Master M2DS-SAF: TP Optimisation

Méthodes de descente: gradient, Newton

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass, field
```

Dans ce TP sont considérés et comparés les algorithmes de gradient et de Newton sans contrainte. Il est suggéré de regarder plus particulièrement le cas d'une fonction quadratique et de la fonction banane de Rosenbrock. L'optimisation se fera par rapport à $x = (x_1, x_2)$ dans R^2.

Le code suivant définit une classe avec la fonction de Rosenbrock et les méthodes `grad`, `hess` qui renvoient le gradient et le Hessien. La fonction est définie par:

$$f(x) = (x_1 - 1)^2 + \alpha(x_1^2 - x2)^2$$

Il s'agit d'une fonction non convexe.

In [2]:
```
class Rosenbrock(object):
    """
    Class for Rosenbrock (banana-like) function (x1-1)**2+alpha*(x1**2-x2)**2

    Attributes
    ----------
    alpha : float, default = 100

    Methods
    -------
    __call__ : computes function value at x
    grad(x) : computes gradient at x
    hess(x) : computes Hessian at x
    """
    def __init__(self, alpha=100):
        self.alpha = alpha

    def __call__(self, x):
        """
        Computes function's value

        Parameters
        ----------
        x : array_like

        Returns
        -------
        y : float
        """
        y = (x[0]-1)**2 + self.alpha*(x[0]**2-x[1])**2
        return y

    def grad(self, x):
        """
        Computes function's gradient

        Parameters
        ----------
        x : array_like
```

```
        Returns
        -------
        g : array_like
        """
        grad = np.c_[(2*(x[0]-1) + 4*self.alpha*x[0]*(x[0]**2-x[1]),
                      -2*self.alpha*(x[0]**2-x[1]))].T
        return grad

    def hess(self, x):
        """
        Computes function's Hessian

        Parameters
        ----------
        x : array_like

        Returns
        -------
        H : array_like
        """
        hessian = np.empty([2, 2])
        hessian[0, 0] = 2 + 4*self.alpha*(3*x[0]**2-x[1])
        hessian[0, 1] = -4*self.alpha*x[0]
        hessian[1, 0] = hessian[0, 1]
        hessian[1, 1] = 2*self.alpha
        return hessian
```

Le code suivant définit une classe avec une fonction quadratique et les méthodes `grad`, `hess` qui renvoient le gradient et le Hessien.

In [3]:
```
class Quadfunc(object):
    """
    class for quadratic function 1/2*x.T@Q@x

    Attributes
    ----------
    A : array_like, default = np.eye(2)

    Methods
    -------
    __call__ : computes function value at x
    grad(x) : computes gradient at x
    hess(x) : computes Hessian at x
    """
    def __init__(self, Q=np.eye(2)):
        self.Q = Q

    def __call__(self, x):
        y = 1/2*((self.Q@x)*x).sum(axis=0)
        return y

    def grad(self, x):
        grad = 1/2*(self.Q+self.Q.T)@x
        return grad

    def hess(self, x):
        hessian = 1/2*(self.Q+self.Q.T)
        return hessian
```

Tracé de la fonction à étudier

In [4]:
```
def plotobjective(f):
    """
```

```
    Function to draw contour plot of objective function.

    Parameters
    ----------
    f : objective function (given as a class with __call__, grad
        and hess methods)

    Returns
    -------
    fig : same as fig, ax returned by plt.subplots
    ax  : same as fig, ax returned by plt.subplots

    Example
    -------
    objfun = Rosenbrock(100) # choisir la fonction
    fig1, ax1 = plotobjective(objfun)
    """
    N = 100
    x = np.linspace(-1.5, 1.5, N)
    y = np.linspace(-1.5, 1.5, N)
    X, Y = np.meshgrid(x, y)
    z = f(np.vstack((X.ravel(), Y.ravel())))

    fig, ax = plt.subplots(figsize=(6, 6), num=1, clear=True)
    ax.contour(X, Y, z.reshape(N, N), 20, linestyles='dashed', linewidths=0.5)
    return fig, ax
```

Algorithmes de gradient et de Newton

Ecrire un algorithme de descente de gradient et un algorithme de Newton. On testera différentes méthodes de choix du pas (fixe, Armijo,...). Prévoir une procédure de recherche de pas par la règle d'Armijo.

In [5]:
```
def armijo(x, d, objfun, alpha=1/4, beta=0.9, maxiter=1000):
    """
    Armijo's linesearch

