COMPTE RENDU DES TRAVAUX PRATIQUES

Roussel Desmond NZOYEM NGUEGUIN

N° étudiant : 21911823

Courriel : desmond.ngueguin@outlook.com

Université de Strasbourg

UFR de Mathématique et d'informatique

Master 1 CSMI

Optimisation

Pr. Yannick PRIVAT

30 juin 2020

Contenu

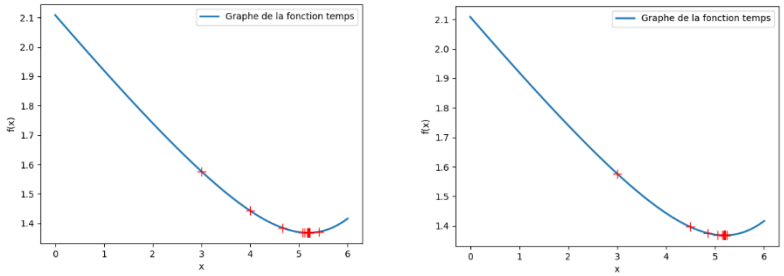
TP 1 - Optimisation sans contrainte en dimensions un et plus, optimisation quadratique

EXERCICE 1 (Optimisation 1D sans contrainte)

Le problème revient à minimiser la fonction temps $f(x) = \frac{x}{8} + \frac{1}{3}\sqrt{4 + (6 - x)^2}$ sur $\mathbb{R}$. On désire obtenir une solution approchée à 10^{-10} près. Nous précéderons par deux méthodes numériques : la **méthode de dichotomie** (en supposant que la solution se trouve dans un intervalle $[a, b]$, voir Listing 2 – fichier *Ex1_dicho.py*) et la **méthode de Newton** (voir Listing 3 – fichier *Ex1_dicho.py*).

1. Comparaison des deux méthodes

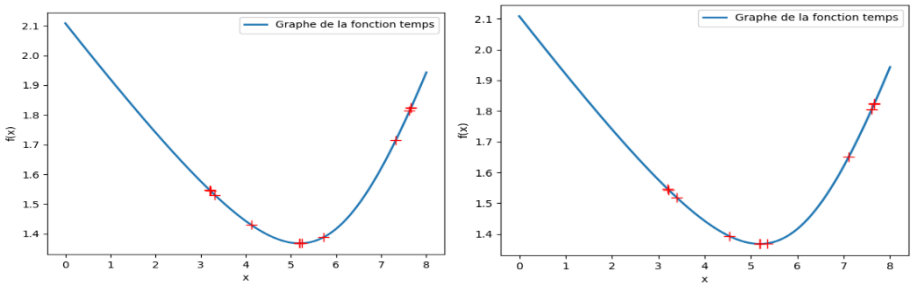
Par la méthode de dichotomie avec initialement $a = 0$, $b = 6$, et un facteur de réduction $\tau = 3$ (correspondant ainsi à l'algorithme dans l'énoncé), il nous faut **50 itérations** pour atteindre la précision voulue ; pour $\tau = 100$, on a besoin de seulement **29 itérations** (ici τ est pris telle que $b' - a' = \frac{1}{\tau}(b - a)$ à chaque itération)**.**



(a): $\tau = 3$ *(b): $\tau = 100$*

Figure 1: Méthode de dichotomie avec $a = 0$ et $b = 6$ initialement

Par la méthode de Newton pour $x^{(0)} = 3.202$, on a besoin de **13 itérations ;** et pour $x^{(0)} = 7.662$, on a besoin de **11 itérations**.



(a): $x^{(0)} = 3.202$ *(b): $x^{(0)} = 7.662$*

Figure 2: Méthode de Newton

On conclut que la méthode de Newton est plus adaptée pour ce problème. La méthode de dichotomie est moins efficace même quand on choisit un facteur de réduction τ très élevé ou un intervalle $[a, b]$ initial très réduit. Cependant, pour pouvoir converger, la méthode de Newton nécessite une très bonne initialisation $x^{(0)} \in [3.202, 7.662]$ proche la solution recherchée $x^* \approx 5.19096$, ce qui réduit malheureusement sa généralisation à d'autres problèmes.

2. *Déterminons analytiquement la solution optimale*

Existence

L'intervalle $K = [0,6]$ est un compact de $\mathbb{R}$ car fermé borné, et $f(x) = \frac{x}{8} + \frac{\sqrt{4+(x-6)^2}}{3}$ est continue sur K. On conclut qu'il existe un minimiseur de f sur K.

Unicité

On a $$f'(x) = \frac{1}{8} + \frac{x-6}{3\sqrt{4+(x-6)^2}}$$

$$f''(x) = \frac{1}{3}\left(\frac{\sqrt{4+(x-6)^2} - \frac{x-6}{\sqrt{4+(x-6)^2}}(x-6)}{4+(x-6)^2}\right)$$
$$= \frac{4 + (x-6)^2 - (x-6)^2}{3(4+(x-6)^2)\sqrt{4+(x-6)^2}}$$
$$= \frac{4}{3(4+(x-6)^2)^{\frac{3}{2}}}$$

Soit x^* le minimiseur recherché. f est deux fois différentiable sur K et $f''(x^*) > 0$. On en déduit que x^* est unique.

Calcul de x^*

On a

$$f'(x) = 0 \Longrightarrow \frac{1}{8} + \frac{x-6}{3\sqrt{4+(x-6)^2}} = 0$$
$$\Longrightarrow \frac{x-6}{3\sqrt{4+(x-6)^2}} = -\frac{1}{8}$$
$$\Longrightarrow -8(x-6) = 3\sqrt{4+(x-6)^2}$$
$$\Longrightarrow 64(x-6)^2 = 9(4+(x-6)^2) \quad \text{et} \quad -8(x-6) \geq 0$$
$$\Longrightarrow (x-6)^2 = \frac{36}{55} \quad \text{et} \quad x \leq 6$$
$$\Longrightarrow x = 6 - \frac{6}{\sqrt{55}}$$

On obtient donc $\boldsymbol{x^* \approx 5.19096}$ qui correspond bien aux résultats retrouvés numériquement.

EXERCICE 2 (Optimisation 2D avec contraintes par pénalisation)

Avec $\varepsilon > 0$, on veut obtenir une solution à 10^{-12} près du minimum de la fonction f_ε définie sur $\mathbb{R}^2$ par

$$f_\varepsilon(x,y) = (x+1)^2 + (y-2)^2 + \frac{1}{\varepsilon}(y-x+1)^2$$

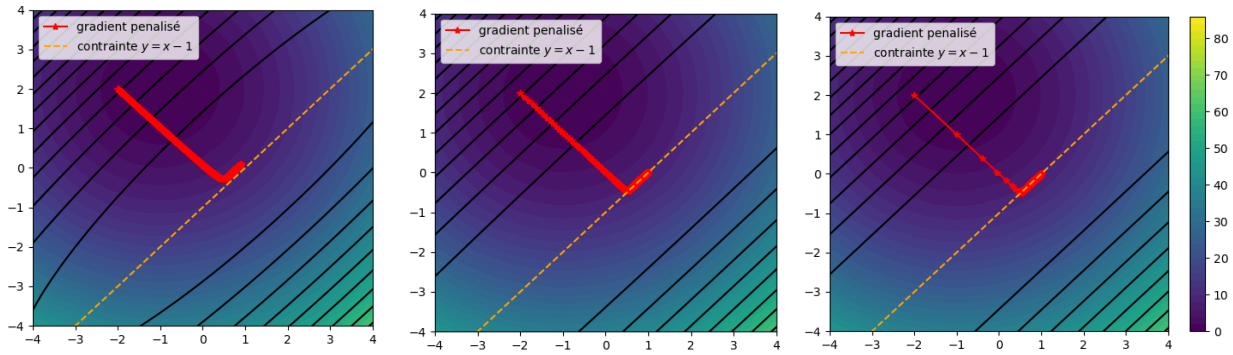
1. *Minimisation par la méthode du gradient à pas fixe*

Pour implémenter la méthode du gradient à pas fixe, on aura besoin :

- D'un point d'initialisation. On choisit $\left(x^{(0)}, y^{(0)}\right) = (-2, 2)$
- D'un pas pour la méthode. On choisit $\rho = 10^{-4}$
- Du gradient de f_ε :

$$\nabla f_\varepsilon = \begin{pmatrix} 2(x+1) - \frac{1}{\varepsilon}(y - x + 1) \\ 2(y-2) + \frac{1}{\varepsilon}(y - x + 1) \end{pmatrix}$$

On implémente l'algorithme du gradient à pas fixe et on obtient les résultats suivants pour différentes valeurs de ε (voir Listing 6 – fichier *Ex2_question1.py)*:



$(a) : \varepsilon = 0.1$ $(a) : \varepsilon = 0.01$ $(c) : \varepsilon = 0.001$

Figure 3 : Méthode du gradient à pas fixe

La figure 3 indique pour lorsque le coût de pénalisation $\frac{1}{\varepsilon}$ augmente, on se rapproche bien plus brusquement de la droite $y - x + 1 = 0$. Tous les trois cas convergent vers le point $(1, 0)$ qui correspond à la solution du problème de minimisation de la fonction $f(x, y) = (x+1)^2 + (y-2)^2$ sous la contrainte $y - x + 1 = 0$. Le tableau 1 ajoute que l'augmentation de $\frac{1}{\varepsilon}$ permet d'obtenir une approximation bien plus rapprochée de ce minimum (tout ceci en gardant le même nombre d'itérations).

