



RÉPUBLIQUE TUNISIENNE
MINISTÈRE DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
Université de Tunis El Manar
Ecole Nationale d'Ingénieurs de Tunis



Mini-projet d'Optimisation (Période 1)

Présenté par :

**Zouaoui Ahmed Anas
Abbouz Hamza**

Spécialité :

**Actuariat, Sciences des données et Contrôle
Stochastique (ASDCS)**

Année Universitaire : **2024-2025**

Table des Matières

1 Optimisation sans contraintes	6
1.1 Analyse mathématique	10
1.1.1 Symétrie de la matrice A	10
1.1.2 Définie positive	10
1.1.3 Solution théorique pour $n = 2$	12
1.2 Affichage des courbes et leurs interprétations :	13
1.2.1 La méthode du gradient à pas fixe	13
1.2.2 La méthode du gradient à pas optimal	15
1.2.3 Méthode du gradient conjugué	16
1.2.4 Comparaison pour des différents valeurs de n :	17
2 Optimisation sous contraintes	18
2.1 Discrétisation et résolution directe	18
2.2 Méthode du gradient projeté	23
2.3 Méthode de pénalisation :	30
2.4 Méthode d'Uzawa	32

Liste des codes

1.1	projection='3d'	9
1.2	vérification que la matrice A est symétrique définie positive en calculant les valeurs propres pour $n = 2$	11
1.3	la solution théorique du problème (4) dans le cas $n = 2$	12
2.1	notre modification	23
2.2	la fonction projK()	25

Table des figures

1.1	projection = '3d'	9
1.2	Affichage de vecteur U pour un rho egale 0.1	13
1.3	Affichage de vecteur U pour un rho egale 0.001	14
1.4	Affichage de vecteur U pour un rho egale 0.01	14
1.5	Affichage de vecteur U avec des pas optimales	15
1.6	Affichage de vecteur U avec des pas conjugué	16
1.7	Teste pour chacune des trois méthodes	17
1.8	Comparaison à l'aide d'un graphique la rapidité de convergence de chacune de ces méthodes, ainsi que le temps de calcul	17
2.1	Solution et obstacle pour divers $f(x)$	20
2.2	Affichage de vecteur U avec des pas optimales	22
2.3	Solutions numériques pour différentes valeurs de n	23
2.4	trajectoire de vecteur U à pas fixe	26
2.5	les solutions approchées de U pour différents n et différents ρ	26
2.6	les courbes de niveaux de J_2 ainsi que le champ de vecteurs de gradient de J_2 avec un ρ optimal	27
2.7	les solutions approchées U ainsi que le graphe de la fonction g pour différents n avec un ρ optimal	28
2.8	les courbes de niveaux de J_2 ainsi que le champ de vecteurs de gradient de J_2 avec nouveau $f(x)$	29
2.9	les solutions approchées U ainsi que le graphe de la fonction g pour différents n avec un nouveau $f(x)$	29
2.10	teste de la fonction pour différents η	30
2.11	resultat de 3.2 avec l'algorithme du gradient à pas optimal.	31
2.12	la méthode du gradient à pas xe pour calculer à chaque étape le minimiseur de V à l'aide de l'algorithme d'Uzawa	33

Objectif :

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

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

où

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

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

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

Chapitre 1

Optimisation sans contraintes

On considère le problème de minimisation (1) :

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

où

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

avec $f \in L^2([0, 1])$ et K est un sous ensemble fermé et convexe $H_0^1([0, 1])$ est défini par :

$$H_0^1([0, 1]) := \{u \in H^1([0, 1]) \text{ tq } u(0) = u(1) = 0\}.$$

1- D'après l'équation d'Euler-Lagrange associée à ce problème de minimisation, on a :

$$\frac{\partial J}{\partial u} - \frac{d}{dx} \left(\frac{\partial J}{\partial u'} \right) = 0$$

or

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

donc, on obtient :

$$-f(x) - u''(x) = 0 \implies -u''(x) = f(x)$$

Conclusion

$$-u''(x) = f(x)$$

2- on a $f(x) = 1$

donc on va résoudre une équation différentielle du second ordre tel que $u''(x) = -1$ avec $u(0) = u(1) = 0$

Première intégration :

$$u'(x) = \int (-1) dx = -x + c_1$$

Deuxième intégration :

$$u(x) = \int (-x + c_1) dx = -\frac{x^2}{2} + c_1 x + c_2$$

or

$$u(0) = 0 \longrightarrow c_2 = 0$$

et

$$u(1) = 0 \iff -\frac{1}{2} + c_1 = 0 \iff c_1 = \frac{1}{2}$$

Ainsi, la solution $u_\epsilon(x)$ qui satisfait $u(0) = u(1) = 0$ est donnée par :

$$u_\epsilon(x) = -\frac{x^2}{2} + \frac{x}{2}$$

3- on approche la solution $u(x)$ par une fonction $u_h(x)$ continue, affine par morceaux donnée par :

$$u_h(x) = \sum_{i=0}^n u_i \phi_i(x), \quad \text{pour tout } i \in \{0, 1, \dots, n\}$$

avec $h = \frac{1}{n+1}$ et $x_i = ih$ $\phi_i(x)$ est définie sur l'intervalle $[0,1]$, donnée par :

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

on veut montrer que le problème de minimisation (1) peut être approché par le problème de minimisation quadratique dans \mathbb{R}^n :

$$\min_{u \in \mathbb{R}^n} J_n(u)$$

où

$$J_n(u) = \frac{1}{2} \langle Au, u \rangle - \langle b, u \rangle$$

avec

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}. \quad \text{et} \quad b = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}$$

on a

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

* pour le terme $\frac{1}{2} \int_0^1 u'(x)^2 dx$:

on approche $u(x)$ par $u_h(x)$ càd

$$(u'(x))^2 \approx \left(\sum_{i=1}^n u_i \phi'_i(x) \right)^2$$

et $u'_h(x) = \sum_{i=1}^n u_i \phi'_i(x)$

avec

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

donc on aura :

$$u'_h(x) = \begin{cases} \sum_{i=0}^n u_i \frac{1}{h}, & \text{si } x \in [x_{i-1}, x_i], \\ \sum_{i=0}^n u_i \left(\frac{-1}{h}\right), & \text{si } x \in [x_i, x_{i+1}], \\ 0 & \text{sinon.} \end{cases}$$

or

$$\begin{aligned} (u'_h(x))^2 &= (u_1\phi'_1(x) + u_2\phi'_2(x) + \dots + u_n\phi'_n(x))^2 \\ &= \sum_{i=0}^n (u_i\phi'_i(x))^2 + 2 \sum_{i=0}^n \sum_{j=0}^n u_i u_j \phi'_i(x) \phi'_j(x) \end{aligned}$$

et alors

$$\begin{aligned} \int_{x_{i-1}}^{x_{i+1}} (u'_h(x))^2 dx &= \int_{x_{i-1}}^{x_{i+1}} \sum_{i=0}^n \frac{u_i^2}{h^2} dx - 2 \sum_{i=0}^n \sum_{j=0}^n \int_{x_i}^{x_{i+1}} \frac{u_i u_j}{h^2} dx \\ &= \frac{2}{h} \sum_{i=0}^n u_i^2 - \frac{2}{h} \sum_{i=0}^n \sum_{j=0}^n u_i u_j \end{aligned}$$

* pour le terme $\int_0^1 f(x)u(x)dx$:

on a :

$$\int_0^1 f(x)u(x) dx \approx \sum_{i=1}^n u_i \int_{x_{i-1}}^{x_{i+1}} f(x)\phi_i(x)dx$$

on pose

$$b_i = \int_{x_{i-1}}^{x_{i+1}} f(x)\phi_i(x)dx = \int_{x_{i-1}}^{x_i} f(x)\phi_i(x)dx + \int_{x_i}^{x_{i+1}} f(x)\phi_i(x)dx$$

Rappel :

La méthode du trapèze est une technique d'intégration numérique qui approxime l'intégrale d'une fonction $f(x)$ sur un intervalle $[a, b]$.

Pour un intervalle divisé en n sous-intervalles égaux $[x_i, x_{i+1}]$ avec $h = \frac{b-a}{n}$, la méthode du trapèze se généralise :

$$\int_a^b f(x) dx \approx \frac{h}{2} \left(f(a) + 2 \sum_{i=1}^{n-1} f(a + ih) + f(b) \right)$$

D'après la méthode de trapéze

$$b_i = \frac{x_i - x_{i-1}}{2} (f(x_i)\phi(x_i) + f(x_{i-1})\phi(x_{i-1})) + \frac{x_{i+1} - x_i}{2} (f(x_i)\phi(x_i) + f(x_{i+1})\phi(x_{i+1}))$$

or

$$\begin{cases} \phi(x_{i-1}) = 0 \\ \phi(x_i) = \frac{h(i-i+1)}{h} = 1 \\ \text{et} \\ \phi(x_{i+1}) = 0 \\ \phi(x_i) = \frac{h(i+1-i)}{h} = 1 \end{cases}$$

on obtient alors :

$$b_i = \frac{h(i-i+1)}{2} f(x_i) + \frac{h(i+1-i)}{2} f(x_i) = h f(x_i)$$

donc

$$J(u) \approx \frac{1}{2} \left(\frac{2}{h} \sum_{i=0}^n u_i^2 - \frac{1}{h} \sum_{i=0}^n \sum_{j=0}^n u_i u_j \right) - h \sum_{i=1}^n u_i f(x_i)$$

alors

$$\frac{J(u)}{h} \approx J_n(u) = \frac{1}{2} \left(\frac{2}{h^2} \sum_{i=0}^n u_i^2 - \frac{1}{h^2} \sum_{i=0}^n \sum_{j=0}^n u_i u_j \right) - \sum_{i=1}^n u_i f(x_i)$$

4-

Listing 1.1 – projection='3d'

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Jn
def Jn(u, A, b):
    return 0.5 * np.dot(u.T, np.dot(A, u)) - np.dot(b, u)    #
    ↪ calcule 1/2 uT*A*u - bT*u
# la representation pour n=2
n = 2
h = 1 / (n + 1)
A = (1 / h**2) * np.array([[2, -1], [-1, 2]])
b = np.array([1, 1])
# maillage de points
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
U, V = np.meshgrid(x, y)
Z = np.array([Jn(np.array([u, v]), A, b) for u, v in zip(
U.flatten(), V.flatten())]).reshape(U.shape)
# Representation 3D
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(U, V, Z, cmap='viridis')
ax.set_xlabel('u1')
ax.set_ylabel('u2')
ax.set_zlabel('Jn(u)')
plt.show()
```

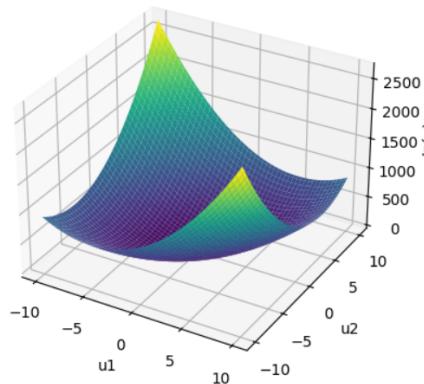


FIGURE 1.1 – projection = '3d'

1.1 Analyse mathématique

1.1.1 Symétrie de la matrice A

La matrice A est définie par :

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \in M_n(\mathbb{R}).$$

Cette matrice est tridiagonale, avec 2 sur la diagonale principale et -1 sur les diagonales adjacentes. Par construction :

$$A^T = A,$$

donc A est symétrique.

1.1.2 Définie positive

Une matrice est définie positive si $x^T Ax > 0$ pour tout vecteur $x \neq 0$. Une condition équivalente est que toutes les valeurs propres de A soient strictement positives :

$$\lambda_i > 0, \quad \forall i.$$

Les valeurs propres λ de A sont les solutions de l'équation :

$$\det(A - \lambda I) = 0,$$

où I est la matrice identité de dimension n .

La matrice $A - \lambda I$ devient :

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

Pour $n = 2$, $h = \frac{1}{3}$, donc $\frac{1}{h^2} = 9$. La matrice A est :

$$A = 9 \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}.$$

L'équation caractéristique est donnée par :

$$\det(A - \lambda I) = \det \left(9 \begin{bmatrix} 2 - \frac{\lambda}{9} & -1 \\ -1 & 2 - \frac{\lambda}{9} \end{bmatrix} \right) = 0.$$

donc on aura :

$$\det(A - \lambda I) = 9[(2 - \frac{\lambda}{9})(2 - \frac{\lambda}{9}) - 1] = \frac{\lambda^2}{9} - 4\lambda + 27 = 0$$

or

$$\delta = 16 - 4 * 3 = 4$$

donc

$$\begin{cases} \lambda_1 = 27 \\ \lambda_2 = 9 \end{cases}$$

Code python :

Listing 1.2 – vérification que la matrice A est symétrique définie positive en calculant les valeurs propres pour n = 2

```
# definir notre matrice A
def matrice_A(n, h):
    A = np.zeros((n, n))
    for i in range(n):
        A[i, i] = 2 / h**2
        if i > 0:
            A[i, i - 1] = -1 / h**2
        if i < n - 1:
            A[i, i + 1] = -1 / h**2
    return A
# verifier si elle est symetrique
def symmetrique(A):
    return np.allclose(A, A.T)          # verification que A = AT
# verifier si elle est definie positive
def definie_positive(A):
    valeurs_propres = np.linalg.eigvals(A)
    return np.all(valeurs_propres > 0), valeurs_propres

# Parametres
n=2
h = 1 / (n + 1)
A = matrice_A(n, h)
print(f"Matrix A for n={n}:")
print(A)
print(f"Symmetrique: {symmetrique(A)}")
is_positive_definite, valeurs_propres = definie_positive(A)
print(f"definie positive: {is_positive_definite}")
print(f"les valeurs propres sont: {valeurs_propres}\n")

for n in [5, 10, 20]:
    h = 1 / (n + 1)
    A = matrice_A(n, h)
    print(f"Matrix A for n={n}:")
    print(A)
    print(f"Symmetrique: {symmetrique(A)}")
    is_positive_definite, valeurs_propres = definie_positive(A)
    print(f"definie positive: {is_positive_definite}")
```

Réultat du code :

