ENS Paris-Saclay

Plassier Vincent vincent.plassier@ens-paris- saclay.fr

TP OPTIMISATION
Méthodes de descente de gradient et algorithmes de Newton

20 mars 2018

1 Méthode de gradient à pas fixe

Dans une première étude nous allons nous intéresser à la méthode de descente de gradient à pas fixe. Nous traitons cette méthode sur deux exemples : d'abord pour la banane de Rosenbrock puis dans un cas quadratique.

1.1 Banane de Rosenbrock

Dans un premier temps, nous considèrons la fonction $f : \mathbb{R}^2 \mapsto \mathbb{R}$ définie par

$$f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2.$$

Remarquons que cette fonction est de classe $\mathcal{C}^1$, qu'elle est positive et atteint son unique minimum en $x^* = (-1, 1)$, pour lequel $f(x^*) = 0$.

Voici un code python qui calcule cette fonction ainsi que son gradient :

```
def rosenbrock(x):
    y = np.asarray(x)
    return ((y[0] - 1)**2 + 100 * (y[1] - y[0]**2)**2)
def rosenbrock_grad(x):
    y = np.asarray(x)
    grad = np.zeros_like(y)
    grad[0] = 400 * y[0] * (y[0]**2 - y[1]) + 2 * (y[0] - 1)
    grad[1] = 200 * (y[1] - y[0]**2)
    return grad
```

Dès lors, ce script nous permet de tracer les lignes de niveaux de la fonction de Rosenbrock sur le domaine $[-1.5, 1.5] \times [-0.5, 1.5]$ (en figure 1). Nous les obtenons à l'aide du code suivant :

```
xmin,xmax,ymin,ymax=-1.5,1.5,-0.5,1.5
xx=np.linspace(xmin,xmax,500)
yy=np.linspace(ymin,ymax,500)
[X,Y]=np.meshgrid(xx,yy)
fXY=rosenbrock([X,Y])
```

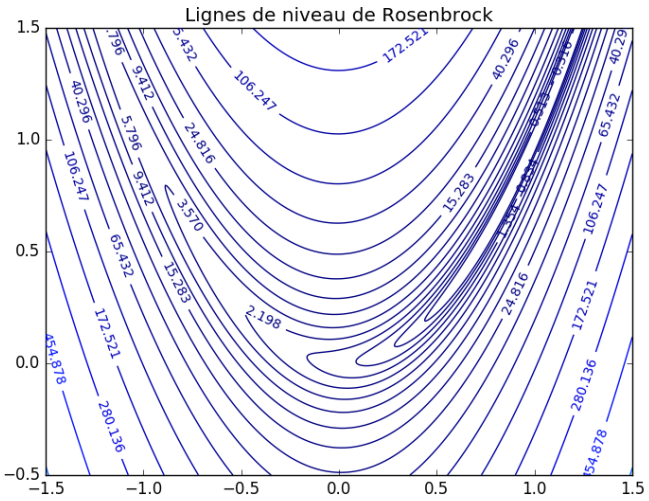


FIGURE 1 – Lignes de niveaux sur $[-1.5, 1.5] \times [-0.5, 1.5]$

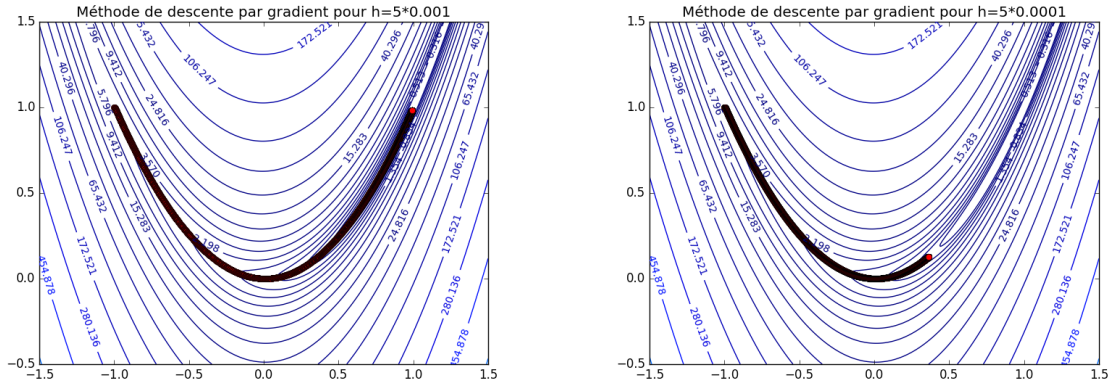
Nous nous proposons maintenant d'implémenter la méthode de descente de gradient à pas fixe, donnée par la relation de récurrence suivante :