$\boldsymbol{\varepsilon}$	**Minimum en** $(\boldsymbol{x^*}, \boldsymbol{y^*})$	**Nombre d'itérations**
0.1	$(0.9048, 0.0952)$	93828
0.01	$(0.9900, 0.0100)$	93828
0.001	$(0.9990, 0.0010)$	93828

Table 1 : Convergence du gradient à pas fixe

Minimisation par la méthode du gradient à pas optimal

On aura besoin :

- D'un point d'initialisation $\left(x^{(0)}, y^{(0)}\right) = (-2, 2)$
- Du gradient de f_ε :

$$\nabla f_\varepsilon\,(x, y) = \begin{pmatrix} 2(x+1) - \frac{1}{\varepsilon}(y - x + 1) \\ 2(y-2) + \frac{1}{\varepsilon}(y - x + 1) \end{pmatrix}$$

- De la matrice Hessienne de f_ε :

$$\mathrm{Hess} f_\varepsilon(x,y) = \begin{pmatrix} 2+\frac{1}{\varepsilon} & -\frac{2}{\varepsilon} \\ -\frac{2}{\varepsilon} & 2+\frac{2}{\varepsilon} \end{pmatrix}$$

Afin de trouver le pas optimal à chaque itération k, on va devoir minimiser la fonction $q^{(k)}$ définie sur $\mathbb{R}$ par :

$$q^{(k)}(\rho) = f_\varepsilon(\mathbf{x}^{(k)} + \rho \mathbf{d}^{(k)})$$

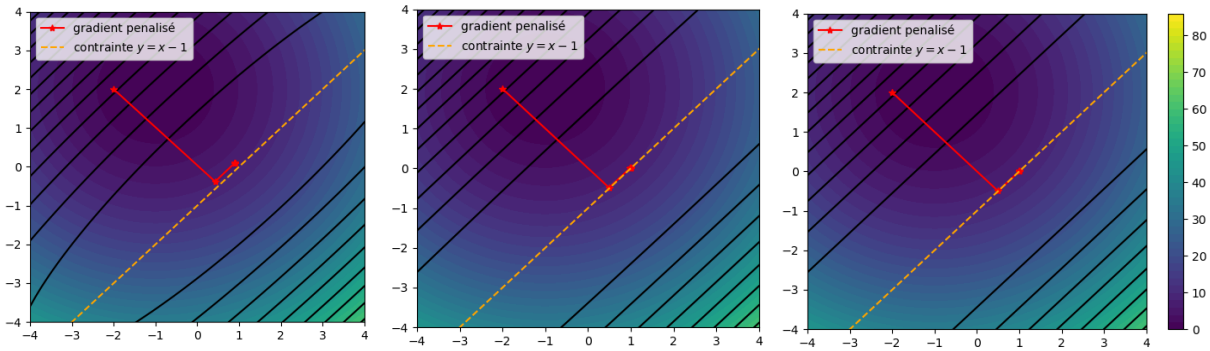
Où $\mathbf{x}^{(k)} = \left(x^{(k)}, y^{(k)}\right) \in \mathbb{R}^2$ est la position du minimum à l'itération k, et $\mathbf{d}^{(k)} = -\nabla f_\varepsilon\left(\mathbf{x}^{(k)}\right) \in \mathbb{R}^2$. Pour simplifier les écritures, on notera tout simplement $q(\rho) = f_\varepsilon(\mathbf{x} + \rho \mathbf{d})$. A priori, on ne connait pas l'intervalle qui contient le pas optimal recherché. On se sert donc de la méthode de Newton (voir Listing 4) pour minimiser q. Pour ce faire, on aura également besoin de :

$$q'(\rho) = \langle \nabla f_\varepsilon(\mathbf{x} + \rho \mathbf{d}), \mathbf{d} \rangle$$

$$q''(\rho) = \langle \mathrm{Hess} f_\varepsilon(\mathbf{x} + \rho \mathbf{d}) \mathbf{d}, \mathbf{d} \rangle$$

Où $\langle \cdot, \cdot \rangle$ désigne le produit scalaire associé à la norme euclidienne dans $\mathbb{R}^2$.

On obtient les résultats suivants (voir Listing 7 – fichier *Ex2_question1.py*) :



$(a) : \varepsilon = 0.1$ $(b) : \varepsilon = 0.01$ $(c) : \varepsilon = 0.001$

Figure 4 : Méthode du gradient à pas optimal

$\boldsymbol{\varepsilon}$	**Minimum en** $(\boldsymbol{x}^*, \boldsymbol{y}^*)$	**Nombre d'itérations**
0.1	$(0.9048, 0.0952)$	11
0.01	$(0.9900, 0.0100)$	9
0.001	$(0.9990, 0.0010)$	7

Table 2 : Convergence du gradient à pas optimal

Avec cette méthode, on obtient bien les résultats attendus. Lorsque le coût de pénalisation augmente, on remarque non seulement une meilleure approximation de la solution exacte, mais aussi une décroissance du nombre d'itérations.

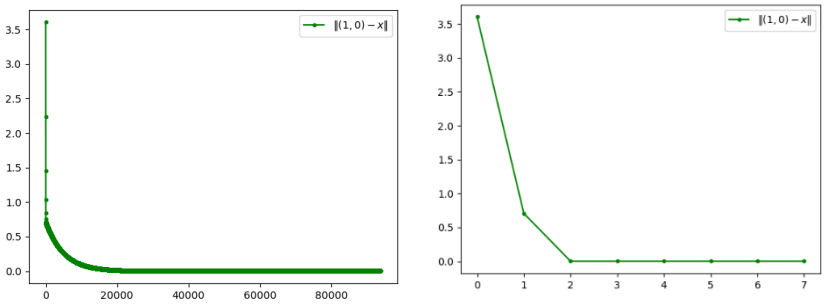
Comparaison des méthodes du gradient à pas fixe et à pas optimal

Le pas pour la méthode de gradient à pas fixe vaut $\rho = 10^{-4}$. On obtient le tableau ci-dessous pour différentes valeurs de ε:

ε	Nombre d'itérations		Temps d'exécution (s)	
	Gradient à pas fixe	Gradient à pas optimal	Gradient à pas fixe	Gradient à pas optimal
0.1	93828	11	1.997	1.349
0.01	93828	9	2.081	0.601
0.001	93828	7	1.923	0.592

Table 3 : Comparaison des méthodes de gradient pénalisé

En termes de nombre d'itérations, la méthode du gradient à pas optimale est radicalement plus performante que celle à pas fixe. En termes de temps d'exécution, elle maintient sa supériorité, ceci malgré le temps supplémentaire qu'elle nécessite pour rechercher son pas optimal. La méthode s'adapte en fonction de la valeur de ε (d'où le nombre d'itérations variable) pour donner le pas le plus efficace possible. Ces résultats sont confirmés par une comparaison de l'erreur de convergence à la figure 5. Pour faire cette comparaison, nous avons calculé à chaque itération, la distance entre la position $\mathbf{x}^{(k)}$ et le point $(1, 0)$, ceci pour $\varepsilon = 0.001$.



(a) : Gradient à pas fixe $\rho = 10^{-4}$ *(b) : Gradient à pas optimal*

Figure 5 : Comparaison de la vitesse de convergence

2. Méthode permettant de déterminer la distance d'un point à l'ellipse

On s'inspire de la question précédente. Déterminer la distance du point $(a, b) \in \mathbb{R}^2$ à l'ellipse $E = \{(x, y) \in \mathbb{R}^2 \mid 2x^2 + y^2 = 1\}$, revient à minimiser, sur $\mathbb{R}^2$, la fonction :

$$f_\varepsilon(x, y) = (x - a)^2 + (y - b)^2 + \frac{1}{\varepsilon}(2x^2 + y^2 - 1)^2$$

En effet, ceci peut être vu comme la minimisation de la fonction $f(x, y) = (x - a)^2 + (y - b)^2$ sous la contrainte $2x^2 + y^2 - 1 = 0$. Afin de s'assurer que la solution obtenue appartient bien à l'ellipse E, il faut prendre un coefficient de pénalisation $\frac{1}{\varepsilon}$ élevé, de façon à ce que les valeurs de $\frac{1}{\varepsilon}(2x^2 + y^2 - 1)^2$ soient très grandes par rapport à celles de f lorsque $2x^2 + y^2 - 1 \neq 0$. Une fois ε choisi, il ne restera plus qu'à trouver (x^*, y^*), solution du problème de minimisation de f_ε sur $\mathbb{R}^2$.

On note que si la contrainte n'avait pas existé, minimiser f_ε aurait été trivial car cela aurait donné le point (a, b). L'ajout de la contrainte a permis de trouver le point le plus proche de (a, b) qui appartient à E. Ce qui explique pourquoi la distance de (a, b) à E est donnée par la distance de (a, b) à (x^*, y^*).