```
... Matrix A for n=2:
[[18. -9.]
 [-9. 18.]]
Symmetrique: True
definie positive: True
les valeurs propres sont: [27.  9.]
```

```

Matrix A for n=5:
[[ 72. -36.  0.  0.  0.]
 [-36.  72. -36.  0.  0.]
 [ 0. -36.  72. -36.  0.]
 [ 0.  0. -36.  72. -36.]
 [ 0.  0.  0. -36.  72.]]
Symmetrique: True
definie positive: True
Matrix A for n=10:
[[ 242. -121.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-121.  242. -121.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -121.  242. -121.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -121.  242. -121.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -121.  242. -121.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -121.  242. -121.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -121.  242. -121.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -121.  242. -121.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -121.  242. -121.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -121.  242. -121.]
 ...
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -441.  882.]]
Symmetrique: True
definie positive: True

```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

1.1.3 Solution théorique pour $n = 2$

Le problème est de minimiser :

$$J_2(u) = \frac{1}{2}u^T Au - b^T u,$$

où $u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$,

$$A = 9 \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

La solution est donnée par :

$$u^* = A^{-1}b.$$

Listing 1.3 – la solution théorique du problème (4) dans le cas $n = 2$

```

n = 2
h = 1 / (n + 1)
A = matrice_A(n, h)
b = np.array([1, 1])
u_min = np.linalg.solve(A, b)
print(f"Solution theorique pour n=2 et b={b} : u* = {u_min}")

```

Resultat du code :

Solution théorique pour n=2 et b=[1 1] : u* = [0.11111111 0.11111111]

1.2 Affichage des courbes et leurs interprétations :

1.2.1 La méthode du gradient à pas fixe

```

import numpy as np
import matplotlib.pyplot as plt
h = 1 / 3 # h pour n=2
A = 1/(h**2) * np.array([[2, -1], [-1, 2]]) # Matrice A
b = np.array([1, 1])
u0 = np.transpose(np.array([-1, 2]))
rho = 0.1

def J2(u):
    return 0.5 * np.dot(u.T, np.dot(A, u)) - np.dot(b, u)

def grad_J2(u,A,b):
    return np.dot(A,u) - b

def gradient_fixe(rho, u0) :
    max_iter = 1000
    r = Tol = 10**(-6)
    k = 0
    u = u0.copy()
    trajectory = [u.copy()]
    while r > Tol and k <= max_iter :
        grad = grad_J2(u,A,b)
        direction = -grad
        u_next = u + rho * direction
        trajectory.append(u_next.copy())
        r = np.linalg.norm(u_next-u)
        print(f'U_min = {u_next} , r = {r} , Max_iteration = {k}')
        k += 1
        u=u_next
    return np.array(trajectory)

x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
U1, U2 = np.meshgrid(x, y)
# Calculer les valeurs de J2 pour chaque point de la grille
Z = np.zeros_like(U1)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i, j], U2[i, j]])
        Z[i, j] = J2(u)
# Calculer les gradients pour la grille
grad_U1, grad_U2 = np.zeros_like(U1), np.zeros_like(U2)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i, j], U2[i, j]])
        grad = grad_J2(u,A,b)
        grad_U1[i, j], grad_U2[i, j] = grad[0], grad[1]

trajectory = gradient_fixe(rho, u0)
# Tracer les courbes de niveau
plt.figure(figsize=(10, 6))
plt.contour(U1, U2, Z, levels=20, cmap='viridis', alpha=0.7)
plt.quiver(U1, U2, grad_U1, grad_U2, color="white", scale=200, alpha=0.5)
# Trajectoire
trajectory_u1, trajectory_u2 = trajectory[:, 0], trajectory[:, 1]
plt.plot(trajectory_u1, trajectory_u2, marker='o', color='blue', label="Trajectoire (u^(k))")
plt.title("Courbes de niveau, champ de gradient et trajectoire de descente")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.colorbar(label="$J_2(u)$")
plt.legend()
plt.grid()
plt.show()

```

✓ 1.2s

Résultat du code :

```

...
U_min = [ 2.7 -2.4] , r = 5.748912940721924 , Max_iteration = 0
U_min = [-4.21 -4.45] , r = 9.736986186700689 , Max_iteration = 1
U_min = [ 7.481 -2.58] , r = 16.552663380858082 , Max_iteration = 2
U_min = [-12.417 -12.6393] , r = 28.13952649370653 , Max_iteration = 3
U_min = [ 21.40897 -21.18674] , r = 47.83719503192575 , Max_iteration = 4
U_min = [-36.09524 -36.3174651] , r = 81.3232151542304 , Max_iteration = 5
U_min = [ 61.6619121 -61.4396898] , r = 138.2494936421914 , Max_iteration = 6
U_min = [-104.525205 -104.74747273] , r = 235.02413919172542 , Max_iteration = 7
U_min = [ 177.9929258 -177.77970363] , r = 399.5410366259332 , Max_iteration = 8
U_min = [-302.28797393 -302.51019168] , r = 679.2197622640865 , Max_iteration = 9
U_min = [ 514.1895573 -513.9673335] , r = 1154.6735958489471 , Max_iteration = 10
U_min = [-873.82224473 -874.04446696] , r = 1962.9451129432105 , Max_iteration = 11
U_min = [ 1485.79781605 -1485.57559382] , r = 3337.0066920834587 , Max_iteration = 12
U_min = [-2525.55628728 -2525.778595] , r = 5672.91137640586 , Max_iteration = 13
U_min = [ 4293.74568838 -4293.52346615] , r = 9643.94933988998 , Max_iteration = 14
U_min = [-7299.06767024 -7299.28989246] , r = 16394.713877812996 , Max_iteration = 15
U_min = [ 12408.71503941 -12408.49281781] , r = 27871.013592382095 , Max_iteration = 16
U_min = [-20000.00000000 -20000.00000000] , r = 47380.723066556 , Max_iteration = 17
U_min = [-35660.97640368 -35660.70000000] , r = 80542.520000000 , Max_iteration = 18
U_min = [-60951.3599886 -60951.5321002] , r = 136590.289778000 , Max_iteration = 19
U_min = [-103638.01198062 -103637.7697584] , r = 232781.4026240993 , Max_iteration = 20
U_min = [-176184.32036706 -176184.54258928] , r = 395720.5374609683 , Max_iteration = 21
U_min = [-299513.644624 -299513.42240178] , r = 672738.513683647 , Max_iteration = 22
U_min = [-509172.89586081 -509173.11808303] , r = 1143655.4732622001 , Max_iteration = 23
U_min = [ 865594.22296337 -865594.00074115] , r = 1944214.3045437406 , Max_iteration = 24

```

Output was truncated. View as a scrollable element or open in a [text editor](#). Adjust cell output settings.

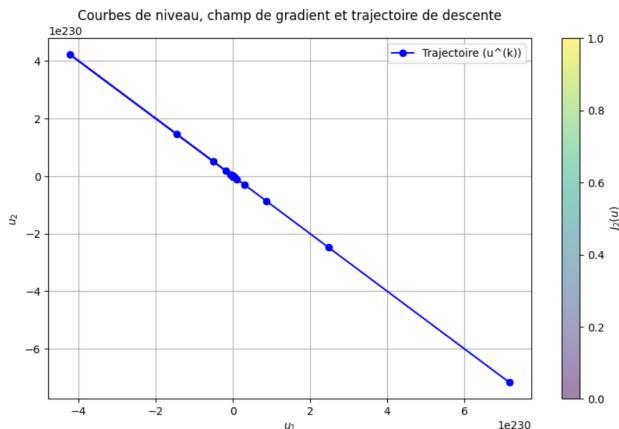


FIGURE 1.2 – Affichage de vecteur U pour un rho égale 0.1

Interprétation :

Les résultats montrent que l'algorithme de descente de gradient fixe diverge au lieu de converger vers le minimum. À chaque itération, les valeurs de u_{min} augmentent et le résidu r , qui mesure la distance entre deux itérations successives, croît rapidement jusqu'à atteindre l'infini.

Ce comportement indique que le pas fixé ($\rho = 0.1$) est trop grand donc c'est un mauvais choix de ρ .

Pour corriger ce problème, il faut réduire ρ à une valeur plus petite ou d'utiliser une méthode adaptative pour ajuster le pas en fonction du gradient.

* pour $\rho = 0.001$

Réultat du code :

```
...
U_min = [-0.963  1.956] , r = 0.05748912940721929 , Max_iteration = 0
U_min = [-0.927062  1.913125] , r = 0.05594466434790722 , Max_iteration = 1
U_min = [-0.89215676  1.87134519] , r = 0.054441970994674224 , Max_iteration = 2
U_min = [-0.85825583  1.83063157] , r = 0.052979922124205825 , Max_iteration = 3
U_min = [-0.82533154  1.79099559] , r = 0.051557420951566395 , Max_iteration = 4
U_min = [-0.79335697  1.75229071] , r = 0.050173400308286854 , Max_iteration = 5
U_min = [-0.76230593  1.71466926] , r = 0.04882682184263752 , Max_iteration = 6
U_min = [-0.73215294  1.67788554] , r = 0.04751667524148816 , Max_iteration = 7
U_min = [-0.70287322  1.64209423] , r = 0.04624197743717303 , Max_iteration = 8
U_min = [-0.67444265  1.60721067] , r = 0.04500177265079033 , Max_iteration = 9
U_min = [-0.64682779  1.57221089] , r = 0.043795112831538762 , Max_iteration = 10
U_min = [-0.62000581  1.54007156] , r = 0.042621140738493365 , Max_iteration = 11
U_min = [-0.59401452  1.50776995] , r = 0.0414789282437392 , Max_iteration = 12
U_min = [-0.56875233  1.47628396] , r = 0.040367613545205665 , Max_iteration = 13
U_min = [-0.54422882  1.44559288] , r = 0.039286422507568565 , Max_iteration = 14
U_min = [-0.52042179  1.41567336] , r = 0.038234483518279155 , Max_iteration = 15
U_min = [-0.49731314  1.38569745] , r = 0.0372110268809595105 , Max_iteration = 16
U_min = [-0.47488294  1.3580745] , r = 0.03621528422143434 , Max_iteration = 17
U_min = [-0.45211237  1.33039521] , r = 0.0352450597915465686 , Max_iteration = 18
U_min = [-0.43198316  1.3033308] , r = 0.034303970524740324 , Max_iteration = 19
U_min = [-0.41147740  1.276983] , r = 0.03338696425344248 , Max_iteration = 20
U_min = [-0.39157804  1.25129401] , r = 0.03249480041735186 , Max_iteration = 21
U_min = [-0.37226799  1.22624651] , r = 0.031626808992762016 , Max_iteration = 22
U_min = [-0.35353095  1.20182367] , r = 0.03078233778847298 , Max_iteration = 23
U_min = [-0.33535098  1.17800906] , r = 0.02996075260846668 , Max_iteration = 24
...
U_min = [0.11119109  0.11119109] , r = 1.02723769902599e-06 , Max_iteration = 938
U_min = [0.11119037  0.11119037] , r = 1.0179925597332716e-06 , Max_iteration = 939
U_min = [0.11118966  0.11118966] , r = 1.0088306266955419e-06 , Max_iteration = 940
U_min = [0.11118895  0.11118895] , r = 9.997511510473897e-07 , Max_iteration = 941
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

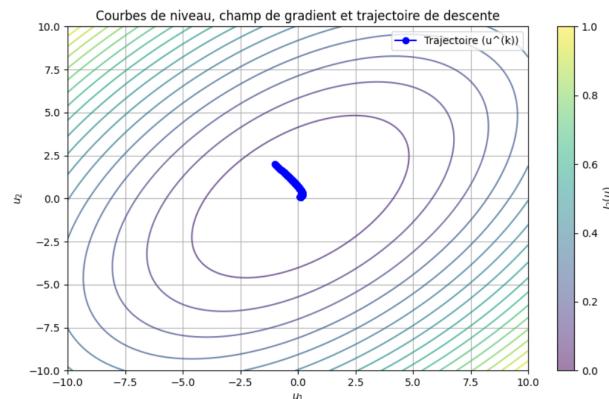


FIGURE 1.3 – Affichage de vecteur U pour un rho égale 0.001

Interprétation :

Les résultats montrent que l'algorithme de descente de gradient fixe converge vers le minimum, mais cette convergence est très lente.

Ce comportement indique que le pas fixé ($\rho = 0.001$) est trop petit donc c'est un mauvais choix de ρ .

Pour corriger ce problème, il faut augmenter ρ à une valeur plus grande.