$$x_{k+1} = x_k - h\nabla f(x_k).$$

La suite ainsi définie est indexée sur tous les entiers, nous devons donc fixer des critères d'arrêt afin d'avoir une descente de gradient numériquement envisageable. Nous proposons alors un critère portant sur le nombre d'itérations, ainsi que sur la distance entre deux termes successifs de la suite $(x_k)_k$. Sous ces conditions, nous avons le code python suivant :

```
def GPF(h,x0,f_grad,epsilon=10**-8,n_max=1000):
    error=1
    increment=0
    XX=[x0]
    while error > epsilon and increment < n_max :
        x1=x0-h*f_grad(x0)
        error=np.linalg.norm(x0-x1,2)
        x0=x1
        increment+=1
        XX.append(x1)
    return np.array(XX)
```

Voici en figure 2 les deux cents premiers termes obtenus par cette méthode pour deux choix de pas différents :



$h = 10^{-3}$ $h = 10^{-4}$

FIGURE 2 – Exemples d'utilisation de la méthode de descente de gradient à pas fixe.

Nous constatons qu'avec la fonction de Rosenbrock, la méthode de descente de gradient à pas fixe donne une suite qui semble converger lentement vers le minimum x^*. Cependant la convergence reste très lente car les itérées de la suite se situent dans une région de faible gradient.

Pour remédier à ce problème, nous proposons d'implémenter la méthode de descente de gradient normalisée, donnée par la relation de récurrence suivante :

$$x_{k+1} = \begin{cases} x_k - h\frac{\nabla f(x_k)}{||\nabla f(x_k)||} & \text{si } \nabla f(x_k) \neq 0, \\ x_k & \text{sinon.} \end{cases}$$

Comme dans la méthode précédente, nous choisissons pour critère d'arrêt le nombre d'itérations de la routine, et seulement celui-ci puisque les termes successifs de la suite $(x_k)_k$ sont distant de h en norme nous n'avons plus l'intérêt d'un critère *epsilon* lié à la distance entre deux itérées sucessives. Voici une implémentation possible de cette méthode :

```
def GPF(h,x0,f_grad,epsilon=10**-8,n_max=1000):
    error=1
    increment=0
    XX=[x0]
    while error > epsilon and increment < n_max :
        x1=x0-h*f_grad(x0)
        error=np.linalg.norm(x0-x1,2)
        x0=x1
        increment+=1
        XX.append(x1)
    return np.array(XX)
```

Remarquons que la suite obtenue par cette méthode ne semble pas converger vers le minimum. En effet, les termes successifs de la suites sont toujours tels que la différence $||x_{k+1} - x_k|| = h \nrightarrow 0$. Néanmoins, les termes de cette suite semblent tout de même se rapprocher du minimum, comme en atteste la figure 3.

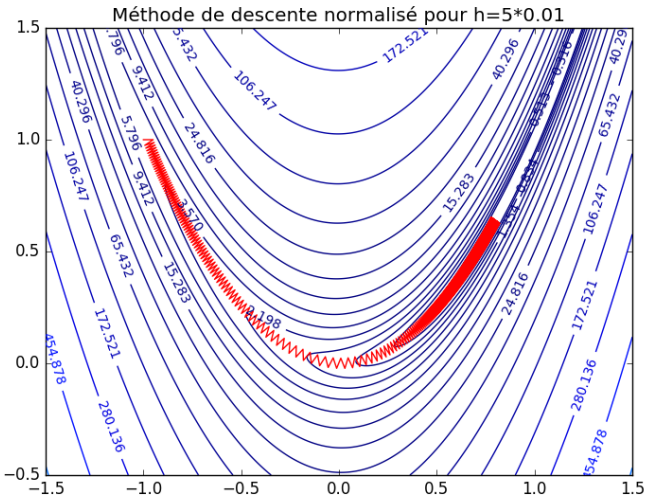


FIGURE 3 – Méthode descente normalisée à pas fixe pour $h = 5.10^{-}2$.