    Parameters
    ----------
    x : array_like. Current point.
    d : array_like. Search direction.
    objfun : objective function (given as a class with __call__, grad
             and hess methods)
    alpha : float (optional, default = 1/4)
    beta : float (optional, default = 0.9)
    maxiter : int (optional, default = 1000)

    Returns
    -------
    t : float. Stepsize.
    """
    t = 1
    fx = objfun(x)
    al = alpha*objfun.grad(x).T@d
    for i in range(maxiter):
        if objfun(x+t*d) < fx + t*al:
            break
        t = beta*t
    return t
```

In [6]:
```
@dataclass
class AlgParam:
    meth: str = field(repr=True)
```

```
    steprule: str = field(repr=True)
    stopcrit: str = field(repr=True, default="nostopcrit")
    maxiter: int = 100
    stepsize: float = 1e-3
    alpha: float = 0.48
    beta: float = 0.9
    prec: float = 1e-3


def descentalg(objfun, xini, params):
    """
    Unconstrained descent algorithm (gradient or Newton)

    Parameters
    ----------
    objfun : objective function (given as a class with __call__, grad
             and hess methods)
    xini : array_like: initial point
    params : AlgParam class with descent algorithm parameters

    Returns
    -------
    c_seq : list. List of criterions values.
    x_set : list. List of successive points generated by algorithm.
    """
    x_seq = []
    d_seq = []
    c_seq = []
    x_cur = xini