* pour $\rho = 0.01$

Réultat du code :

```
...
U_min = [-0.63  1.56] , r = 0.5748912940721924 , Max_iteration = 0
U_min = [-0.3662  1.2325] , r = 0.4205314375882023 , Max_iteration = 1
U_min = [-0.179359  0.987692] , r = 0.30796187449910095 , Max_iteration = 2
U_min = [-0.0481821  0.80376513] , r = 0.22591253307775308 , Max_iteration = 3
U_min = [0.04282954  0.66475102] , r = 0.166156679093085 , Max_iteration = 4
U_min = [0.18497481  0.55895049] , r = 0.1226883491681419 , Max_iteration = 5
U_min = [0.14636279  0.47778471] , r = 0.0921124782845363 , Max_iteration = 6
U_min = [0.17301808  0.41495611] , r = 0.06824996448579 , Max_iteration = 7
U_min = [0.18922088  0.36583564] , r = 0.05172379879781648 , Max_iteration = 8
U_min = [0.19808633  0.3270151] , r = 0.039819972159207115 , Max_iteration = 9
U_min = [0.20186215  0.29598015] , r = 0.031263794776671364 , Max_iteration = 10
U_min = [0.20216517  0.27087132] , r = 0.02511066261279433 , Max_iteration = 11
U_min = [0.20015386  0.25030935] , r = 0.0200660107722301434 , Max_iteration = 12
U_min = [0.19665401  0.23326751] , r = 0.017397502954237126 , Max_iteration = 13
U_min = [0.19225036  0.21897822] , r = 0.01495245943644598 , Max_iteration = 14
U_min = [0.18735334  0.20686467] , r = 0.013065943415707976 , Max_iteration = 15
U_min = [0.18224756  0.19649083] , r = 0.011562247281459615 , Max_iteration = 16
U_min = [0.17712717  0.18752476] , r = 0.010325151342268456 , Max_iteration = 17
U_min = [0.17212151  0.17971175] , r = 0.00927899612609788 , Max_iteration = 18
U_min = [0.16731369  0.17285457] , r = 0.008374722883700768 , Max_iteration = 19
U_min = [0.16275414  0.16679981] , r = 0.007580217867636467 , Max_iteration = 20
U_min = [0.1584793  0.16142304] , r = 0.006874011395063932 , Max_iteration = 21
U_min = [0.15447372  0.15662922] , r = 0.006241263185728941 , Max_iteration = 22
U_min = [0.15076508  0.15233859] , r = 0.005671284658790989 , Max_iteration = 23
U_min = [0.14733784  0.1484865] , r = 0.005156023311175495 , Max_iteration = 24
...
U_min = [0.11112026  0.11112026] , r = 1.2800265628896906e-06 , Max_iteration = 112
U_min = [0.11111944  0.11111944] , r = 1.1648241722256933e-06 , Max_iteration = 113
U_min = [0.11111869  0.11111869] , r = 1.059989967246938e-06 , Max_iteration = 114
U_min = [0.11111801  0.11111801] , r = 9.645090879246696e-07 , Max_iteration = 115
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

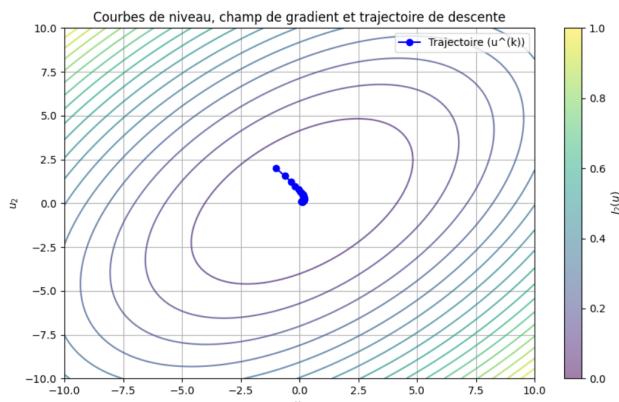


FIGURE 1.4 – Affichage de vecteur U pour un rho égale 0.01

Interprétation :

L'utilisation de $\rho = 0.01$ a permis une convergence parfaite de l'algorithme vers la solution théorique avec un nombre d'itérations très faible.

1.2.2 La méthode du gradient à pas optimal

```

from math import *
import numpy as np
import matplotlib.pyplot as plt
h = 1 / 3 # h pour n=2
A = 1/(h**2) * np.array([[2, -1], [-1, 2]]) # Matrice A
b = np.array([1, 1])
u0 = np.transpose(np.array([-1,2]))
def J2(u):
    return 0.5 * np.dot(u.T, np.dot(A , u)) - np.dot(b, u)
# calculer gradient
def grad_J2(u,A,b):
    return np.dot(A,u) + b
# la fonction section d'ordre 2
def section_doree(u,direction,a,b):
    err = b - a
    Tol = 10**-6
    phi1 = (1+sqrt(5))/2
    k=0
    while err >= Tol :
        a_next = a + (b-a)/(phi1**2)
        b_next = a + (b-a)/phi1
        J2_a_next = J2(u+a_next*direction)
        J2_b_next = J2(u+b_next*direction)
        if J2_a_next<J2_b_next :
            a=a_next
        elif J2_a_next> J2_b_next :
            b=b_next
        else :
            b=b_next
            a=a_next
        k+= 1
        err = b - a
    rho_min = (a+b)/2
    return rho_min

```

```

def gradient_optimale(u0):
    max_iteration = 1000
    k = 0
    r = Tol = 10**(-6)
    u = u0.copy()
    trajectory = [u.copy()]
    while r >= Tol and k <= max_iteration:
        grad = grad_J2(u,A,b)
        direction = -grad
        a = 0
        bb = 1
        rho = -direction.dot(grad)/direction.dot(direction)
        u_next = u + rho*direction
        trajectory.append(u.next.copy())
        r= np.linalg.norm(u.next-u)
        print(f"U_min = {u.next} , r = {r} , Max_iteration = {k} , rho = {rho}")
        u = u.next
    return np.array(trajectory)
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
U1, U2 = np.meshgrid(x, y)
z = np.zeros_like(U1)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i,j], U2[i,j]])
        z[i,j] = J2(u)
# Calculer les gradients pour la grille
grad_U1, grad_U2 = np.gradient(z), np.zeros_like(U2)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i,j], U2[i,j]])
        grad = grad_J2(u,A,b)
        grad_U1[i,j], grad_U2[i,j] = grad[0], grad[1]
trajectory = gradient_optimale(u0)

# Tracer les courbes de niveau
plt.figure(figsize=(10, 6))
plt.contour(U1, U2, z, levels=20, cmap='viridis', alpha=0.7)
plt.quiver(U1, U2, grad_U1, grad_U2, color='white', scale=100, alpha=0.5)
# Trajectoire
plt.plot(trajectory[:,0], trajectory[:,1], marker='o', color='blue', label="Trajectoire ({\nu}(k))")
plt.plot(trajectory[0], trajectory[1], marker='o', color='red', label="U_min")
plt.title("Courbes de niveau, champ de gradient et trajectoire de descente")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.legend()
plt.grid()
plt.show()
    
```

Résultat du code :

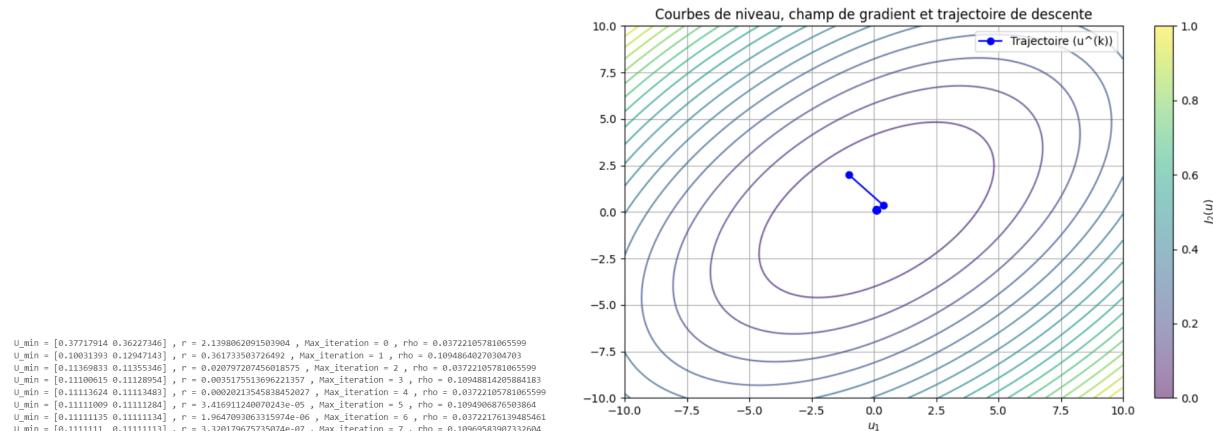


FIGURE 1.5 – Affichage de vecteur U avec des pas optimales

Interprétation :

L'algorithme du gradient à pas optimal est une variante de la méthode du gradient qui vise à optimiser le choix du pas (ou taux d'apprentissage) à chaque itération. Selon les résultats, l'algorithme de descente optimale du gradient converge vers le minimum, avec des étapes moins nombreuses que la méthode de pas fixe .

1.2.3 Méthode du gradient conjugué

```

import numpy as np
import matplotlib.pyplot as plt
h = 1 / 3 # h pour n=2
A = 1/(h**2) * np.array([[2, -1], [-1, 2]]) # Matrice A
b = np.array([1, 1])
u0 = np.transpose(np.array([-1, 2]))

def J2(u):
    return 0.5 * np.dot(u.T, np.dot(A, u)) - np.dot(b, u)

def grad_J2(u,A,b):
    return np.dot(A,u) - b

def gradient_conjugue(u0):
    max_iteration = 1000
    k = 0
    r = Tol = 10**(-6)
    u = u0.copy()
    grad = grad_J2(u,A,b)
    direction = -grad
    trajectory = [u.copy()]
    while r >= Tol and k <= max_iteration:
        rho = -np.dot(grad,direction)/(np.dot(np.dot(A,direction),direction))
        u.next = u + rho*direction
        beta = np.linalg.norm(grad_J2(u.next,A,b))**2/np.linalg.norm(grad)**2
        grad = grad_J2(u.next,A,b)
        direction = -grad + beta*direction
        trajectory.append(u.next.copy())
        r= np.linalg.norm(u.next-u)
        print(f'U_min = {u.next} , r = {r} , Max_iteration = {k} , rho = {rho}')
        u = u.next
        k += 1
    return np.array(trajectory)
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
U1, U2 = np.meshgrid(x, y)

# calculer les valeurs de J2 pour chaque point de la grille
Z = np.zeros_like(U1)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i, j], U2[i, j]])
        Z[i, j] = J2(u)

# calculer les gradients pour la grille
grad_U1, grad_U2 = np.zeros_like(U1), np.zeros_like(U2)
for i in range(U1.shape[0]):
    for j in range(U1.shape[1]):
        u = np.array([U1[i, j], U2[i, j]])
        grad = grad_J2(u,A,b)
        grad_U1[i, j], grad_U2[i, j] = grad[0], grad[1]

trajectory = gradient_conjugue(u0)

# Tracer les courbes de niveau
plt.figure(figsize=(10, 6))
plt.contour(U1, U2, Z, levels=20, cmap='viridis', alpha=0.7)
plt.quiver(U1, U2, grad_U1, grad_U2, color="white", scale=200, alpha=0.5)

# Trajectoire
trajectory_u1, trajectory_u2 = trajectory[:, 0], trajectory[:, 1]
plt.plot(trajectory_u1, trajectory_u2, marker='o', color='blue', label="Trajectoire (u^(k))")

plt.title("Courbes de niveau, champ de gradient et trajectoire de descente")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.colorbar(label="J2(u)")
plt.legend()
plt.grid()
plt.show()
    