1.2 Cas quadratique

Dès à présent nous allons nous intéresser à un deuxième exemple. Cette fois-ci la fonction considérée est une forme quadratique définie par :

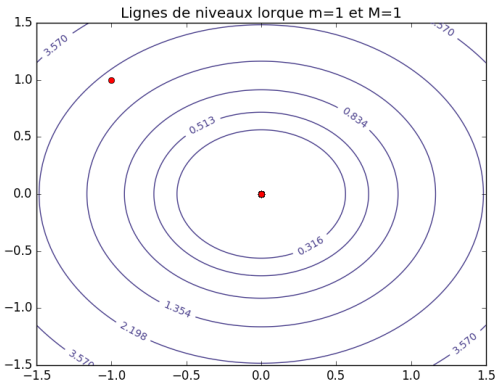
$$f = \begin{cases} \mathbb{R}^n \longrightarrow \mathbb{R} \\ x \longmapsto x^T A x \end{cases}$$

où A est un matrice diagonale appartenant à $\mathcal{M}_n(\mathbb{R})$ avec tous ses coefficients diagonaux valent 1 sauf le premier valant M et le second valant m.
Remarque : Nous avons choisit de suivre la définition de A donnée dans le texte du TP et non celle de l'algorithme mk_quad.
Voici le script python de mk_quad que nous avons utilisé :

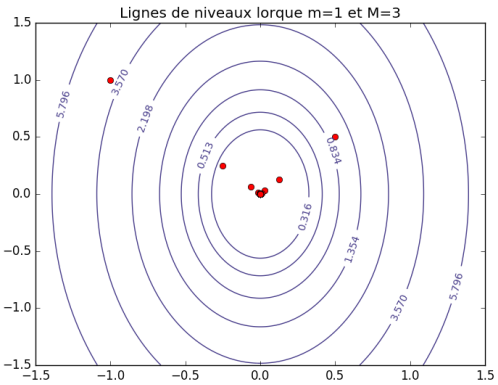
```
def mk_quad(m, M, ndim=2):
    def quad(x):
        y = np.copy(np.asarray(x))
        y = y**2
        y[0]=y[0]*M
        y[1]=y[1]*m
        return np.sum(y,axis=0)
    def quad_grad(x):
        y = np.asarray(x)
        scal = np.ones(ndim)
        scal[0] = M
        scal[1] = m
        return 2 * scal * y
    return quad, quad_grad
```

Par l'intermédiaire d'un script quasiment similaire à celui du début, nous avons pu tracer les lignes de niveaux de la fonction f. Elles sont représentées sur la figure 9 pour différentes valeurs de m et de M. Dans l'optique d'évaluer l'efficacité de la méthode de descente de gradient nous représentons également les itérées obtenues par notre algorithme, toujours sur la figure 9 pour différentes valeurs de (m, M).

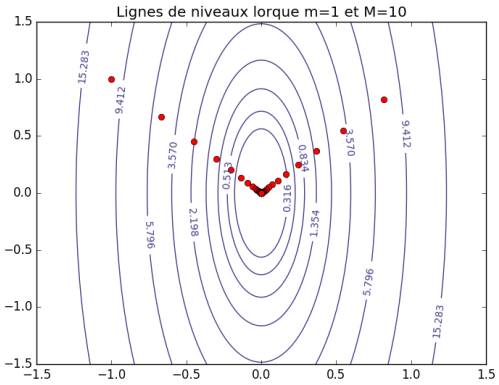
```
def methode_gradient_optimal(x0,m,M,n_max=10**3,eps=10**-8):
    increment=0
    error=1
    XX=[x0]
    while error>eps and increment<n_max:
        d=mk_quad(m, M, ndim=2)[1](x0)
        x1=x0-d/(m+M)
        x0=x1
        XX.append(x0)
        increment+=1
    return np.array(XX)
```



