Projet Optimisation A4

Léon Thomir et Hugo Yeremian

March 2023

1 Partie Théorique

On considèle la fonction f définie sur R^2 par

$$f(x,y) = x^4 + y^4 - 2(x-y)^2 \tag{1}$$

1. Montrer qu'il existe $(\alpha, \beta) \in R_+^2$ (et les déterminer) tels que $f(x,y) \geq \alpha\|(x,y)\|^2 - \beta$ pour tous $(x,y) \in R^2$, où la notation $\|.\|$ désigne la norme euclidienne sur R^2.
En déduire que le problème $\mathcal{P}$ donné par

$$\inf_{(x,y)\in R^2} f(x,y) \tag{2}$$

possède au moins une solution.
Comme fonction polynomiale, $f \in \mathcal{C}^\infty$. On a donc $\forall (x,y) \in R^2$

$$\begin{aligned} f(x,y) &= x^4 + y^4 - 2x^2 - 2y^2 + 4xy \\ &\geq x^4 + y^4 - 4x^2 - 4y^2 \end{aligned} \tag{3}$$

car $\forall (x,y) \in R^2, (x+y)^2 \geq 0 \Leftrightarrow xy \geq -\frac{x^2+y^2}{2}$.
On rappelle que $\forall (K,z) \in R^2, (K+z)^2K^4 + z^4 - 2zK^2 =\geq 0$, d'où

$$f(x,y) \geq 2(2z-4)x^2 + 2(z-4)y^2 - 2z^4 \tag{4}$$

En prenant $z = 4$, on a

$$\begin{aligned} f(x,y) &\geq 4(x^2+y^2) - 512 \\ &\geq 4\|(x,y)\|^2 - 512 \end{aligned} \tag{5}$$

donc avec $\alpha = 4$ et $\beta = 512$, on a $\forall (x,y) \in R^2, f(x,y) \geq \alpha\|(x,y)\|^2 - \beta$.
En utilisant $\alpha\|(x,y)\|^2 - \beta \xrightarrow[\|(x,y)\|^2 \to +\infty]{} +\infty$, et f coercive sur R^2 (car R^2 est fermée et de dimension finie), le problème $\mathcal{P}$ possède au moins une solution.

2. La fonction f est-elle convexe sur R^2 ?
Comme fonction polynomiale, $f \in \mathcal{C}^2$ sur R^2, on peut donc étudier les dérivées de f. $\forall (x,y) \in R^2$,

$\frac{\partial f(x,y)}{\partial x} = 4x^3 - 4x + 4y = 4x^3 + 4(y-x)$

$\frac{\partial f(x,y)}{\partial y} = 4y^3 + 4(y-x)$

$\frac{\partial^2 f(x,y)}{\partial x^2} = 12x^2 - 4$

$\frac{\partial^2 f(x,y)}{\partial y^2} = 12y^2 + 4$

$\frac{\partial^2 f(x,y)}{\partial x \partial y} = 4$

On définit la Hessienne H de f:

$$H(x,y) = \begin{bmatrix} 12x^2 - 4 & 4 \\ 4 & 12y^2 - 4 \end{bmatrix} \Rightarrow H(0,0) = 4\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6}$$

d'où $|H(0,0)| = 0$, donc la Hessienne n'est pas semi-définie positive en tout point. Ainsi, f n'est pas convexe en tout point. **3.** Déterminer les points critiques de f et préciser leur nature. Résoudre alors le problème dans $\mathcal{P}$.