```

Résultat du code :

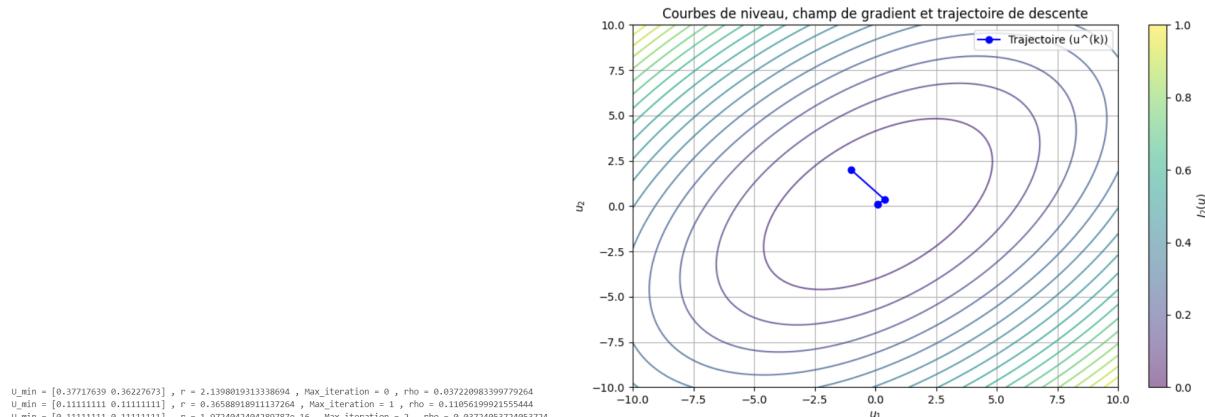


FIGURE 1.6 – Affichage de vecteur \mathbf{U} avec des pas conjugué

Interprétation :

Le gradient conjugué montre ici sa puissance en convergeant très rapidement vers le minimum de la fonction $J_2(u)$, avec un nombre d'itérations réduit et une trajectoire efficace.

La solution obtenue u_{min} est la meilleure approximation du minimum dans le cadre de la précision numérique. Ce comportement reflète l'avantage du gradient conjugué pour résoudre des problèmes quadratiques dans des espaces bien conditionnés.

1.2.4 Comparaison pour des différents valeurs de n :

Courbes des différentes méthodes :

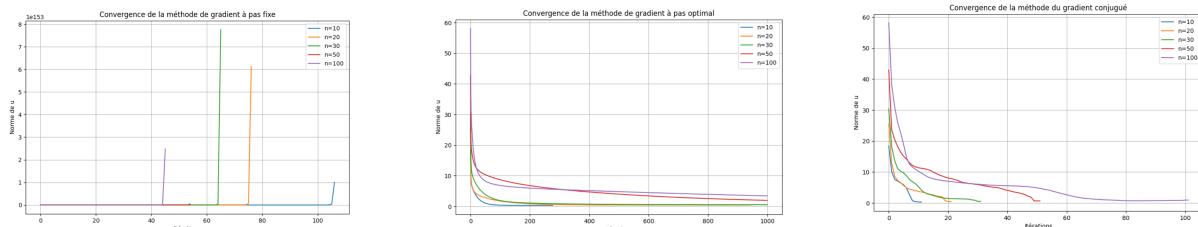


FIGURE 1.7 – Teste pour chacune des trois méthodes

Comparaison des performances :

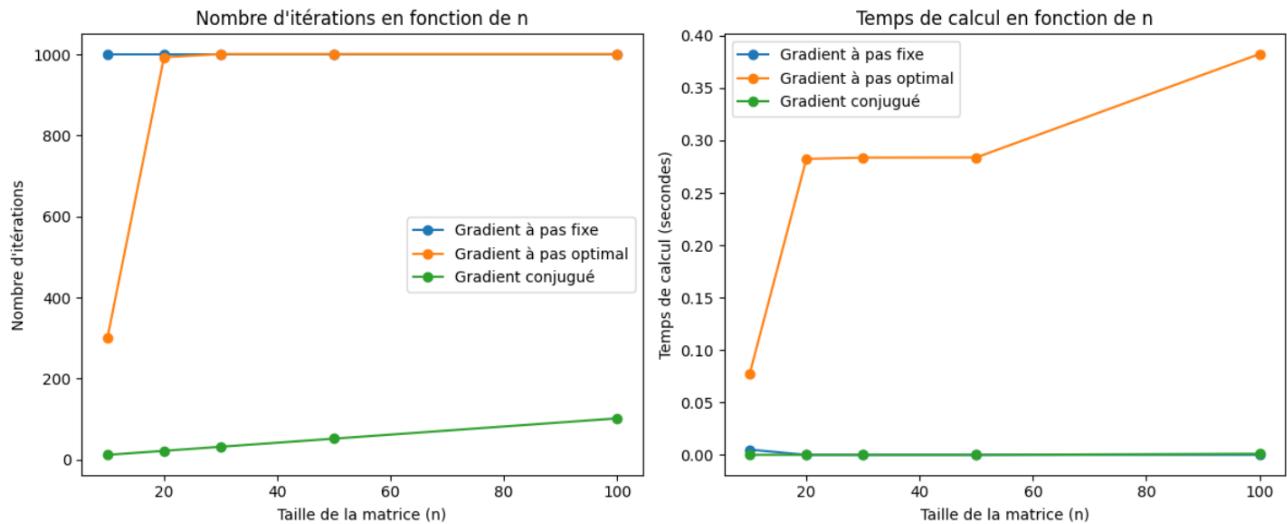


FIGURE 1.8 – Comparaison à l'aide d'un graphique la rapidité de convergence de chacune de ces méthodes, ainsi que le temps de calcul

Interprétation :

- pour le temps de calcul :
 - La méthode à pas fixe nécessite de nombreuses itérations pour converger.
 - Le gradient à pas optimal est plus rapide, mais légèrement dépassé par le gradient conjugué.
 - Le gradient conjugué converge en très peu d'itérations.
- Rapidité de convergence :
 - La méthode du gradient conjugué est la plus rapide.
 - La méthode à pas optimal est efficace, mais légèrement moins performante que le gradient conjugué.
 - La méthode à pas fixe est la moins rapide.

Chapitre 2

Optimisation sous contraintes

2.1 Discrétisation et résolution directe

1.1- Le problème (6) est une version discrète du problème d'optimisation initial (5), obtenu en utilisant une discrétisation de l'intervalle $[0, 1]$ et une approximation de la solution.

En particulier, on a défini la fonction $u_h(x)$ comme une combinaison linéaire des fonctions de base $\phi_i(x)$ associées à la discrétisation, avec u_i étant les valeurs approchées aux points de discrétisation.

On a $u_h(x)$ est une fonction continue, affine par morceaux. Donc d'après l'interpolation linéaire, on obtient :

$$\begin{aligned}\phi_i(x) &= \frac{x_{i+1} - x}{h} \\ \phi_{i+1}(x) &= \frac{x - x_i}{h}\end{aligned}$$

alors

$$\begin{aligned}u_h(x) &= u_i\phi_i(x) + u_{i+1}\phi_{i+1}(x) \\ &= u_i \frac{x_{i+1} - x}{h} + u_{i+1} \frac{x - x_i}{h} \\ &= \frac{u_i x_{i+1} - u_i x + u_{i+1} x - u_{i+1} x_i}{h} \\ &= u_i + \frac{u_{i+1} - u_i}{h}(x - x_i)\end{aligned}$$

d'où $\forall i = 0, \dots, n$

$$u'_h(x) = \frac{u_{i+1} - u_i}{h}$$

sous l'intervalle $[x_i, x_{i+1}]$, on aura :

$$\int_{x_i}^{x_{i+1}} \frac{1}{2} (u'_h(x))^2 dx = \int_{x_i}^{x_{i+1}} \frac{1}{2} \left(\frac{u_{i+1} - u_i}{h} \right)^2 dx = \frac{1}{2} \frac{(u_{i+1} - u_i)^2}{h}$$

d'où

$$\int_{x_i}^{x_{i+1}} \frac{1}{2} (u'(x))^2 dx \approx \frac{1}{2h} \sum_{i=0}^n (u_{i+1} - u_i)^2.$$

pour le terme $u(x)f(x)$:

$$\int_0^1 f(x)u(x)dx \approx \int_{x_i}^{x_{i+1}} f(x)u_h(x)dx$$

Sur le sous-intervalle $[x_i, x_{i+1}]$, $\varphi_i(x)$ est linéaire, et nous approchons $f(x)$ par des valeurs discrètes aux points x_i, x_{i+1} .

En utilisant la méthode des trapèzes :

$$\int_{x_i}^{x_{i+1}} f(x) \phi_i(x) dx \approx \frac{h}{2} (f(x_{i+1}) \phi_i(x_{i+1}) + f(x+i) \phi_i(x+i)).$$

La fonction $J_n(u)$ est définie par :

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

En résumé, le problème (6) peut être exprimé sous la forme :

$$\min_{u \in K_n} J_n(u),$$

où K_n est défini comme :

$$K_n = \{v \in \mathbb{R}^n \mid v_i \geq g(x_i), \forall 1 \leq i \leq n\},$$

et $J_n(u)$ est la version discrète de la fonctionnelle $J(u)$.

1.2- Existance :

$J_n(u)$ est coercive, en effet :

$$J_n(u) = \frac{1}{2} \sum_{i=0}^n \frac{(u_{i+1} - u_i)^2}{h} - h \sum_{i=0}^n \frac{f(x_i) u_i + f(x_{i+1}) u_{i+1}}{2} > \frac{1}{2} \sum_{i=0}^n \frac{(u_{i+1} - u_i)^2}{h}$$

et lorsque $\|u\| \rightarrow \infty$, ce terme tend également vers l'infini. d'où

$$J_n(u) \rightarrow \infty \quad \text{quand} \quad \|u\| \rightarrow \infty.$$

unicité :

La matrice A associée est tridiagonale et définie positive :

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

La matrice A est symétrique définie positive.

et on a $J_n(u)$ est une fonction quadratique.

$\Rightarrow J$ est strictement convexe.

Conclusion : le problème (7) admet une unique solution.

1.3-

```
# Définir les fonctionnelles Jn et son gradient DJn
def J(U, A, fx):
    h = 1 / (n + 1)
    term1 = 1/2 * np.dot(U.T, np.dot(A, U))
    term2 = -h*np.dot(fx.T, U)
    return term1 + term2

def DJ(U, A, fx):
    h = 1 / (n + 1)
    grad_term1 = np.dot(A, U)
    grad_term2 = -h*fx
    return grad_term1 + grad_term2
```

1.4-

```
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# Définir les fonctions f(x) et g(x)
def f_constant(x):
    """Fonction f(x) = 1."""
    return np.ones_like(x)

def f_sin(x):
    """Fonction f(x) = pi^2 * sin(nx)."""
    return (np.pi**2) * np.sin(np.pi * x)

def g(x):
    """Fonction g(x) = max(1.5 - 20(x - 0.6)^2, 0)."""
    return np.maximum(1.5 - 20 * (x - 0.6)**2, 0)

# Définir les fonctionnelles Jn et son gradient DJn
def J(U, A, fx):
    h = 1 / (n + 1)
    term1 = 1/2 * np.dot(U.T, np.dot(A, U))
    term2 = -h*np.dot(fx.T, U)
    return term1 + term2

def DJ(U, A, fx):
    h = 1 / (n + 1)
    grad_term1 = np.dot(A, U)
    grad_term2 = -h*fx
    return grad_term1 + grad_term2

# Paramètres
n = 100
x = np.linspace(0, 1, n + 2) # Points de discréttisation incluant les bords
xv = x[1:-1] # Points intérieurs
gv = g(xv) # Obstacle g(x)
```

```
# Construction de la matrice A (matrice tridiagonale)
A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
fv_constant = f_constant(xv)
Jf_constant = lambda u: J(U, A, fv_constant)
DJf_constant = lambda u: DJ(U, A, fv_constant)
constraints = ({
    'type': 'ineq',
    'fun': lambda u: u - gv,
    'jac': lambda u: np.eye(n)
})
u0 = np.zeros(n)
res_constant = minimize(Jf_constant, u0, method='SLSQP', jac=DJf_constant,
                        constraints=constraints, tol=1e-8,
                        options={'disp': True, 'maxiter': 5000})
u_sol_constant = res_constant.x
# Résolution pour f(x) = pi^2 sin(nx)
fv_sin = f_sin(xv)
Jf_sin = lambda u: J(U, A, fv_sin)
DJf_sin = lambda u: DJ(U, A, fv_sin)
res_sin = minimize(Jf_sin, u0, method='SLSQP', jac=DJf_sin,
                    constraints=constraints, tol=1e-8,
                    options={'disp': True, 'maxiter': 5000})
u_sol_sin = res_sin.x
# Tracer les résultats
plt.figure(figsize=(10, 6))
plt.plot(xv, u_sol_constant, label="Solution u(x) pour f(x) = 1", color="blue")
plt.plot(xv, u_sol_sin, label="Solution u(x) pour f(x) = pi^2 sin(pi x)", color="green")
plt.plot(xv, gv, label="Obstacle g(x)", color="red", linestyle="--")
plt.xlabel("x")
plt.ylabel("u(x) / g(x)")
plt.title("Solutions du problème pour différents f(x) avec l'obstacle g(x)")
plt.legend()
plt.grid()
plt.show()
```

Réultat du code :

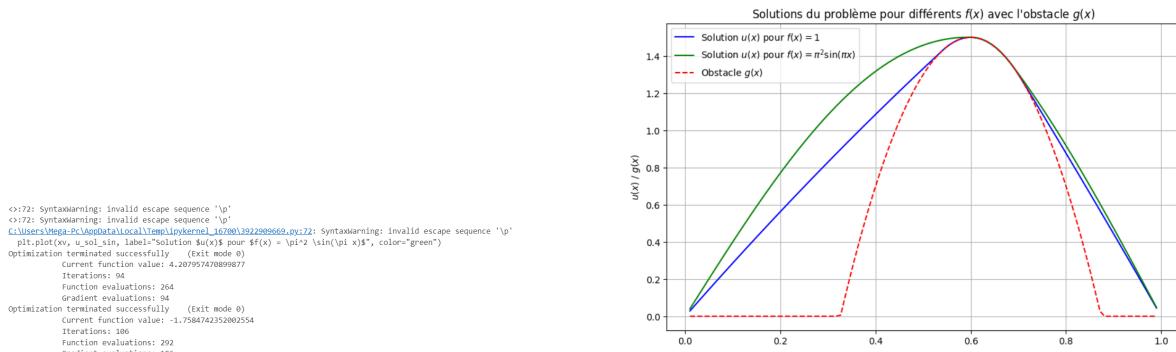


FIGURE 2.1 – Solution et obstacle pour divers $f(x)$

1.5- $\forall x \in]0, 1[$, on a $u \in H_0^1([0, 1])$, de plus u est le minimum de J . donc $u(x)$ est la solution du problème de minimisation sous contraintes, cette condition aux bords est bien satisfaite par la solution.
d'où $u(0) = u(1) = 0$.

vérification de l'égalité 8a :

en appliquant le KKT :

$$\begin{cases} J(u) = \int_0^1 \left(\frac{1}{2}u'(x)^2 - f(x)u(x) \right) dx \\ c(x) = g(x) - u(x) < 0 \end{cases}$$

le grangien : il existe λ un multiplicateur de lagrange tq

$$L(x, \alpha) = \int_0^1 \frac{1}{2}(u'(x))^2 - f(x)u(x) + \alpha(x)(g(x) - u(x)) dx$$

d'après d'Euler-Lagrange on a

$$\frac{\partial J}{\partial u} - \frac{d}{dx} \left(\frac{\partial J}{\partial u'} \right) = 0$$

or

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

donc, on obtient :

$$-f(x) - u''(x) - \alpha(x) = 0 \implies -u''(x) = f(x) + \alpha(x)$$

concusion

$$-u''(x) \geq f(x)$$

vérification de l'égalité 8b :

d'après le question 1.5) on a la courbe de u est toujours d au dessus de g donc

$$u(x) \geq g(x)$$

vérification de l'égalité 8c :

$$(-u''(x) - f(x))(u(x) - g(x)) = 0$$

donc on aura

$$-u''(x) - f(x) = 0 \quad \text{ou} \quad u(x) - g(x) = 0$$

$$\iff -u''(x) = f(x) \quad \text{ou} \quad u(x) = g(x)$$

1.6- pour la tolerance par defaut