    x_seq.append(x_cur)
    c_seq.append(objfun(x_cur))
    for iter in range(params.maxiter):
        # ### search direction
        g = objfun.grad(x_cur)
        if params.meth == "grad":
            d = -g
        elif params.meth == "newt":
            H = objfun.hess(x_cur)
            d = -np.linalg.solve(H, g)
        # ### stopping criterion
        if params.stopcrit == "normgrad":
            if np.linalg.norm(g) < params.prec:
                break
        elif params.stopcrit == "newtdecr":
            sqnewtdecr = np.sqrt(d.T@H@d)
            if sqnewtdecr < 2*params.prec:
                break
        elif params.stopcrit == "nostopcrit":
            pass
        # ### linesearch/step
        if params.steprule == 'armijo':
            step = armijo(x_cur, d, objfun,
                          alpha=params.alpha, beta=params.beta)
        elif params.steprule == 'fixed':
            step = params.stepsize
        x_cur = x_cur + step * d
        # ### store all sequence
        d_seq.append(d)
        x_seq.append(x_cur)
        c_seq.append(objfun(x_cur))
    return c_seq, x_seq
```

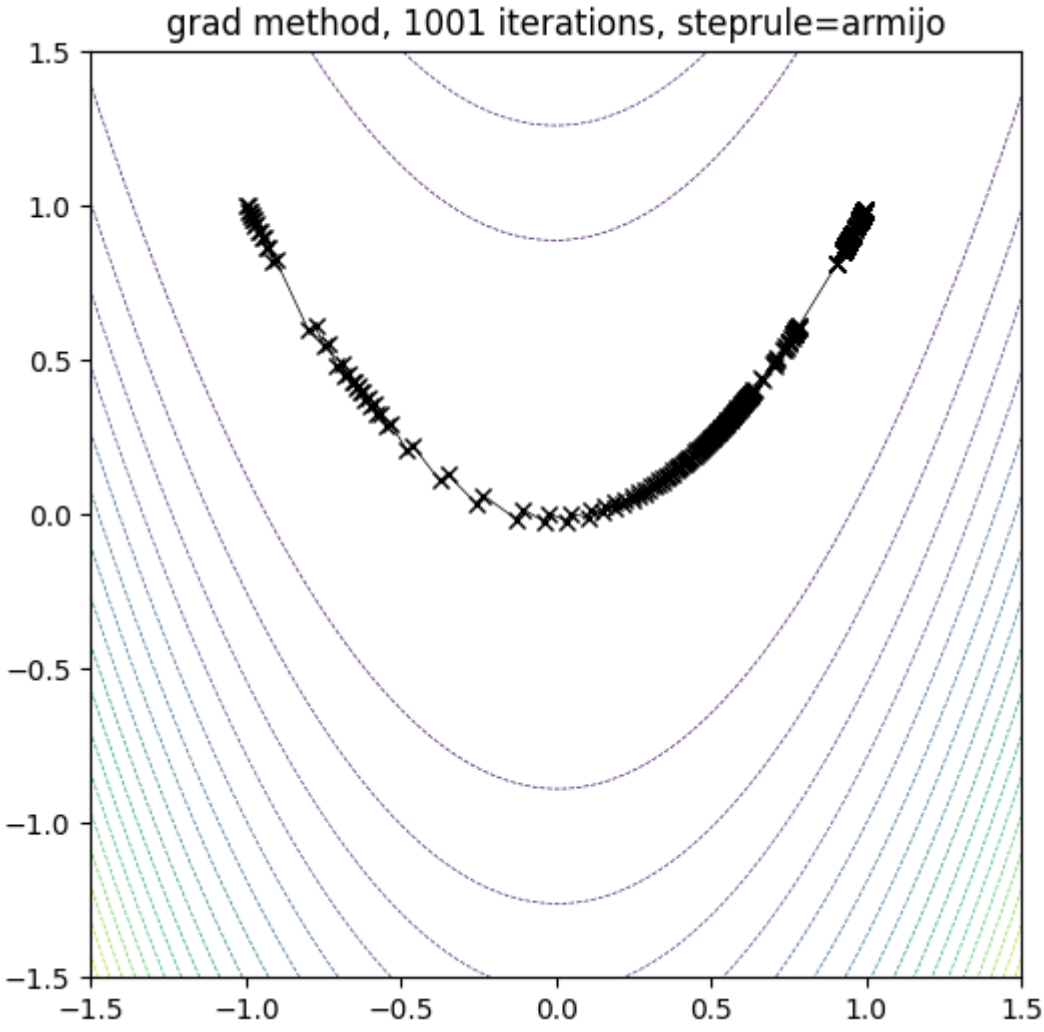
Test algorithme de gradient

Penser à tester un pas fixe et le pas par la méthode d'Armijo avec différentes valeurs.

In [7]:
```
# Choix fonction objectif: déjà fait et point d'initialisation des algorithmes.
objfun = Rosenbrock()
# objfun = Quadfunc(np.array([[1, 0],[0, 10]]))

