0:
        return (k,uk)
    if store==1:
        return (k,U)

def penalisation ( J, DJ , gn , eta , u0 , rho , epsilon , iterMax , store ) :
    Jf=lambda u:J(u)+penalite(gn,eta,u)
    DJf=lambda u:DJ(u)-grad_penalite(gn,eta,u)
    k, u = grad_pas_fixe(Jf ,DJf ,u0 ,rho , epsilon , iterMax , store ) ;
    return (k, u )

```

FIGURE II.31 – code de méthode de pénalisation

II.14 Influence de η sur la solution obtenue par méthode de pénalité

```

# si store =0 on affiche les courbes de g et solution approché et solution de -u"=1
# si store =1 on affiche tous les solution approcher sur le même figure
def tester_solution algo_penalisation(J, DJ , gn , u0 , rho , epsilon , iterMax ,store):
    n=50
    R=np.zeros((7,n**2))
    L=np.zeros((7,n**2))
    k=0
    for eta in [10**(-5),10**(-4),10**(-3),10**(-2),10**(-1),10**(0),10**(+1)]:
        start_time = time.time()
        k1,uk=penalisation (J, DJ , gn , eta , u0 , rho , epsilon , iterMax , 0 )
        end_time = time.time()
        t=end_time-start_time
        print("\n*****\n pour eta= ",eta," \n le nombre d'itérations : ",k1,
              "\n temps de calcul : ", t)
        print("\n solution approchée : ",J(uk)," \n*****")
        X=[i*(1/(n+1)) for i in range(0,n+2)]
        Y=np.zeros(n+2)
        Y[0]=0
        Y[-1]=0
        Y[1:n+1]=uk
        G=[[g(x) for x in X]
            for i in range(n+2)];
        for i in range(n+2):
            R[k,i]=Y[i]
            L[k,i]=G[i]
        k=k+1

```

```

m=[10**(-5),10**(-4),10**(-3),10**(-2),10**(-1),10**(0),10**(1)]
x=[j*(1/n+1) for j in range(0,n+2)]
if store ==0 :
    for i, ax in enumerate(axes.flatten()):
        if(i%6):
            break
        ax.plot(X, [R[i,k] for k in range(n+2)],label="solution approchée")
        ax.plot(X, [L[i,k] for k in range(n+2)],label="fonction g")
        ax.plot(X[1:-1],u_theo,label="solution théorique ",linestyle='--',color='red')
        ax.set_title(' eta= '+str(m[i]))
        ax.legend()
        ax.grid(True)
elif store==1:
    plt.plot(X[1:-1],u_theo,label="solution théorique ",linestyle='--',color='black')
    for i in range(7):
        plt.plot(X, [R[i,k] for k in range(n+2)],label="eta= "+str(m[i]))
    plt.legend()
return R

```

FIGURE II.32 – Code de tracer les courbes de solution en fonction de η

et voici les résultats obtenues :

```

*****
pour eta= 1e-05
le nombre d'itérations : 100001
temps de calcul : 15.202792882919312
solution approchée : 214.39936950952756
*****

pour eta= 0.0001
le nombre d'itérations : 11429
temps de calcul : 2.172682762145996
solution approchée : 213.86695991270045
*****


pour eta= 0.001
le nombre d'itérations : 11271
temps de calcul : 2.035037040710449
solution approchée : 208.9352109587637
*****


pour eta= 0.01
le nombre d'itérations : 11683
temps de calcul : 2.173335313796997
solution approchée : 174.46008795272633
*****


pour eta= 0.1
le nombre d'itérations : 19145
temps de calcul : 3.5070738792419434
solution approchée : 63.61988208548693
*****


pour eta= 1
le nombre d'itérations : 35703
temps de calcul : 6.500007252120972
solution approchée : 1.6586422543294645
*****