On a $\forall (x,y) \in R^2$,

$$\begin{aligned} \nabla f(x,y) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} &\Leftrightarrow \begin{cases} x^3 - (x-y) = 0 \\ y^3 + (x-y) = 0 \end{cases} \\ &\Leftrightarrow \begin{cases} x^3 + y^3 = 0 \\ y^3 + (x-y) = 0 \end{cases} \\ &\Leftrightarrow \begin{cases} x = -y \\ x^3 - 2x = 0 \end{cases} \\ &\Leftrightarrow \begin{cases} x = -y \\ x(x^2-2) = 0 \end{cases} \end{aligned} \tag{7}$$

Donc les points critiques sont $\left\{(-\sqrt{2},\sqrt{2}); (\sqrt{2},-\sqrt{2}); (0,0)\right\}$

- Pour le point $(-\sqrt{2},\sqrt{2})$
  On a $H(-\sqrt{2},\sqrt{2}) = \begin{bmatrix} 20 & 4 \\ 4 & 20 \end{bmatrix}$, de trace 40 et de déterminant 384. Donc la Hessienne a deux valeurs propres strictement positives et $(-\sqrt{2},\sqrt{2})$ est un minimum local avec $f(-\sqrt{2},\sqrt{2}) = -8$
- Pour le point $(\sqrt{2},-\sqrt{2})$
  On a $H(\sqrt{2},-\sqrt{2}) = \begin{bmatrix} 20 & 4 \\ 4 & 20 \end{bmatrix}$, de trace 40 et de déterminant 384. Donc la Hessienne a deux valeurs propres strictement positives et $(-\sqrt{2},\sqrt{2})$ est un minimum local avec $f(-\sqrt{2},\sqrt{2}) = -8$
- Pour le point $(0,0)$
  On a $H(0,0) = \begin{bmatrix} -4 & 4 \\ 4 & -4 \end{bmatrix}$ donc on ne peut pas conclure avec la matrice Hessienne

Prenons $\epsilon \in R$ avec $|\epsilon| < 2$

$$\begin{aligned} f(\epsilon,-\epsilon) &= 2\epsilon^4 - 8\epsilon^2 \\ &= -2\epsilon^2(4-\epsilon^2) \end{aligned} \tag{8}$$

avec $4 - \epsilon^2 > 0$ donc $f(\epsilon,-\epsilon) < 0$ et $f(\epsilon,\epsilon) = 2\epsilon^4 \geq 0$
Ainsi, on peut conclure que $(0,0)$ est un point selle de f. On sait que $\mathcal{P}$ possède au moins une solution, et donc

$$\inf_{(x,y)\in R^2} f(x,y) = f(\sqrt{2},-\sqrt{2}) = (-\sqrt{2},\sqrt{2}) = -8 \tag{9}$$

2 Partie Théorique

Nous allons maintenant nous intéresser au problèeme de minimisation de la fonction de Rosenbrock définie sur R^2 par

$$f(x,y) = (x-1)^2 + 100(x^2-y)^2 \tag{10}$$

2.1 Etude théorique

(a) Trouver les points critiques de f.
Comme fonction polynomiale, $f \in \mathcal{C}^\infty$ sur R^2. On détermine les dérivées partielles :

$$\frac{\partial f(x,y)}{\partial x} = 2x - 2 + 400(x^2-y)x \text{ and } \frac{\partial f(x,y)}{\partial y} = 200(-x^2+y)$$

On a donc

$$\begin{aligned} \nabla f(x,y) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} &\Leftrightarrow \begin{cases} 2x - 2 + 400(x^2-y)x = 0 \\ 200(-x^2+y) = 0 \end{cases} \\ &\Leftrightarrow \begin{cases} x = 1 \\ y = 1 \end{cases} \end{aligned} \tag{11}$$

Donc f admet un unique point critique au point $(1,1)$

(b) Démontrer que f admet un minimum global qu'elle atteint en $(1,1)$.
On détermine les dérivées secondes de f sur R^2 avec $f \in \mathcal{C}^\infty$. $\forall (x,y) \in R^2$:

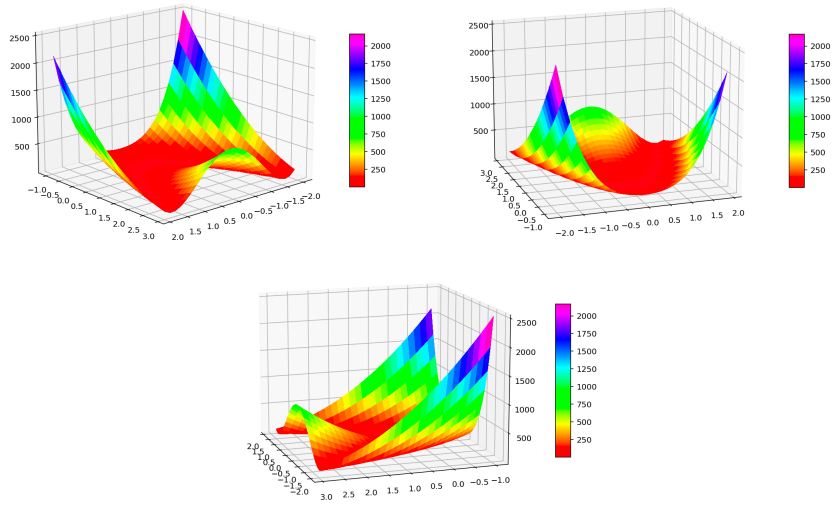
$$\frac{\partial^2 f(x,y)}{\partial x^2} = 2 + 400(3x^2 - y) \; ; \; \frac{\partial^2 f(x,y)}{\partial y^2} = 200 \text{ and } \frac{\partial^2 f(x,y)}{\partial x \partial y} = -400x$$

On a donc $H(x,y) = \begin{bmatrix} 2 + 400(3x^2 - y) & -400x \\ -400x & 200 \end{bmatrix}$ et $H(1,1) = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$ La trace vaut 1002 et le déterminant vaut 400. Ainsi, la Hessienne a deux valeurs propres strictement positives au point $(1,1)$ donc f admet un minimum local en $(1,1)$ avec $f(1,1) = 0$. De plus, f est positive sur R^2 donc f admet un minimum global en $(1,1)$.

(c) Déterminer le Jacobien de la fonction f ainsi que sa Hessienne.
D'après les questions précédentes,

$$J_f = \begin{bmatrix} 2(x-1) + 400(x^2 - y)x \\ -200(x^2 - y) \end{bmatrix} \text{ et } H(x,y) = \begin{bmatrix} 2 + 400(3x^2 - y) & -400x \\ -400x & 200 \end{bmatrix}$$

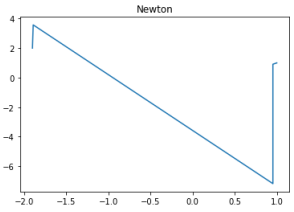
2.2 Plot



3 Partie Numérique : Methodes de descente

```python
def NewtonR(x, y, epsilon):
    x = [x]
    y = [y]
    start_time = time.time()
    Matr = -np.linalg.inv(hessrosen(x[0], y[0])).dot(diffrosen(x[0], y[0])) + np.array([x[0], y[0]])
    x.append(Matr[0])
    y.append(Matr[1])
    n = 0
    nmax = 10000
    while (np.sqrt((x[n+1]-x[n])**2+(y[n+1]-y[n])**2) > epsilon and n < nmax):
        Matr = -np.linalg.inv(hessrosen(x[n+1], y[n+1])).dot(diffrosen(x[n+1], y[n+1])) + np.array([x[n+1], y[n+1]])
        x.append(Matr[0])
        y.append(Matr[1])
        n += 1
    rep = [x[-1], y[-1]]
    X = np.transpose([x, y])
    temps = time.time() - start_time
    print(temps)
    print(n)
    return rep, X, temps
```