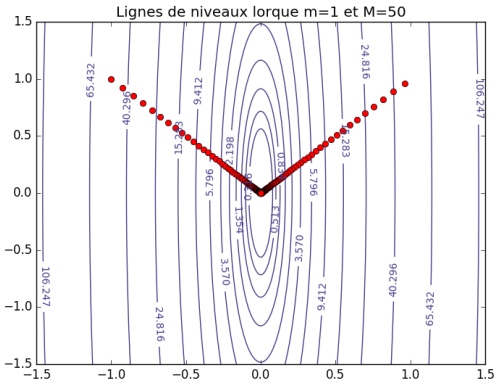
(a) $m = 1$ et $M = 1$



(b) $m = 1$ et $M = 3$



(c) $m = 1$ et $M = 10$



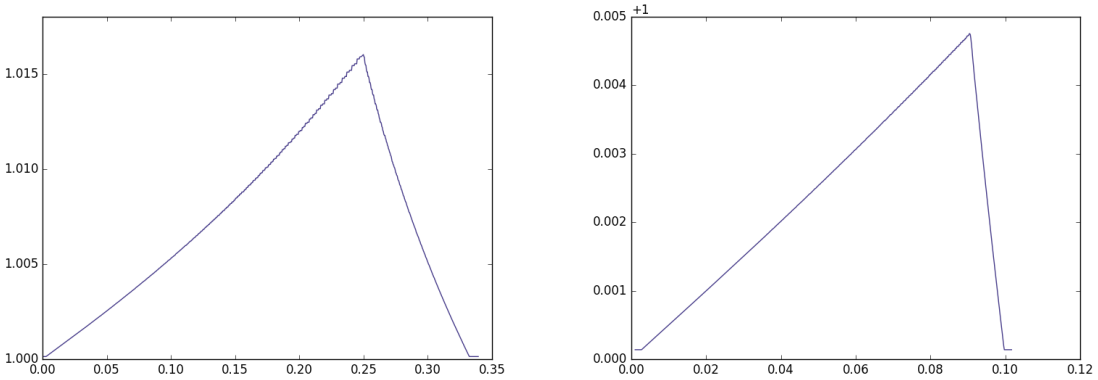
(d) $m = 1$ et $M = 50$

FIGURE 4 – Lignes de niveaux

Nous pouvons remarquer que l'algorithme converge d'autant plus rapidement que m et M sont proches. En effet, dans le cas où $(m, M) = (1, 1)$ nous pouvons constater sur la figure 4(a) que l'algorithme converge en une itération alors que dans le cas où $(m, M) = (1, 50)$ les itérations zigzaguent jusqu'à atteindre le minimum. Cela s'explique par le fait que la descente s'effectue dans une direction perpendiculaire à la ligne de niveau. Ainsi, lorsque les lignes de niveaux ressemblent à des cercles (c'est-à-dire lorque $m \simeq M$) la méthode converge rapidement. De plus, nous remarquons que le cas $m \simeq M$ correspond au cas dans lequel le conditionnement de la matrice A est proche de 1.

Il semble naturel de s'interesser à l'ordre de convergence de la méthode de descente de gradient.

Voici deux graphiques (cf figures 9) qui représente les ordres de convergences obtenus pour différentes valeurs de (m, M) en fonction du pas h :



(a) $m = 1$ et $M = 3$ (b) $m = 1$ et $M = 10$

FIGURE 5 – Ordres de convergences en fonction du pas

Nous constatons des ordres de convergence supérieurs à 1 ou inexistants, cela concorde avec l'intuition : soit l'ordre de convergence existe et les itérations convergent, sinon l'ordre n'est pas défini et les itérations divergent (ce n'est pas toujours binaire pour d'autres méthodes). Nous constatons également que le maximum de l'ordre de convergence est atteint pour un pas valant environ $\frac{1}{m+M}$, cela concorde donc avec nos résultats théoriques (cf TD 3). Par exemple, pour la figure 5(a) le maximum semble atteint aux alentours de $\frac{1}{m+M} \simeq 0.25$ alors que pour la figure 5(b) le maximum se situe dans les environs de $\frac{1}{m+M} \simeq 0.09$. De plus, l'ordre de convergence (lorsqu'elle a lieu) est très proche de 1, la méthode de descente de gradient est bien loin de l'efficacité quadratique des méthodes quasi-Newton.

