Imports classiques

Nous allons tout d'abord lancer les imports classiques

In [ ]:
```
%load_ext autoreload
%autoreload 2

%matplotlib inline

# Imports classiques
# - %load_ext autoreload et %autoreload 2 permettent de recharger automat
#   les modules modifiés sans relancer le kernel (pratique en développeme
# - %matplotlib inline active l'affichage des figures à l'intérieur du no
import numpy as np
import matplotlib.pyplot as plt
import Optim as opt
import functions as func
```

Vous devez copier-coller ici votre algorithme de descente de Wolfe ainsi que `ls_wolfe_step_is_one`

In [ ]:

```
def ls_wolfe(x,function,step,descent,f,df) :
    """
    Recherche de pas satisfaisant les conditions de Wolfe (Armijo + curva

    Paramètres
    ---------
    x : np.array
        Point courant.
    function : objet
        Doit fournir value(x) et grad(x) pour évaluer la fonction et son
    step : float
        Pas initial proposé pour la recherche de pas.
    descent : np.array
        Direction de descente (par exemple -H*grad ou -grad).
    f : float
        Valeur de la fonction en x (f(x)).
    df : np.array
        Gradient en x (\nabla f(x)).

    Retour
    -----
    x2, f2, df2, step2 : (np.array, float, np.array, float)
        Nouveau point, valeur, gradient et pas choisi.
    """
    step_min,step_max=0.,np.inf
    i=0
    mycontinue=True
    # dérivée directionnelle initiale <=> (\nabla f(x) , d); devrait être
    scal=np.dot(df,descent)
    step2=step
    # paramètres c1 (Armijo) et c2 (curvature)
    eps1,eps2=0.1,0.9
    while mycontinue :
        i=i+1
        mycontinue=False
        x2=x+step2*descent
        f2=function.value(x2)
        # Condition d'Armijo : si non satisfaite, réduire le pas
        if f2>f+eps1*step2*scal :
            step_max=step2
            step2=0.5*(step_min+step_max)
            mycontinue=True
        else :
            # Vérifier la condition de curvature
            df2=function.grad(x2)
            if np.dot(df2,descent) < eps2*scal :
                # si la condition de curvature n'est pas satisfaite, augm
                step_min=step2
                step2=min(0.5*(step_min+step_max),2*step_min)
                mycontinue=True
    return x2,f2,df2,step2


def ls_wolfe_step_is_one(x,function,step,descent,f,df) :
    """Variante qui force le pas initial à 1 (utile pour certaines méthod
    return ls_wolfe(x,function,1.,descent,f,df)
```

In [ ]:

L-BFGS

Nous allons nous intéresser à l'algorithme Limited Memory BFGS. Cet algorithme est du type BFGS, c'est à dire qu'il estime l'inverse de la Hessienne de f. Le L dans le nom de l'algorithme signifie qu'il est à mémoire limitée, c'est à dire qu'il ne garde en mémoire que les L dernières itérations de calcul pour estimer la Hessienne. L'algorithme est le suivant : Nous sommes à l'itération k, nous notons x_k l'itéré et nous avons stocké les vecteurs suivants pour tout $k_{min} \leq i \leq k$.

$$s_i = x_i - x_{i-1} \text{ et } y_i = \nabla f(x_i) - \nabla f(x_{i-1})$$

Et on a aussi stocké $\rho_i = \frac{1}{(s_i, y_i)}$. Tous les ρ_i doivent être positifs. L'algorithme est le suivant

$r = -\nabla f(x_k)$

Pour $i = k, k-1, \dots k_{min}$

$\alpha_i = \rho_i(s_i \cdot r)$

$r = r - \alpha_i y_i$

$r = \frac{(s_{k_{min}} \cdot y_{k_{min}})}{(y_{k_{min}} \cdot y_{k_{min}})} r$ Pour $i = k_{min}, k_{min}+1, \dots, k$ //