```

import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
# definir f(x) et g(x)
def f(x):
    return np.ones_like(x)
def g(x):
    return np.maximum(1.5 - 20 * (x - 0.6)**2, 0)
# definir les fonctions h et D
def D(u, A, f_val):
    h = 1/(n+1)
    u_xx = (u[1:-1] - 2 * u[1:-1] + u[2:]) / h**2 # Approximation de u''(x)
    f_val = f(x)
    g_val = g(x)
    prop1 = np.all(u[0] >= 0) # u(0) = u(1) = 0
    prop2 = np.all(u_xx >= f_val) # u''(x) >= f(x) sans tolérance
    prop3 = np.all((u[1:-1] * g_val) + u[0] >= g(x))
    prop4 = np.all((u[1:-1] - f_val)*(u[1:-1] - g_val) == 0)
    if len(violations) > 0:
        print("Pour n = " + str(n), violation détectée aux points : (violations)")
    else:
        print("Pour n = " + str(n), aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.")
    return prop1, prop2, prop3, prop4
n_values = [10, 20, 30, 50, 100]
results = []
for n in n_values:
    x = np.linspace(0, 1, n+1)
    x[0], x[-1] = 0, 1 # points internes
    f = g = lambda x: 0
    A = (n+1) * (A * np.eye(n) - np.diag(np.ones(n-1), -1) - np.diag(np.ones(n-1), 1))
    const = ("type": "Ineq", "fun": lambda u: u - g(x),
             "jac": lambda u: np.gradient(u))
    ob = np.inf
    res = minimize(f, ob, method="SLSQP", jac=D, constraints=const,
                  tol=1e-8, options={"disp": True, "maxiter": 5000}) # Tolerance 1e-8 de prof
    u = np.concatenate(([0], res.x, [0])) # Ajouter les bords
    results[n] = {
        "solution": u,
        "properties": check_properties(u, xv, n, f, g),
        "x": x
    }
    # vérification des violations
    violations = np.sum(x < g(x))
    if len(violations) > 0:
        print("Pour n = " + str(n), violation détectée aux points : (violations)")
    else:
        print("Pour n = " + str(n), aucune violation de u(x) >= g(x).")
    print("Optimization terminated successfully (Exit mode 0)\n"
          "Current function value: " + str(res.fun) + "\n"
          "Iterations: " + str(res.nit) + "\n"
          "Function evaluations: " + str(res.nfev) + "\n"
          "Gradient evaluations: " + str(res.njev) + "\n\n")
print("Pour n = 10, violation détectée aux points : [0.72727273]\n"
      "Pour n = 10, aucune violation de u(x) >= g(x).\n"
      "Optimization terminated successfully (Exit mode 0)\n"
      "Current function value: 4.117121838638147\n"
      "Iterations: 14\n"
      "Function evaluations: 24\n"
      "Gradient evaluations: 14\n\n")
print("Pour n = 20, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.\n"
      "Pour n = 20, aucune violation de u(x) >= g(x).\n"
      "Optimization terminated successfully (Exit mode 0)\n"
      "Current function value: 4.183142677450998\n"
      "Iterations: 23\n"
      "Function evaluations: 49\n"
      "Gradient evaluations: 23\n\n")
print("Pour n = 30, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.\n"
      "Pour n = 30, violation détectée aux points : [0.5483871  0.58064516 0.61290323 0.64516129 0.67741935 0.70967742]\n"
      "Optimization terminated successfully (Exit mode 0)\n"
      "Current function value: 4.205001580144658\n"
      "Iterations: 51\n"
      "Function evaluations: 131\n\n")
...
Propriété 1 (u(0) = u(1) = 0) : True
Propriété 2 (-u''(x) >= f(x)) : False
Propriété 3 (u(x) >= g(x)) : True
Propriété 4 (complémentarité) : False

```

```

# Tracer g(x)
x_fine = np.linspace(0, 1, 1000)
plt.plot(x_fine, g(x_fine), 'k--', label="Obstacle g(x)")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.legend()
plt.title("Solutions numériques pour différentes valeurs de n")
plt.show()

```

Réultat du code :

```

... Optimization terminated successfully (Exit mode 0)
Current function value: 4.117121838638147
Iterations: 14
Function evaluations: 24
Gradient evaluations: 14
Pour n = 10, violation détectée aux points : [0.72727273]
Pour n = 10, aucune violation de u(x) >= g(x).
Optimization terminated successfully (Exit mode 0)
Current function value: 4.183142677450998
Iterations: 23
Function evaluations: 49
Gradient evaluations: 23
Pour n = 20, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.
Pour n = 20, aucune violation de u(x) >= g(x).
Optimization terminated successfully (Exit mode 0)
Current function value: 4.198275649657233
Iterations: 33
Function evaluations: 75
Gradient evaluations: 33
Pour n = 30, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.
Pour n = 30, violation détectée aux points : [0.5483871  0.58064516 0.61290323 0.64516129 0.67741935 0.70967742]
Optimization terminated successfully (Exit mode 0)
Current function value: 4.205001580144658
Iterations: 51
Function evaluations: 131
...
Propriété 1 (u(0) = u(1) = 0) : True
Propriété 2 (-u''(x) >= f(x)) : False
Propriété 3 (u(x) >= g(x)) : True
Propriété 4 (complémentarité) : False

```

FIGURE 2.2 – Affichage de vecteur U avec des pas optimales

Conclusion :

Observation de violence au niveau des propriétés car il ya des marche d'erreur comme l'exemple : $-u''(x) = 0.99999$ et $f(x) = 1.00000$

pour la tolerance personnalisée :
 dans un cas, j'ai ajouté une tolerance personnalisée et certains cas j'ai modifié la valeur de talerance

Listing 2.1 – notre modification

```
# 1er modification
prop2 = np.all(-u_xx >= f_val - 1e-3)
# 2eme modification
prop4 = np.allclose( (-u_xx - f_val )*(u[1:-1] - g_val) , 0,
                    rtol=1e-2, atol=1e-3)
# 3eme modification
res = minimize(Jf, u0, method='SLSQP', jac=DJf, constraints=
                const,
                tol=1e-12, options={'disp': True, 'maxiter':
                    5000})
```

Résultat du code :

```
...
Optimization terminated successfully  (Exit mode 0)
    Current function value: 4.117121838216883
    Iterations: 15
    Function evaluations: 26
    Gradient evaluations: 15
Pour n = 10, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.
Pour n = 10, aucune violation de u(x) >= g(x).
Optimization terminated successfully  (Exit mode 0)
    Current function value: 4.183142675540745
    Iterations: 25
    Function evaluations: 51
    Gradient evaluations: 25
Pour n = 20, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.
Pour n = 20, aucune violation de u(x) >= g(x).
Optimization terminated successfully  (Exit mode 0)
    Current function value: 4.19827564933328
    Iterations: 34
    Function evaluations: 77
    Gradient evaluations: 34
Pour n = 30, aucune violation de (-u_xx - f_val)*(u[1:-1] - g_val) == 0.
Pour n = 30, aucune violation de u(x) >= g(x).
Optimization terminated successfully  (Exit mode 0)
    Current function value: 4.20590015801402925
    Iterations: 52
    Function evaluations: 132
...
Propriété 1 (u(0) + u(1) = 0) : True
Propriété 2 (u'(x) >= f(x)) : True
Propriété 3 (u(x) >= g(x)) : True
Propriété 4 (complémentarité) : True
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

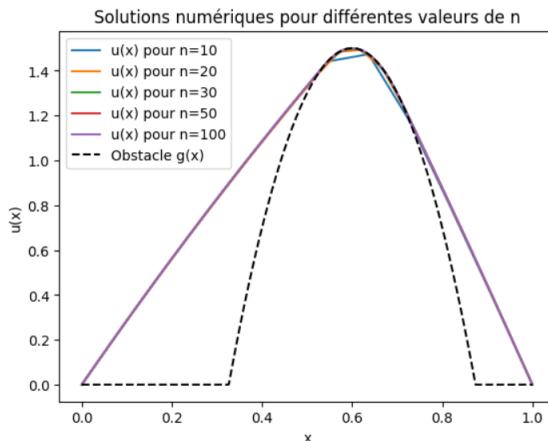


FIGURE 2.3 – Solutions numériques pour différentes valeurs de n

2.2 Méthode du gradient projeté

1- On a l'ensemble K_n est défini comme :

$$K_n = \{\mathbf{v} \in \mathbb{R}^n : v_i \geq g_i(x_i), \forall i = 1, \dots, n\},$$

K_n est non vide :

En effet, on a v_i , un vecteur non nul de \mathbb{R}^n appartenant à K_n , qui satisfait la condition $v_i \geq g_i(x_i)$

convexité :

Soit $\mathbf{v}, \mathbf{w} \in K_n$ et $\alpha \in [0, 1]$, cela signifie que :

$$v_i \geq g_i(x_i) \quad \text{et} \quad w_i \geq g_i(x_i) \quad \forall i = 1, \dots, n$$

on obtient alors :

$$\lambda v_i + (1 - \lambda)w_i \geq \lambda g(x_i) + (1 - \lambda)g(x_i) = g(x_i)$$

D'où K_n est convexe

fermé :

Soit $\{v_k\} \subset K_n$. Cela signifie que

$$(v_k)_i \geq g(x_i) \quad \text{pour tout } k \text{ et tout } i.$$

Si $v_k \rightarrow v$ (convergence dans \mathbb{R}^n), alors la limite satisfait :

$$v_i = \lim_{k \rightarrow \infty} (v_k)_i \quad \text{et, par passage à la limite, } v_i \geq g(x_i).$$

Donc, $v \in K_n$.

Cela montre que K_n est fermé.

L'opérateur de projection $P_{K_n}(v)$ sur l'ensemble convexe K_n est défini comme la solution du problème :

$$P_{K_n}(v) = \arg \min_{w \in K_n} \|w - v\|_2^2,$$

où

$$\|w - v\|_2^2 = \sum_{i=1}^n (w_i - v_i)^2.$$

le problème de minimisation sera

$$\min_{w_i \geq g_i} (w_i - v_i)^2$$

Cependant, la contrainte $w_i \geq g_i$ donne que

- si $v_i \geq g_i$ alors $w_i = v_i$
- si $v_i < g_i$ alors $w_i = g_i$

D'où

$$(P_{K_n}(v))_i = \max(g_i, v_i) \quad , \quad i = \{0, \dots, n\}$$

2- Cas 1 : $v_i \in [a_i, b_i]$

Si v_i est dans l'intervalle $[a_i, b_i]$, alors le minimum est atteint pour $w_i = v_i$, car $f(w_i)$ est convexe avec un minimum global à $w_i = v_i$ (non contraint).

Cas 2 : $v_i < a_i$ Si $v_i < a_i$, alors w_i ne peut pas descendre en dessous de a_i à cause de la contrainte $w_i \geq a_i$. Le minimum est donc atteint à $w_i = a_i$.

Cas 3 : $v_i > b_i$ Si $v_i > b_i$, alors w_i ne peut pas dépasser b_i à cause de la contrainte $w_i \leq b_i$. Le minimum est donc atteint à $w_i = b_i$.

alors, on aura

$$(P_{K_n}(v))_i = \begin{cases} v_i & \text{si } a_i \leq v_i \leq b_i, \\ a_i & \text{si } v_i < a_i, \\ b_i & \text{si } v_i > b_i. \end{cases}$$

d'où

$$(P_{K_n}(v))_i = \max(a_i, \min(v_i, b_i))$$

Si $a_i = -\infty$, il n'y a plus de contrainte inférieure sur w_i , donc $\max(a_i, \cdot)$ n'impose aucune restriction.

De même, si $b_i = +\infty$, il n'y a plus de contrainte supérieure sur w_i , donc $\min(\cdot, b_i)$ n'impose aucune restriction.

La formule reste donc valide, car :

- Si $a_i = -\infty$, alors $\max(a_i, x) = x$.
- Si $b_i = +\infty$, alors $\min(x, b_i) = x$.

2.3-

Listing 2.2 – la fonction projK()

```
import numpy as np
def projK(u, gn):
    """ Projette le vecteur u sur l'ensemble Kn

    Param tres :
    - u : numpy array, vecteur      projeter.
    - gn : numpy array, vecteur des contraintes g(xi).

    Retourne :
    - numpy array, projection de u sur Kn.

    """
    return np.maximum(gn, u)

def gradient_projetee_pas_fixe(J, DJ, gn, u0, rho, Tol, iterMax,
                                 store):
    u = [u0] if store else u0
    k = 0
    rk = Tol + 1
    while rk >= Tol and k < iterMax:
        grad = DJ(u[-1] if store else u)
        u_next = projK(u[-1] if store else u - rho * grad, gn, np
                        .inf)
        if store:
            u.append(u_next)
        else:
            u = u_next
        rk = np.linalg.norm(u_next - (u[-2] if store else u))
        k += 1
    return (u if store else u_next, k)
```

2.5-

```
# Définition des paramètres du problème
n = 2
k = 1000
rho = 0.1
Tol = 1e-5
iterMax = 1000
# Définition de g2 et de son gradient
def f0(A, x):
    return np.dot(A[0], x) - np.dot(A[1], x) + b[0]*np.dot(fx, x, u)
def Df0(A, fx):
    return np.array([np.dot(A[0], u) - fx[0],
                    np.dot(A[1], u) - fx[1]])
# projection sur le convexe K
def projK(U, g):
    return np.maximum(U, g)
def gradient_projetee_pas_fixe(J, Df, gn, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    k = 0
    history = [U.copy()] if store else None
    rk = Tol + 1
    while rk >= Tol and iterMax < iterMax:
        grad = Df(U, g) # Calcul du gradient
        U = projK(U - rho * grad, g) # mise à jour avec projection
        rk = np.linalg.norm(U - u0) # critère de convergence
        U = U.next
        k += 1
        if store:
            history.append(U.copy()) # stockage des itérations
    print("Nombre d'itérations : ", iterMax - len(history))
    return (np.array(history), iterMax - len(history), k)
# Construction de la matrice A et des vecteurs f(x) et g(x)
A = np.array([[1, 1], [-1, 1], [1, -1], [-1, -1]])
b = np.array([1, 1, 1, 1])
fx = np.array([1, 1]) # Valeur de f(x) pour n = 2
gx = np.array([-1, 1]) # Valeur de g(x) pour n = 2 (aucune contrainte spécifique ici)
u0 = np.array([0, 0]) # Point initial
# Appel de la fonction avec "store" activé pour suivre les itérations
history, _, iterMax = gradient_projetee_pas_fixe(J, Df, gn, u0, rho, Tol, iterMax, store=True)
# Grille pour les courbes de niveau et le champ de gradients
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x, y)
Z = np.zeros_like(X)

# Calcul du champ de vecteurs (gradient)
DX = np.zeros_like(X)
DY = np.zeros_like(Y)
for j in range(K.shape[0]):
    for i in range(K.shape[1]):
        U = np.array([X[i, j], Y[i, j]])
        Z[i, j] = Df(U, g)

# Configuration du graphique
plt.title("Gradient projeté pour n=2")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.legend()
plt.grid()
plt.show()

# Normalisation pour le champ de gradients
norm = np.sqrt(DX**2 + DY**2)
DX /= norm
DY /= norm
# Tracé des courbes de niveau et des gradients
plt.figure(figsize=(10, 6))
plt.contour(X, Y, Z, levels=10, colors='black', alpha=0.7)
plt.quiver(X, Y, DX, DY, color='white', alpha=0.5, scale=50)
# Tracé des itérations
history_x, history_y, history_z, history_u = np.zeros(len(history)), np.zeros(len(history)), np.zeros(len(history)), np.zeros(len(history))
for i in range(len(history)):
    history_x[i] = history[i][0]
    history_y[i] = history[i][1]
    history_z[i] = Z[history[i][0], history[i][1]]
    history_u[i] = np.array([history[i][0], history[i][1], history_z[i]])
    plt.plot(history_x[:i+1], history_y[:i+1], 'o--', color='blue', label="Iterations")
    plt.scatter(history_x[i], history_y[i], color='green', label="Depart (u0)")
    plt.scatter(history_x[i-1], history_y[i-1], color='red', label="Arrivée (u*)")
```