En ce qui concerne l'implémentation, on décide, pour des raisons de performances, d'appliquer la méthode de gradient à pas optimal avec $\varepsilon = 0.001$. Comme vu précédemment, on aura besoin :

- D'un point d'initialisation $(x^{(0)}, y^{(0)}) = (-2, 2)$
- Du gradient de f_ε :

$$\nabla f_\varepsilon(x,y) = \begin{pmatrix} 2(x-a) - \dfrac{8x}{\varepsilon}(2x^2+y^2-1) \\ 2(y-b) + \dfrac{4y}{\varepsilon}(2x^2+y^2-1) \end{pmatrix}$$

- De la matrice Hessienne de f_ε :

$$\mathrm{Hess} f_\varepsilon(x,y) = \begin{pmatrix} 2 + \dfrac{8}{\varepsilon}(6x^2+y^2-1) & \dfrac{16xy}{\varepsilon} \\ \dfrac{16xy}{\varepsilon} & 2 + \dfrac{4}{\varepsilon}(2x^2+3y^2-1) \end{pmatrix}$$

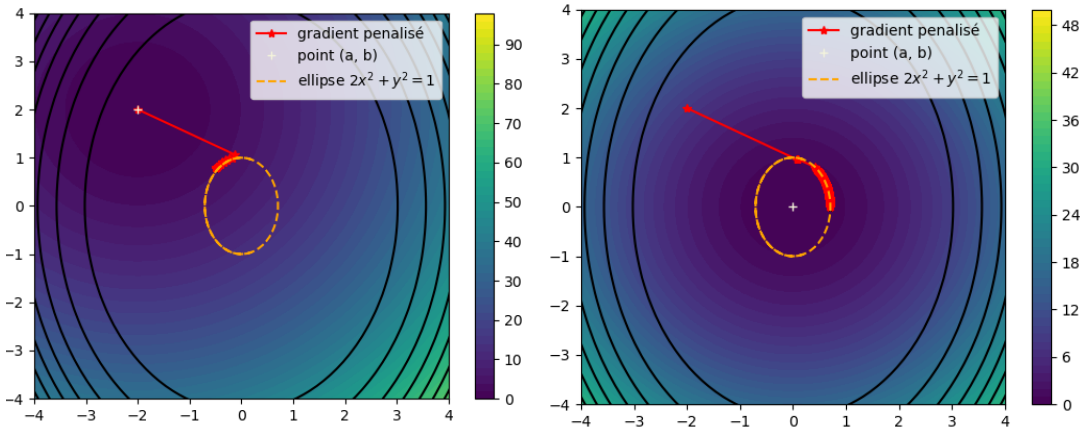
Comme à la question 1, on se sert de la méthode de Newton (voir Listing 4) pour minimiser $q(\rho) = f_\varepsilon(\mathbf{x} + \rho\mathbf{d})$, on a :

$$q'(\rho) = \langle \nabla f_\varepsilon(\mathbf{x} + \rho\mathbf{d}), \mathbf{d} \rangle$$

$$q''(\rho) = \langle \mathrm{Hess} f_\varepsilon(\mathbf{x} + \rho\mathbf{d})\mathbf{d}, \mathbf{d} \rangle$$

Où $\langle \cdot,\cdot \rangle$ désigne le produit scalaire associe à la norme euclidienne dans $\mathbb{R}^2$.

Pour tester la méthode, on choisit deux cas : $(a,b) = (-2,2)$ et $(a,b) = (0,0)$. On obtient les résultats suivants (voir Listing 7 – fichier *Ex2_question2.py*) :



$(a) : (a,b) = (-2,2)$ $(b) : (a,b) = (0,0)$

Figure 6 : Détermination de la distance de (a,b) à l'ellipse

Les distances observées sur la figure 6 sont récapitulées ci-dessous :

Point (a,b)	**Minimum obtenu en (x^*,y^*)**	**Distance de (a,b) à E**
$(-2,2)$	$(-0.4647, 0.7542)$	1.9771
$(0,0)$	$(7.0702 \times 10^{-1}, 4.9683 \times 10^{-9})$	0.7070

Table 4 : Distance du point (a, b) à E par optimisation sous contrainte par pénalisation

On confirme que la méthode fonctionne correctement. Elle nous donne un moyen numérique de calculer la distance d'un point à un sous-ensemble (différentiable) de $\mathbb{R}^2$.

EXERCICE 3 (Méthode de gradient, du gradient conjugué)

En utilisant la méthode du gradient conjugué, on souhaite obtenir un minimum à 10^{-12} près de la fonction

$$J_n : \mathbb{R}^n \ni \mathbf{x} \mapsto \frac{1}{2}\langle A_n\mathbf{x}, \mathbf{x} \rangle_{\mathbb{R}^n} - \langle \mathbf{b}_n, \mathbf{x} \rangle_{\mathbb{R}^n}$$

Avec

$$A_n = \begin{pmatrix} 4 & -2 & 0 & \cdots & 0 \\ -2 & 4 & -2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -2 & 4 & -2 \\ 0 & \dots & 0 & -2 & 4 \end{pmatrix} \quad \text{et} \quad \mathbf{b}_n = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

Cas n = 2

Courbes de niveau et gradient

En dimension 2, on pose $\mathbf{x} = (x_1, x_2)$. Pour obtenir les courbes de niveau de J_2 et le champ de vecteur correspondant à ∇J_2, il faut expliciter leurs expressions respectives. On obtient :

$$J_2(x_1, x_2) = 2x_1^2 + 2x_2^2 - 2x_1x_2 - x_1 - x_2$$

$$\nabla J_2(x_1, x_2) = \begin{pmatrix} 4x_1 - 2x_2 - 1 \\ -2x_1 + 4x_2 - 1 \end{pmatrix}$$

En utilisant les fonctions ***contourf()*** et ***quiver()*** de la librairie Matplotlib, on obtient les résultats suivants :

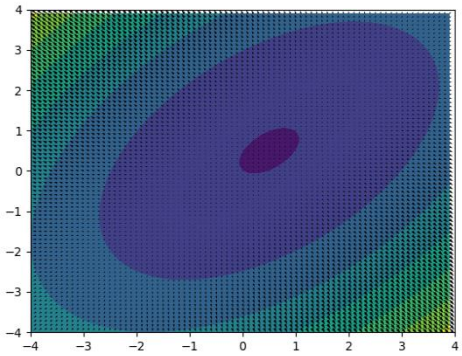
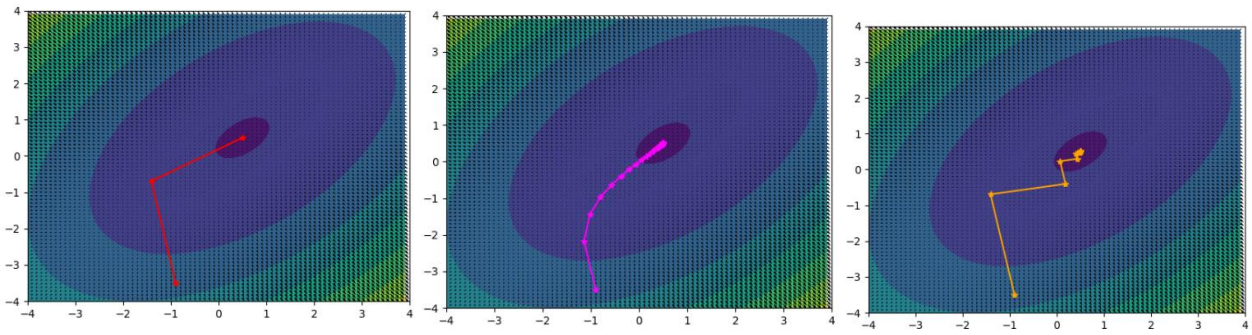


Figure 7 : Lignes de niveau et gradient de la fonction J_2

Solution par différentes méthodes de gradient

On implémente ensuite la méthode du gradient conjugué, et on compare avec les méthodes de gradient à pas fixe ($\rho = 10^{-1}$) et à pas optimal (déjà implémentées à l'exercice 2). Toutes ces méthodes sont prises avec $\mathbf{x}^{(0)} = \begin{pmatrix} -0.9 \\ -3.5 \end{pmatrix}$ comme position initiale. Les programmes utilisés pour obtenir les figures ci-dessous sont indiqués en annexe (voir Listing 9, Listing 11, et Listing 10 – fichier *Ex3_gradconj.py*).



(a) : Gradient conjugué *(b) : Gradient à pas fixe* *(c) : Gradient à pas optimal*

Figure 8 : Convergence des différentes méthodes de gradient

Un graphe récapitulatif est donné ci-dessous.

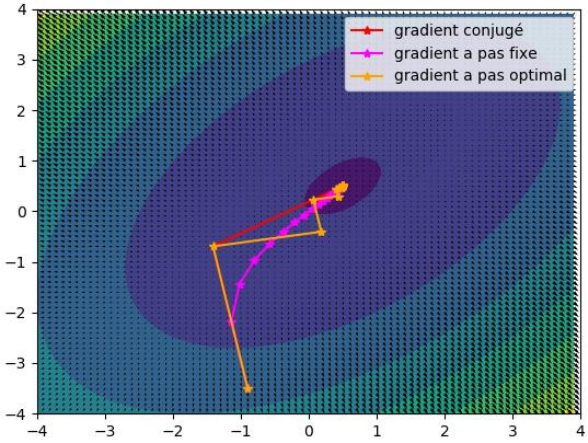


Figure 9 : Comparaison des différentes méthodes de gradient

On constate immédiatement que la méthode du gradient conjugué est de loin la plus efficace. Elle converge vers la solution exacte $x_{exact} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ en seulement **2** itérations, alors que la méthode de gradient à pas fixe $\rho = 10^{-1}$ demande considérablement plus d'itérations (environ **124**). On notera néanmoins que le choix d'un meilleur pas (en l'occurrence $\rho = 2.5 \times 10^{-1}$) permet de réduire cette disparité. Quant à la méthode de gradient à pas optimal, elle nécessite en général moins de **50** itérations, ce qui reste bien loin des performances obtenues avec le gradient conjugué.