;$\beta_i = \rho_i(y_i \cdot r)$

$r = r + (\alpha_i - \beta_i)s_i$

rend r

Fonctions sur les listes

Vous aurez sans doute besoin des fonctions suivantes pour les listes

In [ ]:
```
# Exemples d'opérations sur les listes en Python
# Ces courtes démonstrations servent à rappeler la syntaxe et quelques mé

a=[(2*i,3*i) for i in range(5) ]  # création d'une liste de tuples
print(a)
print(a[3],a[-1])                # accès par index (positif et négatif)
print('*** Pop ***')
a.pop(0)                          # suppression du premier élément
print(a)
print('*** Parcours ***')
for e,f in a :                    # parcours de la liste de tuples
      print(e,'et',f)
print('*** Parcours Inverse***')
for e in reversed(a) :            # parcours inversé
      print(e)
b=[e**2 for e,f in a]            # comprehension de liste
print(b)
print('*** Parcours de deux listes ensembles***')
for (m,(t,p)) in zip(b,a) :       # zipper deux listes pour itérer en par
    print(m,'et',t,'et encore',p)
print('*** Append ***')
print(b)
b.append(546)                     # ajout d'un élément
print(b)
print('*** Inversion ***')
c=list(reversed(b))               # inversion en créant une nouvelle list
print(c)

print(len(b))                     # longueur de la liste
```

```
[(0, 0), (2, 3), (4, 6), (6, 9), (8, 12)]
(6, 9) (8, 12)
*** Pop ***
[(2, 3), (4, 6), (6, 9), (8, 12)]
*** Parcours ***
2 et 3
4 et 6
6 et 9
8 et 12
*** Parcours Inverse***
(8, 12)
(6, 9)
(4, 6)
(2, 3)
[4, 16, 36, 64]
*** Parcours de deux listes ensembles***
4 et 2 et encore 3
16 et 4 et encore 6
36 et 6 et encore 9
64 et 8 et encore 12
*** Append ***
[4, 16, 36, 64]
[4, 16, 36, 64, 546]
*** Inversion ***
[546, 64, 36, 16, 4]
5
```

Class BFGS

Créez une classe `BFGS` ci dessous, sa fonction `__init__` sera de la forme `__init__(self,nb_stock_max=8)` où `nb_stock_max` est le nombre maximum d'itérations prises en compte. Cette fonction créera aussi une liste vide appelée `stock` qui conserve les s_i, g_i, ρ_i. Elle devra aussi créer une liste vide nommée `last_iter`.

In [ ]:
```
class BFGS() :
    """Classe L-BFGS (mémoire limitée) qui stocke les derniers (s,y,rho)

    nb_stock_max : nombre maximum de couples (s,y) à conserver.
    """
    def __init__(self,nb_stock_max=8,verbose=True) :
        self.nb_stock_max=nb_stock_max
        # ``stock`` contiendra des triplets [s, y, rho]
        self.stock = []
        # ``last_iter`` garde le dernier (x, g) pour calculer s et y
        self.last_iter =[]


    def push(self, x, g):
        """Enregistre (s_k, y_k, rho_k) calculés à partir de x et g actue

        Si rho <= 0, on détecte un problème numérique et on vide le stock
        """

        if self.last_iter != [] :
            s = x - self.last_iter[0]
            y = g - self.last_iter[1]
            denom = np.dot(y, s)
            # éviter division par zéro
            if denom == 0:
                rho = np.inf
            else:
                rho = 1.0 / denom
            if rho >0 :
                # maintien de la taille maximale du stock
                if len(self.stock) == self.nb_stock_max :
                    self.stock.pop(0)

                self.stock.append([s,y,rho])

            else :
                # en cas d'instabilité, on vide le stock (sécurité simple
                self.stock = []

        # sauvegarde pour la prochaine itération
        self.last_iter = [x,g]



    def get(self, grad) :
        """Calcule la direction r = H_k * (-grad) via la "two-loop recurs

        Si aucun élément en stock, on rend simplement -grad.
```

```
        """
        q = -grad
        if len(self.stock) == 0 :
           # pas d'information de quasi-Newton disponible
           pass
        else :
            # Liste pour stocker les alpha temporaires (dans l'ordre d'ap
            L_a = []

            # Première passe : on parcourt le stock de la fin au début
            for s,y,rho in reversed(self.stock) :
                alpha = rho * np.dot(s,q)
                L_a.append(alpha)
                q = q - alpha * y

            L_a.reverse()

            # Scaling initial (gamma) : assure une bonne échelle initiale
            gamma = np.dot(self.stock[0][0], self.stock[0][1]) / np.dot(s

            # B0 est implicitement gamma*I, on applique au vecteur q
            B0 = gamma * np.ones(len(grad))

            q = B0 * q

            # Deuxième passe : on réapplique les s avec les beta correspo
            for (s, y, rho), alpha in zip(self.stock, L_a):
                beta = rho * np.dot(y, q)
                q = q + s * (alpha - beta)

        return q


    def dc(self,x,function,df) :
        """Mise à jour du stock puis calcul de la direction via `get`.

        Conçu pour être passé comme paramètre `dc` aux algorithmes de des
        """
        self.push(x,df)
        return self.get(df)
```