pour eta= 10
le nombre d'itérations : 40362
temps de calcul : 7.672910928726196
solution approchée : -1.9489028738459195
*****
```

FIGURE II.33 – solution approché par méthde de pénalisation en fonction de η

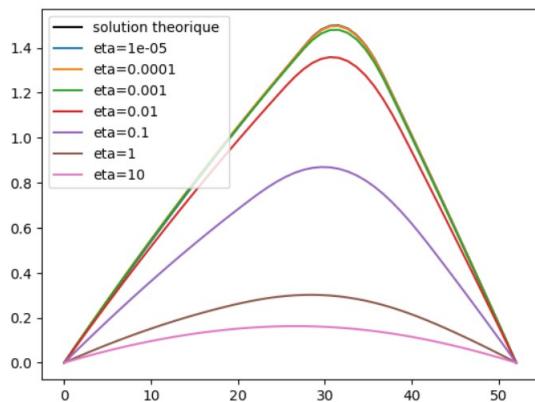


FIGURE II.34 – Les courbes de solution en fonction de η

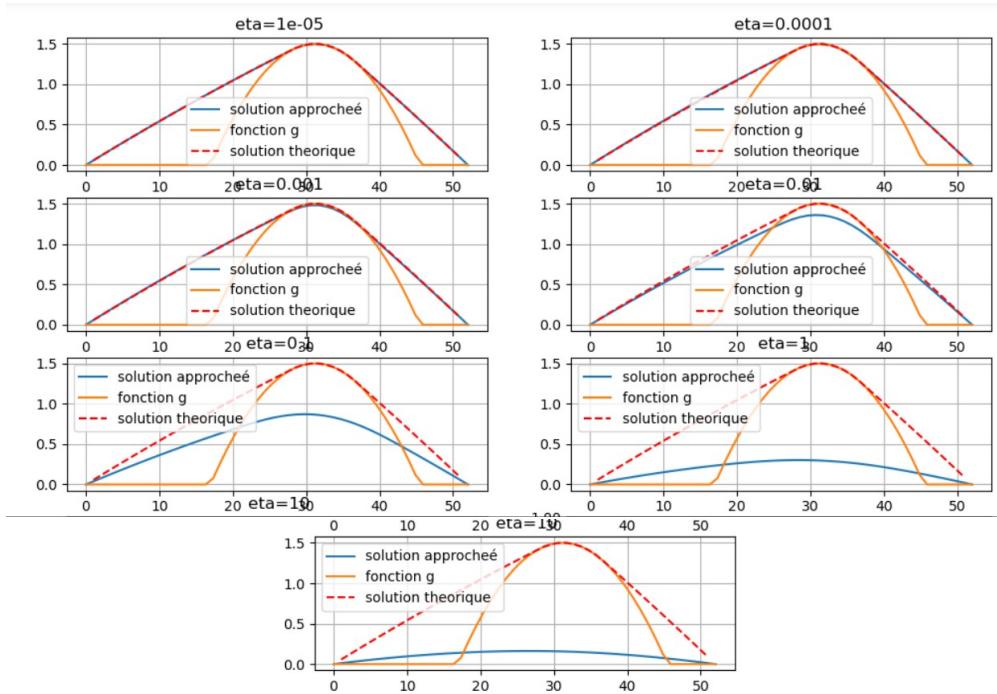


FIGURE II.35 – les courbes de solution approché , fonction g et solution theorique en fonction de η

Et voici un tableau illustrant la variation de l'erreur en fonction de η , ainsi que le code permettant de calculer l'erreur pour chaque η :

η	Solution par méthode de pénalisation	Solution théorique	Erreur
10^{-5}	214.39936		0.05571
10^{-4}	213.8669		0.588120
10^{-3}	208.93521		5.5198
10^{-2}	174.46008		39.99499
10^{-1}	63.61988		150.8351
1	1.65864		212.7964
10^1	-1.94890	214.45508	216.4039

FIGURE II.36 – Tableau de variation de l'erreur en fonction de η