xini = np.array([[-1], [1]]) # point initial
```

In [8]:
```
myparams = AlgParam(meth='grad',
                    steprule='armijo', stepsize=1e-2,
                    alpha = 0.48, beta=0.9,
                    stopcrit='normgrad', maxiter=1000)

c1, x1 = descentalg(objfun, xini, myparams)
x1 = np.asarray(x1)

fig1, ax1 = plotobjective(objfun)
ax1.plot(x1[:, 0], x1[:, 1], marker='x', color='black', linewidth=0.5)
ax1.set_title(myparams.meth+' method, '+str(len(c1))+' iterations, steprule='+myparam
print(f'Final value {c1[-1]}')
```

```
Final value [1.73433208e-05]
```



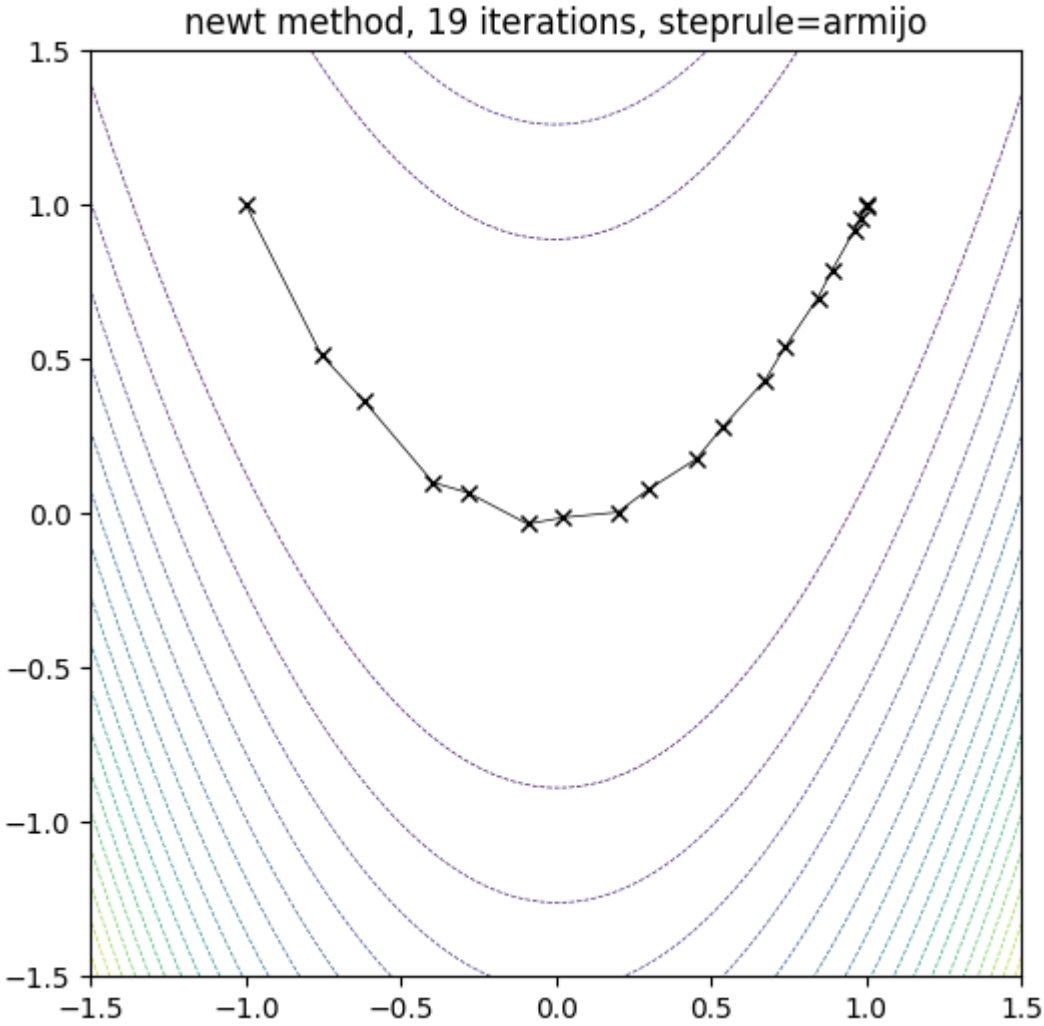
Test algorithme de Newton

Penser à tester un pas fixe et le pas par la méthode d'Armijo avec différentes valeurs.

In [9]:
```
myparams = AlgParam(meth='newt',
                    steprule='armijo', stepsize=1e-2,
                    stopcrit='newtdecr',
                    maxiter=2000)

c2, x2 = descentalg(objfun, xini, myparams)
```

```
x2 = np.asarray(x2)

fig2, ax2 = plotobjective(objfun)
ax2.plot(x2[:, 0], x2[:, 1], marker='x', color='black', linewidth=0.5)
ax2.set_title(myparams.meth+' method, '+str(len(c2))+' iterations, steprule='+myparam
print(f'Final value {c2[-1]}')
```

```
Final value [1.096387e-08]
```



In [ ]:

Compléments possibles

In [ ]:

```
class Quartfunc(object):
    """
    Class for function x1**4 + alpha x2**4

    Attributes
    ----------
    alpha : float, default = 1000

    Methods
    -------
    __call__ : computes function value at x
    grad(x) : computes gradient at x
    hess(x) : computes Hessian at x
    """
    def __init__(self, alpha=1):
        self.alpha = alpha

    def __call__(self, x):
        y = x[0]**4 + self.alpha*x[1]**4
        return y
```

```
    def grad(self, x):
        grad = np.c_[(4*x[0]**3,
                      self.alpha*4*x[1]**3)].T
        return grad

    def hess(self, x):
        hessian = np.empty([2, 2])
        hessian[0, 0] = 12*x[0]**2
        hessian[0, 1] = 0
        hessian[1, 0] = hessian[0, 1]
        hessian[1, 1] = self.alpha*12*x[1]**2
        return hessian
```