Push

Nous allons maintenant créer une fonction `push(self, x, grad)` qui enregistre s_k, g_k, ρ_k. Pour cela, on a besoin de $x_{k-1}, \nabla f(x_{k-1})$. Si ils existent, ils se trouvent dans la liste `self.last_iter`. Ensuite on peut calculer s_k, g_k et ρ_k.

Si ρ_k est positif, alors on enregistre le triplet (s_k, g_k, ρ_k) à la fin de la liste `self.stock`, en vérifiant `self.stock` ne doit contenir au maximum que les dernières `self.nb_stock_max` itérations (si nécessaire on retire le tout premier élément de `self.stock`).

Si ρ_k est négatif, quelquechose c'est mal passé, on vide le `self.stock`.

A la fin, on n'oublie pas de mettre x_k et $\nabla f(x_k)$ dans `self.last_iter` pour être sûr de les y trouver la prochaine fois.

Get

Nous allons maintenant créer une fonction `get(self, grad)` qui modifie la direction de descente et applique l'algorithme ci-dessus. Cette fonction doit nous rendre le `r` final. Si le `self.stock` est vide, cette fonction doit nous rendre `-grad`

dc

Nous créeons maintenant une fonction `dc(self,x,function,df)` qui appplique tout d'abord `self.push` puis `self.get`, elle rend le résultat de la fonction de `self.get`.

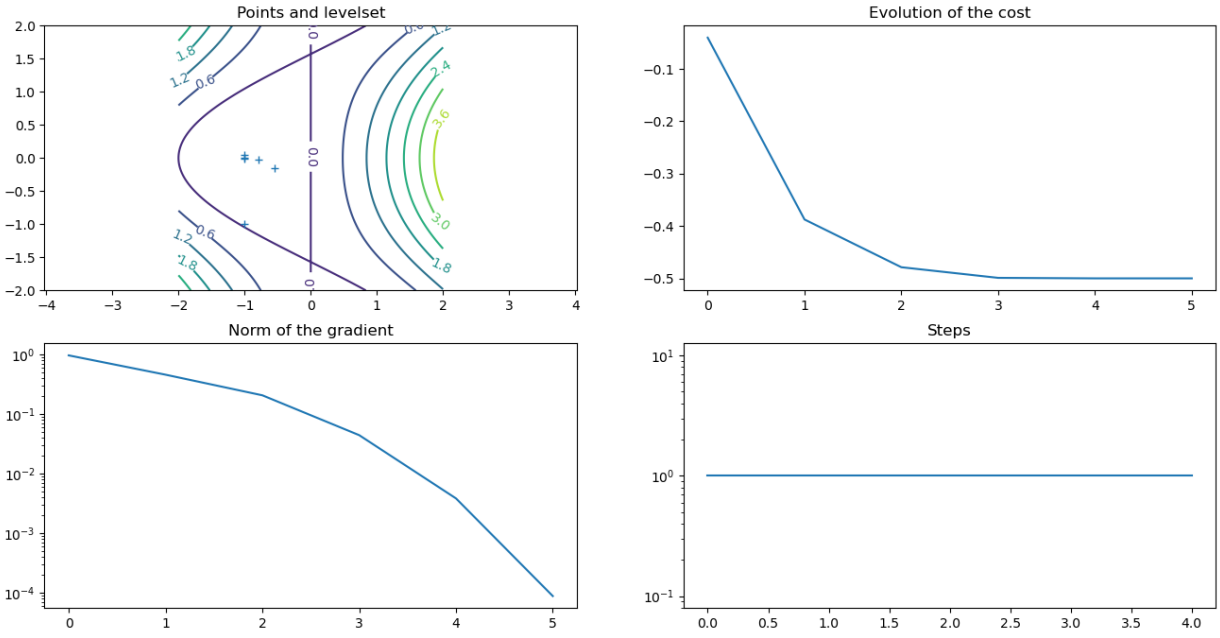
C'est l'heure de tester ...