```
m=[10**(-5),10**(-4),10**(-3),10**(-2),10**(-1),10**(0),10**(1)]
for i in range(7):
    sol_penalite=[[R[i,j] for j in range(1,51)]]
    print("*****\n")
    print("teta = ",m[i],"\n solution approchee=",sol_penalite)
    print("\n solution theorique : ",sol_theo)
    print("\n erreur : ",abs(sol_penalite-sol_theo) )
    print("\n*****\n")
#plus eta augmente erreur augmente
```

FIGURE II.37 – Code de calcule de l'erreur pour chaque η

On constate que plus la valeur de η est grande, plus l'erreur est importante, comme clairement illustré dans le tableau et les figures. En effet, pour $\eta \leq 10^{-2}$, l'erreur est considérée comme très élevée (près de 40 en termes d'erreur), ce qui explique pourquoi la courbe de la solution approchée s'éloigne progressivement de la solution théorique, comme le montre la Figure II.37.

II.15 Implementation de code de méthode de pénalisation à pas optimal

Dans cette section, nous allons modifier le code précédent en remplaçant l'algorithme de pas fixe par l'algorithme de pas optimal. Ce dernier utilise l'algorithme de la section dorée pour déterminer $\rho_{optimal}$ tel que $\rho_{optimal} = \min_{\rho \in \mathbb{R}} J_n(u + \rho d)$. Voici le code permettant d'implémenter cet algorithme :

```

Jf=lambda u:J(u)+penalite(gn,eta,u)
DJf=lambda u: DJ(u)-grad_penalite(gn,eta,u)

def J_n_doree(u,d,t,Jf):
    return J(u*t*d)

def minimiser_section_dorée(a,b,u,d,Tol,Jf):
    k=0
    e=b-a
    ksi=(1+m.sqrt(5))/2
    while e>= Tol:
        a1=a+e/ksi**2
        b1=a+e/ksi
        if( J_n_doree(u,d,a1,Jf)>J_n_doree(u,d,b1,Jf)):
            a=a1
        elif( J_n_doree(u,d,a1,Jf)<J_n_doree(u,d,b1,Jf)):
            b=b1
        else:
            a=a1
            b=b1
            e=b-a
        k+=1
    #x min=min(J_n_doree(u,d,a1,f),J_n_doree(u,d,b1,f))
    x_min=(a+b)/2
    return [x_min,k]

def minimiser_grad_pas_optimal(Jf,DJf,u0,rho,Tol,iterMax,f):
    Ux1=[]
    Uy1=[]
    k=0
    rk=Tol
    rho_k=rho
    uk=u0
    n=np.shape(u0)[0]
    Ux1.append(uk[0])
    Uy1.append(uk[1])
    while (rk>=Tol) & (k< iterMax):
        dk=-DJf(uk)
        rho_k=minimiser_section_dorée(-1000,1000,uk,dk,Tol,Jf)[0]
        uk=uk+rho_k*dk
        rk=np.linalg.norm(rho_k * dk)
        Ux1.append(uk[0])
        Uy1.append(uk[1])
        k+=1
    return (k,uk,Ux1,Uy1)

def penalite_pas_optimal(J, DJ , gn , eta , u0 , rho , epsilon , iterMax,f):
    k,uk,Ux1,Uy1=minimiser_grad_pas_optimal(Jf,DJf,u0,rho,Tol,iterMax,f)
    return (k,uk)

```

FIGURE II.38 – Code de l’algorithme de pénalisation à pas optimal

et voici les résultats obtenus :

```

*****
pour eta= 1e-05
le nombre d'itérations : 100000
temps de calcul : 1721.7659485340118
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****


*****
pour eta= 0.0001
le nombre d'itérations : 100000
temps de calcul : 1744.89547681800847
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****


*****
pour eta= 0.001
le nombre d'itérations : 100000
temps de calcul : 1774.7967524528503
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****


*****
pour eta= 10
le nombre d'itérations : 100000
temps de calcul : 1735.061152458191
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****


*****
pour eta= 0.1
le nombre d'itérations : 100000
temps de calcul : 1819.7485753032227
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****