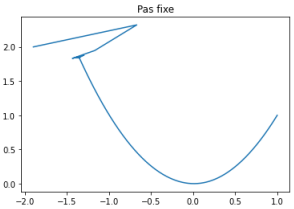
Listing 1: Méthode Newton



3.1 Méthode de descente à pas fixe

```python
def gradfixe(x, y, eps, rho):
    start_time = time.time()
    k = 0
    epsk = 2*eps
    nmax = 100000
    x_list = [x]
    y_list = [y]
    while (epsk > eps) and (k < nmax):
        wk = - np.array(diffrosen(x_list[k], y_list[k]))
        Xnew = np.array([x_list[k], y_list[k]]) + rho * wk
        x_list.append(Xnew[0])
        y_list.append(Xnew[1])
        epsk = np.sqrt((x_list[k+1]-x_list[k])**2 + (y_list[k+1]-y_list[k])**2)
        k += 1
    rep = np.array([x_list[-1], y_list[-1]])
    X = np.transpose(np.array([x_list, y_list]))
    temps = time.time() - start_time
    print(k)
    print(epsk)
    return rep, X, temps
```

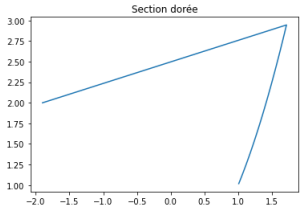
Listing 2: Méthode de descente à pas fixe



3.2 Section Dorée

```python
def gradopt_section_dor e(x, y, eps):
    start_time = time.time()
    k = 0
    epsk = 2 * eps
    nmax = 100000
    x_list = [x]
    y_list = [y]
    while epsk > eps and k < nmax:
        wk = - np.array(diffrosen(x_list[k], y_list[k]))
        rhok = section_dor e(x_list[k], y_list[k], wk, 0, 1, 50)
        Xnew = np.array([x_list[k], y_list[k]]) + rhok * wk
        x_list.append(Xnew[0])
        y_list.append(Xnew[1])
        epsk = np.sqrt((x_list[k + 1] - x_list[k]) ** 2 + (y_list[k + 1] - y_list[k]) ** 2)
        k += 1
    rep = np.array([x_list[-1], y_list[-1]])
    X = np.transpose(np.array([x_list, y_list]))
    temps = time.time() - start_time
    print(k)
    print(epsk)
    return rep, X, temps
```

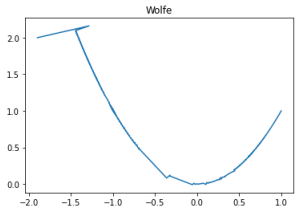
Listing 3: Grad Opt Section Dorée



3.3 GradOpt Méthode Wolfe

```python
def gradopt_m thode_wolfe(x, y, eps, rho):
    start_time = time.time()
    k = 0
    epsk = 2 * eps
    nmax = 100000
    x_list = [x]
    y_list = [y]
    rhok = rho
    while (epsk > eps) and (k < nmax):
        wk = -np.array(diffrosen(x_list[k], y_list[k]))
        rhok = m thode_wolfe(x_list[k], y_list[k], rhok, wk, 0.1, 0.9)
        Xnew = np.array([x_list[k], y_list[k]]) + rhok * wk
        x_list.append(Xnew[0])
        y_list.append(Xnew[1])
        epsk = np.sqrt((x_list[k+1] - x_list[k])**2 + (y_list[k+1] - y_list[k])**2)
        k += 1
    rep = np.array([x_list[-1], y_list[-1]])
    X = np.transpose(np.array([x_list, y_list]))
    temps = time.time() - start_time
    print(k)
    print(epsk)
    return rep, X, temps
```