Résultat du code :

```
U_min=[0.11112673 0.11112673] , iter_max = 36
```

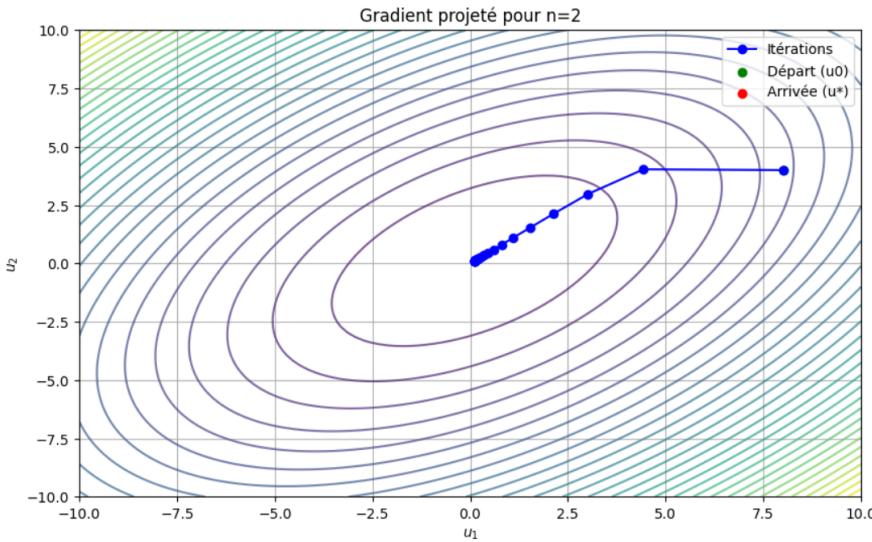


FIGURE 2.4 – trajectoire de vecteur U à pas fixe

2.6 - 2.7-

```
import numpy as np
import matplotlib.pyplot as plt
from time import time

# Définition des fonctions Jn et son gradient
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - np.dot(fx.T, U)

def D(J, U, A, fx):
    return np.dot(A, U) - fx

# Projection sur le convexe K
def projK(U, gn):
    return np.maximum(U, gn)

# Gradient projeté à pas fixe
def gradient_projetee_pas_fixe(J, D, gn, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [U] if store else None
    rk = Tol + 1

    while rk >= Tol and iter_count < iterMax:
        grad = D(U, A, fx)
        U_next = projK(U - rho * grad, gn)
        rk = np.linalg.norm(U_next - U)
        U = U_next
        iter_count += 1
        if store:
            history.append(U)

    return (U if not store else np.array(history)), iter_count

# Paramètres
n_values = [5, 20, 50, 100]
rho_values = [0.1, 0.5, 1]
Tol = 1e-5
iterMax = 1000

# Définir f(x) et g(x)
f = lambda x: np.maximum(1.5 - 20 * (x - 0.6)**2, 0)

# Résolution pour différentes valeurs de n
for rho in rho_values:
    print(f"\nRésultats pour \rho = {rho}")

    for n in n_values:
        x = np.linspace(0, 1, n + 2)
        xv = x[1:-1]
        fx = f(xv)
        gx = g(xv)

        A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
        u0 = np.zeros(n)

        start_time = time()
        U, iter_count = gradient_projetee_pas_fixe(J, D, gx, u0, rho, Tol, iterMax, store=False)
        elapsed_time = time() - start_time

        print(f"\n{n}: Itérations = {iter_count}, Temps = {elapsed_time:.4f}s")
```

Résultat du code :

```
Résultats pour \rho = 0.1
n = 5: Itérations = 1000, Temps = 0.0070s
n = 20: Itérations = 502, Temps = 0.0020s
n = 50: Itérations = 310, Temps = 0.0015s
n = 100: Itérations = 240, Temps = 0.0010s

Résultats pour \rho = 0.5
n = 5: Itérations = 434, Temps = 0.0020s
n = 20: Itérations = 238, Temps = 0.0010s
n = 50: Itérations = 184, Temps = 0.0000s
n = 100: Itérations = 157, Temps = 0.00005s

Résultats pour \rho = 1
n = 5: Itérations = 306, Temps = 0.0050s
n = 20: Itérations = 195, Temps = 0.0010s
n = 50: Itérations = 158, Temps = 0.0010s
n = 100: Itérations = 137, Temps = 0.0010s
C:\Users\11625\AppData\local\Temp\ipykernel_26420\352748582.py:25: RuntimeWarning: invalid value encountered in subtract
u_next = projK(U - rho * grad, gn)
```

FIGURE 2.5 – les solutions approchées de U pour différents n et différents ρ

Interpretation :

Effet de ρ (le pas d'apprentissage) :

- Pour $\rho = 0.1$: l'algorithme est lent : le nombre d'itérations est élevé (exemple : 1000 pour $n = 5$)
- Pour $\rho = 0.5$: les performances s'améliorent considérablement. Le nombre d'itérations diminue presque de moitié par rapport à $\rho = 0.1$ avec une convergence stable et rapide.
- Pour $\rho = 1$: l'algorithme atteint son pic d'efficacité. Le nombre d'itérations est encore réduit, avec un minimum de 137 pour $n = 100$.

Observation générale :

- Un petit ρ conduit à une convergence lente (plus d'itérations).
- Un ρ plus grand accélère la convergence jusqu'à un certain point.

2.8- pour la question 2.5 avec ρ optimal :

```

import numpy as np
import matplotlib.pyplot as plt
# Définition des paramètres
Tol = 1e-5
iterMax = 1000
# Définition de Jn et de son gradient
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - h*np.dot(fx.T, U)
def D(J, U, A, fx):
    return np.dot(A, U) - h*fx
# Projection sur le convexe K
def projK(U, g0):
    return np.maximum(U, g0)
# Gradient projeté à pas fixe avec historique
def gradient_projeté_pas_fixe(D, J, gn, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [U.copy()] if store else None
    rk = Tol
    while rk >= Tol and iter_count < iterMax:
        grad = D(J, U, fx)
        U.next = projK(U - rho * grad, gn)
        rk = np.linalg.norm(U.next - U)
        U = U.next
        iter_count += 1
        if store:
            history.append(U.copy())
    return (np.array(history) if store else U), iter_count
# Fonction principale pour l'expérience
n = 2
A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
fx = np.ones(n) # f(x) = 1
g0 = np.zeros(n) # Pas de contrainte supplémentaire
u0 = np.array([8, 4]) # Point initial (adapté à n)

# Calcul des valeurs propres de A
eigenvalues = np.linalg.eigvals(A)
lambda_min = min(eigenvalues)
lambda_max = max(eigenvalues)
rho_optimal = 2 / (lambda_min + lambda_max)
print("Pas optimal pour n = {} : rho = {}".format(n, rho_optimal))

# Résolution
history, iter_count = gradient_projeté_pas_fixe(D, J, gx, u0, rho_optimal, Tol, iterMax, store=1)

# Grille pour les courbes de niveau
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x, y)
Z = np.zeros_like(X)
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        U = np.array([X[i, j], Y[i, j]])
        Z[i, j] = J(U, A, fx)

# Tracé des courbes de niveau
plt.figure(figsize=(10, 6))
plt.contour(X, Y, Z, levels=30, cmap="viridis", alpha=0.7)

# Tracé des itérations
history_x, history_y = history[:, :, 1]
plt.plot(history_x, history_y, 'o', color="blue", label="Iterations")
plt.scatter(history_x[0], history_y[0], color="green", label="Départ (u0)")
plt.scatter(history_x[-1], history_y[-1], color="red", label="Arrivée (u*)")

# Configuration du graphique
plt.title("Gradient projeté avec pas optimal (n={})".format(n))
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.legend()
plt.grid()
plt.show()

```

Résultat du code :

Pas optimal pour n = 2 : rho = 0.1667
 $u_{\min} = [0.11111297 \quad 0.11111487]$, iter_max = 21

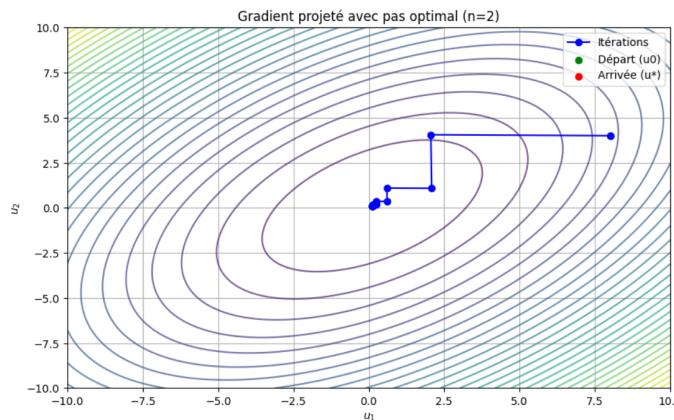


FIGURE 2.6 – les courbes de niveaux de J_2 ainsi que le champ de vecteurs de gradient de J_2 avec un ρ optimal

pour la question 2.6 avec ρ optimal :

```

import numpy as np
import matplotlib.pyplot as plt
from time import time
# Définition des fonctions Jn et son gradient
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - h*np.dot(fx.T, U)
def DJ(U, A, fx):
    return np.dot(A, U) - h*fx
# Projection sur le convexe K
def projK(U, gN):
    return np.maximum(U, gN)
# Gradient projeté à pas fixe
def gradient_projetee_pas_fixe(J, DJ, gn, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [u0] if store else None
    rk = Tol + 1
    while rk >= Tol and iter_count < iterMax:
        grad = DJ(U, A, fx)
        U_next = projK(U - rho * grad, gn)
        rk = np.linalg.norm(U_next - U)
        U = U_next
        iter_count += 1
        if store:
            history.append(U)
    return (U if not store else np.array(history)), iter_count
# Paramètres
n_values = [5, 20, 50, 100]
Tol = 1e-5
iterMax = 1000

```

```

# Définir f(x) et g(x)
f = lambda x: 1
g = lambda x: np.maximum(1.5 - 20 * (x - 0.6)**2, 0)

for n in n_values:
    x = np.linspace(0, 1, n + 2)
    xv = x[1:-1]
    fx = f(xv)
    gx = g(xv)
    # Calcul des valeurs propres de A
    eigenvalues = np.linalg.eigvals(A)
    lambda_min = min(eigenvalues)
    lambda_max = max(eigenvalues)
    rho_optimal = 2 / (lambda_min + lambda_max)

    print(f"Pas optimal pour n = {n} : rho = {rho_optimal:.4f}")

    A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
    u0 = np.zeros(n)

    start_time = time()
    U, iter_count = gradient_projetee_pas_fixe(J, DJ, gx, u0, rho, Tol, iterMax, store=False)
    elapsed_time = time() - start_time

    print(f"n = {n}: Iterations = {iter_count}, Temps = {elapsed_time:.4f}s")

```

Résultat du code :

```

.. Pas optimal pour n = 5 : rho = 0.1667
n = 5: Iterations = 306, Temps = 0.0024s
Pas optimal pour n = 20 : rho = 0.0833
n = 20: Iterations = 194, Temps = 0.0034s
Pas optimal pour n = 50 : rho = 0.0238
n = 50: Iterations = 157, Temps = 0.0000s
Pas optimal pour n = 100 : rho = 0.0098
n = 100: Iterations = 137, Temps = 0.0000s
C:\Users\21625\AppData\Local\Temp\ipykernel_12096\1905405569.py:25: RuntimeWarning: invalid value encountered in subtract
      U_next = projK(U - rho * grad, gn)

```

FIGURE 2.7 – les solutions approchées U ainsi que le graphe de la fonction g pour différents n avec un ρ optimal

2.9- pour la question 2.5 avec $f(x) = \pi^2 \sin(\pi x)$:

```

import numpy as np
import matplotlib.pyplot as plt
# Définition des paramètres
Tol = 1e-5
iterMax = 1000
# Définition de Jn et de son gradient
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - h * np.dot(fx.T, U)
def DJ(U, A, fx):
    return np.dot(A, U) - h * fx
# Projection sur le convexe K
def projK(U, gN):
    return np.maximum(U, gN)
# Gradient projeté à pas fixe avec historique
def gradient_projetee_pas_fixe(J, DJ, gn, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [U.copy()] if store else None
    rk = Tol + 1 # Pour initialiser correctement la boucle
    while rk >= Tol and iter_count < iterMax:
        grad = DJ(U, A, fx)
        U_next = projK(U - rho * grad, gn)
        rk = np.linalg.norm(U_next - U)
        U = U_next
        iter_count += 1
        if store:
            history.append(U.copy())
    print(f"U_min = {U}, iter_max = {iter_count}")
    return (np.array(history) if store else U), iter_count
n = 2
h = 1/(n+1)
A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
fx = (np.pi**2) * np.sin(np.pi * np.linspace(1, n, n))
gx = np.zeros(n) # Pas de contrainte supplémentaire
u0 = np.array([8, 4]) # Point initial (adapté à n)