*****
pour eta= 1
le nombre d'itérations : 100000
temps de calcul : 1788.366930961609
solution approchée : 214.48101028129201
erreur: 0.025929649428832136
*****
```

FIGURE II.39 – Solution et erreur en fonction de η

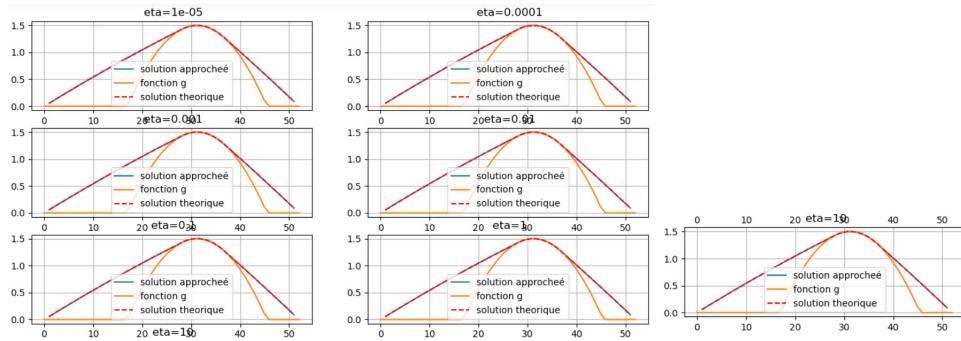


FIGURE II.40 – les courbes de solution approchée , fonction g et solution theorique en fonction de η

II.15.1 Comparaison entre la méthode de pénalisation à pas fixe et à pas optimal

Voici ces deux tableau qui décrivent la différence entre ce 2 méthodes en point de vue rapidité et précision :

η	Solution par méthode de pénalisation à pas fixe	Solution par méthode de pénalisation à pas optimal	Solution théorique	Erreur de méthode de pénalisation à pas fixe	Erreur de méthode de pénalisation à pas optimal
10^{-5}	214.39936	214.481		0.056	0.02592
10^{-4}	213.8669	214.481		0.588120	0.02592
10^{-3}	208.93521	214.481			5.5198 0.02592
10^{-2}	174.46008	214.481		39.99499	0.02592
10^{-1}	63.61988	214.481		150.8351	0.02592
1	1.65864	214.481		212.7964	0.02592
10^1	-1.94890	214.481		216.4039	0.02592
214.45508					
η	Méthode de pénalisation à pas fixe	Méthode de pénalisation à pas optimal	η	Méthode de pénalisation à pas fixe	Méthode de pénalisation à pas optimal
10^{-5}	>100001	>100000	10^{-5}	15,202	>1721.765
10^{-4}	11429	>100000	10^{-4}	2,177	>1744.895
10^{-3}	11271	>100000	10^{-3}	2,035	>1730,12
10^{-2}	11683	>100000	10^{-2}	2,17	>1730,12
10^{-1}	19145	>100000	10^{-1}	3,5	>1773.817
1	35703	>100000	1	6,5	>1788.3669
10^1	40362	>100000	10^1	7,6729	1735,8
Nombre d'itérations			temps d'exécution		

FIGURE II.41 – Tableaux comparatifs de méthode de pénalisation à pas fixe et à pas optimal

On constate que la méthode de pénalisation à pas optimal est plus précise que celle à pas fixe en variant η (erreur de l'ordre de 0.05), mais malheureusement elle est très lente (presque 30 min) et nécessite beaucoup de temps d'exécution. C'est pourquoi elle nécessite un algorithme permettant de calculer η_{opt}

Méthode d'Uzawa

II.16 Relation entre méthode d'Uzawa et problème Dual

Pour $(\mathbf{x}, \boldsymbol{\lambda}) \in \mathbb{R}^n \times \mathbb{R}^n$, on définit le lagrangien :

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = J_n(\mathbf{v}) + \sum_{i=1}^n \boldsymbol{\lambda}_i (g_i(\mathbf{x}_i) - v_i(\mathbf{x}_i)) = \mathcal{L}_{\boldsymbol{\lambda}}(\mathbf{x})$$

On définit le problème primal (P) et son problème dual (D) :

$$\begin{cases} \inf_{\mathbf{v} \in \mathbb{R}^n} \sup_{\boldsymbol{\lambda} \in \mathbb{R}^n, \boldsymbol{\lambda}_I \geq 0} \mathcal{L}(\mathbf{v}, \boldsymbol{\lambda}) & (\text{P}) \\ \sup_{\boldsymbol{\lambda} \in \mathbb{R}^n, \boldsymbol{\lambda}_I \geq 0} \inf_{\mathbf{v} \in \mathbb{R}^n} \mathcal{L}(\mathbf{v}, \boldsymbol{\lambda}) & (\text{D}) \end{cases}$$