Lancez une méthode de Newton_Wolfe sur votre fonction préférée et à chaque itération calculez ce que donnerait un L-BFGS. Comparez les angles des directions entre la méthode de Newton et le L-BFGS, comparez aussi le ration des normes. Ensuite lancez un LBFGS avec recherche de pas de Wolfe sur vos tests préférez et obtenez le comportement de Newton_Wolfe sans le calcul de la Hessienne...

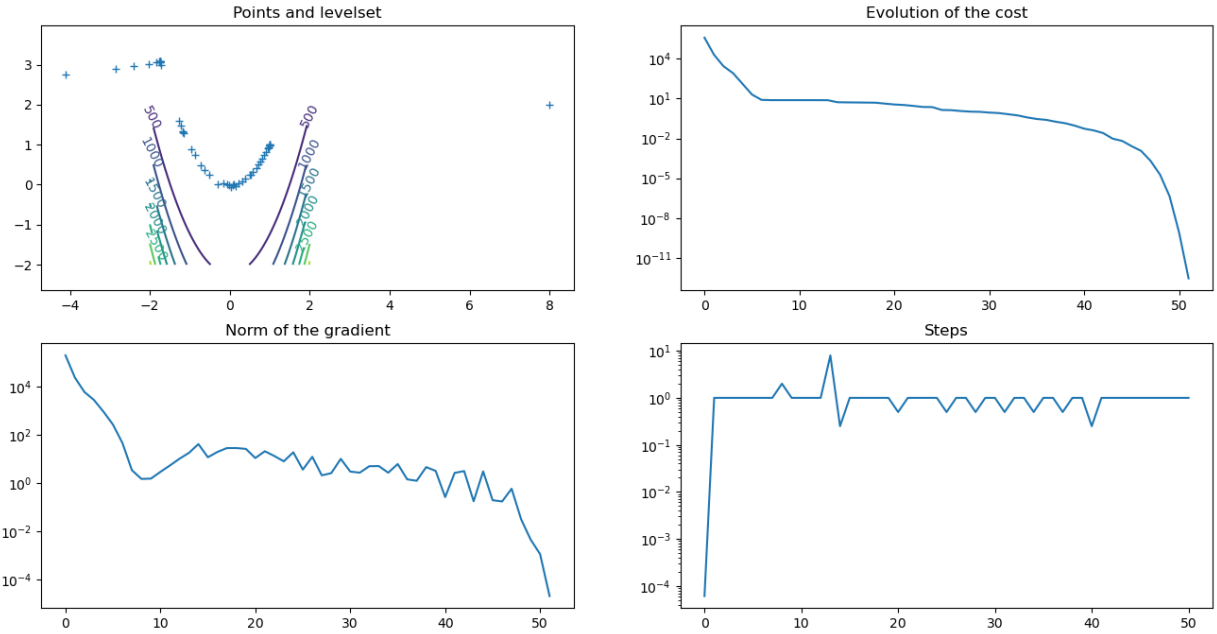
In [ ]:
```
f=func.oscill()
x0=np.array([-1,-1])
B=BFGS()
res=opt.main_algorithm(f,0.1,x0,ls=ls_wolfe_step_is_one,dc=B.dc,verbose=T
opt.graphical_info(res,f)


f=func.Rosen()
x0=np.array([8,2])
B=BFGS()
res=opt.main_algorithm(f,1,x0,ls=ls_wolfe_step_is_one,dc=B.dc,verbose=Fal
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .format(len(res['list_costs']),res['list_costs'][-1],res['list_grad
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
iter=   0 f=-4.030e-02 df=9.589e-01 comp=[   1,   1,   0]
iter=   1 f=-3.876e-01 df=4.552e-01 comp=[   2,   3,   0]
iter=   2 f=-4.787e-01 df=2.055e-01 comp=[   3,   5,   0]
iter=   3 f=-4.990e-01 df=4.415e-02 comp=[   4,   7,   0]
iter=   4 f=-5.000e-01 df=3.843e-03 comp=[   5,   9,   0]
iter=   5 f=-5.000e-01 df=8.867e-05 comp=[   6,  11,   0]
Success !!! Algorithm converged !!!
```