```

```

# Calcul des valeurs propres de A
eigenvalues = np.linalg.eigvals(A)
lambda_min = min(eigenvalues)
lambda_max = max(eigenvalues)
rho_optimal = 2 / (lambda_min + lambda_max)
print(f"Pas optimal pour n = {n} : rho = {rho_optimal:.4f}")

# Résolution
history, iter_count = gradient_projetee_pas_fixe(J, DJ, gx, u0, rho_optimal, Tol, iterMax, store=True)
# Grille pour les courbes de niveau
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x, y)
Z = np.zeros_like(X)
# Correction ici pour calculer Z[i, j] comme un scalaire
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        U = np.array([X[i, j], Y[i, j]])
        Z[i, j] = J(U, A, fx) # Maintenant J retourne un scalaire

# Tracé des courbes de niveau
plt.figure(figsize=(10, 6))
plt.contour(X, Y, Z, levels=30, cmap="viridis", alpha=0.7)
# Tracé des itérations
history_x, history_y = history[:, 0], history[:, 1]
plt.plot(history_x, history_y, 'o', color="blue", label="Iterations")
plt.scatter(history_x[0], history_y[0], color="green", label="Départ (u0)")
plt.scatter(history_x[-1], history_y[-1], color="red", label="Arrivée (u*)")
# Configuration du graphique
plt.title("Gradient projeté avec pas optimal (n={n})")
plt.xlabel("$u_1$")
plt.ylabel("$u_2$")
plt.legend()
plt.grid()
plt.show()

```

Résultat du code :

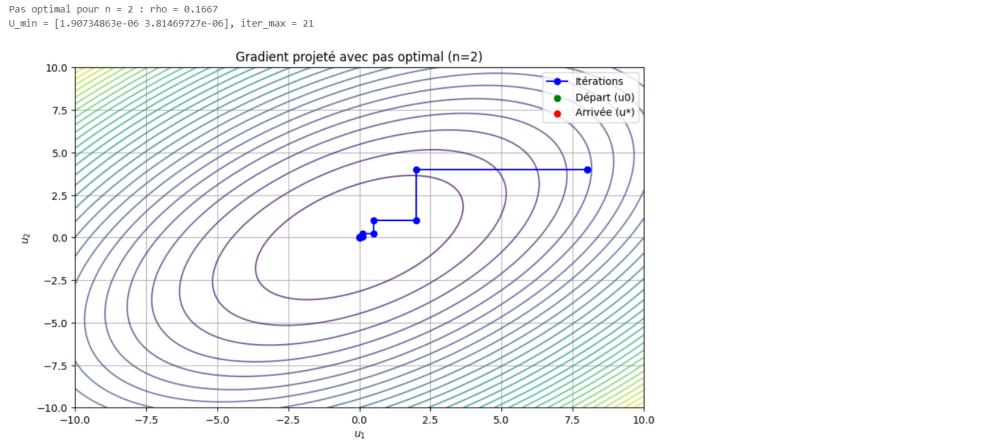


FIGURE 2.8 – les courbes de niveaux de J_2 ainsi que le champ de vecteurs de gradient de J_2 avec nouveau $f(x)$

pour la question 2.6 avec $f(x) = \pi^2 \sin(\pi x)$:

```
import numpy as np
import matplotlib.pyplot as plt
from time import time
# Définition des fonctions Jn et son gradient
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - h*np.dot(fx.T, U)
def D(J, U, A, fx):
    return np.dot(A, U) - h*fx
# Projection sur le convexe K
def projK(U, gN):
    return np.maximum(U, gN)
# Gradient projeté à pas fixe
def gradient_projetee_pas_fixe(J, D, gN, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [U] if store else None
    rk = Tol + 1
    while rk >= Tol and iter_count < iterMax:
        grad = D(J, U, fx)
        U_next = projK(U - rho * grad, gN)
        rk = np.linalg.norm(U_next - U)
        U = U_next
        iter_count += 1
        if store:
            history.append(U)
    return (U if not store else np.array(history)), iter_count
# Paramètres
n_values = [5, 20, 50, 100]
Tol = 1e-5
iterMax = 1000
```

```
# Définir f(x) et g(x)
f = lambda x: (np.pi**2) * np.sin(np.pi * x)
g = lambda x: np.maximum(1.5 - 20 * (x - 0.6) ** 2, 0)

for n in n_values:
    x = np.linspace(0, 1, n + 2)
    xv = x[1:-1]
    fx = f(xv)
    gx = g(xv)
    # Calcul des valeurs propres de A
    eigenvalues = np.linalg.eigvals(A)
    lambda_min = min(eigenvalues)
    lambda_max = max(eigenvalues)
    rho_optimal = 2 / (lambda_min + lambda_max)

    print(f"Pas optimal pour n = {n} : rho = {rho_optimal:.4f}")

    A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
    u0 = np.zeros(n)

    start_time = time()
    U, iter_count = gradient_projetee_pas_fixe(J, D, gx, u0, rho, Tol, iterMax, store=False)
    elapsed_time = time() - start_time

    print(f"n = {n}: Itérations = {iter_count}, Temps = {elapsed_time:.4f}s")
```

Résultat du code :

```
Pas optimal pour n = 5 : rho = 0.1667
n = 5: Itérations = 307, Temps = 0.0017s
Pas optimal pour n = 20 : rho = 0.0833
n = 20: Itérations = 201, Temps = 0.0000s
Pas optimal pour n = 50 : rho = 0.0238
n = 50: Itérations = 163, Temps = 0.0000s
Pas optimal pour n = 100 : rho = 0.0098
n = 100: Itérations = 142, Temps = 0.0000s
```

FIGURE 2.9 – les solutions approchées U ainsi que le graphe de la fonction g pour différents n avec un nouveau $f(x)$

2.3 Méthode de pénalisation :

3.1-

```

import numpy as np
import matplotlib.pyplot as plt
# Fonctionnelle Jn
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) - h * np.dot(fx.T, U)
# Gradient de Jn
def DJ(U, A, fx):
    return np.dot(A, U) - h * fx
# Terme de pénalisation
def penalite(U, gN, eta):
    return (1 / eta) * np.sum(np.maximum(gN - U, 0)**2)
# Gradient du terme de pénalisation
def grad_penalite(U, gN, eta):
    return (-2 / eta) * np.maximum(gN - U, 0)
# Fonction J pénalisée
def J_penalise(U, A, fx, gN, eta):
    return J(U, A, fx) + penalite(U, gN, eta)
# Gradient de J pénalisé
def grad_J_penalise(U, A, fx, gN, eta):
    return DJ(U, A, fx) + grad_penalite(U, gN, eta)

```

```

# Algorithme du gradient à pas fixe pour le problème pénalisé
def penalisation(J, grad_J, gN, eta, u0, rho, Tol, iterMax, store):
    U = u0.copy()
    iter_count = 0
    history = [U.copy()] if store else None
    rk = Tol + 1 # Pour entrer dans la boucle
    while rk > Tol and iter_count < iterMax:
        grad = grad_J(U) # Utilisation du gradient pénalisé
        U_next = U - rho * grad
        rk = np.linalg.norm(U_next - U)
        # Vérification des valeurs
        if not np.all(np.isfinite(U_next)):
            print(f"Divergence détectée pour η = {eta}")
            break
        U = U.next
        iter_count += 1
        if store:
            history.append(U.copy())
    print(f"U_min pour η = {eta}: {U}, Nombre d'itérations: {iter_count}")
return (np.array(history) if store else U), iter_count

```

3.2-

```

# Paramètres du problème
n = 50
eta_vals = [1e5, 1e4, 1e3, 1e2, 10, 1, 0.1]
x = np.linspace(0, 1, n + 2)
xv = x[1:-1] # Points internes
h = 1 / (n + 1) # Discréétisation
A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
fx = (np.pi**2) * np.sin(np.pi * xv) # Fonction f(x)
gx = np.maximum(1.5 - 20 * (xv - 0.6)**2, 0) # Obstacle g(x)
u0 = np.zeros(n) # Initialisation
rho = 0.1 # Pas fixe
Tol = 1e-5
iterMax = 500
# Résolution pour différentes valeurs de eta
solutions = []
for eta in eta_vals:
    print(f"Résolution pour η = {eta}")
    Jf = lambda U: J(U, A, fx) + penalite(U, gx, eta)
    grad_Jf = lambda U: grad_J_penalise(U, A, fx, gx, eta)
    u_sol, iter_count = penalisation(Jf, grad_Jf, gx, eta, u0, rho, Tol, iterMax, store=0)
    solutions.append((eta, u_sol, iter_count))
# Tracé des solutions pour différentes valeurs de η
plt.figure(figsize=(10, 6))
for eta, u_sol, _ in solutions:
    plt.plot(xv, u_sol, label=f"η = {eta}")
# Tracé de l'obstacle
plt.plot(xv, gx, label="Obstacle g(x)", linestyle="--", color="black")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title("Solutions pour différentes valeurs de η")
plt.legend()
plt.grid()
plt.show()

```

Résultat du code :

```

-- Résolution pour η = 100000.0
Divergence détectée pour η = 100000.0
U_min pour η = 100000.0: [-1.21761829e+305 -4.13067626e+305 6.45381170e+305 -8.58064183e+305
1.0682859e+306 -1.27501812e+306 1.47608494e+306 -1.67270919e+306
1.80312039e+306 -2.01287824e+306 2.20468759e+306 -2.39358251e+306
2.49317878e+306 -2.61156494e+306 -2.79925182e+306
2.84766273e+306 -2.88312566e+306 2.89967416e+306 -2.88838566e+306
2.86972359e+306 -2.81380054e+306 2.75903711e+306 -2.67172856e+306
2.83482359e+306 -2.77777777e+306 2.74462777e+306 -2.64282399e+306
2.15469517e+306 -2.04263717e+306 1.93499718e+306 -1.83682630e+306
1.73192908e+306 -1.63661034e+306 1.54728397e+306 -1.46252151e+306
1.3814193e+306 -1.30276089e+306 1.22316172e+306 -1.14708431e+306
1.03821123e+306 -9.69689012e+306 8.90937813e+306 -8.20527392e+306
7.0301123e+305 -5.97589320e+305 4.86309624e+305 -3.69806037e+305
2.40952599e+305 -1.25001166e+305], Nombre d'itérations: 266
Résolution pour η = 10000.0
Divergence détectée pour η = 10000.0
U_min pour η = 10000.0: [-1.1121090506e+305 2.22353181e+305 -3.32925047e+305 4.42079346e+305
5.51104324e+305 6.57930555e+305 -7.44711439e+305 8.63792000e+305
9.09520000e+305 -1.08330000e+306 1.18830000e+306 -1.28330000e+306
-1.28766809e+306 1.34011322e+306 -1.40857276e+306 1.44137076e+306
-1.47103082e+306 1.48910506e+306 -1.49621984e+306 1.49204780e+306
-1.47733326e+306 1.45287826e+306 -1.41969033e+306 1.37897225e+306
-1.43366626e+306 1.40578026e+306 -1.37085033e+306 1.33577225e+306
-1.1185471e+306 1.05299136e+306 -9.9698901e+305 9.43626284e+305
...
5.46950294e+306 -5.01201515e+306 5.24289908e+306 -4.47427595e+306
5.61161774e+306 -3.89898517e+306 3.94711418e+306 -3.22757114e+306
3.14180314e+306 -2.43401018e+306 2.19620688e+306 -1.51595988e+306
1.13238499e+306 -5.17236832e+305], Nombre d'itérations: 238
Output truncated due to a console overflow or open in a text editor. Adjust cell output settings.
C:\Users\l2623\appdata\local\temp\ipykernel_4640\2070294465.pyz:2: RuntimeWarning: overflow encountered in add
return DJ(U, A, fx) + grad_penalite(U, gN, eta)
C:\Users\l2623\appdata\local\temp\ipykernel_4640\2070294465.pyz:18: RuntimeWarning: overflow encountered in multiply
return (...) + eta * np.maximum(gN - U, 0)