Cas n = 10, 20, 30, 50, 100

Les simulations sont effectuées avec pour vecteur initial $\mathbf{x}^{(0)} = (0, \dots, 0)$. Le pas pour la méthode du gradient à pas fixe reste $\rho = 10^{-1}$. Pour la méthode du gradient à pas optimale, on se sert dans un premier temps de la méthode de Newton (avec une précision de 10^{-2}, et une initialisation de ρ à 0) pour trouver le pas optimal à chaque itération (voir Listing 4). Les nombres d'itérations nécessaires sont indiqués dans le tableau 5 ci-après. La figure 10 compare les erreurs de convergence.

n	Gradient conjugué	Gradient à pas fixe	Gradient à pas optimal
10	5	1619	638
20	10	5974	2430
30	15	13114	5432
50	25	35819	14857
100	50	141944	58874

Table 5 : Nombre d'itérations

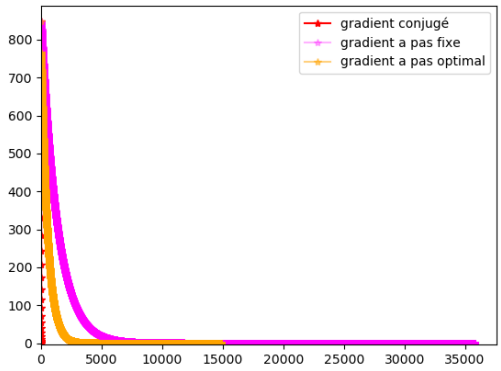


Figure 10 : Comparaison des erreurs de convergence pour n = 50

Le tableau 5 montre qu'en général, la méthode de gradient conjugue est radicalement plus performante que les autres, surtout lorsque la dimension du problème augmente. Elle nécessite toujours moins de n itérations, peu importe le point d'initialisation. La méthode de gradient à pas fixe nécessite un nombre relativement élevé d'itération, bien qu'au bout de 8000 itérations, on est déjà assez proche de la solution recherchée (figure 10).

Comparons à présent les temps d'exécution. Le tableau 6 indique les temps d'exécution approximatif (en secondes).

n	Gradient conjugué	Gradient à pas fixe	Gradient à pas optimal
10	0.001	0.110	0.146
20	0.003	0.475	0.590
30	0.003	0.917	1.417
50	0.004	3.124	4.319
100	0.036	36.293	79.477

Table 6 : Temps d'exécution (s)

Ce tableau permet de confirmer l'efficacité de la méthode du gradient conjugué. Il nous indique aussi que malgré le nombre d'itérations élevé de la méthode de gradient à pas fixe, elle reste plus rapide que la méthode de gradient à pas optimal. Cette relative lenteur est due à la recherche du pas optimal implémentée avec la méthode de Newton (voir Listing 4).

Ceci dit, la fonction à minimiser est une fonction quadratique et la matrice A_n est symétrique définie positive. On connait donc exactement l'expression de son pas optimal. A l'itération k, il est donné par :

$$\rho^{(k)} = \frac{\left\|\mathbf{d}^{(k)}\right\|^2}{\langle A\mathbf{d}^{(k)}, \mathbf{d}^{(k)}\rangle}, \text{ où } \mathbf{d}^{(k)} = A\mathbf{x}^{(k)} - \mathbf{b}$$

On remplace donc la fonction ***newtonmin()*** (voir Listing 4) par la fonction ***calculPasOptimal()*** (voir Listing 8) dans la méthode du gradient à pas optimal (voir Listing 10) et on refait la comparaison pour obtenir les deux tableaux suivants :

n	Gradient conjugué	Gradient à pas fixe	Gradient à pas optimal
10	5	1619	638
20	10	5974	2430
30	15	13114	5432
50	25	35819	14885
100	50	141944	59078

Table 7 : Nombre d'itérations

n	Gradient conjugué	Gradient à pas fixe	Gradient à pas optimal
10	0.001	0.110	0.081
20	0.003	0.475	0.358
30	0.003	0.917	0.805
50	0.004	3.124	2.713
100	0.036	36.293	45.844

Table 8 : Temps d'exécution (s)

Cette astuce nous permet d'augmenter l'efficacité de la méthode du gradient à pas optimal. Elle reste néanmoins la plus lente pour des valeurs de n élevées ($n = 100$ par exemple), le calcul de ce pas optimal ayant toujours un coût non négligeable.

Tout ceci nous permet de conclure que la méthode du gradient conjugué est à préconiser dès que la situation le permet. En effet, elle n'est valable que pour des matrices symétriques définies positives, ce qui constitue une limitation non négligeable.

TP 2 - Méthode du gradient projeté, d'Uzawa

On souhaite résoudre le problème $\min_{v \in K} J(v)$ avec

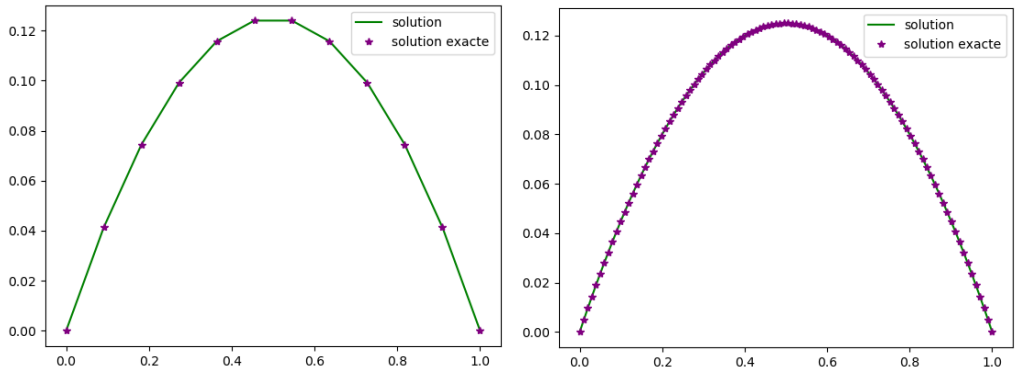
$$J(v) = \frac{1}{2}(Av, v) - (b, v) \qquad \text{et} \qquad K = \{v \in \mathbb{R}^n : v \geq g\}$$

1. Écriture d'une méthode de gradient à pas fixe

Dans un premier temps, on minimise la fonction J sur $\mathbb{R}^n$. On choisit la méthode de gradient à pas fixe dont l'algorithme fut donné lors du TP 1. On le rappelle ci-dessous :

choisir $\rho = \dfrac{2}{\lambda_1(A) + \lambda_n(A)}$
poser $k = 0$
choisir $v^{(0)} = (0, \dots, 0)$
tant que $\left\| v^{(k+1)} - v^{(k)} \right\|_{\mathbb{R}^n} \geq 10^{-12}$ **et** $k \leq 10^6$ **faire**
    calculer $d^{(k)} = -\nabla J\left(v^{(k)}\right)$
    poser $v^{(k+1)} = v^{(k)} + \rho d^{(k)}$
    poser $k = k + 1$
fin tant que

La solution exacte de ce problème est le vecteur v (s'il existe), tel que $Av = b$. On l'obtient à l'aide de la fonction ***solve()*** du module Numpy. On implémente notre algorithme et on vérifie la convergence à travers les figures suivantes (voir Listing 15 - fichier *TP2_gradproj_uzawa.py*).



(a): n=10 *(b): n=100*

Figure 11: Méthode du gradient à pas fixe (pas choisi optimal pour une fonction quadratique)

Le graphe ci-dessous permet d'observer l'erreur en norme à chaque itération. Il permet de mieux observer la convergence de la méthode.

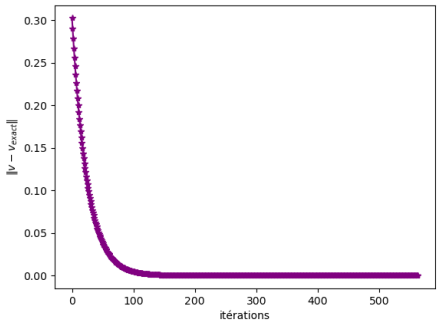


Figure 12 : Convergence de la méthode de gradient à pas fixe pour n = 10

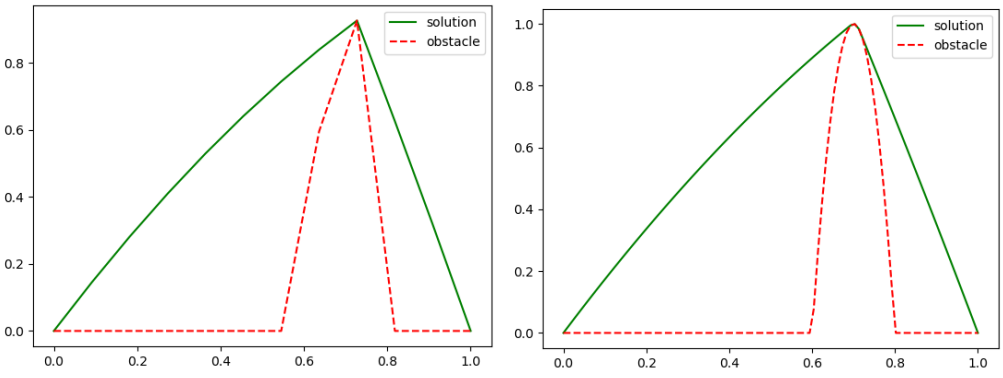
2. Méthode du gradient projeté à pas constant

On adapte le programme précèdent pour programmer la méthode de gradient projeté. On y a ajouté la projection sur l'ensemble des contraintes $K = \{v \in \mathbb{R}^n : v \geq g\}$ pour obtenir l'algorithme suivant :

choisir $\rho = \dfrac{2}{\lambda_1(A) + \lambda_n(A)}$
poser $k = 0$
choisir $v^{(0)} = (0, \dots, 0)$
tant que $\left\|v^{(k+1)} - v^{(k)}\right\|_{\mathbb{R}^n} \geq 10^{-12}$ **et** $k \leq 10^6$ **faire**
    calculer $d^{(k)} = -\nabla J\left(v^{(k)}\right)$
    poser $v^{(k+1)} = \max\left(v^{(k)} + \rho d^{(k)}, g\right)$
    poser $k = k + 1$
fin tant que

3. Vérification de la méthode

Après exécution de l'algorithme, on obtient les figures suivantes (voir Listing 14 – fichier *TP2_gradproj_uzawa.py*) :



(a): n=10 *(b): n=100*

Figure 13: Méthode du gradient projeté

On vérifie effectivement qu'en chaque point x, la condition $(-u''(x) - 1)(u(x) - g(x)) = 0$ est vérifiée. La figure 13 nous permet en outre d'observer que $-u''(x) \geq 1$, et que $u(x) \geq g(x)$(voir lignes 186-188 du fichier *TP2_gradproj_uzawa.py*).