Listing 4: Grad Opt Méthode Wolfe

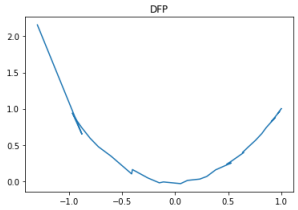


3.4 DFP

```python
def dfp(x, y, eps, rho):
    tic = time.time()
    k = 0
    rhok = rho
    Bk = np.eye(2)
    epsk = 2 * eps
    nmax = 100000
    x=[x]
    y=[y]
    while epsk > eps and k < nmax:
        wk = -np.array(diffrosen(x[k], y[k]))
        rhok = m thode_wolfe(x[k], y[k], rhok, wk, 0.01, 0.99)
        Mat1 = rhok * Bk @ wk + np.array([x[k], y[k]])
        x.append( Mat1[0])
        y.append( Mat1[1])
        sk = np.array([x[k+1], y[k+1]]) - np.array([x[k], y[k]])
        yk = diffrosen(x[k+1], y[k+1]) - diffrosen(x[k], y[k])
        Bk = Bk + np.outer(sk, sk.T) / (sk.T @ yk) - (Bk @ yk @ yk.T @ Bk.T) / (yk.T @ Bk @ yk
    )
        epsk = np.sqrt((x[k+1] - x[k]) ** 2 + (y[k+1] - y[k]) ** 2)
        k += 1
    rep = np.array([x[-1], y[-1]])
    X = np.vstack((x, y)).T
    print(k)
    temps = time.time() - tic
```

```
    return rep, X, temps
```

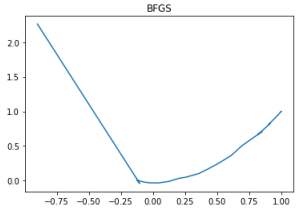
Listing 5: DFP



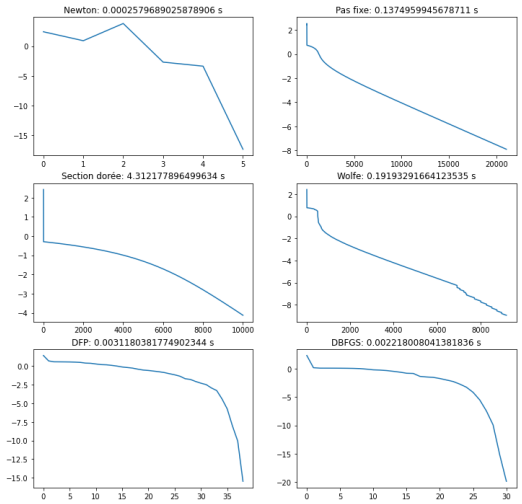
3.5 BFGS

```
def bfgs(x, y, eps, rho):
    tic = time.time()
    k = 0
    rhok = rho
    Bk = np.eye(2)
    epsk = 2 * eps
    nmax = 100000
    x=[x]
    y=[y]
    while epsk > eps and k < nmax:
        wk = -np.array(diffrosen(x[k], y[k]))
        rhok = m thode_wolfe(x[k], y[k], rhok, wk, 0.01, 0.99)
        Mat1 = rhok * Bk @ wk + np.array([x[k], y[k]])
        x.append( Mat1[0])
        y.append( Mat1[1])
        sk = np.array([x[k+1], y[k+1]]) - np.array([x[k], y[k]])
        yk = diffrosen(x[k+1], y[k+1]) - diffrosen(x[k], y[k])
        Bk = Bk + np.outer(yk, yk.T) / (yk.T @ sk) - (Bk @ sk @ sk.T @ Bk.T) / (sk.T @ Bk @ sk
    )
        epsk = np.sqrt((x[k+1] - x[k]) ** 2 + (y[k+1] - y[k]) ** 2)
        k += 1
    rep = np.array([x[-1], y[-1]])
    X = np.vstack((x, y)).T
    print(k)
    temps = time.time() - tic
    return rep, X, temps
```

Listing 6: BFGS



3.6 Recapitulatif des Méthodes



Les temps de calcul sont inclus à chaque début de graphique, et nous précisions les nombres d'itérations de chaque méthode :

$$\begin{pmatrix} Newton & 6 \\ Pasfixe & 21056 \\ sectiondorée & 9996 \\ Wolfe & 9196 \\ DFP & 38 \\ BFGS & 30 \end{pmatrix}$$

Gradient à pas fixe: Cette méthode utilise un pas fixe pour toutes les itérations, ce qui signifie que la mise à jour des variables est effectuée en utilisant un pas constant. Le temps d'itération et le temps de calcul sont généralement faibles, mais la convergence peut être lente, en particulier pour des fonctions complexes ou mal conditionnées.