Numériquement nous pouvons remarquer que la vitesse de l'algorithme varie très peu en fonction de la dimension de A. Pour cela nous traçons un tableau dans lequel on remplie la première ligne avec différentes valeurs de la dimension de la matrice A, puis dans la seconde ligne nous marquons le ratio entre la 10^4-ième itération pour dim=d sur la 10^4-ième itération pour dim=2. Pour $(m, M) = (1, 3)$, nous trouvons :

dimension de la matrice A	2	3	5	10	20	30	40	50
ratio	1	1.3	1.7	2.6	3.7	4.6	5.4	6.0

En mettant le tableau sous forme d'un graphique, nous obtenons une courbe logarithmique représentée en vert. cette courbe est obtenue pour un pas fixe $h = 0.01$. Nous représentons également en rouge le nombre d'itérations nécessaire à l'algorithme pour approcher à $epsilon = 10^{-8}$ le minimum pour un pas $h_{opt} = \frac{1}{m+M}$:

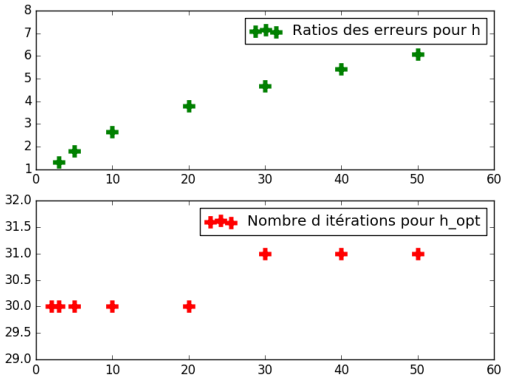


FIGURE 6 – Ratios des erreur ainsi que le nombre d'itérations nécessaire pour une tolérance 10^{-8} en fonction de la dimension

2 Choix du pas de descente : règle d'Armijo et règle de Wolfe

Dans cette seconde partie nous allons nous intéresser à une méthode très similaire à la méthode de descente par gradient, à ceci près qu'elle cherche à estimer un pas de descente optimal. Nous avons implémenter les méthodes d'Armijo et de Wolfe de la manière suivante :

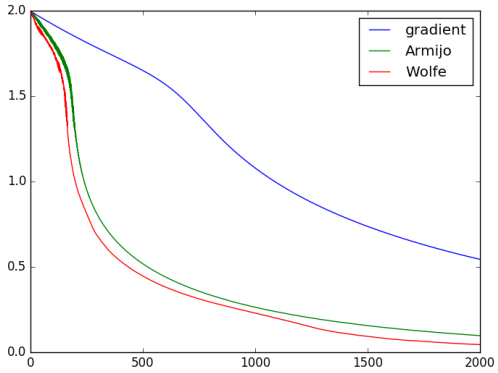
```
def Armijo(x0,f,f_grad,c,L,epsilon,n_max):
    increment=0
    error=1
    XX=[x0]
    while error>epsilon and increment<n_max:
        d=-f_grad(x0) # direction de descente
        h=1/L # initialise
        y=f(x0)
        A=c*norm(d)**2
        while f(x0+h*d)>y-h*A:
            h*=0.9
        x1=x0+h*d
        error=norm(x1-x0)
        x0=x1
        XX.append(x0)
        increment+=1
    return np.array(XX)
def Wolfe(x0,f,f_grad,c1,c2,L,alpha,epsilon,n_max):
    increment=0
    error=1
    XX=[x0]
    while error>epsilon and increment<n_max:
        d=-f_grad(x0) # direction de descente
        h=1/L # initialise
        y=f(x0)
        A=c1*norm(d)**2
        h_g,h_d=0,0
        while f(x0+h*d)>y-h*A or np.sum(d*f_grad(x0+h*d))<-c2*norm(d)**2:
            if f(x0+h*d)>y-h*A:
```

```
            h_d=h
        elif np.sum(d*f_grad(x0+h*d))<-c2*norm(d)**2:
            h_g=h
        if h_d==0:
            h*=alpha
        else:
            h=(h_g+h_d)/2
    x1=x0+h*d
    error=norm(x1-x0)
    x0=x1
    XX.append(x0)
    increment+=1
return np.array(XX)
```

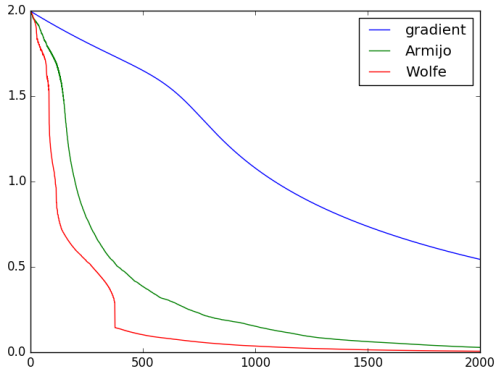
Reprenons les notations de l'algorithme, un développement limité de la fonction considérée (notée f) au point $x0 + h * d$ nous donne :