4. Algorithme d'Uzawa

Après avoir posé,

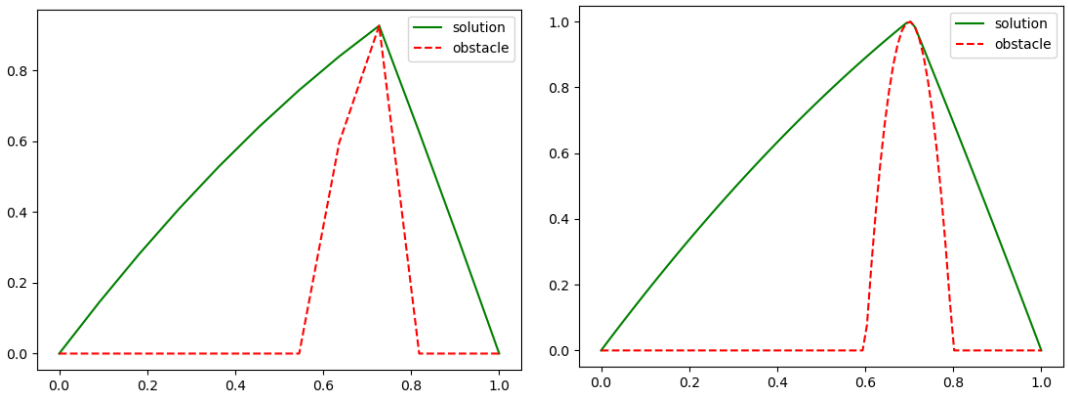
$$ctrt(v) = g - v$$
$$\nabla ctrt(v) = \begin{pmatrix} -1 & & \\ & \ddots & \\ & & -1 \end{pmatrix}$$
$$\mathcal{L}(v, \mu) = J(v) + \mu \cdot ctrt(v)$$
$$\nabla_v \mathcal{L}(v, \mu) = \nabla J(v) + \nabla ctrt(v)^T \cdot \mu$$

Nous utilisons l'algorithme d'Uzawa indiqué ci-dessous (on marque en vert l'étape de minimisation du lagrangien $\mathcal{L}$ par rapport à v qui sera réexaminée plus tard) :

choisir $\rho = 9.8$
poser $k = 0$
choisir $\mu^{(0)} = (0, \dots, 0)$ et $v^{(0)} = (0, \dots, 0)$
tant que $\left\| v^{(k+1)} - v^{(k)} \right\|_{\mathbb{R}^n} \geq 10^{-12}$ **et** $k \leq 10^6$ **faire**

  choisir $\rho_v = \dfrac{2}{\lambda_1(A) + \lambda_n(A)}$
  poser $l = 0$ **et** $v^{(0)} = v^{(k)}$
  tant que $\left\| v^{(l+1)} - v^{(l)} \right\|_{\mathbb{R}^n} \geq 10^{-4}$ **et** $k \leq 10^3$ **faire**
    calculer $d^{(l)} = -\nabla_v \mathcal{L}\left(v^{(l)}, \mu^{(k)}\right)$
    poser $v^{(l+1)} = v^{(l)} + \rho_v d^{(l)}$
    poser $l = l + 1$
  fin tant que
  poser $\mu^{(k+1)} = \max\left(0, \mu^{(k)} + \rho\, ctrt\left(v^{(l)}\right)\right)$
  poser $k = k + 1$
fin tant que

On obtient les résultats suivants après implémentation de l'algorithme d'Uzawa pour l'obstacle de base $g(x) = \max(0, 1 - 100(x - 0.7)^2)$ (voir Listing 16 – fichier *TP2_gradproj_uzawa.py*).



(a): $n = 10$ et $\rho = 9.8$ *(b): $n = 100$ et $\rho = 9.8$*

Figure 14: Méthode d'Uzawa

Le choix de ρ est conditionné par le théorème de convergence de l'algorithme d'Uzawa. Ce théorème nous garantit que si $\rho \in]0, \alpha/M^2]$, l'algorithme d'Uzawa converge vers la solution recherchée v^*. Ici α et M sont définis tels que J soit α-convexe, et la contrainte $ctrt$ soit M-lipschitzienne. Or pour nos choix de n, la matrice A est symétrique définie positive, donc J est λ_1-convexe (λ_1 étant la plus petite valeur propre de A) et la contrainte

$ctrt(v) = g - v$ est 1-lipschitzienne. Dans cette étude, nous choisissons donc $\boldsymbol{\rho = 9.8 \leq \lambda_1}$ pour avoir les résultats de convergence les plus rapides.

Comme avec la méthode de gradient projeté, on vérifie bien que si $u(x) \neq g(x)$ alors $-u''(x) = 1$ et vice-versa, ce qui nous permet de confirmer notre approche (voir lignes 303-305 du fichier *TP2_gradproj_uzawa.py*).

Comparaison des méthodes

Les tableaux suivants permettent de comparer le nombre d'itérations et le temps d'exécution pour les méthodes de gradient projeté et d'Uzawa avec l'obstacle $g(x) = \max(0, 1 - 100(x - 0.7)^2)$.

n	Gradient projeté	Uzawa
5	78	164
10	324	377
100	21919	44295
500	482579	877056

Table 10 : Comparaison du nombre d'itérations

n	Gradient projeté	Uzawa
5	0.0030	0.0250
10	0.0090	0.0840
100	4.3771	22.6770
500	135.5049	1864.1173

Table 9 : Comparaison du temps d'exécution (en seconde)

Qu'on regarde le nombre d'itérations ou le temps d'exécution, la méthode du gradient projeté est plus efficace que la méthode d'Uzawa. En particulier, lorsque $n = 500$, l'algorithme d'Uzawa prends 1864 secondes (plus de **30 minutes**) pour converger ; alors que la méthode du gradient projeté n'a besoin que de 135 secondes (soit **2 minutes**) sous les même conditions (même tolérance = 10^{-12} , mêmes conditions initiales, etc.).

L'inefficacité de la méthode d'Uzawa est dû à la rigueur que nous imposons lors de l'étape de minimisation du lagrangien en fonction de v. Comme indiqué en vert dans l'algorithme d'Uzawa, nous avons opté pour la méthode du gradient à pas optimal avec une tolérance de 10^{-4} et un nombre d'itérations maximal de 10^3. Si nous voulons comparer plus honnêtement la méthode d'Uzawa à la méthode de gradient projeté, il serait judicieux d'optimiser cette étape.

Algorithme d'Uzawa optimisé

La définition du lagrangien est telle que nous pouvons optimiser l'algorithme d'Uzawa en remplaçant l'étape de minimisation par la résolution d'un système linéaire.

Le lagrangien est défini comme ceci :

$$\begin{aligned}\mathcal{L}(v, \mu) &= J(v) + \mu \cdot ctrt(v) \\ &= \frac{1}{2}\langle Av, v\rangle - \langle b, v\rangle + \langle \mu, g - v\rangle \\ &= \frac{1}{2}\langle Av, v\rangle - \langle b', v\rangle + \langle \mu, g\rangle \quad \text{avec} \quad b' = b + \mu\end{aligned}$$

Minimiser $\mathcal{L}$ par rapport à v revient donc à prendre $v^* = A^{-1}(b + \mu)$ (pour les cas que nous traitons ($n \geq 5$), la matrice A est symétrique définie positive, donc inversible).