```

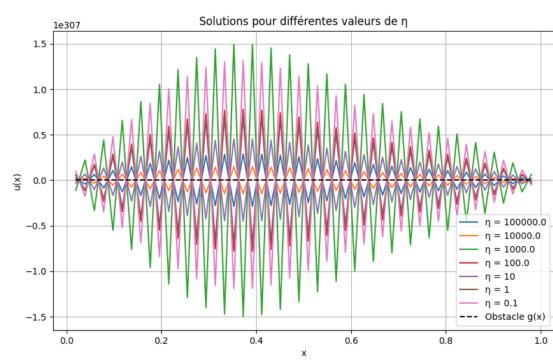


FIGURE 2.10 – teste de la fonction pour différents η

3.3-

```

import numpy as np
import matplotlib.pyplot as plt
# Fonctionnelle Jn
def J(U, A, fx):
    return 0.5 * np.dot(U.T, np.dot(A, U)) + h * np.dot(fx.T, U)
# Terme de pénalisation
def D(U, A, fx):
    return np.dot(A, U) - h * fx
# Termes de pénalisation
def penalite(U, gH, eta):
    return (1 / eta) * np.sum(np.maximum(gH - U, 0)**2)
# Calcul du pas optimal de penalisation
def grad_penalite(U, gH, eta):
    return -(2 / eta) * np.maximum(gH - U, 0)
# Fonctionnelle pénalisée
def J_penalise(U, A, fx, gH, eta):
    return J(U, A, fx) + penalite(U, gH, eta)
# Gradient de la fonctionnelle pénalisée
def grad_J_penalise(U, A, fx, gH, eta):
    return D(U, A, fx) + grad_penalite(U, gH, eta)
# Calcul du pas optimal pour recherche linéaire
def calcule_pas_optimal(x, grad, J_func):
    # Fonction unidimensionnelle à minimiser
    phi = lambda rho: J_func(x - rho * grad)
    # Intervalle initial pour la recherche
    rho_min, rho_max = 0, 1
    tol = 1e-5
    # Recherche par dichotomie
    while rho_max - rho_min > tol:
        rho_mid = (rho_min + rho_max) / 2
        if phi(rho_mid) < phi(rho_mid + tol):
            rho_max = rho_mid
        else:
            rho_min = rho_mid
    return (rho_min + rho_max) / 2

# algorithme du gradient à pas optimal
def gradient_pas_optimal(J_func, grad_J_func, gH, eta, u0, w0, Tol, iterMax):
    u = u0.copy()
    iter_count = 0
    rk = np.linalg.norm(u - w0) # norme de la différence initiale
    while rk > Tol and iter_count < iterMax:
        grad = grad_J_func(u) # Gradient pénalisé
        rho_h_k = call_gradient_pas_optimal(u, grad, J_func) # Calcul du pas optimal
        u = u - rho_h_k * grad # Mise à jour
        rk = np.linalg.norm(u - w0)
        iter_count += 1
    print(f"\nMin pour \eta = {eta}: Nombre d'itérations: {iter_count}")
    return u, iter_count

# Paramètres du problème
n = 50
x = np.linspace(0, 1, n + 2)
xv = x[1:-1]
h = 1 / (n + 1)
A = (np.eye(n) * np.ones(n)) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1)
fx = (np.pi**2) * np.sin(np.pi * xv)
gH = np.maximum(1.5 - 20 * (xv - 0.6)**2, 0)
u0 = np.zeros(n)
# Processus de la pénalisation
eta_vals = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
for eta in eta_vals:
    iter_count = 0
    # Résolution pour différentes valeurs de \eta
    solutions = []
    for eta in eta_vals:
        print(f"\nRésolution pour \eta = {eta}")
        Jf = lambda U: J_penalise(U, A, fx, gH, eta)
        grad_Jf = lambda U: grad_J_penalise(U, A, fx, gH, eta)
        u_sol, iter_count = gradient_pas_optimal(Jf, grad_Jf, gH, eta, u0, Tol, iterMax)
        solutions.append((eta, u_sol))

# Tracé des solutions pour différentes valeurs de \eta
plt.figure(figsize=(10, 6))
for eta, u_sol in solutions:
    plt.plot(xv, u_sol, label=f"\eta = {eta}")
plt.title("Solutions pour différentes valeurs de \eta (Gradient à Pas Optimal)")
plt.legend()
plt.grid()
plt.show()

# Tracé de l'obstacle
plt.plot(xv, gx, label="Obstacle g(x)", linestyle="--", color="black")
plt.xlabel("x")
plt.ylabel("g(x)")
plt.title("Obstacle g(x)")
plt.legend()
plt.grid()
plt.show()

```

Résultat du code :

```

... Résolution pour \eta = 100000.0
U_min pour \eta = 100000.0: [0.06153219 0.12283282 -0.18366689 0.24380056 0.30301627 0.36107672
0.6177724 0.47298831 0.52610964 0.57751364 0.62664243 0.67339262
0.7175903 0.75906385 0.79766012 0.83228866 0.86563863 0.89476184
0.92049521 0.94273361 0.96139833 0.97641383 0.98772797 0.99529293
0.9998405 0.9998454 0.99929675 0.98773902 0.97642690 0.961040403
0.9427422 0.92050285 0.89477335 0.86564733 0.83323966 0.79766905
0.7598745 0.71759858 0.67340228 0.62664698 0.57751281 0.52626105
0.4728867 0.4177771 0.3610830 0.30301963 0.24380694 0.18366882
0.1228346 0.06153386], Nombre d'itérations: 3352
Résolution pour \eta = 10000.0
U_min pour \eta = 10000.0: [0.061542 0.12285044 0.18369323 0.24383891 0.30306035 0.36113168
0.41783419 0.47295139 0.52627522 0.57769221 0.6267399 0.67349913
0.7175903 0.75906385 0.79773737 0.83228866 0.86563863 0.89476184
0.92049521 0.94273361 0.96139833 0.97641383 0.98772797 0.99529293
0.9998405 0.9998454 0.99929675 0.98773902 0.97642690 0.961040403
0.9427422 0.92050285 0.89477335 0.86564733 0.83323966 0.79766905
0.7598745 0.71759858 0.67340228 0.62664698 0.57751281 0.52626105
0.4728867 0.4177771 0.3610830 0.30301963 0.24380694 0.18371541
0.12286551 0.06154939], Nombre d'itérations: 3332
Résolution pour \eta = 1000.0
U_min pour \eta = 1000.0: [0.06163839 0.12304353 0.1839824 0.24422505 0.30354222 0.36171706
0.4185086 0.47372291 0.52714213 0.5785671 0.62779907 0.67465602
0.71895681 0.76053744 0.7992367 0.83491228 0.86742498 0.8966563
0.92249117 0.94483672 0.963606372 0.97872637 0.99014261 0.99781212
...
1.4789203 1.46165592 1.43169304 1.38892147 1.33415611 1.26957462
1.19952832 1.12660266 1.05095158 0.97727449 0.89215892 0.80938861
0.72460501 0.63803413 0.54987724 0.46634965 0.36967173 0.27806828
0.18576758 0.09300059], Nombre d'itérations: 1535
Output was truncated. View as a scrollable element or open in a text editor. Adjust cell output settings.

```

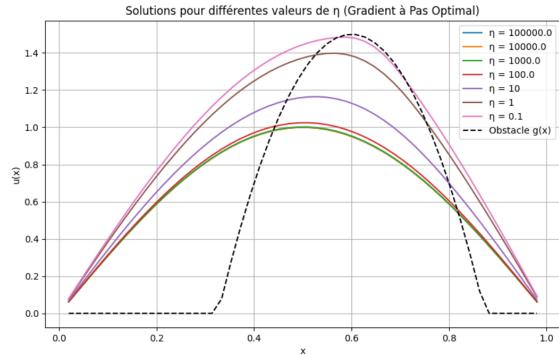


FIGURE 2.11 – résultat de 3.2 avec l'algorithme du gradient à pas optimal.

3.4-

Une méthode où η varie progressivement est essentielle pour la méthode de pénalisation, car elle :

- Améliore la précision des solutions en respectant mieux les contraintes.
- Réduit le coût computationnel global.
- Évite les problèmes de stabilité associés à des η fixes très petits.

En pratique, une bonne stratégie consiste à commencer avec un η initial modéré et à le diminuer selon une loi géométrique ou exponentielle jusqu'à atteindre une convergence satisfaisante.

2.4 Méthode d'Uzawa

4.1- le problème peut être réécrit avec un multiplicateur de Lagrange $\lambda = (\lambda_1, \dots, \lambda_n)$, en utilisant la formulation du lagrangien :

$$L_\lambda(v) = J_n(v) + \sum_{i=1}^n \lambda_i(g(x_i) - v_i).$$

On peut formuler le problème dual donc comme suit :

$$\max_{\lambda \geq 0} \min_{v \in \mathbb{R}^n} L_\lambda(v).$$

À chaque itération, $v^{(k+1)}$ est obtenu en minimisant le lagrangien pour un $\lambda^{(k)}$ donné :

$$v^{(k+1)} = \arg \min_{v \in \mathbb{R}^n} L_{\lambda^{(k)}}(v).$$

Cela revient à résoudre :

$$v^{(k+1)} = \arg \min_{v \in \mathbb{R}^n} \left[J_n(v) - \sum_{i=1}^n \lambda_i^{(k)} v_i \right].$$

Dans le problème dual, l'objectif est de maximiser $D(\lambda)$, où :

$$D(\lambda) = \min_{v \in \mathbb{R}^n} L_\lambda(v).$$

$v^{(k+1)}$ minimise $L_\lambda(v)$, donc la valeur $v^{(k+1)}$ satisfait :

$$\nabla_v L_{\lambda^{(k)}}(v^{(k+1)}) = 0.$$

et pour $\lambda^{(k+1)}$ utilise un pas de gradient projeté pour maximiser $D(\lambda)$, avec :

$$\lambda_i^{(k+1)} = \Pi_{\mathbb{R}^+} \left(\lambda_i^{(k)} + \rho \frac{\partial D(\lambda)}{\partial \lambda_i} \right),$$

où $\Pi_{\mathbb{R}^+}$ est l'opérateur de projection sur \mathbb{R}^+ , et :

$$\frac{\partial D(\lambda)}{\partial \lambda_i} = g(x_i) - v_i^{(k+1)}.$$

4.2- Tel que présenté, l'algorithme de la méthode d'Uzawa ne peut pas toujours assurer la convergence des multiplicateurs de Lagrange $\{\lambda^{(k)}\}$ sans des hypothèses supplémentaires :

- Convexité du problème primal,
- Dualité forte,
- Choix approprié du pas ρ .

Dans la pratique, des ajustements ou des extensions peuvent être nécessaires pour garantir la convergence dans tous les cas.

4.3- Pour une matrice A définie positive avec spectre $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$, une plage de choix pour ρ est donnée par :

$$0 < \rho < \frac{2}{\lambda_n}.$$

Un choix optimal est :

$$\rho = \frac{2}{\lambda_1 + \lambda_n}.$$

Ce choix assure une convergence rapide et stable de l'algorithme d'Uzawa.

4.5-

```

import numpy as np

def gradient_fixe_lambda(J, D, gN, rho, epsilon, iterMax, u0, lambda0):
    u = u0.copy()
    history = [u.copy()] if store else None

    for i in range(iterMax):
        grad = Dj(u) + lambda_k # Gradient de L
        u_next = u - rho * grad # Mise à jour du gradient fixe

        if np.linalg.norm(u_next - u) < epsilon:
            break

        u = u_next
        if store:
            history.append(u.copy())

    return (history if store else u, i + 1)

✓ 0.0s

```

```

def Uzawa(J, Dj, gN, rho, epsilon, iterMax, u0, lambda0):
    u = u0.copy()
    lambda_k = lambda0.copy()
    rk = epsilon + 1 # Critère d'arrêt
    k = 0

    while rk >= epsilon and k < iterMax:
        # Étape 1 : Minimiser Lλ(v) avec respect à v (gradient fixe)
        u, _ = gradient_fixe_lambda(J, Dj, lambda_k, rho, u, epsilon, iterMax, store=0)

        # Étape 2 : Mettre à jour les multiplicateurs de Lagrange
        lambda_k = np.maximum(lambda_k + rho * (gN - u), 0)

        # Étape 3 : Vérifier la convergence
        rk = np.linalg.norm(u - u0)
        u0 = u.copy()
        k += 1

    return u, lambda_k, k

✓ 0.0s

```

```

import matplotlib.pyplot as plt
from time import time
# Définir les fonctions f(x), g(x), et Jn(u)
def f(x):
    return np.ones_like(x)
def g(x):
    return np.maximum(1.5 - 20 * (x - 0.6) ** 2, 0)
def J(u, A, f_val):
    h = 1 / (len(u) + 1)
    return 0.5 * (u.T @ A @ u) - h * f_val @ u
def Dj(u, A, f_val):
    h = 1 / (len(u) + 1)
    return A @ u - h * f_val
# Discréétisation et paramètres
n = 50
x = np.linspace(0, 1, n + 2)
xv = x[1:-1]
fv, gv = f(xv), g(xv)
A = (n + 1) * (2 * np.eye(n) - np.diag(np.ones(n - 1), -1) - np.diag(np.ones(n - 1), 1))
# Conditions initiales
u0 = np.zeros(n)
lambda0 = 2 / np.max(np.linalg.eigvals(A))
rho_max = 2 / np.max(np.linalg.eigvals(A))
rho = rho_max * 0.9
epsilon = 1e-5
iterMax = 5000

```

```

# Résolution avec Uzawa
start_time = time()
u_sol, lambda_sol, num_iters = Uzawa(
    lambda_u: J(u, A, fv),
    lambda_u: Dj(u, A, fv),
    gv,
    rho,
    epsilon,
    iterMax,
    u0,
    lambda0
)
end_time = time()
# Résultats
print(f"Solution calculée en {num_iters} itérations et {(end_time - start_time):.2f} secondes.")
# Tracer les résultats
plt.plot(xv, u_sol, label="Solution u(x)")
plt.plot(xv, gv, 'k--', label="Obstacle g(x)")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.legend()
plt.title("Résolution par la méthode d'Uzawa")
plt.show()
✓ 2m 36.6s

```

Réultat du code :

Solution calculée en 5000 itérations et 156.43 secondes.

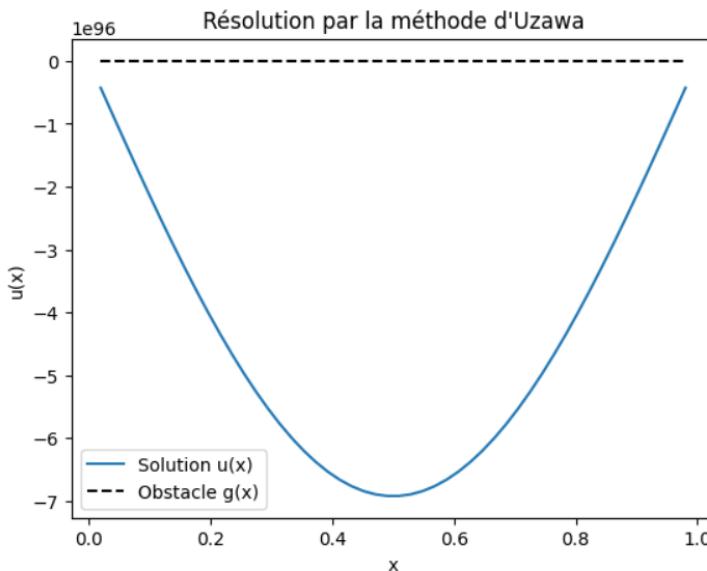


FIGURE 2.12 – la méthode du gradient à pas xe pour calculer à chaque étape le minimiseur de V à l'aide de l'algorithme d'Uzawa

4.6-

- Si ρ dépasse une certaine limite, la méthode d'Uzawa peut devenir instable.
- les oscillations des multiplicateurs de Lagrange λ_k peuvent empêcher la convergence.
càd, si ρ est trop grand, la convergence de l'algorithme n'est plus garantie.