$$f(x0 + h * d) = f(x0) + h\langle \nabla f(x0)|x1\rangle + o(h) \tag{1}$$

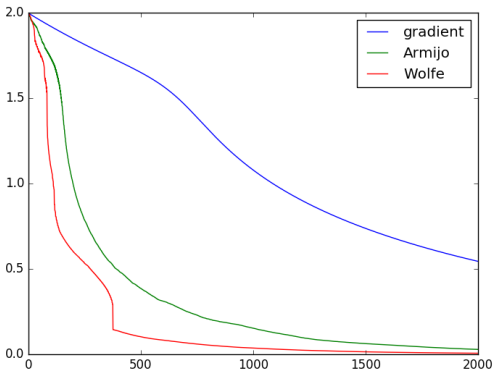
Nous en déduisons que pour h suffisamment petit nous avons $f(x0+h*d) \leq f(x0)+h*c\langle \nabla f(x0)|d\rangle = y - h * A$, ce qui prouve que la recherche linéaire d'Armijo s'arrête en un nombre fini d'itérations. De même, dans notre cours nous avons montré que la recherche linéaire de Wolfe se termine en un nombre fini d'itérations.
Dès lors, ces deux scripts vont nous permettent de comparer les trois méthodes de descentes par gradient étudiées. Dans les figures suivantes nous représentons les distances entre les itérées x_k et point réalisant le minimum $x^* = (0, 0)$ obtenues pour les trois méthodes. La première figure 7 montrent nos résultats obtenus pour $h = 10^{-3}$,$L = 100$ ainsi que pour différentes valeurs de c,c_1 et c_2.



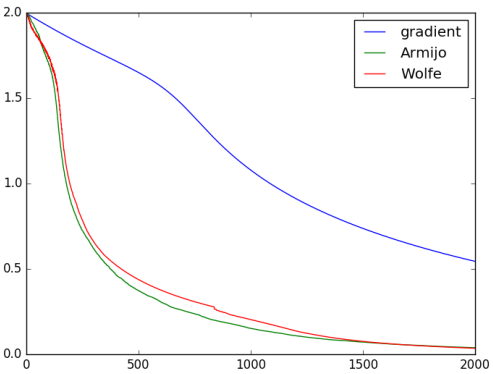
(a) $c = 0,1$, $c1 = 10^{-3}$ et $c2 = 0,9$



(b) $c = 0,1$, $c1 = 10^{-3}$ et $c2 = 0,9$



(c) $c = 0,95$, $c1 = 0.1$ et $c2 = 0.9$

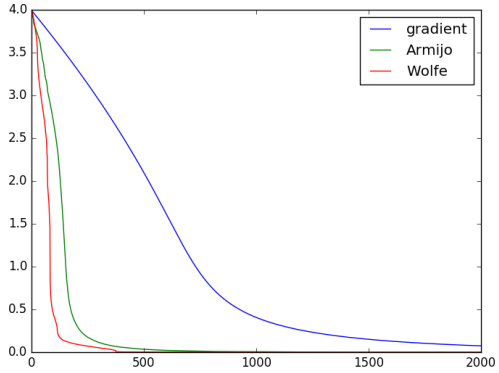


(d) $c = 0,95$, $c1 = 0.3$ et $c2 = 0.7$

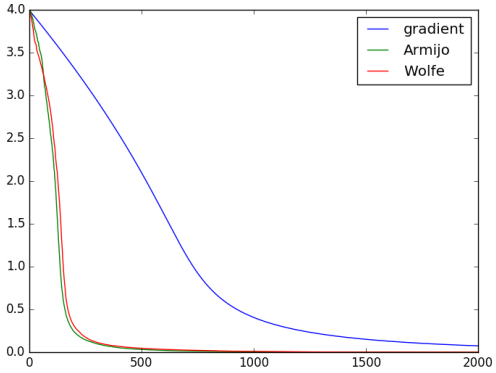
FIGURE 7 – Distance entre x_k et le minimum x^*

Nous pouvons constater que les itérés obtenus par la méthode de Wolfe semblent converger beaucoup plus rapidement que ceux obtenus par la méthode de descente par gradient pour tous les jeux de paramètres. Toutefois, la méthode de Wolfe recherche un pas h de manière plus sophistiqué, l'algorithme effectue davantage d'opérations à chaque itérations. Il semble donc avantageux d'utiliser une méthode d'Armijo voir de descente de gradient lorsque la différentielle est extrèmement simple à calculer, mais lorsque la différentielle se complexifie, mieux vaut maximiser le pas de descente.