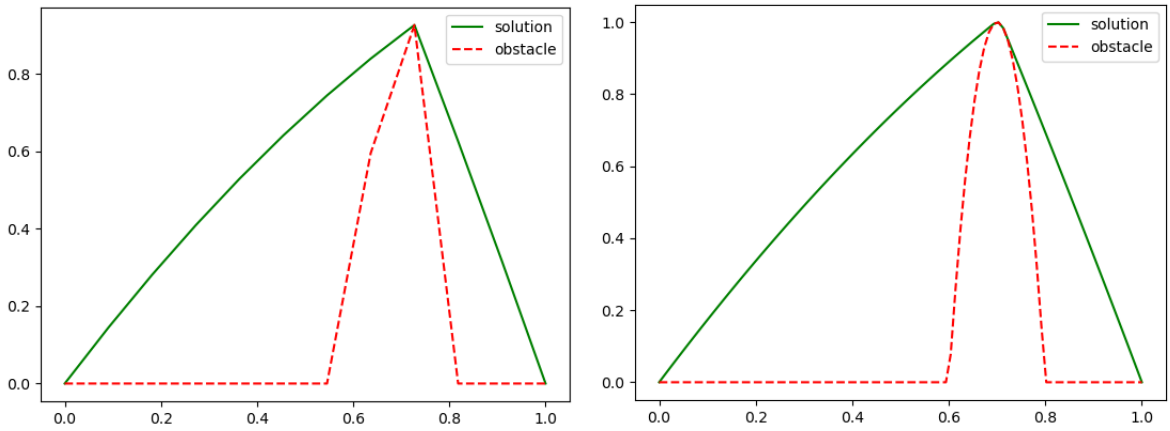
L'algorithme d'Uzawa optimisé est donc :

choisir $\rho = 9.8$
poser $k = 0$
choisir $\mu^{(0)} = (0, \dots, 0)$ et $v^{(0)} = (0, \dots, 0)$
tant que $\left\| v^{(k+1)} - v^{(k)} \right\|_{\mathbb{R}^n} \geq 10^{-12}$ **et** $k \leq 10^6$ **faire**

$$\textbf{poser } v^{(k+1)} = A^{-1}\left(b + \mu^{(k)}\right)$$
$$\textbf{poser } \mu^{(k+1)} = \textbf{max}\left(0, \mu^{(k)} + \rho\, ctrt\left(v^{(k+1)}\right)\right)$$

poser $k = k + 1$
fin tant que

On implémente cet algorithme et on obtient effectivement les mêmes figures qu'avec l'algorithme d'Uzawa ordinaire (voir Listing 17 – fichier *TP2_gradproj_uzawa.py*).



(a): n=10 et $\rho = 9.8$ *(b): n=100 et $\rho = 9.8$*

Figure 15: Méthode d'Uzawa optimisée

On refait la comparaison entre cette nouvelle méthode et la méthode du gradient projeté.

n	Gradient projeté	Uzawa optimisée
5	78	62
10	324	141
34	2794	4970
100	21919	43691
500	482579	620969

Table 11 : Comparaison du nombre d'itérations

n	Gradient projeté	Uzawa optimisée
5	0.0020	0.0040
10	0.0090	0.0060
34	0.0930	0.1630
100	4.3771	8.8380
500	135.5049	189.3455

Table 12 : Comparaison du temps d'exécution (en seconde)

L'observation du nombre d'itérations (tableau 11) permet de remarquer que la méthode du gradient conjugué reste plus performante, mais seulement pour $n \geq 34$.

Quant au temps d'exécution, le tableau 12 nous permet de constater que la méthode du gradient conjugué reste plus rapide. Cependant cet avantage est moins prononcé par rapport à la comparaison de la section précédente, ou nous avons utilisé l'algorithme d'Uzawa ordinaire.

Réduction de la tolérance

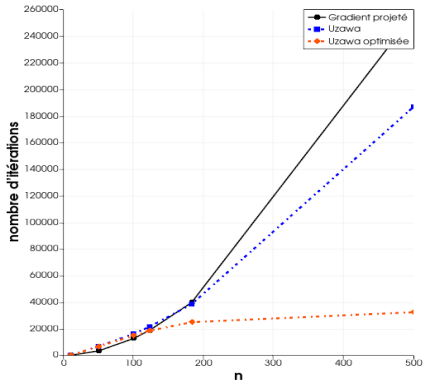
On modifie les algorithmes de gradient projeté, d'Uzawa et d'Uzawa optimisé en remplaçant la tolérance. Cette fois, la condition d'arrêt est $\left\|v^{(k+1)} - v^{(k)}\right\|_{\mathbb{R}^n} \geq 10^{-8}$ **et** $k \leq 10^6$. Les mêmes programmes qu'aux Listing 14, Listing 16, et Listing 17 sont utilisés à une différence près. On refait donc la comparaison des trois méthodes et on obtient les résultats suivants.

n	Gradient projeté	Uzawa	Uzawa optimisée
10	207	199	94
50	3457	6864	6745
100	12777	16333	14864
123	18951	21601	18661
183	39899	38607	25093
500	256512	187037	32585

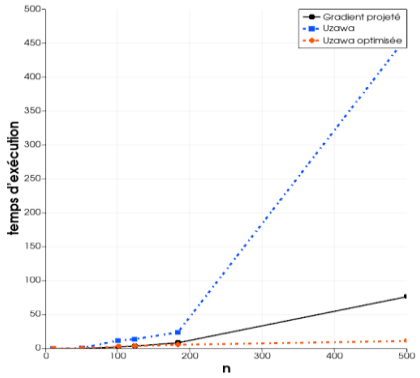
Table 13 : Comparaison du nombre d'itérations pour une tolérance de 10^{-8}

n	Gradient projeté	Uzawa	Uzawa optimisée
10	0.0050	0.0700	0.0090
50	0.1020	0.9160	0.2420
100	2.4290	11.8800	3.0520
123	3.7360	13.9460	3.8250
183	8.5590	23.9590	5.6910
500	76.4982	459.4660	11.2710

Table 14 : Comparaison du temps d'exécution (s) pour une tolérance de 10^{-8}



(a): Nombre d'itérations



(b): Temps d'exécution (s)

Figure 16: Illustrations des comparaisons des méthodes du gradient projeté, d'Uzawa, et d'Uzawa optimisée pour une tolérance de 10^{-8}

Le nombre d'itérations traduit un effet contraire à celui précédemment observé. La méthode d'Uzawa devient meilleure avec l'augmentation de la taille du problème. En général, à partir de $n = 123$, la methode d'Uzawa optimisée demande moins d'itérations que la méthode du gradient projeté. Il faut par contre attendre jusqu'à $n = 183$ pour que la méthode d'Uzawa ordinaire devienne meilleur.

En termes de temps d'exécution, il constate bien que la méthode d'Uzawa optimisée s'exécute en général plus rapidement que la méthode du gradient projeté, contrairement à l'étude précédente faite avec une tolérance de 10^{-12}.

Il est évident que la méthode d'Uzawa optimisée surpasse sa version ordinaire sur tous les plans. On conclut donc qu'il faut préconiser la méthode du gradient projeté pour des problèmes de petite taille ($n < 123$), et la méthode d'Uzawa optimisée dans le cas contraire.

Testons d'autres choix d'obstacles

On choisit trois obstacles qui sont généralement lissés (cas *b* et *c*) pour conserver la propriété de différentiation. Mais cette fois, la comparaison des méthodes ne s'effectue que pour $n = 10, 100, 500$ et la tolérance vaut 10^{-8}. Comme précédemment, le pas pour la méthode d'Uzawa utilisée est de $\rho = 9.8$.

a. *Un saut d'obstacles*

On considère pour cette section l'obstacle

$$g(x) = \begin{cases} 0.25 & si\ 0.2 \leq x \leq 0.21 \\ 0.3 & si\ 0.4 \leq x \leq 0.41 \\ 0.45 & si\ 0.6 \leq x \leq 0.61 \\ 0.15 & si\ 0.8 \leq x \leq 0.81 \\ 0 & sinon \end{cases}$$

On obtient les graphes suivants (voir Listing 14 et Listing 17 – fichier *TP2_gradproj_uzawa.py*) :

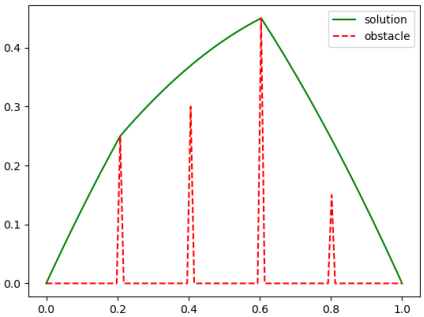


Figure 17 : Gradient projeté avec n=100

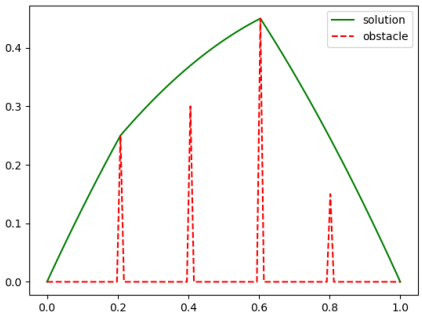


Figure 18 : Uzawa optimisée avec n=100

Le tableau de comparaison est le suivant :

n	Gradient projeté	Uzawa optimisée
10	341	2
100	4382	1089
500	88592	140560

Table 15 : Nombre d'itérations pour un obstacle en forme de saut d'obstacles

n	Gradient projeté	Uzawa optimisée
10	0.0090	0.0060
100	0.9240	0.2550
500	27.9518	44.8970

Table 16 : Temps d'exécution (en seconde) pour un obstacle en forme de saut d'obstacles

La méthode du gradient projeté devient plus efficace à partir d'un certain rang, dans ce cas $n \geq 500$.

b. *Un plateau*

On considère pour cette section l'obstacle

$$g(x) = \begin{cases} 1 & si\ 0.4 \leq x \leq 0.8 \\ 0 & sinon \end{cases}$$

On obtient les graphes suivants (voir Listing 14 et Listing 17 – fichier *TP2_gradproj_uzawa.py*) :