Pas optimal (section dorée): La méthode de la section dorée est utilisée pour déterminer le pas optimal à chaque itération. Le temps d'itération et le temps de calcul sont généralement moyens, mais la convergence est généralement rapide. Cette méthode nécessite moins d'itérations pour converger vers la solution optimale que le gradient à pas fixe. Ici nous avons une petite erreur de temps de calcul pour la section dorée, donc c'est plutôt assez lent.

Pas optimal (Wolfe): La méthode de Wolfe est une autre approche pour déterminer le pas optimal. Elle est similaire à la section dorée en termes de temps d'itération et de temps de calcul, avec une convergence rapide.

Quasi-Newton (DFP): La méthode DFP (Davidon-Fletcher-Powell) est une méthode de Quasi-Newton qui utilise une approximation de la matrice hessienne pour améliorer la convergence. Le temps d'itération est généralement plus long, et le temps de calcul est plus élevé que les autres méthodes. Toutefois, elle présente une convergence très rapide, ce qui la rend souvent préférable pour des problèmes complexes.

Quasi-Newton (BFGS): La méthode BFGS (Broyden-Fletcher-Goldfarb-Shanno) est une autre méthode de Quasi-Newton. Elle est similaire à la méthode DFP en termes de temps d'itération et de temps de calcul, mais elle est généralement considérée comme plus efficace et plus robuste. La convergence est également très rapide.

En résumé, nous devrions avoir :

Méthode	Temps d'itération	Temps de calcul	Convergence	Complexité
Gradient à pas fixe	Rapide	Faible	Lente	Faible
Pas optimal (section dorée)	Moyen	Moyen	Rapide	Moyen
Pas optimal (Wolfe)	Moyen	Moyen	Rapide	Moyen
Quasi-Newton (DFP)	Lent	Élevé	Très rapide	Élevé
Quasi-Newton (BFGS)	Lent	Élevé	Très rapide	Élevé

Or nous obtenons des résultats légèrement différents, mais explicables, en effet : les méthodes de Quasi Newton sont sensées être plus lentes, il y a environ 30 itérations c'est très rapide.
On peut dire que cela est expliqué par le fait qu'on inverse seulement une matrice 2*2, ainsi ce serait intéressant de tester avec une plus grande dimension.

3.7 Complexité

La complexité exacte de ces algorithmes dépend de plusieurs facteurs, tels que la taille du problème, la qualité de l'implémentation et les conditions initiales. Cependant, voici une estimation générale de la complexité en termes d'opérations pour chaque algorithme mentionné:

Gradient à pas fixe: La complexité de cet algorithme est généralement de l'ordre de O(n**2) pour n variables, car chaque itération nécessite le calcul du gradient et la mise à jour des variables. Le nombre total d'itérations dépend de la fonction et du pas fixe choisi.

Pas optimal (section dorée): La complexité de cet algorithme est également de l'ordre de O(n**2), en

raison du calcul du gradient et de la recherche unidimensionnelle du pas optimal. Cependant, la convergence est généralement plus rapide, ce qui signifie que moins d'itérations sont nécessaires pour atteindre la solution optimale.

Pas optimal (Wolfe): La complexité de cet algorithme est similaire à celle de la section dorée, soit de l'ordre de O(n**2). Là encore, la convergence est généralement rapide, nécessitant moins d'itérations que le gradient à pas fixe.