Soit $L_n(\boldsymbol{\lambda}) = \inf_{\mathbf{v} \in \mathbb{R}^n} \mathcal{L}_{\boldsymbol{\lambda}}(\mathbf{v})$

Donc le problème dual (D) s'écrit $\sup_{\boldsymbol{\lambda} \in \mathbb{R}^n, \boldsymbol{\lambda}_I \geq 0} L_n(\boldsymbol{\lambda})$

Donc pour résoudre ce problème on peut utiliser la méthode de Gradient Projété sur l'espace $K = \{\mathbf{v} \in \mathbb{R}^n, v_i \geq 0\} = \mathbb{R}_+^n$ avec les propriétés suivantes :

-La direction de descente : $d^{(k)} = -\nabla L_n(\boldsymbol{\lambda}^{(k)}) = g(x) - v$

Mettre à jour : $\boldsymbol{\lambda}^{(k+1)} = P_{\mathbb{R}_+^n}(\boldsymbol{\lambda}^{(k)} + \rho d^{(k)})$ car on a $\boldsymbol{\lambda} \in K$

Or $P_{\mathbb{R}_+^n}(\boldsymbol{\lambda}^{(k)} + \rho d^{(k)}) = \max(\lambda_i^{(k)} + \rho(g(x_i) - v_i), 0)_{1 \leq i \leq n}$ ceci d'après question 2 .

On conclut que l'algorithme d'Uzawa est un algorithme de gradient projeté appliqué au problème dual du problème (7).

II.17 Convergence de méthode d'Uzawa

K_n est un ensemble de contraintes d'inégalités affines.

$$\begin{aligned} \langle \nabla J_n(x) - \nabla J_n(y), x - y \rangle &= \langle Ax - Ay, x - y \rangle \\ &= \langle A(x - y), x - y \rangle \\ &= \langle AX, X \rangle; X = x - y \\ &\geq \lambda_1 \|X\|^2 \end{aligned}$$

avec $\lambda_1 = \min\{\lambda, \lambda \in sp(A)\}$

or on a A est symétrique définie positive donc $sp(A) \subset \mathbb{R}_+^*$ Donc J_n est strictement convexe de plus J_n est coercive .

$$\begin{cases}
J_n \text{ est } \lambda_1\text{stictement convexe} \\
J_n \text{ est coercive} \\
K_n \text{ est non vide}
\end{cases} \implies \text{-Il existe un unique minimiseur } u \text{ de } J_n \text{ sur } K_n$$

$K_n = \{v \in \mathbb{R}^n, v_i \leq b_i\} \cap \{v \in \mathbb{R}^n, v_i \leq -a_i\}$
 $= \{v \in \mathbb{R}^n, I_nv \leq b_i\} \cap \{v \in \mathbb{R}^n, I_nv \leq -a\}$
 $= \{v \in \mathbb{R}^n, \mathbf{C}_1 v \leq b\} \cap \{v \in \mathbb{R}^n, \mathbf{C}_2 v \leq -a\}$
 $K_n = \{v \in \mathbb{R}^n, \mathbf{C}_1 v \leq f_1, \mathbf{C}_2 v \leq f_2\}$
soit $\mathbf{C} = \begin{bmatrix} \mathbf{C}_1 \\ \mathbf{C}_2 \end{bmatrix}$ $\mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}$
on peut montrer comme en TD 4 que si $0 < \rho < \frac{2\lambda_{\min}(A)}{\|\mathbf{C}\|^2}$, alors il existe $\gamma > 0$ tel que

$$\|u_k - u\|_2^2 \leq \gamma (\|\lambda_k - \lambda\|_2^2 - \|\lambda_{k+1} - \lambda\|_2^2)$$

or on a $(\|\lambda_k - \lambda\|_2^2)_k$ est une suite décroissante minorée par 0 donc c'est une suite convergente est convergente

d'où $\|u_k - u\|^2$ converge vers 0 d'où u_k converge vers u

Conclusion :

- Il existe un unique minimiseur u de J_n sur K_n
- pour tous $0 < \rho < \frac{2\lambda_{\min}(A)}{\|\mathbf{C}\|^2}$ on a l'algorithme d'Uzawa converge

Code de l'algorithme d'Uzawa