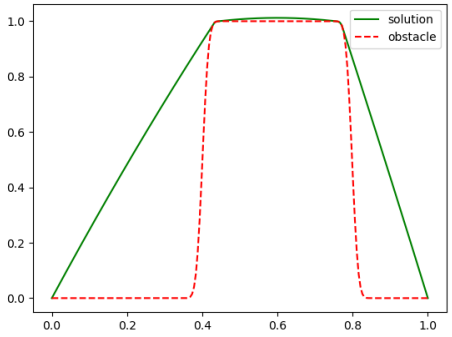


Figure 19 : Gradient projeté avec n=200

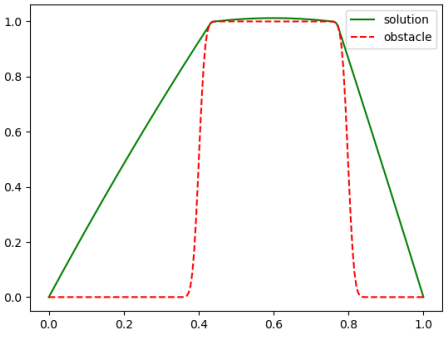


Figure 20 : Uzawa optimisée avec n=200

Le tableau de comparaison est :

n	Gradient projeté	Uzawa optimisée
10	82	194
100	4769	5938
500	95186	128201

Table 17 : Nombre d'itérations pour un obstacle en forme de plateau

n	Gradient projeté	Uzawa optimisée
10	0.0040	0.0130
100	0.9890	1.2140
500	29.7671	40.8730

Table 18 : Temps d'exécution (en seconde) pour un obstacle en forme de plateau

Dans la lancée du précédent obstacle, ce choix d'obstacle semble indiquer que l'algorithme d'Uzawa, même optimisée, est moins efficace que le gradient projeté. Ceci autant en nombre d'itérations qu'en temps d'exécution.

Le point commun entre cet obstacle et le précèdent est que chacun d'eux vaut 0 sur une portion moyenne du domaine.

c. *Une cuve*

On considère pour cette section l'obstacle

$$g(x) = \begin{cases} 1 + x^3 - x & si\ 0.05 \leq x \leq 0.95 \\ 0 & sinon \end{cases}$$

On obtient les graphes suivants (voir Listing 14 et Listing 17 – fichier *TP2_gradproj_uzawa.py*):

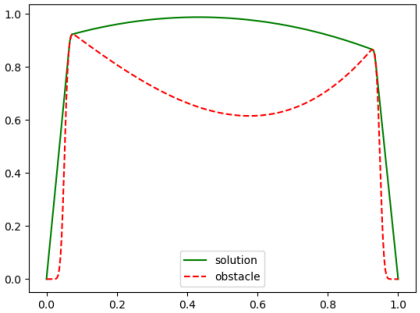


Figure 21 : Gradient projeté avec n=100

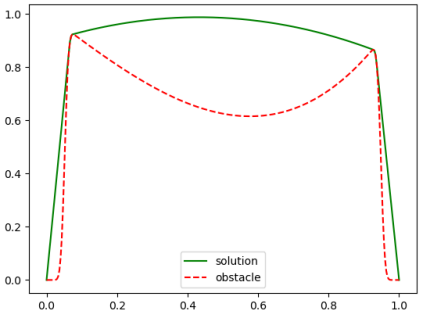


Figure 22 : Uzawa optimisée avec n=100

Le tableau de comparaison est :

n	Gradient projeté	Uzawa optimisée
10	247	225
100	19296	5784
500	393376	106954

Table 19 : Nombre d'itérations pour un obstacle en forme de cuve

n	Gradient projeté	Uzawa optimisée
10	0.0070	0.0180
100	3.9420	1.2230
500	115.0060	32.4700

Table 20: Temps d'exécution (en seconde) pour un obstacle en forme de cuve

Avec cet obstacle omniprésent sur presque tout le domaine, on remarque que la méthode d'Uzawa est presque quatre fois plus rapide que la méthode du gradient projeté. On se rappelle aussi que pour l'obstacle de base qui était très peu présent sur le domaine, la méthode d'Uzawa devenait plus efficace à partir de $n = 123$.

En conclusion générale, afin de résoudre un problème d'optimisation comme le nôtre, il serait intéressant d'utiliser l'algorithme d'Uzawa sous sa version optimisée lorsque la taille de l'obstacle sur le domaine est **soit trop grande, soit trop petite**. Dans les cas contraires, il est recommandé d'utiliser la méthode du gradient projeté.

ANNEXE

Les sections les plus marquantes des programmes utilisés sont indiquées ci-dessous. Aussi sont précisées les figures illustrant ces programmes, et les fichiers dans lesquels ils sont écrits. Cependant, les lignes de codes pour afficher les résultats à l'écran ne sont pas données ici, elles sont écrites et commentées dans les fichiers correspondants à chaque exercice.

EXERCICE 1 (Optimisation 1D sans contrainte)

```python
def f(x):
    return x/8 + np.sqrt((6-x)**2+4)/3

def f_prime(x):
    return 1/8 + (1/3)*(x-6) / np.sqrt((6-x)**2+4)

def f_second(x):
    return (4/3) / (((6-x)**2+4)*np.sqrt((6-x)**2+4))
```

Listing 1: Quelques fonctions pour l'exercice 1
Ex1_dicho.py et Ex1_newton.py

```python
xx = [] #liste
x0 = 0 # borne gauche de l'intervalle initial
x1 = 6 # borne droite de l'intervalle initial
xx.append((x0+x1)/2) # ajouter a une liste
compteur = 0
tau = 3     # facteur de reduction
while ((x1-x0)>1e-10 and compteur<100):
    xt0 = x0 + (1-1/tau)*(x1-x0)/2
    xt1 = x0 + (1+1/tau)*(x1-x0)/2
    M0 = f(x0)
    M1 = f(x1)
    if M0 < M1:
        x1 = xt1
    elif M0 > M1:
        x0 = xt0
    else:
        x0 = xt0
        x1 = xt1
    xx.append((x0+x1)/2) # ajouter a une liste
    compteur=compteur+1
```

Listing 2: Méthode de dichotomie
– Figure 1 –
Ex1_dicho.py

```python
xx = [] #liste
# x = 3.202          # x_0_min pour l'initalisation qui converg
x = 7.662            # x_0_max pour une initalisation qui conv
x_suiv = x - f_prime(x)/f_second(x) # position suivante
xx.append(x) # ajouter a une liste
compteur = 0
while (abs(x_suiv-x)>1e-10 and compteur<100):
    x = x_suiv
    x_suiv = x - f_prime(x)/f_second(x)
    xx.append(x) # ajouter a une liste
    compteur=compteur+1
```

Listing 3: Méthode de Newton
– Figure 2 –
Ex1_newton.py

EXERCICE 2 (Optimisation 2D avec contraintes par pénalisation)

```python
def f(x,y):
    return (x-a)**2 + (y-b)**2

def feps(x, eps):
    return (x[0]-a)**2 + (x[1]-b)**2 + (1/eps)*(2*x[0]**2 + x[1]**2 -

def gradfeps(x, eps):
    return np.array([2*(x[0]-a) + (8*x[0]/eps)*(2*x[0]**2+x[1]**2-1),
b) + (4*x[1]/eps)*(2*x[0]**2+x[1]**2-1)])

def hessfeps(x, eps):
    return np.array([[2 + (8/eps)*(6*x[0]**2+x[1]**2-
1), 16*x[0]*x[1]/eps], [16*x[0]*x[1]/eps, 2 + (4/eps)*(2*x[0]**2+3*x[1
```

Listing 5: Quelques fonctions pour l'exercice 2
Ex2_question1.py et Ex2_question2.py

```python
def newtonmin(x, d, feps, gradfeps, hessfeps, eps):
    # Fonction a minimiser
    def q(rho, eps):
        return feps(x + rho*d, eps)
    def q_prime(rho, eps):
        return gradfeps(x + rho*d, eps)@d.T
    def q_second(rho, eps):
        return (hessfeps(x + rho*d, eps)@d) @ d.T

    rho = 0 # position initiale
    rho_suiv = rho - q_prime(rho, eps)/q_second(rho, eps) # position s
    compteur = 0
    while (abs(rho_suiv-rho)>1e-10 and compteur<100):
        rho = rho_suiv
        rho_suiv = rho - q_prime(rho, eps)/q_second(rho, eps)
        compteur=compteur+1

    return rho
```

Listing 4: Méthode de Newton utilisée pour la détermination du pas optimal - Figures 4 et 6 -
Ex2_question1.py et Ex2_question2.py

```python
def gradientpasfixe(xinit, gradJ,eps):

    pas = 1e-4
    #---------------
    nbit = 0
    x = xinit.copy()
    xold = x.copy()

    grad = gradJ(x,eps)
    err = 1.
    xlist = [list(x)]

    while (err > 1e-12 and nbit < 1e5):
        xold[:] = x
        x -= pas * grad
        xlist.append(list(x))
        grad[:] = gradJ(x,eps)
        err = np.linalg.norm(x - xold)
        nbit += 1

    return x, np.array( xlist), nbit
```

Listing 6: Méthode du gradient à pas fixe
– Figure 3 –
Ex2_question1.py et Ex2_question2.py

```python
def gradientpasoptimal(xinit, J, gradJ, HessJ, eps):

    nbit = 0
    x = xinit.copy()
    xold = x.copy()

    grad = gradJ(x,eps)
    err = 1.
    xlist = [list(x)]
    paslist = []

    while (err > 1e-12 and nbit < 1e5):
        xold[:] = x
        pas = newtonmin(x, -grad, J, gradJ, HessJ, eps)
        paslist.append(pas)
        x -= pas * grad
        xlist.append(list(x))
        grad[:] = gradJ(x,eps)
        err = np.linalg.norm(x - xold)
        nbit += 1

    return x, np.array(xlist), np.array(paslist), nbit
```