Les figures suivantes 8 représentent les distances de $f(x_k)$ au minimum $f(x^*) = 0$ pour les mêmes jeux de paramètres.



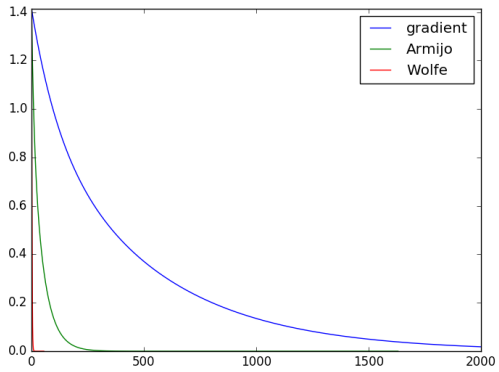
(a) $c = 0,9$, $c1 = 0,1$ et $c2 = 0,9$



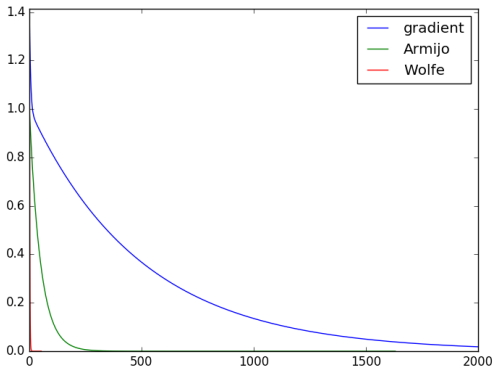
(b) $c = 0,5$, $c1 = 0,5$ et $c2 = 0,5$

FIGURE 8 – Distance entre x_k et le minimum x^*

Nous allons maintenant reprendre l'étude précédente mais cette fois-ci pour la fonction quadratique définie dans la section **1.2** avec comme paramètre d'entrée (m, M). La figure 9 repésente la quatité $||x_k - x^*||$ en fonction de k.



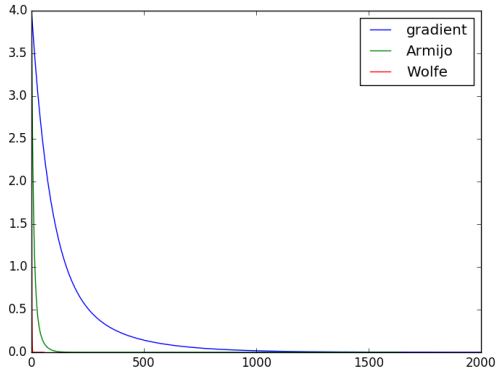
(a) $m = 1$ et $M = 3$



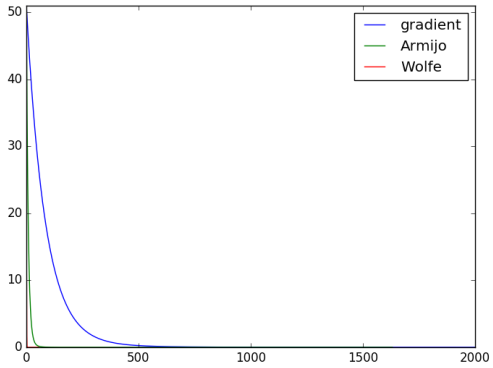
(b) $m = 1$ et $M = 50$

FIGURE 9 – Distance entre x_k et le minimum x^*

Nous constatons une très rapide convergence de l'algorithme de Wolfe vers le minimum de la fonction quadratique. En effet, pour une tolérance $epsilon = 10^{-16}$ avec $(m, M) = (1, 5)$ notre algorithme s'arrête au bout de seulement 341 itérations, alors que la méthode d'Armijo en nécessite 1632.
Enfin, la dernière figure 10 correspond à $|f(x_k) - f(x^*)|$ en fonction de k.