```

#ReDefinition des variables:
store=1

def f(x):
    return 1
def g(x):
    return np.maximum(1.5-20*(x-0.6)**2,0)

#Dimension de l'espace:
n=2
#Pas de subdivision:
h = 1/(n + 1)
#La matrice A:
A = (1/(h**2))*diags([-1*np.ones(n-1),2*np.ones(n),-1*np.ones(n-1)],[ -1,0,1]).toarray()
#Subdivision de l'intervalle [0,1]:
x=np.ones((n+2,1))
for i in range(0,n+2):
    x[i]=i*h
xv = x[1: -1]
#le vecteur b:
b=np.ones((n,1))
for i in range(1,n):
    b[i]=f(x[i])

#Definition de la fonction Lambda du probleme dual:
def Lambda_dual(u,lambda):
    y=DJ(u, A, b)+sum([lambda[i]*(g(i)-u[i]) for i in range(np.shape(u)[0])])
    return y

#Definition de la fonction grad Lambda du probleme dual:
def Lambda_dual_grad(u,lambda):
    u_vect=np.zeros((np.shape(u)[0],1))
    u_vect[:,0]=u
    y=DJ(u_vect, A, b).flatten()-lambda
    return y

#Methode du gradient a pas fixe pour la fonction Lambda
def grad_pas_fixe_lambda(lambda, u0, rho, epsilon=10**(-6), Max_iter=1000) :
    k = 0
    ue=u0.copy()
    epsilon = Tol
    #Boucle:
    while ((epsilon >= Tol)and(k <=Max_iter)) :
        d=-1*Lambda_dual_grad(ue,lambda)
        up=ue
        ue=ue+rho*d
        epsilon=np.linalg.norm(up-ue)
        k = k + 1
    return ue

```

FIGURE II.42 – Code de l'algorithme d'Uzawa

II.18 Influence de variation de ρ

Maintenant on vas etudie l'influence de variation de rho sur la solution ainsi sur la rapidite de l'algorithem et voici les resultat obtenue :

```

def Uzawa(u0, lamda0, gx, rho=0.01, Tol=10**(-6), Max_iter=10000
          u = u0.copy()
          k = 0
          err = Tol
          lamda=lamda0.copy()
          while err >= Tol and k <= Max_iter:
              # Calculer  $u^{(k+1)}$  minimiseur de la fonction Lambda
              u_new = grad_pas_fixe_lambda(lamda, u, rho)

              # Mise à jour  $\lambda^{(k+1)}$ 
              lamda_new = np.maximum(lamda + rho * (gx - u_new), 0)
              # Calculate  $r^{k+1}$ 
              err = np.linalg.norm(u_new - u)

              # Mise à jour  $u$  et Incrementer le compteur
              lamda=lamda_new
              u = u_new
              k = k+1

return [u,k]

#script_uzawa.py
n=2
Tol=10**(-5)
x = np.linspace(0,1,n+2)
xv = x[1:-1]
fv = np.array ([ f( e ) for e in xv ])
gv = np.array ([ g( k ) for k in xv ])
u0=np.zeros((n,))
lamda0=np.zeros(np.shape(u0)[0])
u,k=Uzawa(u0, lamda0, gv)
print("solution par methode Uzawa:",J(u,A,fv))