Listing 7: Méthode du gradient à pas optimal
– Figures 4, 6 –
Ex2_question1.py et Ex2_question2.py

EXERCICE 3 (Méthode de gradient, du gradient conjugué)

```python
def J(x, n):
    A = matrixA(n)
    b = np.ones((n), dtype=np.float64)
    return (x.T@A@x)/2 - b.T@x

# Gradient de J en dimension n quelconque
def gradJ(x, n):
    A = matrixA(n)
    b = np.ones((n), dtype=np.float64)
    return A@x - b

# Matrice Hesseisnne de J
def HessJ(x, n):
    return matrixA(n)
```

Listing 7: Quelques fonctions pour l'exercice 3
Ex3_gardconj.py

```python
def calculPasOptimal(x, gradJ):
    n = len(x)
    A = matrixA(n)
    b = np.ones_like(x)

    d = gradJ(x, n) # d=A@x-b
    return (d.T@d)/(d.T@A@d)
```

Listing 8: Formule de calcul du pas optimal pour les fonctions quadratiques avec $A \in S_n^{++}(\mathbb{R})$
– Tables 7 et 8 –
Ex3_gradconj.py

```python
def gradConj(xinit, n):
    A = matrixA(n)
    b = np.ones((n), dtype=np.float64)
    # Initialisation
    r = A@xinit - b
    d = -r
    x = xinit.copy()
    xlist = [list(x)]
    # Conditions d'arret
    err = 1.        # norme du residu
    nbit = 0

    while (err > 1e-12 and nbit < 1e5):
        rho = (-r.T@d) / (d.T@A@d)
        x = x + rho*d

        rold = r.copy()
        r = A@x - b
        beta = (r.T@r) / (rold.T@rold)
        d = -r + beta*d

        xlist.append(list(x))
        err = np.linalg.norm(r)
        nbit += 1

    return x, np.array(xlist), nbit
```

Listing 9: Méthode de gradient conjugue
– Figures 8a et 9 –
Ex3_gradconj.py

```python
def gradPasFixe(xinit, gradJ, n):
    nbit = 0
    err = 1.

    pas = 1e-1
    x = xinit.copy()
    xold = x.copy()
    grad = gradJ(x, n)
    xlist = [list(x)]

    while (err > 1e-12 and nbit < 1e5):
        xold[:] = x
        x -= pas * grad
        xlist.append(list(x))
        grad[:] = gradJ(x, n)
        err = np.linalg.norm(x - xold)
        nbit += 1

    return x, np.array(xlist), nbit
```

Listing 11: Méthode du gradient à pas fixe
– Figures 8b et 9 –
Ex3_gradconj.py

```python
def gradPasOptimal(xinit, gradJ, HessJ, n):
    nbit = 0
    x = xinit.copy()
    xold = x.copy()

    grad = gradJ(x,n)
    err = 1.
    xlist = [list(x)]

    while (err > 1e-12 and nbit < 1e5):
        xold[:] = x
        # pas = newtonmin(x, -grad, gradJ, Hes
        pas = calculPasOptimal(x, gradJ)
        x -= pas * grad
        xlist.append(list(x))
        grad[:] = gradJ(x, n)
        err = np.linalg.norm(x - xold)
        nbit += 1

    return x, np.array(xlist), nbit
```

Listing 10: Méthode du gradient à pas optimal
– Figures 8c et 9 –
Ex3_gradconj.py

TP 2 - Méthode du gradient projeté, d'Uzawa

```python
#------ Fonction à minimiser J -------#
def J(A, b, v):
    X = A @ v
    J = 0.5 * (X @ v) - (b @ v)
    return J

#------ Gradient de J -------#
def gradJ(A, b, v):
    grad = A @ v - b
    return grad

#------ Contrainte pour le problème d'optimisation -
def ctrt(g, v):
    return g - v

#------ Matrice Jacobienne de la contrainte -------#
def gradctrt(v):
    return np.diag(-np.ones_like(v), k=0)

#------ Lagrangien L -------#
def L(A, b, g, v, mu):
    return J(A, b, v) + mu.T@ctrt(g, v)

#------ Gradient de L en fonction de v -------#
def gradL(A, b, v, mu):
    return gradJ(A, b, v) + gradctrt(v).T@mu
```

Listing 13: Quelques fonctions utiles pour le TP2 - TP2_gradproj_uzawa.py -

```python
#------ L'obstacle de base -------#
def g(x):
    return np.maximum(np.zeros_like(x),
                      1. - 100 * (x - 0.7)**2)

#------ Un obstacle en forme de saut d'obstacle ----
def g(x):
    ret = np.zeros_like(x)
    n_prime = len(x)
    for i in range(n_prime):
        if 0.2 <= x[i] <= 0.21:
            ret[i] = 0.6
        if 0.4 <= x[i] <= 0.41:
            ret[i] = 0.7
        if 0.6 <= x[i] <= 0.61:
            ret[i] = 1
        if 0.8 <= x[i] <= 0.81:
            ret[i] = 0.4
    return ret

#------ Un obstacle en forme de plateau -------#
def g(x):
    ret = np.where(0.4<=x, np.ones_like(x),
                           np.zeros_like(x))
    return np.where(x<=0.8, ret, np.zeros_like(x))

#------ Un obstacle en forme de cuve -------#
def g(x):
    ret = 1+x**3-x
    ret1 = np.where(0.05<=x, ret, np.zeros_like(x))
    return np.where(x<=0.95, ret1, np.zeros_like(x))
```

Listing 12: Quelques choix d'obstacles pour le TP2 - TP2_gradproj_uzawa.py -

```python
def gradient(A, b):
    val_propres = np.real(np.linalg.eig(A)[0])
    pas = 2 / (min(val_propres) + max(val_propres))
    nbit = 0
    err = 1.
    v = np.zeros_like(b)    # v initial
    vold = v.copy()
    grad = gradJ(A, b, v)
    vlist = [list(v)]

    while (err > 1e-8 and nbit < 1e6):
        vold[:] = v
        v -= pas * grad
        vlist.append(list(v))
        grad[:] = gradJ(A, b, v)
        err = np.linalg.norm(v - vold)
        nbit += 1

    return v, np.array(vlist), nbit
```

Listing 15: Méthode du gradient à pas fixe (sans contrainte) – Figure 11 – TP2_gradproj_uzawa.py

```python
def gradient_projete(A, b, gvec):
    val_propres = np.real(np.linalg.eig(A)[0])
    pas = 2 / (min(val_propres) + max(val_propres)
    nbit = 0
    err = 1.
    v = np.zeros_like(b)            # v initial
    v = np.maximum(v, gvec)         # projection
    vold = v.copy()
    grad = gradJ(A, b, v)
    vlist = [list(v)]

    while (err > 1e-8 and nbit < 1e6):
        vold[:] = v
        v -= pas * grad
        v = np.maximum(v, gvec)
        vlist.append(list(v))
        grad[:] = gradJ(A, b, v)
        err = np.linalg.norm(v - vold)
        nbit += 1

    return v, np.array(vlist), nbit
```

Listing 14: Méthode du gradient projeté – Figures 13, 17, 19, et 21 – TP2_gradproj_uzawa.py

```python
def uzawa(A, b, gvec):
    val_propres = np.real(np.linalg.eig(A)[0])

    pas = 9.8
    v = np.zeros_like(b)    # v initial
    mu = np.zeros_like(b)   # mu initial
    vold = v.copy()
    vlist = [list(v)]
    err = 1.
    nbit = 0

    pas_v = 2 / (min(val_propres) + max(val_propres))

    while (err > 1e-12 and nbit < 1e6):
        vold[:] = v

        # Minimisation du lagrangien par rapport a v
        err_v = 1.
        nbit_v = 0
        vold_v = v.copy()
        while (err_v > 1e-4 and nbit_v < 1e3):
            vold_v[:] = v
            grad = gradL(A, b, v, mu)
            v -= pas_v * grad
            err_v = np.linalg.norm(v - vold_v)
            nbit_v += 1

        # Maximisation du lagrangien par rapport a mu
        mu += pas*ctrt(gvec, v)
        mu = np.maximum(np.zeros_like(mu), mu)

        vlist.append(list(v))
        err = np.linalg.norm(v - vold)
        nbit += 1

    return v, np.array(vlist), nbit
```

Listing 16: Méthode d'Uzawa
- Figures 14-
TP2_gradproj_uzawa.py

```python
def uzawa_optimisee(A, b, gvec):
    pas = 9.8
    v = np.zeros_like(b)    # v initial
    mu = np.zeros_like(b)   # mu initial
    vold = v.copy()
    vlist = [list(v)]
    err = 1.
    nbit = 0

    inv_A = nplin.inv(A)

    while (err > 1e-12 and nbit < 1e6):
        vold[:] = v

        # Minimisations du lagrangien par rapport a v
        v = inv_A @ (b+mu)
        vlist.append(list(v))

        # Maximisation du lagrangien par rapport a mu
        mu += pas*ctrt(gvec, v)
        mu = np.maximum(np.zeros_like(mu), mu)

        err = np.linalg.norm(v - vold)
        nbit += 1

    return v, np.array(vlist), nbit
```

Listing 17: Méthode d'Uzawa optimisée
- Figures 15, 18, 20, et 22 -
TP2_gradproj_uzawa.py