(a) $m = 1$ et $M = 3$



(b) $m = 1$ et $M = 50$

FIGURE 10 – Ecart entre $f(x_k)$ et $f(x^*)$ pour la fonction quadratique

3 Gradient conjugué

Dans cette partie, nous allons nous interesser à nouvelle méthode de descente par gradient. Cette fois-ci nous étudierons la méthode de *Polak-Ribière*. Cette méthode peut être implémentée sur Python de la manière suivante :

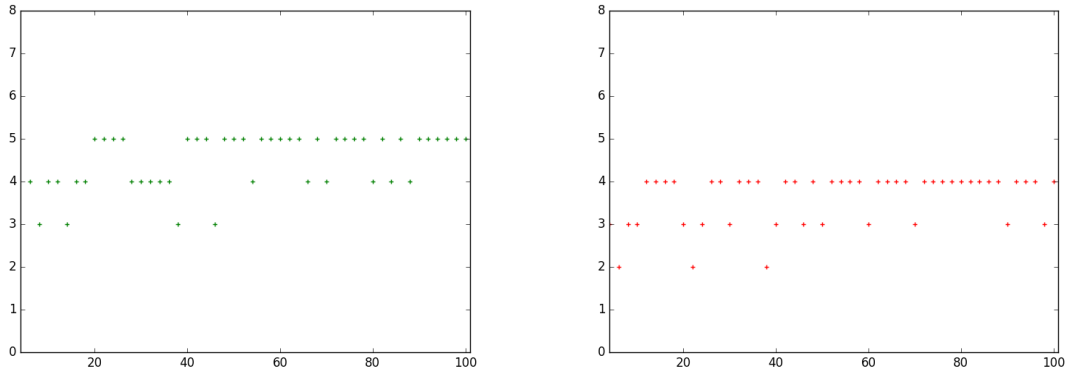
```
def Polak_Ribiere(x0,f,f_grad,epsilon=10**-8,n_max=10**4):
    c1=0.01
    c2=0.9
    error=1
    increment=0
    XX=[x0]
    x1=x0
    d=0
    g0=f_grad(x0)
    while error>epsilon and increment<n_max:
        g1=f_grad(x1)
        x0=x1
        beta=np.dot((g1-g0),g1)/norm(g0)**2
        d=-g1+beta*d
        d=-np.sign(np.sum(g1*d))*d # bien direction de descente
        h=recherche_Wolfe(x0,f,f_grad,c1,c2,L,alpha,epsilon,n_max)
        x1=x0+h*d
        g0=g1
        error=norm(x1-x0)
        increment+=1
        XX.append(x1)
    return np.array(XX)
```

Cet algorithme nous fournit une méthode nécessitant peu d'itérations. En effet, dans les tableaux suivants nous avons rentré le nombre d'itérations nécessaires pour obtenir une précision de l'ordre de 10^{-12} sur le minimum :

Rosenbrock	Gradient	Armijo	Wolfe	Polak-Ribière
Nombre d'itérations	49331	12260	10556	119

Cas quadratique $(m, M) = (1, 20)$	Gradient	Armijo	Wolfe	Polak-Ribière
Nombre d'itérations	10700	1177	112	296

Nous pouvons constater un facteur de l'ordre de 10^2 itérations d'écart entre la méthode de Polak-Ribère et celle de descente par gradient à pas fixe. Il faut toutefois nuancé cette différence puisque la méthode implémentée utilise une recherche linéaire de Wolfe, et celle-ci nécessite plusieurs boucles supplémentaires non comptées dans le nombre final d'itérations.
Nous allons maintenant nous intéresser à tester l'une des proposition de notre cours. En effet, nous savons que dans le cas quadratique, la méthode du gradient conjugué converge en au plus n itérations. Pour cela, nous avons choisi une liste de 50 dimensions comprisent entre 2 et 100 et nous avons regardé le nombre necessaire d'itérations avant convergence. Les résultats obtenus sont les suivants (**??**) :



(a) Resultats de notre algorithme

(b) Résultats de python

FIGURE 11 – Nombre d'itérations en fonction de la dimension

Nous pouvons constater que le nombre nécessaire d'itérations est compris entre 3 et 5, ce qui est bien inférieur à la dimension des espaces. Cependant, pour $dimension = 2$ notre algorithme effectue 4 itérations. Cela peut s'expliquer par des erreurs d'approximations lors des calculs. De même l'algorithme de python utilise 3 itérations en dimension 2. Les résultats obtenus sont fortement similaires avec ceux obtenus par la fonctoin dédiée *scipy.sparse.linalg.cg* de python (qui correspond à la méthode du gradient conjugué dans notre cadre quadratique).