solution par methode Uzawa: 11.275379602843385

```

FIGURE II.43 – Code de l'algorithme d'Uzawa et résultat obtenu

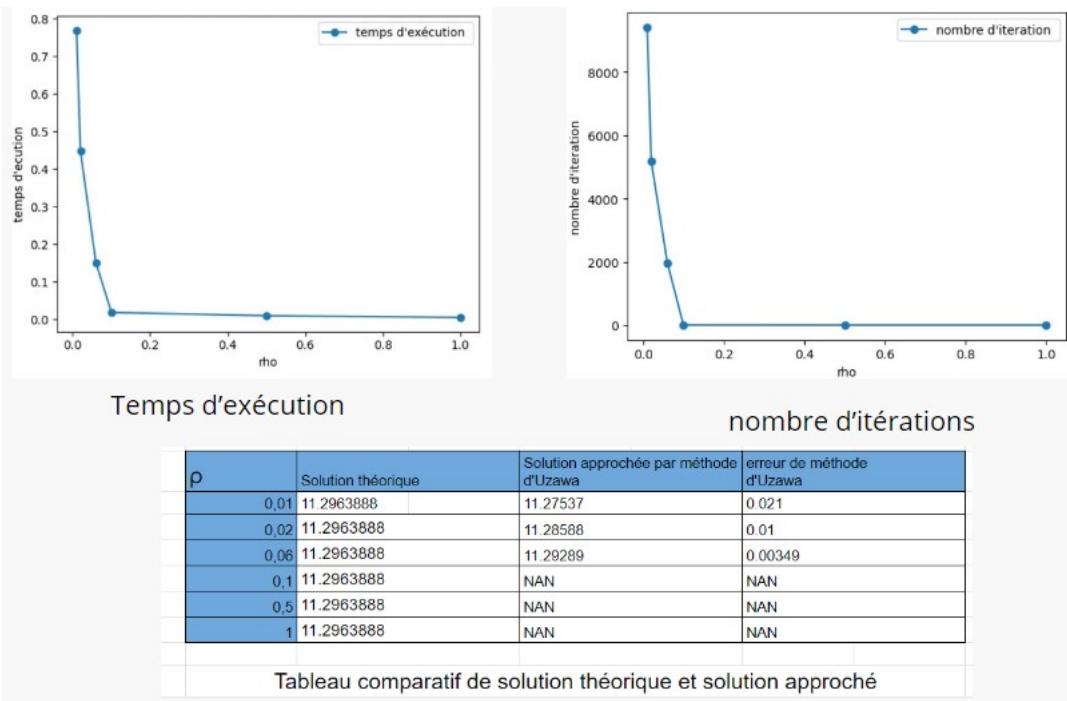


FIGURE II.44 – Comparaison entre les méthodes d’Uzawa avec des différents ρ

On constate que :

- L'algorithme ne converge que pour $\rho \in]0, 0.07[$
- Dans les cas où l'algorithme converge, plus ρ est grand, plus l'erreur (que l'on considère comme petite) est petite, indiquant une solution plus précise. De plus, l'algorithme est rapide, car le nombre d'itérations ainsi que le temps d'exécution diminuent.

Conclusion

En conclusion, ce projet d'optimisation visait à optimiser la fonction J en utilisant plusieurs algorithmes, tant dans le cas avec contraintes que dans le cas sans contraintes. À travers votre démarche, nous avons exploré différentes approches et techniques pour optimiser la performance des algorithmes, avec l'objectif ultime d'obtenir des résultats plus efficaces et robustes.

Dans le contexte des problèmes sans contraintes, notre observation des trois algorithmes de gradient à pas fixe, de pas optimal et de pas conjugué révèle que chacun présente des défauts spécifiques. Malgré leurs limitations respectives, tous parviennent à converger vers une solution. Toutefois, notre analyse comparative souligne que la méthode du gradient à pas conjugué se distingue comme la plus performante. Ses avantages apparents dans la résolution de problèmes sans contraintes renforcent la conviction en sa pertinence et son efficacité dans ces contextes.

En ce qui concerne les problèmes avec contraintes, notre exploration des méthodes a révélé des nuances importantes. La méthode du gradient projeté à pas fixe s'est avérée fonctionnelle, mais avec une restriction liée à la valeur du pas ρ , nécessitant une recherche laborieuse pour trouver une valeur optimale. L'approche de la pénalisation, bien que résolvant ce problème, dépend elle-même d'un paramètre crucial, η introduisant ainsi une nouvelle couche de complexité. La méthode d'Uzawa a été identifiée comme une solution à ce défi.

Ces observations soulignent l'importance critique du choix des paramètres tels que le pas ρ et le paramètre de pénalisation η , qui exercent une influence significative sur l'erreur et la rapidité de convergence vers la solution. La dépendance de ces méthodes aux paramètres souligne la nécessité d'une approche soigneuse et réfléchie lors de la mise en œuvre des techniques d'optimisation dans des environnements réels.

En résumé, ce projet a enrichi notre compréhension de l'optimisation en mettant en lumière les avantages et les limitations des algorithmes dans des contextes variés. Ces conclusions offrent des indications précieuses pour orienter des développements futurs dans le domaine de l'optimisation, soulignant l'importance de l'adaptabilité des méthodes en fonction des spécificités des problèmes rencontrés.

Bibliographie

- [1] Maher Moakher. *Analyse Mathématique et Numérique de la Méthode des Éléments Finis.* 2023. Année Universitaire 2023-2024.