Quasi-Newton (DFP): La complexité de la méthode DFP est généralement de l'ordre de O(n**3) pour n variables, car elle nécessite le calcul du gradient et la mise à jour de la matrice hessienne inverse. Cependant, la convergence est généralement beaucoup plus rapide que celle des autres méthodes, ce qui peut compenser la complexité accrue.

Quasi-Newton (BFGS): La complexité de la méthode BFGS est également de l'ordre de O(n**3) pour n variables, en raison du calcul du gradient et de la mise à jour de la matrice hessienne inverse. Tout comme la méthode DFP, la convergence est généralement très rapide, ce qui peut compenser la complexité accrue.

Il est important de noter que ces estimations de complexité sont approximatives et peuvent varier en fonction des conditions spécifiques du problème et de l'implémentation utilisée.

Ici, le problème est à deux dimensions, et manque peut-être de représentativité quid de la dimension de nos méthodes, il serait donc intéressant de comparer ces résultats avec ceux obtenus en dimension 3 ainsi que dans des dimensions supérieures.

3.8 Gradient Conjugué

```python
from scipy.sparse import diags
from numpy.linalg import inv

def GC(A, b, x0, eps):
    start_time = time.time()
    k = 0
    epsk = np.sqrt(np.dot(A @ x0 - b, A @ x0 - b))
    w0 = -(A @ x0 - b)
    epsk = 2 * eps
    nmax = 100
    xk = x0
    wk = w0
    while (epsk > eps and k < nmax):
        rhok = -np.dot(A @ xk - b, wk) / np.dot(A @ wk, wk)
        xnew = xk + rhok * wk
        wnew = -(A @ xnew - b) + ((np.dot(A @ xnew - b, A @ xnew - b)) / (np.dot(A @ xk - b, A @ xk - b))) * wk
        epsk = np.dot(A @ xnew - b, A @ xnew - b)
        k += 1
        wk = wnew
        xk = xnew
    temps = time.time() - start_time
    return temps

def main_GC():
    n_list=[10,20,30,50,100]
    time_list=[]
    for i in n_list:
        n = i
        vec4 = 4 * np.ones(n)
        vec_m2 = -2 * np.ones(n-1)
        A = diags([vec_m2, vec4, vec_m2], [-1, 0, 1], shape=(n, n)).toarray()
        #print(A)
        b= 1 * np.ones(n)
        x= 0 * np.ones(n)
        temps=GC(A, b, x, eps=10e-3)
        time_list.append(temps)
    print(time_list)
```

Listing 7: Gradient Conjugué

3.8.1 Résultats Gradient Conjugué

$$\text{nlist} = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 50 \\ 100 \end{bmatrix} \quad \text{temps correspondants} = \begin{bmatrix} 0.00026702880859375 \\ 0.0003027915954589844 \\ 0.00041103363037109375 \\ 0.0007979869842529297 \\ 0.0018019676208496094 \end{bmatrix}$$

Analysons ces résultats :

Comme prévu, le temps d'exécution augmente avec la taille du problème (n). Cela est cohérent avec la complexité théorique de la méthode du gradient conjugué, qui dépend du nombre de variables (n).

Les temps d'exécution restent relativement faibles même pour n = 100, ce qui suggère que l'algorithme est efficace pour résoudre les problèmes de cette taille. Cela est probablement dû à la structure particulière de notre matrice A (presque diagonales et symétrique).

Les temps d'exécution n'augmentent pas de manière linéaire ou quadratique avec la taille du problème. Cela est la preuve que la convergence est atteinte en moins d'itérations que n. Cela est en adéquation avec le fait que la méthode de gradient conjugué est une méthode exacte qui converge donc en au plus n itérations.

En conclusion, ces résultats montrent que la méthode du gradient conjugué est efficace pour résoudre des problèmes de taille modérée (jusqu'à n = 100). Il serait intéressant de modifier la matrice A tant par sa structure que par sa taille.