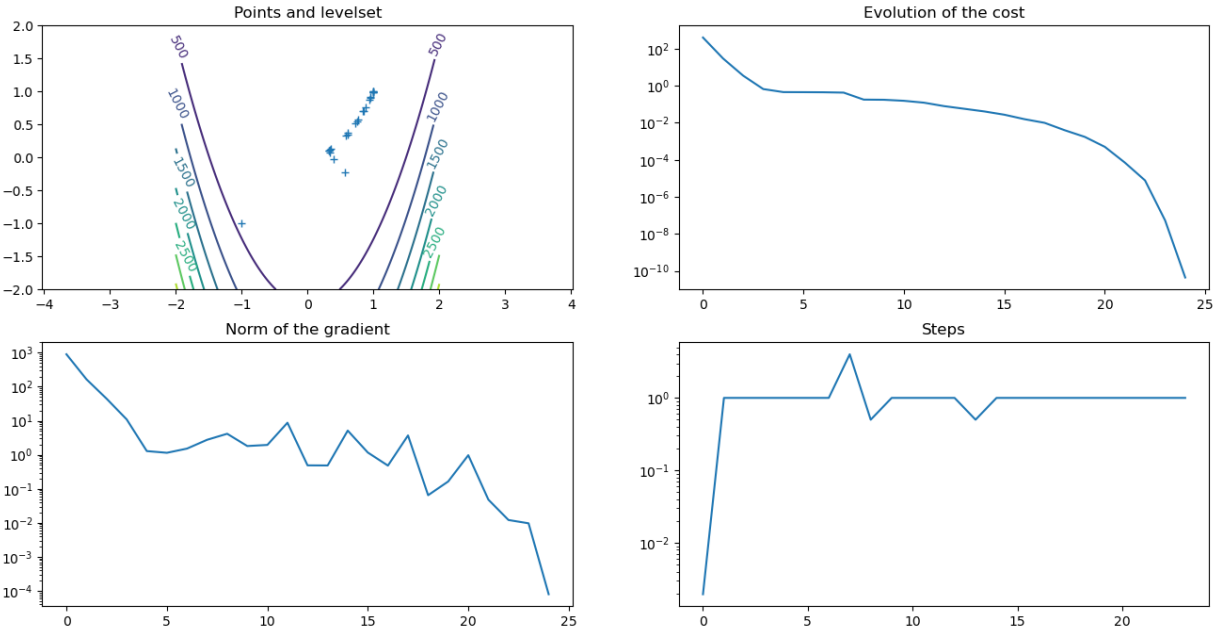



```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :  52 cost :2.871e-13 grad :2.117e-05 comp=[  80, 107,   0]
```



In [ ]:
```
f=func.Rosen()
x0=np.array([-1,-1])
B=BFGS()
res=opt.main_algorithm(f,1,x0,ls=ls_wolfe_step_is_one,dc=B.dc,verbose=Fal
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .format(len(res['list_costs']),res['list_costs'][-1],res['list_grad
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :  25 cost :4.379e-11 grad :7.989e-05 comp=[  38,  51,   0]
```



Comparaison

Votre objectif est de regarder la documentation de `scipy.optimize`, de comprendre le code suivant et de comparer votre algorithme de BFGS+Wolfe avec les algorithmes disponibles dans scipy.optimize

In [ ]:
```
# Comparaison avec scipy.optimize.minimize
# On utilise la fonction de Rosenbrock comme exemple; minimize utilise de
# bien optimisées (BFGS, L-BFGS-B, etc.) ; on compare le comportement et
# d'évaluations.
f=func.Rosen()
x0=np.array([-1,-1])
from scipy.optimize import minimize
# appelée avec jac=f.grad pour fournir le gradient analytique
res=minimize(f.value,x0,jac=f.grad)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
```

In [ ]:
```
# Affichage du résultat retourné par scipy et des compteurs d'évaluations
print(res)
print(f.nb_eval,f.nb_grad)
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 1.8499214547674284e-16
        x: [ 1.000e+00  1.000e+00]
      nit: 31
      jac: [ 2.763e-07 -1.261e-07]
 hess_inv: [[ 5.083e-01  1.016e+00]
            [ 1.016e+00  2.036e+00]]
     nfev: 40
     njev: 40
40 40
```

In [ ]:

In [ ]: