TP1_minimisation_sans_contrainte

January 14, 2026

1 Imports sous Notebook Python

Dans ce TP, vous allez essentiellement programmer des classes dans un fichier que vous pourrez garder pour plus tard. Ces classes seront enregistrées dans des fichiers `Optim.py` et `functions.py`. Cependant, le comportement par défaut d'un Notebook quand on demande d'importer un fichier est de ne pas le relire !!! Ainsi vos modifications dans les fichiers ne seront pas prises en compte. Pour que ce soit le cas, il faut lancer les commandes suivantes :

[1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import numpy as np

import matplotlib.pyplot as plt
```

1.1 Fonctionnement des classes et des fichiers de librairie sous python

Dans cette section, nous allons nous chauffer un peu et apprendre (si on ne le sait pas déjà) comment fonctionnent les classes et les fichiers sous python. Dans le fichier `functions.py` contient déjà une classe nommée `square`. Ouvrez le fichier `function.py`. Cette classe `square` a 4 sous-fonctions, la fonction `__init__` se lance à l'appel de la classe (instanciation) et les autres fonctions se lancent avec les commandes suivantes

[2]:
```
import functions as  func
print("***** INSTANCIATION")
J=func.square()
print("***** METHODES DE LA CLASSE")
a=np.array([1,2])
print(J.value(a))
print(J.grad(a))
print(J.Hess(a))
```

```
***** INSTANCIATION
Fonction (x,y) --> x^2/2+7/2*y^2
***** METHODES DE LA CLASSE
14.5
[ 1 14]
[[1. 0.]
 [0. 7.]]
```

2 Implémentation de nouvelles fonctions

TODO : Dans le fichier `functions.py`, créez une classe nommée `Rosen()` sur le modèle de `square()` qui calcule la fonction, le gradient ou la Hessienne de :

$$f : (x, y) \mapsto 100 * (y - x^2)^2 + (1 - x)^2$$

Créez aussi une classe `oscill()` qui calcule la fonction, le gradient ou la Hessienne de :

$$g : (x, y) \mapsto \frac{1}{2}x^2 + x\cos(y)$$

[3]:
```
import functions as func

R=func.Rosen()
O=func.oscill()
a=np.array([1.3,2.45])
print(R.value(a)) # 57.85

print(O.value(a)) # -0.15630063026149965

print(R.grad(a))
print(R.Hess(a))
print(O.grad(a))

print(O.Hess(a))
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
57.85
-0.15630063026149965
[-394.6  152. ]
[[1050. -520.]
 [-520.  200.]]
[ 0.52976875 -0.82909411]
[[ 1.         -0.6377647 ]
 [-0.6377647   1.00130063]]
```

3 Tests de dérivées numériques

Volontairement, je ne vous ai pas donné les réponses pour le gradient et la Hessienne. Avant de continuer, il faut vérifier que vos calculs sont bons. Pour cela on va faire des tests avec le gradient et la dérivée numérique. Pour ce faire on va partir d'un point a quelconque et on prend une direction d aléatoire. On compare ensuite pour plusieurs valeurs de ε les valeurs suivantes :

$$\frac{J(a + \varepsilon d) - J(a)}{\varepsilon} \simeq (\nabla J(a), d)$$

$$\frac{\nabla J(a + \varepsilon d) - \nabla J(a))}{\varepsilon} \simeq HJ(a)d$$

On rappelle que quand on compare deux nombres b et c, on s'intéresse au nombre b/c. Quand on compare deux vecteurs b et c, on s'intéresse au ratio des normes et à l'angle donné par

$$\frac{(b,c)}{\|b\|\|c\|}$$

On vous donne une fonction `deriv_num(J,a,d,compute_grad=True,compute_Hess=True)` dans `Optim.py` qui teste la dérivée numérique d'une fonction `J`. Les arguments `compute_grad` et `compute_Hess` sont optionnels et déterminent si on doit vérifier le calcul de `J` pour son gradient et sa Hessienne. Ensuite testez votre code pour les 3 fonctions.

[4]:
```
import Optim as opt
np.random.seed(42)
a=np.random.randn(2)
d=np.random.randn(2)
opt.deriv_num(func.square(),a,d)
opt.deriv_num(func.Rosen(),a,d)
opt.deriv_num(func.oscill(),a,d)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
eps 1.0e-01 grad 7.2e-01 ratio 0.0e+00 angle 0.0e+00
eps 1.0e-02 grad 7.2e-02 ratio 2.2e-16 angle 0.0e+00
eps 1.0e-03 grad 7.2e-03 ratio 1.5e-14 angle 1.1e-16
eps 1.0e-04 grad 7.2e-04 ratio 1.4e-13 angle 0.0e+00
eps 1.0e-05 grad 7.2e-05 ratio 5.9e-14 angle 0.0e+00
eps 1.0e-06 grad 7.2e-06 ratio 1.2e-12 angle 0.0e+00
eps 1.0e-07 grad 7.2e-07 ratio 7.3e-11 angle 1.1e-16
eps 1.0e-08 grad 7.0e-08 ratio 8.9e-10 angle 1.1e-16
eps 1.0e-09 grad 1.9e-09 ratio 1.3e-08 angle 0.0e+00
eps 1.0e-10 grad 4.6e-08 ratio 3.8e-08 angle 0.0e+00
eps 1.0e-11 grad 1.6e-06 ratio 3.6e-07 angle 3.2e-15
eps 1.0e-12 grad 6.5e-06 ratio 3.9e-06 angle 7.5e-13
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
eps 1.0e-01 grad 1.5e-01 ratio 1.0e-02 angle 3.4e-03
eps 1.0e-02 grad 1.6e-02 ratio 1.1e-03 angle 3.7e-05
eps 1.0e-03 grad 1.6e-03 ratio 1.1e-04 angle 3.8e-07
eps 1.0e-04 grad 1.6e-04 ratio 1.1e-05 angle 3.8e-09
eps 1.0e-05 grad 1.6e-05 ratio 1.1e-06 angle 3.8e-11
eps 1.0e-06 grad 1.6e-06 ratio 1.1e-07 angle 3.8e-13
eps 1.0e-07 grad 1.6e-07 ratio 1.2e-08 angle 3.8e-15
eps 1.0e-08 grad 1.1e-08 ratio 2.1e-09 angle 0.0e+00
eps 1.0e-09 grad 6.7e-08 ratio 8.0e-08 angle 4.6e-15
eps 1.0e-10 grad 5.0e-07 ratio 6.4e-07 angle 3.4e-15
eps 1.0e-11 grad 2.6e-06 ratio 4.4e-06 angle 5.6e-12
eps 1.0e-12 grad 1.5e-08 ratio 8.7e-05 angle 4.9e-10
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
eps 1.0e-01 grad 2.8e-02 ratio 1.6e-02 angle 1.0e-02
eps 1.0e-02 grad 2.2e-03 ratio 2.4e-03 angle 1.0e-04
eps 1.0e-03 grad 2.1e-04 ratio 2.4e-04 angle 1.0e-06
```

```
eps 1.0e-04 grad 2.1e-05 ratio 2.5e-05 angle 1.0e-08
eps 1.0e-05 grad 2.1e-06 ratio 2.5e-06 angle 1.0e-10
eps 1.0e-06 grad 2.1e-07 ratio 2.5e-07 angle 1.0e-12
eps 1.0e-07 grad 2.1e-08 ratio 2.4e-08 angle 1.0e-14
eps 1.0e-08 grad 1.2e-08 ratio 1.3e-09 angle 0.0e+00
eps 1.0e-09 grad 8.5e-08 ratio 2.6e-09 angle 0.0e+00
eps 1.0e-10 grad 6.0e-07 ratio 2.8e-07 angle 5.5e-14
eps 1.0e-11 grad 1.5e-06 ratio 3.5e-06 angle 1.1e-12
eps 1.0e-12 grad 8.9e-06 ratio 3.6e-05 angle 1.2e-09
```

3.1 Algorithme d'optimisation

Dans `Optim.py`, on vous donne une fonction `main_algorithm(function,step,xini,dc,ls,itermax,tol,verbos` qui lance un algorithme d'optimisation. Les différentes variables sont expliquées dans le bloc de commentaire, nous nous concentrons dans ce paragaphe sur les variables les plus compliquées à comprendre, ce sont les variables `dc` et `ls`, ce sont deux fonctions. La fonction `ls(x, function, step, descent,f,df)` calcul le pas dans la direction de descent donnée par `descent`. Certaines méthodes on besoin de la valeur de la fonction ou de son gradient au point `x`, ces variables sont stockées dans `f` et `df`. La fonction en elle-même est le stockée dans `function`. Pour des raisons qui seront plus claires plus tard, la fonction `ls` rend `x2,f2,df2,step2` où `step2` est le pas choisi par la méthode, `x2,f2,df2` sont, respectivement, les nouvelles valeurs de `x`, de `f` et de `df`. Une exemple de fonction codant l'algorithme de calcul de pas fixe est donné dans `ls_constant`.

La fonction `dc(x,function,df)` calcule la direction de descente. L'algorithme de gradient est donné ci-dessous.

Ensuite on donne un exemple d'optimisation de méthode de gradient à pas fixe

[5]:
```
def ls_constant(x, function, step, descent,f,df) :
    ## FIXED STEP
    step2=step
    x2=x+step2*descent
    f2=function.value(x2)
    df2=function.grad(x2)
    return x2,f2,df2,step2

def dc_gradient(x,function,df) :
    descent=-df
    return descent

x0=np.array([7,1.5])
f=func.square()
res=opt.main_algorithm(f,0.1,x0,ls=ls_constant,dc=dc_gradient)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
iter=   0 f=3.238e+01 df=1.262e+01 comp=[   1,   1,   0]
iter=   1 f=2.055e+01 df=7.044e+00 comp=[   2,   2,   0]
iter=   2 f=1.614e+01 df=5.748e+00 comp=[   3,   3,   0]
iter=   3 f=1.303e+01 df=5.111e+00 comp=[   4,   4,   0]
```

```
iter=   4 f=1.055e+01 df=4.593e+00 comp=[   5,   5,   0]
iter=   5 f=8.543e+00 df=4.134e+00 comp=[   6,   6,   0]
iter=   6 f=6.920e+00 df=3.720e+00 comp=[   7,   7,   0]
iter=   7 f=5.605e+00 df=3.348e+00 comp=[   8,   8,   0]
iter=   8 f=4.540e+00 df=3.013e+00 comp=[   9,   9,   0]
iter=   9 f=3.677e+00 df=2.712e+00 comp=[  10,  10,   0]
iter=  10 f=2.979e+00 df=2.441e+00 comp=[  11,  11,   0]
iter=  11 f=2.413e+00 df=2.197e+00 comp=[  12,  12,   0]
iter=  12 f=1.954e+00 df=1.977e+00 comp=[  13,  13,   0]
iter=  13 f=1.583e+00 df=1.779e+00 comp=[  14,  14,   0]
iter=  14 f=1.282e+00 df=1.601e+00 comp=[  15,  15,   0]
iter=  15 f=1.039e+00 df=1.441e+00 comp=[  16,  16,   0]
iter=  16 f=8.413e-01 df=1.297e+00 comp=[  17,  17,   0]
iter=  17 f=6.814e-01 df=1.167e+00 comp=[  18,  18,   0]
iter=  18 f=5.519e-01 df=1.051e+00 comp=[  19,  19,   0]
iter=  19 f=4.471e-01 df=9.456e-01 comp=[  20,  20,   0]
iter=  20 f=3.621e-01 df=8.510e-01 comp=[  21,  21,   0]
iter=  21 f=2.933e-01 df=7.659e-01 comp=[  22,  22,   0]
iter=  22 f=2.376e-01 df=6.893e-01 comp=[  23,  23,   0]
iter=  23 f=1.925e-01 df=6.204e-01 comp=[  24,  24,   0]
iter=  24 f=1.559e-01 df=5.584e-01 comp=[  25,  25,   0]
iter=  25 f=1.263e-01 df=5.025e-01 comp=[  26,  26,   0]
iter=  26 f=1.023e-01 df=4.523e-01 comp=[  27,  27,   0]
iter=  27 f=8.284e-02 df=4.070e-01 comp=[  28,  28,   0]
iter=  28 f=6.710e-02 df=3.663e-01 comp=[  29,  29,   0]
iter=  29 f=5.435e-02 df=3.297e-01 comp=[  30,  30,   0]
iter=  30 f=4.403e-02 df=2.967e-01 comp=[  31,  31,   0]
iter=  31 f=3.566e-02 df=2.671e-01 comp=[  32,  32,   0]
iter=  32 f=2.889e-02 df=2.404e-01 comp=[  33,  33,   0]
iter=  33 f=2.340e-02 df=2.163e-01 comp=[  34,  34,   0]
iter=  34 f=1.895e-02 df=1.947e-01 comp=[  35,  35,   0]
iter=  35 f=1.535e-02 df=1.752e-01 comp=[  36,  36,   0]
iter=  36 f=1.243e-02 df=1.577e-01 comp=[  37,  37,   0]
iter=  37 f=1.007e-02 df=1.419e-01 comp=[  38,  38,   0]
iter=  38 f=8.158e-03 df=1.277e-01 comp=[  39,  39,   0]
iter=  39 f=6.608e-03 df=1.150e-01 comp=[  40,  40,   0]
iter=  40 f=5.353e-03 df=1.035e-01 comp=[  41,  41,   0]
iter=  41 f=4.336e-03 df=9.312e-02 comp=[  42,  42,   0]
iter=  42 f=3.512e-03 df=8.381e-02 comp=[  43,  43,   0]
iter=  43 f=2.845e-03 df=7.543e-02 comp=[  44,  44,   0]
iter=  44 f=2.304e-03 df=6.788e-02 comp=[  45,  45,   0]
iter=  45 f=1.866e-03 df=6.110e-02 comp=[  46,  46,   0]
iter=  46 f=1.512e-03 df=5.499e-02 comp=[  47,  47,   0]
iter=  47 f=1.225e-03 df=4.949e-02 comp=[  48,  48,   0]
iter=  48 f=9.919e-04 df=4.454e-02 comp=[  49,  49,   0]
iter=  49 f=8.034e-04 df=4.008e-02 comp=[  50,  50,   0]
iter=  50 f=6.508e-04 df=3.608e-02 comp=[  51,  51,   0]
iter=  51 f=5.271e-04 df=3.247e-02 comp=[  52,  52,   0]
```

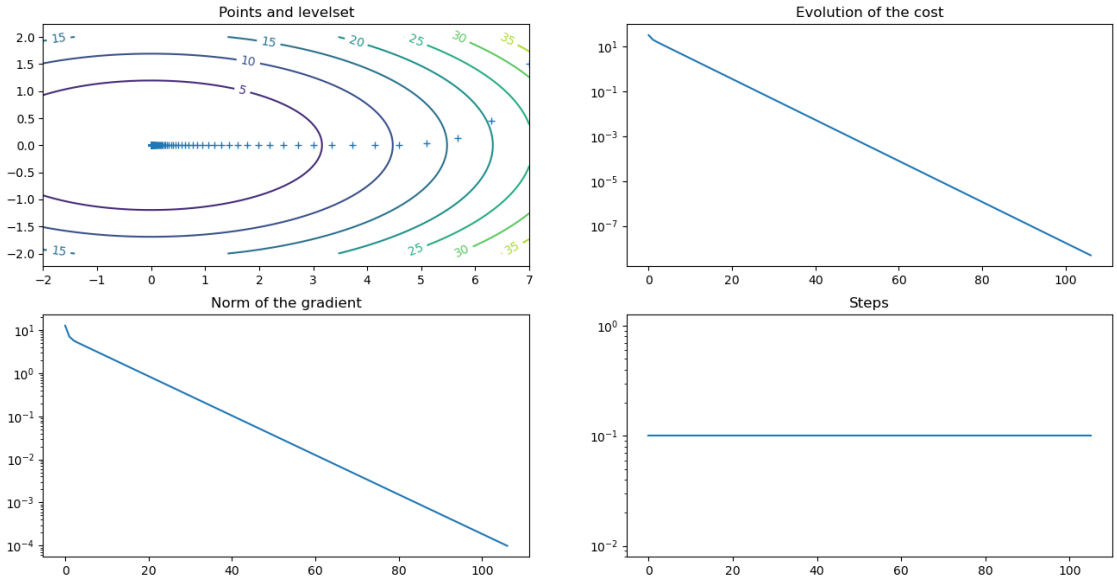
```
iter=  52 f=4.270e-04 df=2.922e-02 comp=[  53,  53,   0]
iter=  53 f=3.458e-04 df=2.630e-02 comp=[  54,  54,   0]
iter=  54 f=2.801e-04 df=2.367e-02 comp=[  55,  55,   0]
iter=  55 f=2.269e-04 df=2.130e-02 comp=[  56,  56,   0]
iter=  56 f=1.838e-04 df=1.917e-02 comp=[  57,  57,   0]
iter=  57 f=1.489e-04 df=1.726e-02 comp=[  58,  58,   0]
iter=  58 f=1.206e-04 df=1.553e-02 comp=[  59,  59,   0]
iter=  59 f=9.767e-05 df=1.398e-02 comp=[  60,  60,   0]
iter=  60 f=7.912e-05 df=1.258e-02 comp=[  61,  61,   0]
iter=  61 f=6.408e-05 df=1.132e-02 comp=[  62,  62,   0]
iter=  62 f=5.191e-05 df=1.019e-02 comp=[  63,  63,   0]
iter=  63 f=4.205e-05 df=9.170e-03 comp=[  64,  64,   0]
iter=  64 f=3.406e-05 df=8.253e-03 comp=[  65,  65,   0]
iter=  65 f=2.759e-05 df=7.428e-03 comp=[  66,  66,   0]
iter=  66 f=2.234e-05 df=6.685e-03 comp=[  67,  67,   0]
iter=  67 f=1.810e-05 df=6.017e-03 comp=[  68,  68,   0]
iter=  68 f=1.466e-05 df=5.415e-03 comp=[  69,  69,   0]
iter=  69 f=1.187e-05 df=4.873e-03 comp=[  70,  70,   0]
iter=  70 f=9.619e-06 df=4.386e-03 comp=[  71,  71,   0]
iter=  71 f=7.791e-06 df=3.947e-03 comp=[  72,  72,   0]
iter=  72 f=6.311e-06 df=3.553e-03 comp=[  73,  73,   0]
iter=  73 f=5.112e-06 df=3.197e-03 comp=[  74,  74,   0]
iter=  74 f=4.141e-06 df=2.878e-03 comp=[  75,  75,   0]
iter=  75 f=3.354e-06 df=2.590e-03 comp=[  76,  76,   0]
iter=  76 f=2.717e-06 df=2.331e-03 comp=[  77,  77,   0]
iter=  77 f=2.200e-06 df=2.098e-03 comp=[  78,  78,   0]
iter=  78 f=1.782e-06 df=1.888e-03 comp=[  79,  79,   0]
iter=  79 f=1.444e-06 df=1.699e-03 comp=[  80,  80,   0]
iter=  80 f=1.169e-06 df=1.529e-03 comp=[  81,  81,   0]
iter=  81 f=9.472e-07 df=1.376e-03 comp=[  82,  82,   0]
iter=  82 f=7.673e-07 df=1.239e-03 comp=[  83,  83,   0]
iter=  83 f=6.215e-07 df=1.115e-03 comp=[  84,  84,   0]
iter=  84 f=5.034e-07 df=1.003e-03 comp=[  85,  85,   0]
iter=  85 f=4.077e-07 df=9.030e-04 comp=[  86,  86,   0]
iter=  86 f=3.303e-07 df=8.127e-04 comp=[  87,  87,   0]
iter=  87 f=2.675e-07 df=7.315e-04 comp=[  88,  88,   0]
iter=  88 f=2.167e-07 df=6.583e-04 comp=[  89,  89,   0]
iter=  89 f=1.755e-07 df=5.925e-04 comp=[  90,  90,   0]
iter=  90 f=1.422e-07 df=5.332e-04 comp=[  91,  91,   0]
iter=  91 f=1.152e-07 df=4.799e-04 comp=[  92,  92,   0]
iter=  92 f=9.328e-08 df=4.319e-04 comp=[  93,  93,   0]
iter=  93 f=7.556e-08 df=3.887e-04 comp=[  94,  94,   0]
iter=  94 f=6.120e-08 df=3.499e-04 comp=[  95,  95,   0]
iter=  95 f=4.957e-08 df=3.149e-04 comp=[  96,  96,   0]
iter=  96 f=4.015e-08 df=2.834e-04 comp=[  97,  97,   0]
iter=  97 f=3.252e-08 df=2.550e-04 comp=[  98,  98,   0]
iter=  98 f=2.634e-08 df=2.295e-04 comp=[  99,  99,   0]
iter=  99 f=2.134e-08 df=2.066e-04 comp=[ 100, 100,   0]
```

```
iter= 100 f=1.728e-08 df=1.859e-04 comp=[ 101, 101,   0]
iter= 101 f=1.400e-08 df=1.673e-04 comp=[ 102, 102,   0]
iter= 102 f=1.134e-08 df=1.506e-04 comp=[ 103, 103,   0]
iter= 103 f=9.186e-09 df=1.355e-04 comp=[ 104, 104,   0]
iter= 104 f=7.441e-09 df=1.220e-04 comp=[ 105, 105,   0]
iter= 105 f=6.027e-09 df=1.098e-04 comp=[ 106, 106,   0]
iter= 106 f=4.882e-09 df=9.881e-05 comp=[ 107, 107,   0]
Success !!! Algorithm converged !!!
```

On vous donne aussi une fonction `graphical_info` qui permet de donner des informations sur le tableau `res` de convergence

[6]:
```
#TEST 1 : fonctionnne très bien
x0,f=np.array([7,1.5]),func.square()
res=opt.main_algorithm(f,0.1,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
opt.graphical_info(res,f,xmax=7)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
```



TODO :Essayer les 3 fonctions avec plusieurs pas et affichez le nombre d'itérations nécessaires et la valeur finale de la fonction obtenue et la valeur finale du gradient, vous devez faire apparaître le fait qu'un pas trop petit ou trop grand ne fait pas converger. Vous pouvez vous aider du code qui est déjà fait pour la fonction `square` et remplir uniquement les `TODO` par des listes de pas adéquates.

[7]:
```
x0=np.array([-1,1])
f=func.square()
for s in [0.25,0.125,0.05,1e-3,1e-4] :
    res=opt.main_algorithm(f,s,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
```

```
    print("step : {:1.2e} iter : {:3d} cost :{:1.3e} grad :{:1.3e}".
 ↪format(s,len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1]))

x0=np.array([-1,1])
f=func.oscill()
TODO = []
for s in [0.325,0.25,0.125,0.05,1e-3,1e-4]:
    res=opt.main_algorithm(f,s,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
    print("step : {:1.2e} iter : {:3d} cost :{:1.3e} grad :{:1.3e}".
 ↪format(s,len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1]))

f=func.Rosen()
x0=np.array([-1.,1])
TODO = []
for s in [0.325,0.25,0.125,0.05,1e-3,1e-4]:
    res=opt.
 ↪main_algorithm(f,s,x0,ls=ls_constant,dc=dc_gradient,verbose=False,itermax=30000)
    print("step : {:1.2e} iter : {:3d} cost :{:1.3e} grad :{:1.3e}".
 ↪format(s,len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1]))
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
step : 2.50e-01 iter :  40 cost :7.192e-10 grad :9.481e-05
step : 1.25e-01 iter :  70 cost :4.967e-09 grad :9.967e-05
step : 5.00e-02 iter : 181 cost :4.780e-09 grad :9.778e-05
step : 1.00e-03 iter : 9207 cost :4.997e-09 grad :9.997e-05
step : 1.00e-04 iter : 20001 cost :9.156e-03 grad :1.353e-01
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
step : 3.25e-01 iter :  27 cost :-5.000e-01 grad :7.862e-05
step : 2.50e-01 iter :  36 cost :-5.000e-01 grad :8.365e-05
step : 1.25e-01 iter :  75 cost :-5.000e-01 grad :8.968e-05
step : 5.00e-02 iter : 191 cost :-5.000e-01 grad :9.698e-05
step : 1.00e-03 iter : 9679 cost :-5.000e-01 grad :9.991e-05
step : 1.00e-04 iter : 20001 cost :-4.822e-01 grad :1.763e-01
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
step : 3.25e-01 iter :   8 cost :nan grad :nan
step : 2.50e-01 iter :   9 cost :nan grad :nan
step : 1.25e-01 iter :   8 cost :nan grad :nan
step : 5.00e-02 iter :   9 cost :nan grad :nan
step : 1.00e-03 iter : 20508 cost :1.251e-08 grad :9.997e-05
```

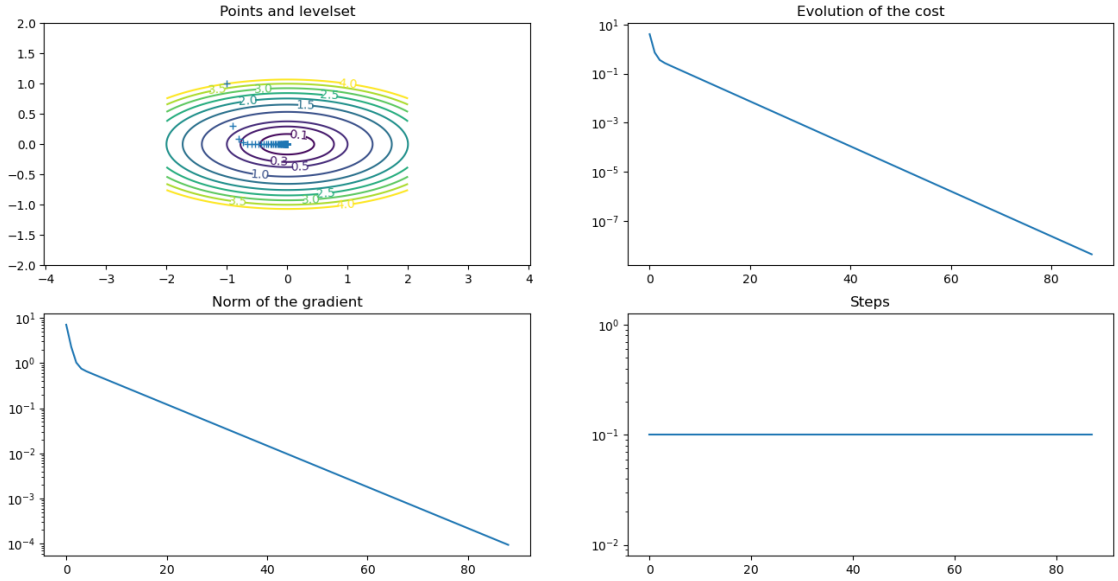
```
/home/rosas/Bureau/4A/OPTI/functions.py:39: RuntimeWarning: overflow encountered
in scalar power
  return 100*(x[1]-x[0]**2)**2+(1 - x[0])**2
/home/rosas/Bureau/4A/OPTI/functions.py:45: RuntimeWarning: overflow encountered
in scalar power
  -400*x[0]*(x[1]-x[0]**2)+2*(x[0]-1),
/home/rosas/Bureau/4A/OPTI/functions.py:46: RuntimeWarning: overflow encountered
```

```
in scalar power
  200*(x[1]-x[0]**2)
/home/rosas/Bureau/4A/OPTI/functions.py:39: RuntimeWarning: invalid value
encountered in scalar subtract
  return 100*(x[1]-x[0]**2)**2+(1 - x[0])**2
/home/rosas/Bureau/4A/OPTI/functions.py:45: RuntimeWarning: invalid value
encountered in scalar subtract
  -400*x[0]*(x[1]-x[0]**2)+2*(x[0]-1),
/home/rosas/Bureau/4A/OPTI/functions.py:46: RuntimeWarning: invalid value
encountered in scalar subtract
  200*(x[1]-x[0]**2)
/home/rosas/Bureau/4A/OPTI/functions.py:45: RuntimeWarning: overflow encountered
in scalar multiply
  -400*x[0]*(x[1]-x[0]**2)+2*(x[0]-1),
/tmp/ipykernel_7048/2981709185.py:4: RuntimeWarning: invalid value encountered
in add
  x2=x+step2*descent

step : 1.00e-04 iter : 30001 cost :2.385e-02 grad :1.571e-01
```

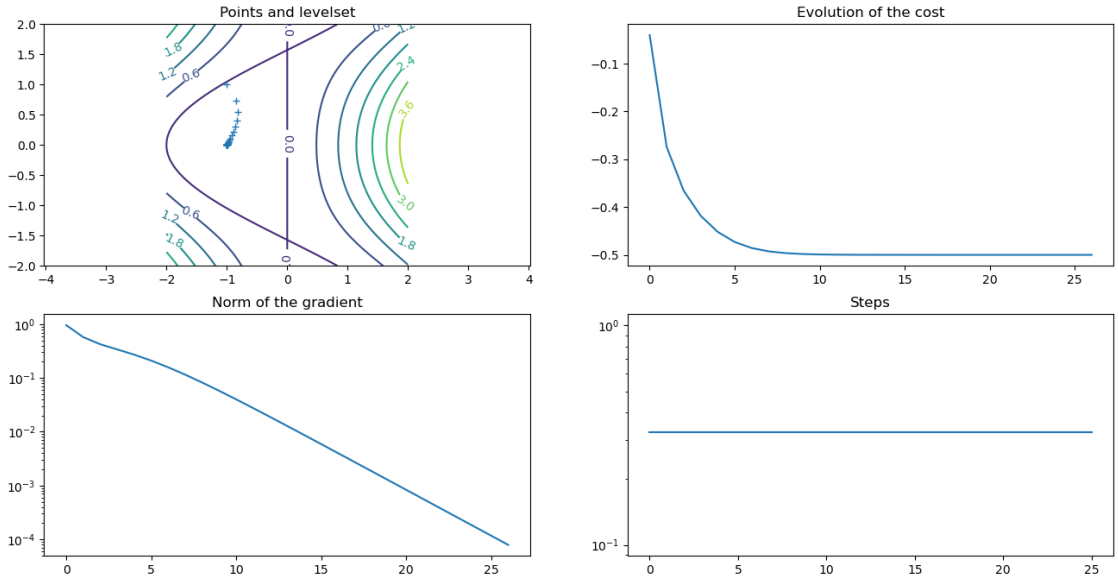
Vous pouvez garder ici trois tests convergents. Remplacez `TODO` par ce qu'il faut

[8]:
```
## TEST 1 convergent test for the function 'square':
f=func.square()
x0=np.array([-1,1])
res=opt.main_algorithm(f,0.1,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
opt.graphical_info(res,f,[0,0.1,0.3,0.5,1,1.5,2,2.5,3,3.5,4])

## TEST 2 convergent test for the function 'Oscill':
f=func.oscill()
x0=np.array([-1,1])
TODO=0.
res=opt.main_algorithm(f,0.325,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
opt.graphical_info(res,f)

## TEST 3 convergent test for the function 'Rosen':
f=func.Rosen()
TODO=0.
x0=np.array([-1,1])
res=opt.main_algorithm(f,1e-3,x0,ls=ls_constant,dc=dc_gradient,verbose=False)
opt.graphical_info(res,f,[0,1,10,100,1000])
```

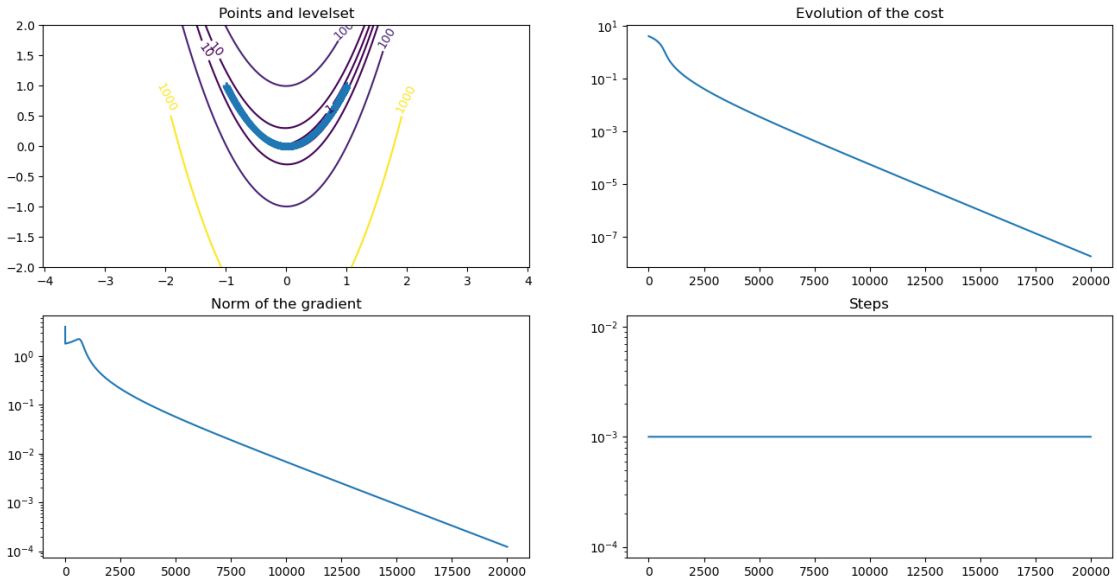
```
Fonction (x,y) --> x^2/2+7/2*y^2
```



Fonction (x,y) --> 1/2*x^2 +x*cos(y)



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



3.2 Line search Backtracking

Implémenter une fonction de recherche linéaire `ls_backtracking` qui calcule un pas par rebroussement, c'est à dire qui vérifie que la fonction décroît et qui divise le pas par 2 si elle ne décroît pas. La fonction doit s'écrire sous la forme

`x2,f2,df2,step2 = ls_backtracking(x, function, step, descent,f,df)`

Les arguments en entrée sont - `x` : l'itéré actuel - `function` : qui est la fonction que l'on minimise - `step` : qui est le pas initial de la line-search - `descent` : la direction de descente - `f` : la valeur de la fonction au point `x` - `df` : la valeur du gradient de la fonction au point `x`

Les arguments en sortie sont - `x2` : le nouvel itéré, il vaut `x+step2*descent` - `f2` : la valeur de la fonction au point `x2` - `df2` : la valeur du gradient de la fonction au point `x2` - `step2` : le pas calculé par la méthode

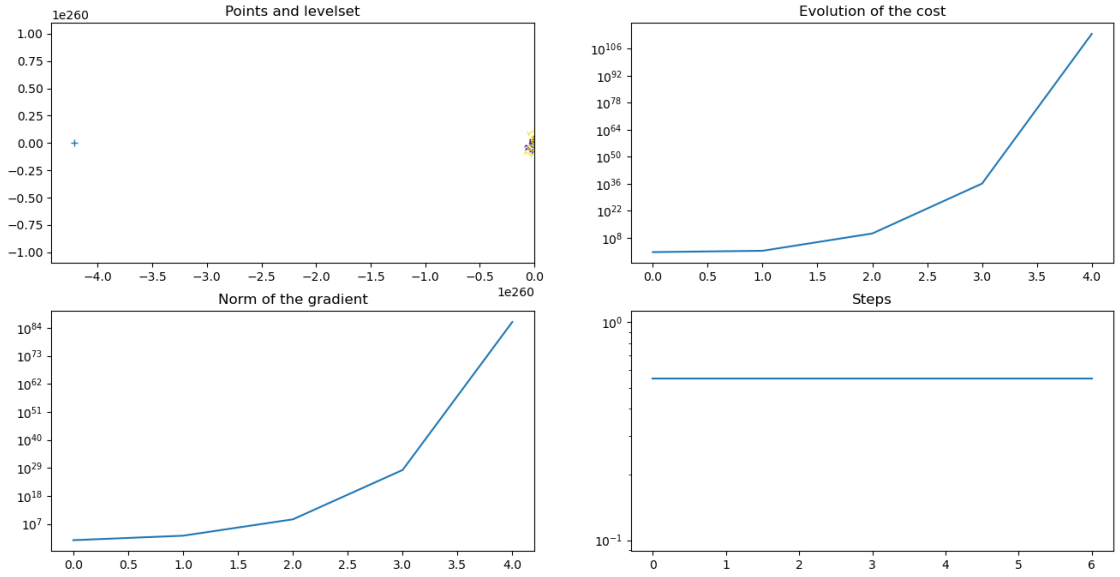
Tester l'algorithme de descente de gradient avec cette recherche linéaire et observer que cette recherche linéaire est plus stable que la précédente. Vérifier que pour `step=1` la méthode avec `ls_constant` **diverge** pour la fonction `Rosen` mais elle **converge** avec `ls_backtracking`. Montrez que le coût supplémentaire de cette méthode est négligeable.

[9]:
```
def ls_backtracking(x,function,step,descent,f,df) :
    step2 = step
    x2 = x + step2 * descent
    f2 = function.value(x2)
    while f2 >= f:
        step2 /= 2
        x2 = x + step2 * descent
        f2 = function.value(x2)
    df2 = function.grad(x2)
```

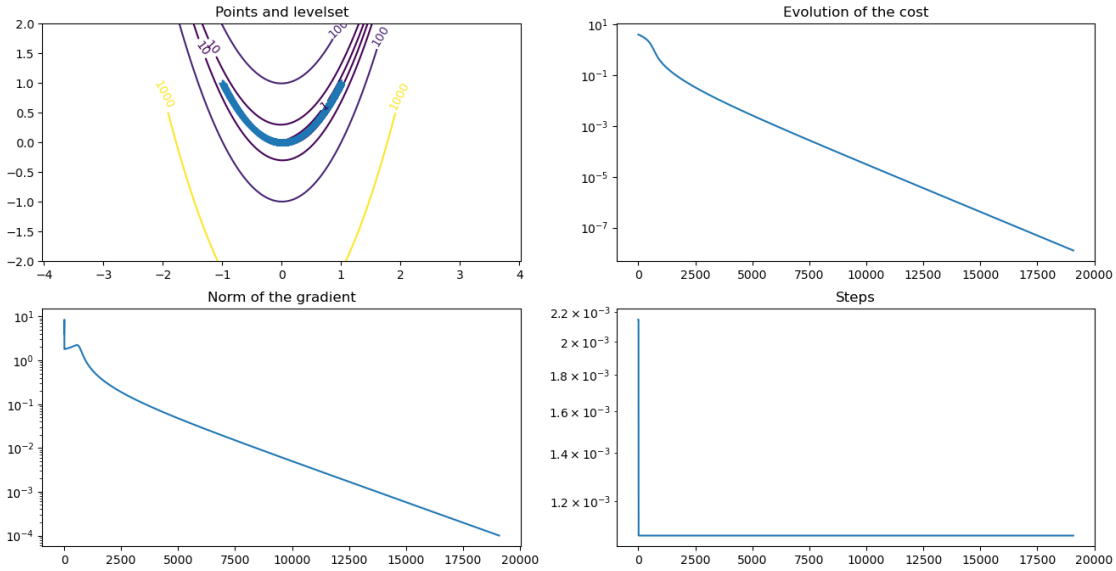
```
    return x2, f2, df2, step2
```

[10]:
```
f=func.Rosen()
x0=np.array([-1,1])
res=opt.main_algorithm(f,0.
 ↪55,x0,ls=ls_constant,dc=dc_gradient,verbose=False,itermax=40000)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f,[0,1,10,100,1000])

f=func.Rosen()
x0=np.array([-1,1])
res=opt.main_algorithm(f,0.
 ↪55,x0,ls=ls_backtracking,dc=dc_gradient,verbose=False,itermax=40000)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f,[0,1,10,100,1000])
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :   8 cost :nan grad :nan comp=[   8,   8,   0]
```




```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter : 19083 cost :1.251e-08 grad :9.996e-05 comp=[19092,19083,   0]
```



3.3 Partial Line search

Implémenter une fonction de recherche linéaire `ls_partial_backtrack` qui commence par diviser le pas par 10 tant que la fonction ne diminue pas. Puis elle calcule le pas s_{k+1} parmi

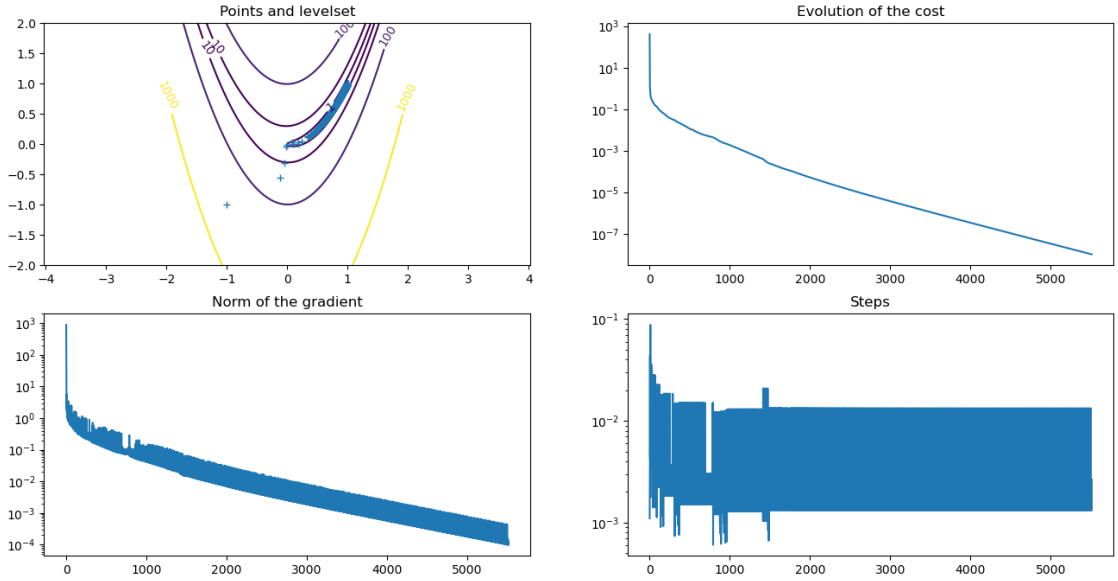
$$\{0.1s_k, 0.5s_k, s_k, 2s_k, 10s_k\}$$

qui minimise $f(x_k + sd_k)$. Tester à nouveau l'algorithme de gradient et comparer la variable `nb_computations` entre cette méthode et les précédentes.

[11]:
```
def ls_partial_backtrack(x,function,step,descent,f,df) :

    step2 = step
    x2 = x + step2 * descent
    f2 = function.value(x2)
    s = np.array([0.1,0.5,1,2,10])
    while f2 >= f:
        step2 /= 10
        x2 = x + step2 * descent
        f2 = function.value(x2)

    sk = step2*s
    min = np.argmin([function.value(x + sk[i]*descent) for i in range(len(s))])
    step2 = sk[min]
    x2 = x + step2 * descent
    f2 = function.value(x2)
    df2 = function.grad(x2)
    return x2,f2,df2,step2
```

```
f=func.Rosen()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,0.
  ↪55,x0,ls=ls_partial_backtrack,dc=dc_gradient,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
  ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
  ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f,[0,1,10,100,1000])
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter : 5513 cost :1.089e-08 grad :9.908e-05 comp=[39296,5513,   0]
```



[ ]:

3.4 Algorithme de Wolfe

On rappelle les conditions de Wolfe:

$$f(x_k + sd_k) \leq f(x_k) + \epsilon_1 s(\nabla f(x_k)^T d_k)$$

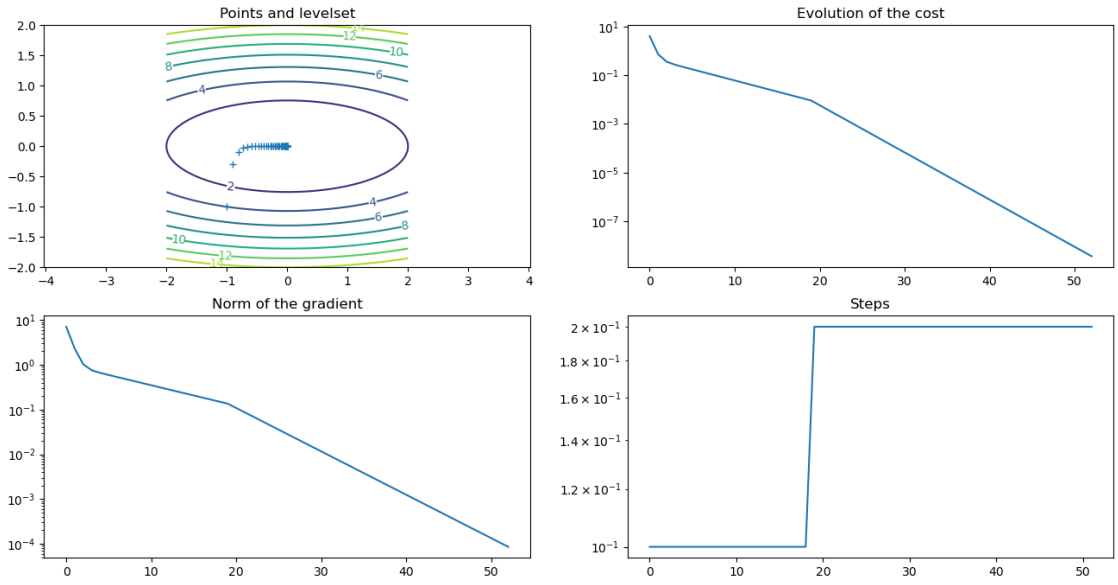
$$\nabla f(x_k + sd_k)^T d_k \geq \epsilon_2 (\nabla f(x_k)^T d_k)$$

avec, en pratique: $\epsilon_1 = 10^{-4}$ et $\epsilon_2 = 0.9$.
Implémenter une fonction `ls_wolfe`. Tester l'algorithme de gradient avec pas de Wolfe sur les 3 fonctions tests proposées.

[ ]:
```
def ls_wolfe(x,function,step,descent,f,df) :
    step_min,step_max=0.,np.inf
    i=0
    mycontinue=True
    scal=np.dot(df,descent)
    step2=step
    eps1,eps2=0.1,0.9
    while mycontinue :
        i=i+1
        mycontinue=False
        x2=x+step2*descent
        f2=function.value(x2)
        if f2>f+eps1*step2*scal :
            # step is too big, decrease it
            step_max=step2
            step2=0.5*(step_min+step_max)
            mycontinue=True
        else :
            df2=function.grad(x2)
            if np.dot(df2,descent) < eps2*scal :
                # step is too small, increase it
                step_min=step2
                step2=min(0.5*(step_min+step_max),2*step_min)
                mycontinue=True
    return x2,f2,df2,step2
```

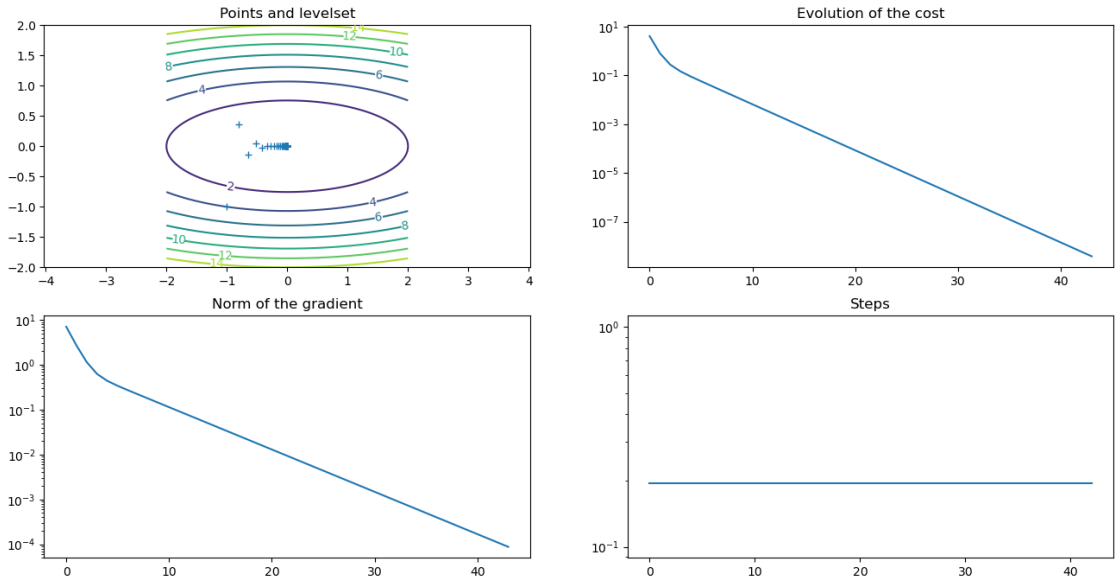
[13]:
```
# TEST1 : SQUARE FUNCTION : here Wolfe is useless
f=func.square()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,0.1,x0,ls=ls_wolfe,dc=dc_gradient,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
iter :  53 cost :3.665e-09 grad :8.562e-05 comp=[  54, 106,   0]
```



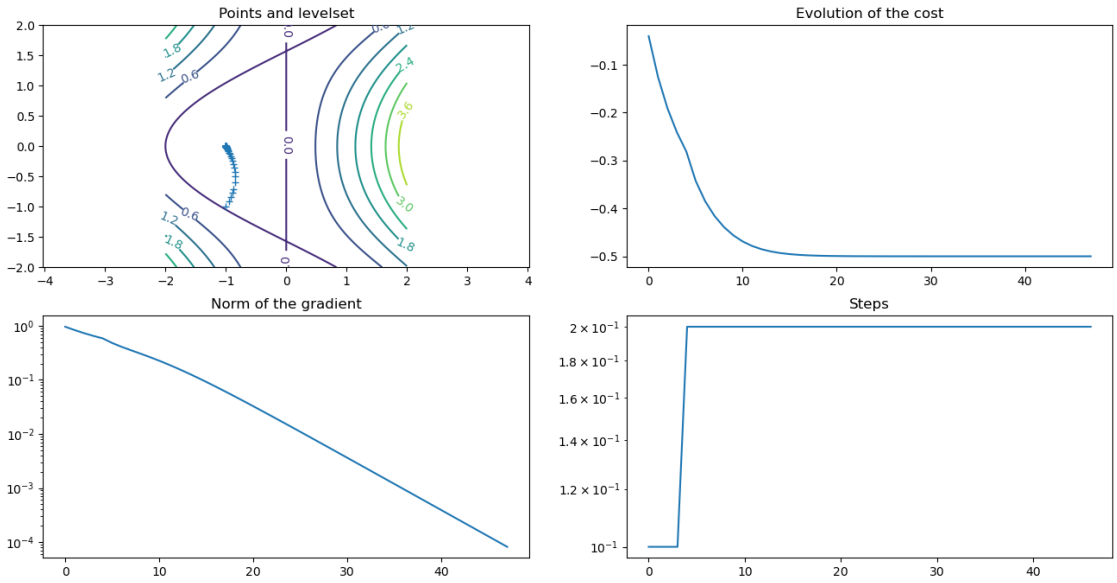
[14]:
```
# TEST2 : SQUARE FUNCTION : here Wolfe is usefull but only for the first␣
 ↪iteration
f=func.square()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,100,x0,ls=ls_wolfe,dc=dc_gradient,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
iter :  44 cost :3.827e-09 grad :8.749e-05 comp=[  53,  87,   0]
```



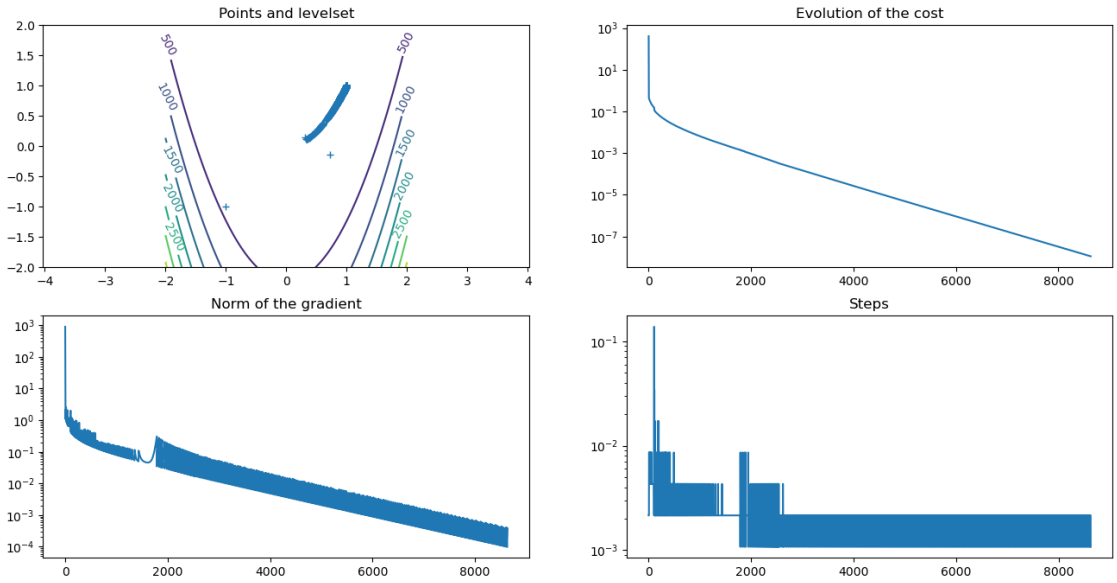
[15]:
```
# TEST3 : OSCILL FUNCTION : Wolfe increases the step here
f=func.oscill()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,0.1,x0,ls=ls_wolfe,dc=dc_gradient,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
iter :  48 cost :-5.000e-01 grad :8.181e-05 comp=[  49,  96,   0]
```



[16]:
```
# TEST4 : ROSEN FUNCTION : Wolfe is kind of erratic but good
f=func.Rosen()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,0.55,x0,ls=ls_wolfe,dc=dc_gradient,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter : 8631 cost :1.132e-08 grad :9.944e-05 comp=[9700,17791,   0]
```



3.5 Algorithme de Newton

Dans cette deuxième partie, nous allons implémenter les algorithmes de Newton. Il s'agit de prendre comme direction de descente d_k solution de

$$d_k = Hf(x_k)^{-1}\nabla f(x_k)$$

Attention ce choix de d_k ne donne pas toujours une direction de descente. On va donc calculer l'angle entre d_k et $\nabla f(x_k)$, i.e, on calcule

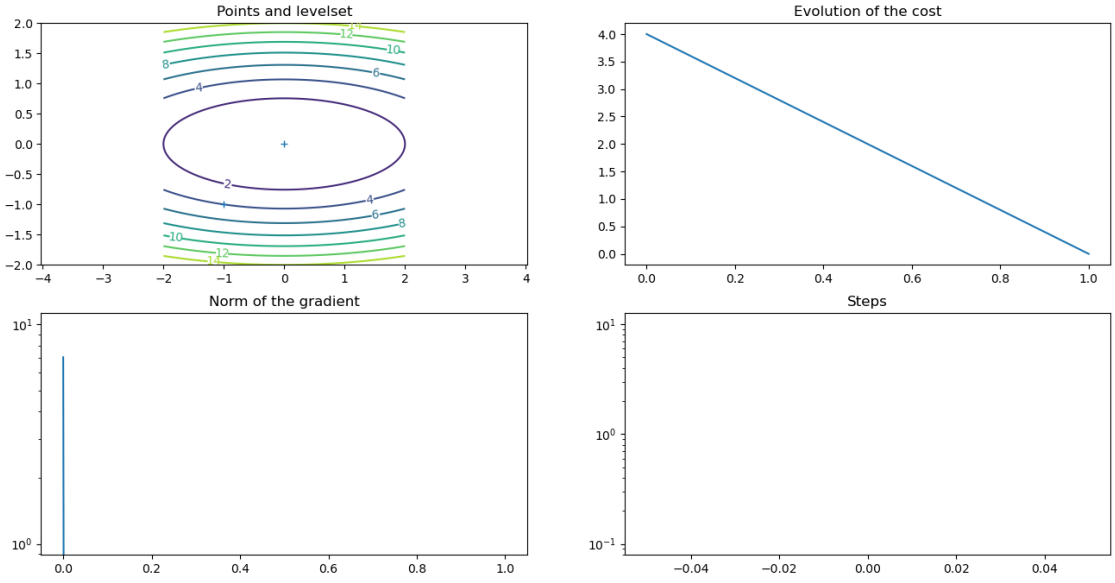
$$\cos(\theta_k) = \frac{\langle d_k, -\nabla f(x_k)\rangle}{\|d_k\|\|\nabla f(x_k)\|}$$

Si $\cos(\theta_k) > 0.1$ alors l'algorithme de Newton rend d_k sinon il se transforme en algorithme de gradient et rend $-\nabla f(x_k)$. Essayez la méthode de Newton avec pas constant. De préférence avec un pas de 1.

[57]:
```
def dc_Newton(x,function,df) :
    hess=function.Hess(x)
    d=-np.linalg.solve(hess,df)

    cos = np.dot(d,-df)/ (np.linalg.norm(d,np.inf)*np.linalg.norm(df, np.inf))

    if cos > 0.1 :
        descent = d
    else :
        descent = -df
    return descent
```

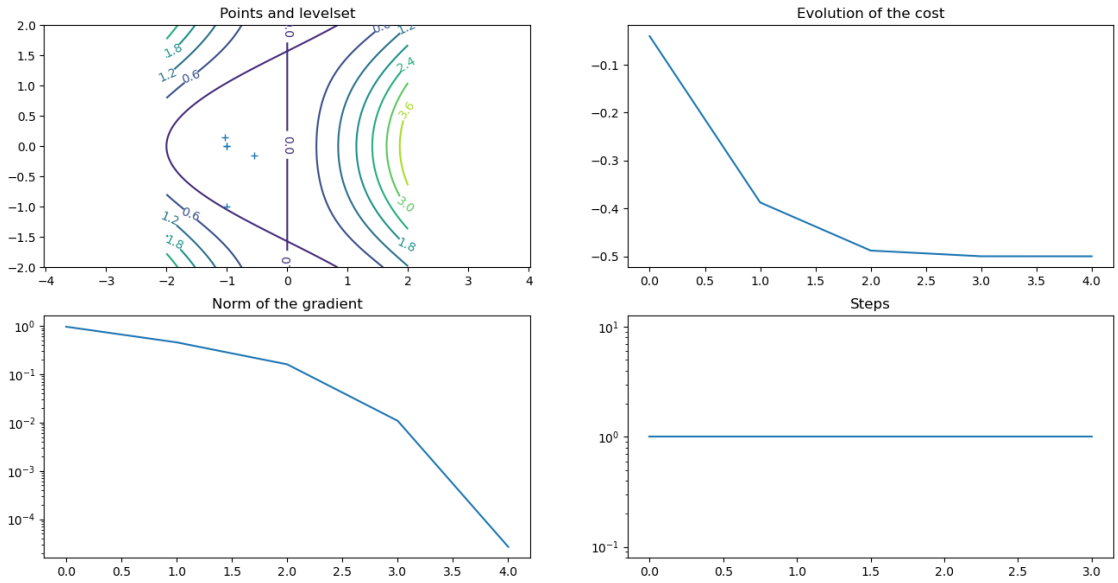
[58]:
```
# TEST1 : SQUARE FUNCTION
f=func.square()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,1,x0,ls=ls_constant,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> x^2/2+7/2*y^2
iter :   2 cost :0.000e+00 grad :0.000e+00 comp=[   2,   2,   1]
```



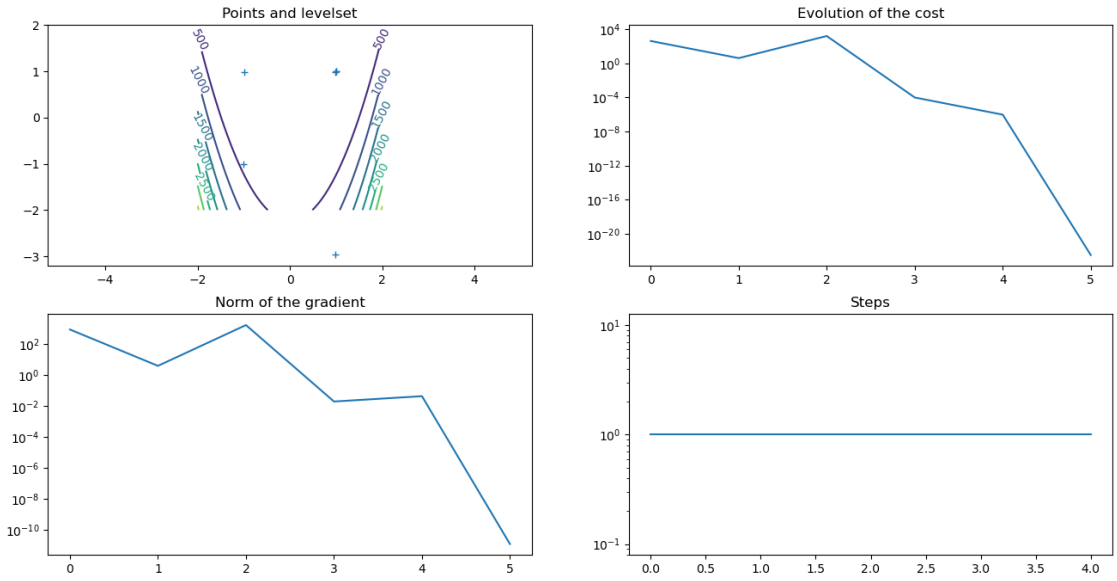
[59]:
```
# TEST2 : OSCILL FUNCTION  :
f=func.oscill()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,1,x0,ls=ls_constant,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 1/2*x^2 +x*cos(y)
iter :   5 cost :-5.000e-01 grad :2.677e-05 comp=[   5,   5,   4]
```



[60]:
```
# TEST3 : ROSEN FUNCTION
# convergence ultra rapide, mais on a l'impression que c'est de la chance.
f=func.Rosen()
x0=np.array([-1,-1])
res=opt.main_algorithm(f,1,x0,ls=ls_constant,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :   6 cost :3.478e-23 grad :1.175e-11 comp=[   6,   6,   5]
```

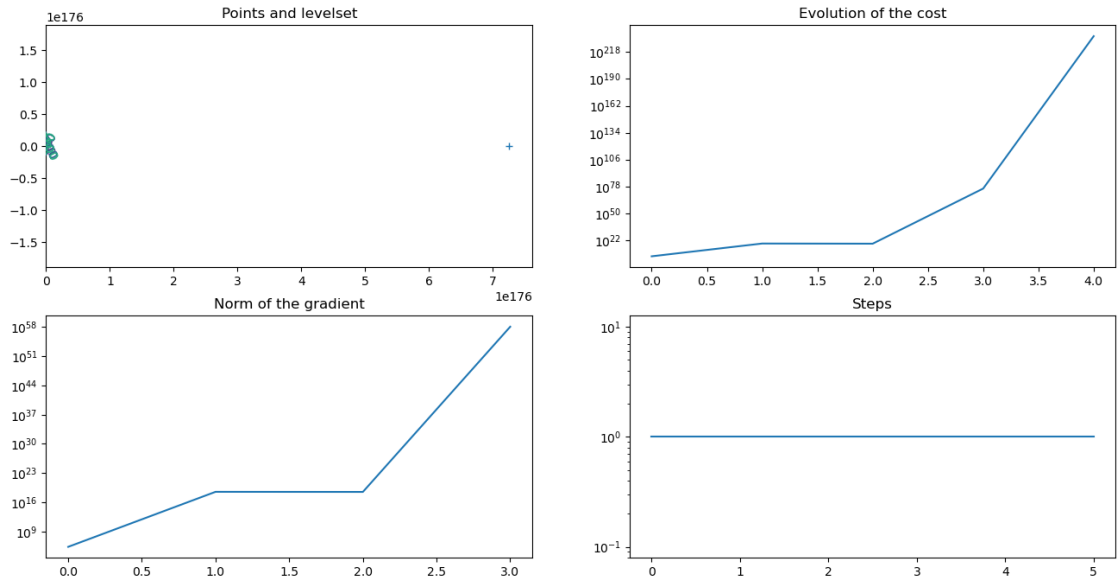


Tester la méthode de Newton avec pas de Wolfe sur les 3 fonctions et comparer les résultats obtenus avec ceux des algos de gradient avec pas de Wolfe et l'algorithme de Newton classique. On essaiera aussi une nouvelle fonction linesearch qui met comme premier pas de Wolfe le pas 1. Elle est définie de la manière suivante

```
def ls_wolfe_step_is_one(x,function,step,descent,f,df) :     return
ls_wolfe(x,function,1.,descent,f,df)
```

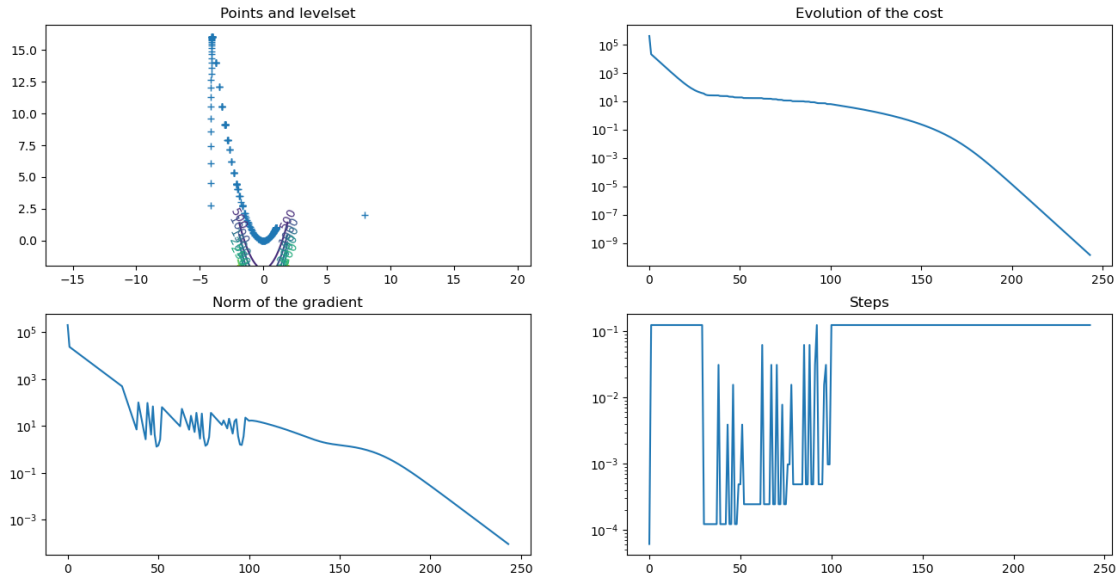
[46]:
```
def ls_wolfe_step_is_one(x,function,step,descent,f,df) :
    return ls_wolfe(x,function,1.,descent,f,df)

# TEST3 : ROSEN FUNCTION
f=func.Rosen()
x0=np.array([8,2])
res=opt.main_algorithm(f,1,x0,ls=ls_constant,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :   7 cost :nan grad :nan comp=[   7,   7,   6]
```



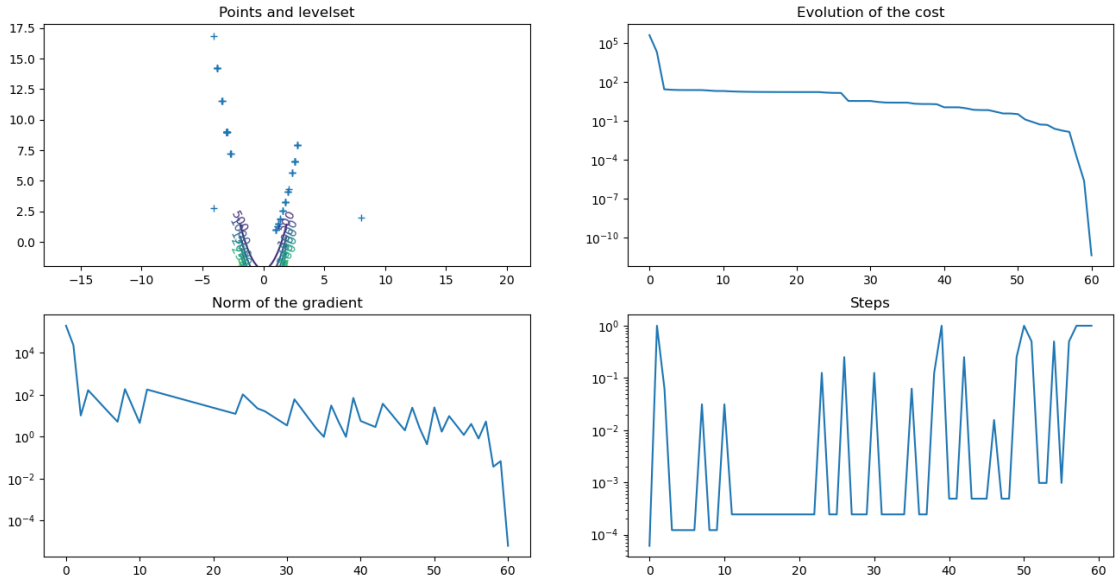
[61]:
```
f=func.Rosen()
x0=np.array([8,2])
res=opt.main_algorithm(f,1,x0,ls=ls_wolfe,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter : 244 cost :1.431e-10 grad :9.285e-05 comp=[ 455, 591, 243]
```



[62]:
```
f=func.Rosen()
x0=np.array([8,2])
res=opt.
 ↪main_algorithm(f,1,x0,ls=ls_wolfe_step_is_one,dc=dc_Newton,verbose=False)
print("iter : {:3d} cost :{:1.3e} grad :{:1.3e} comp=[{:4d},{:4d},{:4d}]"
      .
 ↪format(len(res['list_costs']),res['list_costs'][-1],res['list_grads'][-1],f.
 ↪nb_eval,f.nb_grad,f.nb_hess))
opt.graphical_info(res,f)
```

```
Fonction (x,y) --> 100*(y-x^2)^2 +(1-x)^2
iter :  61 cost :4.261e-12 grad :6.075e-06 comp=[ 578, 121,  60]
```



[ ]:
```

```