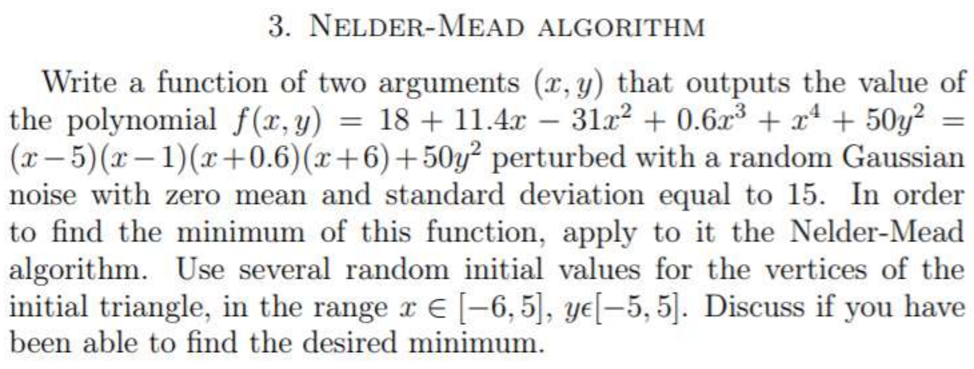
S_Assignment_3 Report


3. Nelder-Mead algorithm

Write a function of two arguments (x, y) that outputs the value of the polynomial $f(x,y) = 18 + 11.4x - 31x^2 + 0.6x^3 + x^4 + 50y^2 = (x-5)(x-1)(x+0.6)(x+6) + 50y^2$ perturbed with a random Gaussian noise with zero mean and standard deviation equal to 15. In order to find the minimum of this function, apply to it the Nelder-Mead algorithm. Use several random initial values for the vertices of the initial triangle, in the range $x \in [-6,5]$, $y\epsilon[-5,5]$. Discuss if you have been able to find the desired minimum.

Part 1: Code

The problem has been solved using **Python** programming language:

```
# We import the following modules to implement operations in the algorithm
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.collections  as mc

# Function used by the algorithm
def f(x):
    return 18 + 11.4*x[0] - 31*(x[0]**2) + 0.6*(x[0]**3) + x[0]**4 + 50*(x[1]**2) +
random.gauss(0,15) # Add Gaussian noise with Mean=0 and StdDev=15

# Function used to plot
def f2(x1,x2):
    return 18 + 11.4*x1 - 31*(x1**2) + 0.6*(x1**3) + x1**4 + 50*(x2**2)

# Methods defined to plot the final/intermediate result
def set_plot_parameters():
    levels = np.arange(-400,1100,100)
    x = np.linspace(-8,8,1000)
    y = np.linspace(-7.0,7.0,1000)
    X, Y = np.meshgrid(x,y)
    Z = f2(X,Y)
    return X, Y, Z, levels

def contour_plot(X, Y, Z, levels):
    _, ax = plt.subplots(figsize=(9,7))
    CS = plt.contour(X, Y, Z, levels=levels, cmap='ocean')
    ax.clabel(CS, inline=1, fontsize=8)
    return ax


# Method defined to update the simplex with the new found point
def update(simplex,res,point,score):
    # Update the simplex with new point
    simplex[2] = point
    # Substitute the new tuple (x,f(x)) in the score list
    res[2] = point, score
    return simplex, res
```

```
# Method that shrinks the surface of the simplex
def shrink(simplex,sigma):
    for i in [ 1, 2 ]:
        simplex[i] = simplex[0] + sigma * ( simplex[i] - simplex[0] )
    return simplex


def contract(simplex,res,contraction,worst_score):

    contraction_score = f(contraction)

    # Compare the contraction score with the worst score
    if contraction_score <= worst_score:
        simplex, res = update(simplex,res,contraction,contraction_score)

    else:
        # Perform shrinkrage
        simplex = shrink(simplex,0.5)

        # Substitute the new tuples (x,f(x)) in the score list
        res[1] = simplex[1],f(simplex[1])
        res[2] = simplex[2],f(simplex[2])

    return simplex, res


# Main method of the algorithm. Here it is possible to choose the maximum number of
# iterations, the maximum number of iterations without changes,
# the tollerance and if you want to plot or not the result
def nelder_mead(f, first_simplex, max_iter=1000, max_iter_no_change=10, toll=0.001,
plot=True):

    # Initialize first simplex
    simplex = first_simplex

    # Initialize number of iteration
    iters = 0
    iter_no_change = 0

    # Initialize best score list
    best_score_list = []

    # Generate tuples (X,f(X)) and sort the score list and the simplex on the scores
    res = [ (x,f(x)) for x in simplex ]
    res.sort(key=lambda w: w[1])
    simplex = [ x for x,fx in res ]

    # In order to not recompute meshgrids at each iteration
    if plot:
        X, Y, Z, levels = set_plot_parameters()

    # We repeat all calculations until the number of iterations is lower than the
    # maximum number of iterations
    # OR until the result doesn't change for a fixed number of
    # iterations(max_iter_no_change)
    while iters <= max_iter and iter_no_change < max_iter_no_change:
        iters += 1

        # Retrieve ordered tuples (X,f(X))
```

```
best, best_score = res[0] # first_Score
second_worst, second_score = res[1]
worst, worst_score = res[2] # last_Score


# Add best score to the best score list
best_score_list.append(best_score)

# Check if best_score has changed from the previous iteration more than toll
# (obviously this has to been checked after iteration 1)
if iters > 1:
    if np.abs( best_score_list[iters-1] - best_score_list[iters-2] ) <= toll:
        iter_no_change += 1
    else:
        iter_no_change = 0

# Compute the centroid
centroid = ( best + second_worst ) / 2

# Compute and evaluate the reflective point
reflection = centroid + ( centroid - worst )
reflection_score = f(reflection)

# Now we have some different cases (obviously one for every "if") based
# on the values of reflection_score and best_score:

# 1st case:
if reflection_score <= best_score:

    # Compute and evaluate the expanded point
    expansion = centroid + 2 * ( reflection - centroid )
    expansion_score = f(expansion)

    # Expansion
    if expansion_score <= reflection_score:
        simplex, res = update(simplex,res,expansion,expansion_score)

    # Reflection
    else:
        simplex, res = update(simplex,res,reflection,reflection_score)

# 2nd case: Reflection
elif best_score < reflection_score <= second_score:
    simplex, res = update(simplex,res,reflection,reflection_score)

# 3rd case: Contraction 1 or Shrink
elif second_score < reflection_score <= worst_score:

    # Compute and evaluate the contracted point
    contraction = centroid + ( reflection - centroid ) / 2
    simplex, res = contract(simplex,res,contraction,worst_score)

# 4th case: Contraction 2 or Shrink
else:
    # Compute and evaluate the contracted point
    contraction = centroid + ( worst - centroid ) / 2
    simplex, res = contract(simplex,res,contraction,worst_score)
```

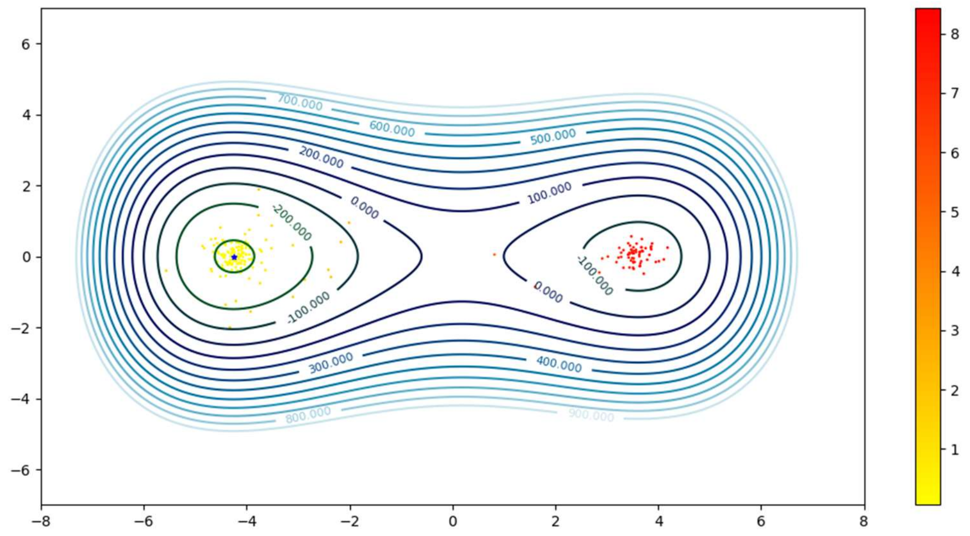
```
        # Sort the list (x,f(x)) and the simplex on the scores
        res.sort(key=lambda w: w[1])
        simplex = [ x for x,fx in res ]

        # With this part of code we plot the current simplex. It works only if
        # plot=True
        if plot:
            ax = contour_plot(X, Y, Z, levels)
            ax.set_title('Iteration '+str(iters)+": "+str(round(res[0][1],3)))
            lines = [ [simplex[0],simplex[1]], [simplex[1],simplex[2]],
            [simplex[2],simplex[0]] ]
            lc = mc.LineCollection(lines, colors='red', linewidths=1)
            ax.add_collection(lc)
            ax.autoscale()
            ax.margins(0.1)
            plt.pause(1/(iters*10)**2)
            plt.close()
            plt.show()


    # Plot final result
    if plot:
        X, Y, Z, levels = set_plot_parameters()
        ax = contour_plot(X, Y, Z, levels)
        ax.set_title("Minimum at:("+str(res[0][0][0])+"; "+str(res[0][0][1])+")")
        plt.scatter(res[0][0][0],res[0][0][1], c='blue',s=15, marker='*')
        plt.show()

    # At the end, we return the result
    return res[0]


# Here we run the algorithm (with a ‘good’ initial simplex)
x_min, score = nelder_mead(f, [np.array([0,4]), np.array([-2,5]), np.array([0,6])],
toll=1, plot=True)

# Random initial simplex in x ∈ [-6, 5], y ∈ [-5, 5]
x_min, score = nelder_mead(f, [np.array([gauss(0,6),gauss(0,5)]),
np.array([gauss(0,6),gauss(0,5)]),
np.array([gauss(0,5),gauss(0,5)])],toll=1,plot=False)

'''Bad case'''
x_min, score = nelder_mead(f, [np.array([1,1]), np.array([-1,-3]),
np.array([3,3])],toll=1,plot=True) #near local minimum

x_min, score = nelder_mead(f, [np.array([-4.5,0.5]), np.array([-4,0]),
np.array([2,3])], toll=1,plot=True) #too streched

x_min, score = nelder_mead(f, [np.array([0.35,0]), np.array([0.15, 0]),
np.array([0.25,3])],toll=1,plot=True) #too streached
```

Part 2: Summary of results

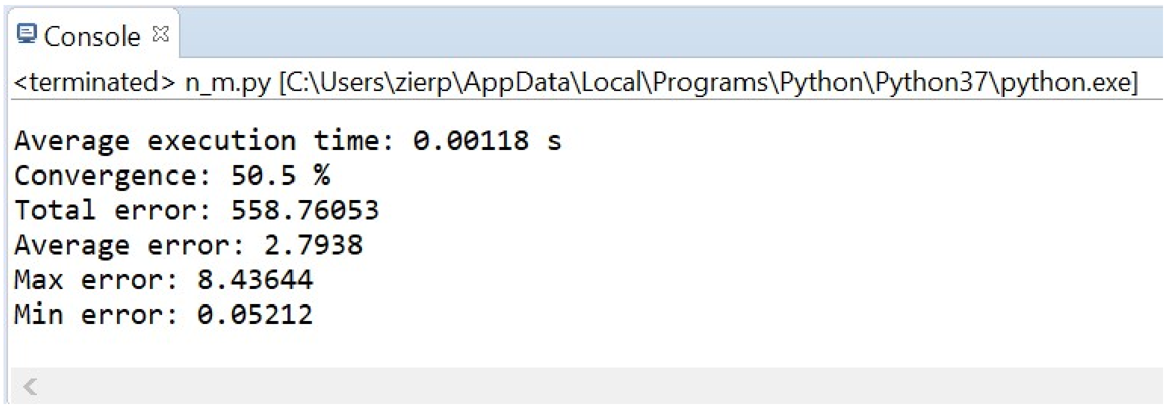
The Nelder-Mead algorithm is a non-derivative method, indeed it is based on function evaluations of few points at each iteration.
The function in analysis is **not convex**, so we expect that the **convergence is not guaranteed**.
First, we run **k=200** times the algorithm with the initial points of the simplex at each repetition that are randomly generated in this way:

```
x ∈ gauss(0,6) and y ∈ gauss(0,5)
```

The scatter plot of the results of the **k** repetitions is the following:



The blue point '*' represents the global minimum from which the *error* (Euclidian distance) is calculated. The output of the console is:



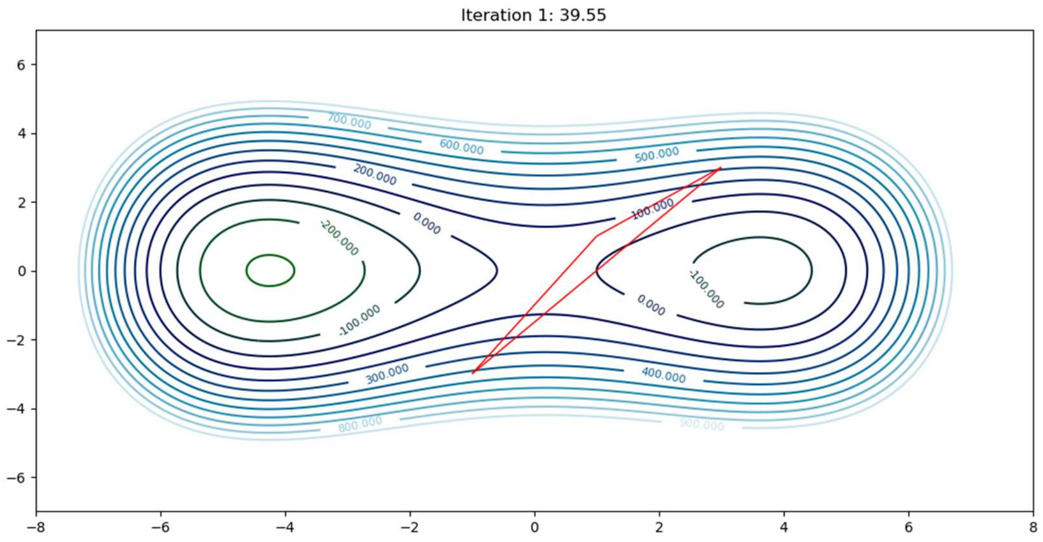
```
<terminated> n_m.py [C:\Users\zierp\AppData\Local\Programs\Python\Python37\python.exe]

Average execution time: 0.00118 s
Convergence: 50.5 %
Total error: 558.76053
Average error: 2.7938
Max error: 8.43644
Min error: 0.05212
```

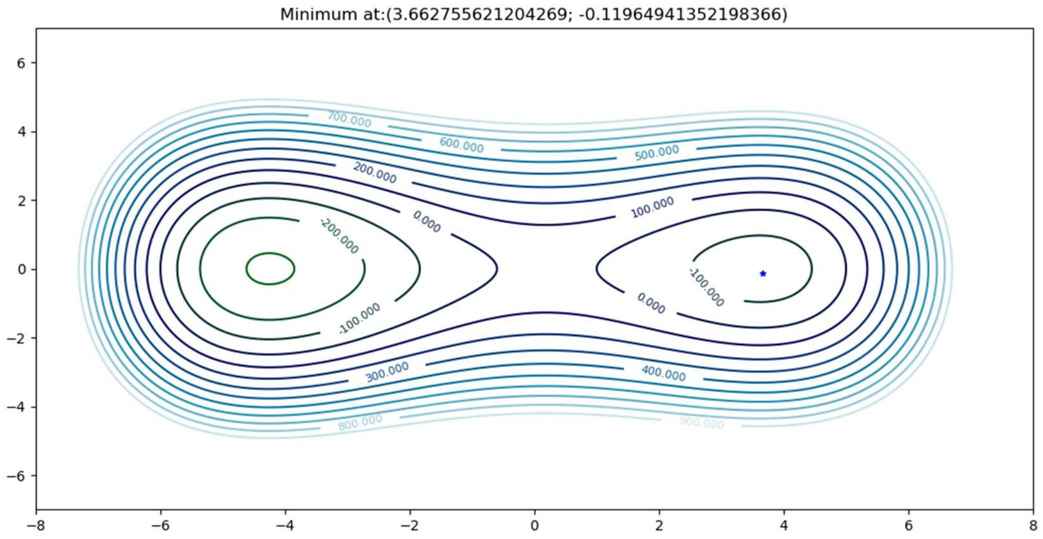
(Note: % of convergence is calculated using points with *error ≤ 0.5*):

As we expected the algorithm does not always converge: **let's inspect his behavior**.

➢ Bad starting simplex

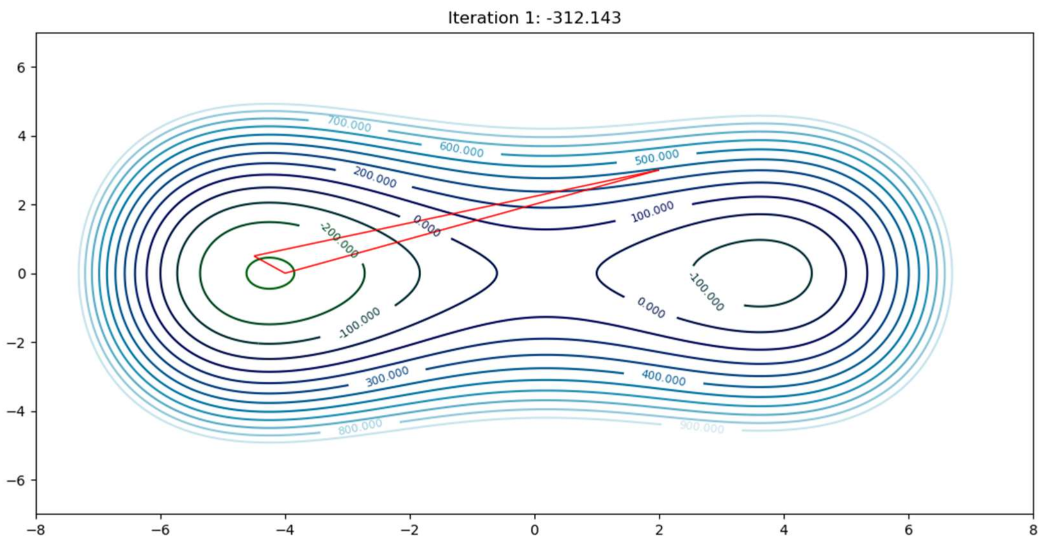
Choosing an initial simplex too near to a local minimum, for example a triangle made by points `[(1,1), (-1,-3), (3,3)]`, leads to the following result:

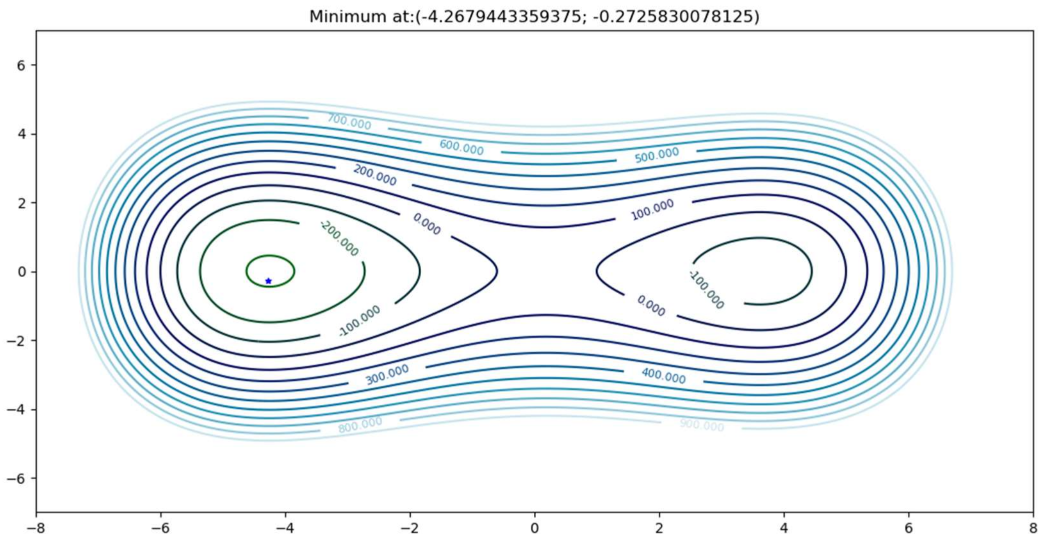




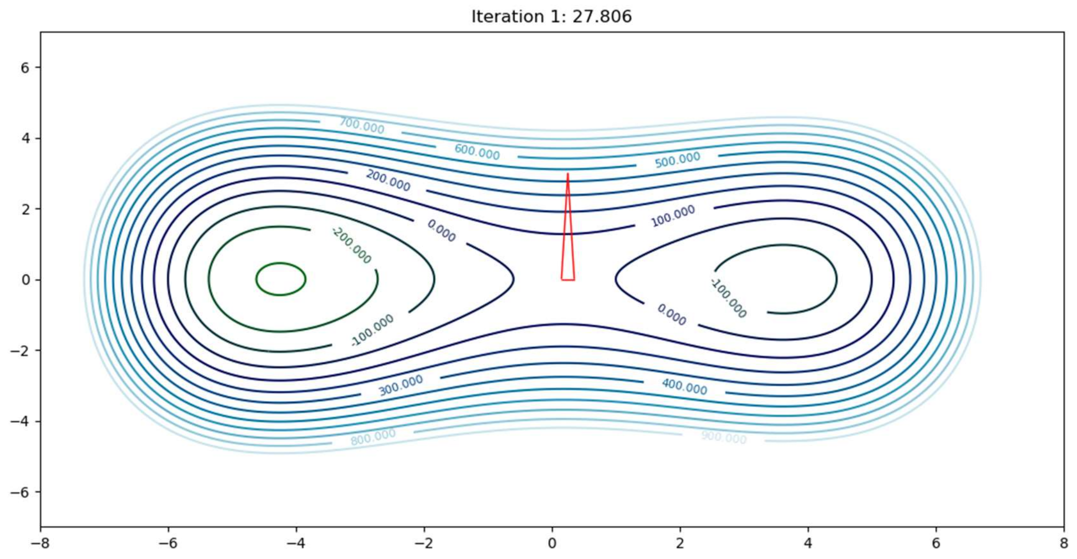
As we can see from the contours, the algorithm choose the *"wrong side"* (right side) of the function: it finds a local minimum **but not the global minimum**, that it is in the left side of the graph.

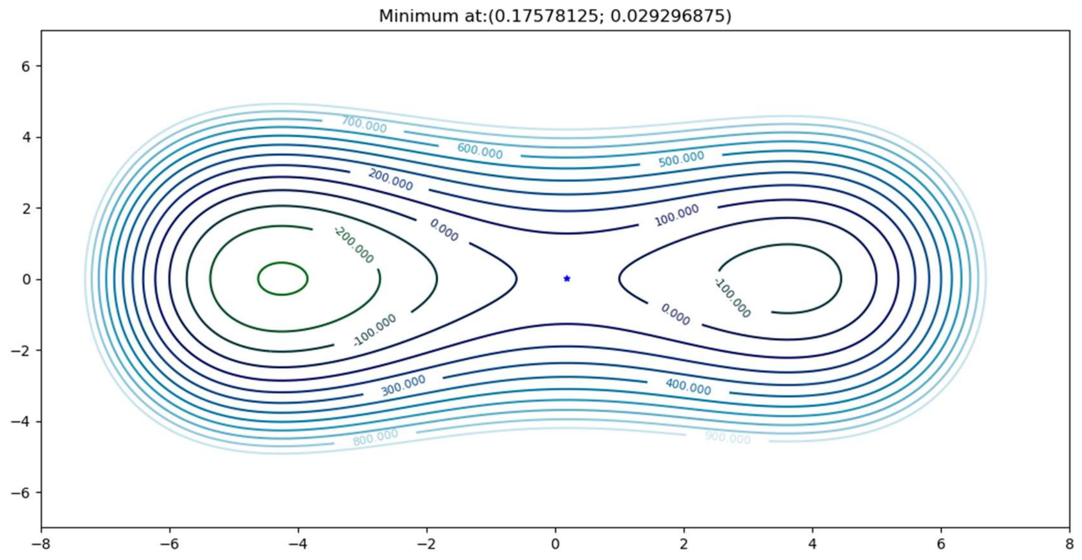
Choosing an initial simplex with some points near to the global minimum and others too far, making the simplex surface too "*streched*", for example `[(-4,-0.5), (-4,-0), (2,3)]`, leads to a solution **that it is not precise**, because the algorithm spends too much iterations without changing the actual minimum. Increasing the number of iterations may improve the result, leading to a better precision.





Choosing an initial simplex too *"small and/or streched"* into the saddle point, for example `[(0.35,0), (0.15,0), (0.25,3)]` can generate problems.
In particular, it can happen that the simplex cannot get out the saddle point and the algorithm finds the solution in the same spot of the initial simplex, i.e. into the *saddle point*). The plot of the final result is:



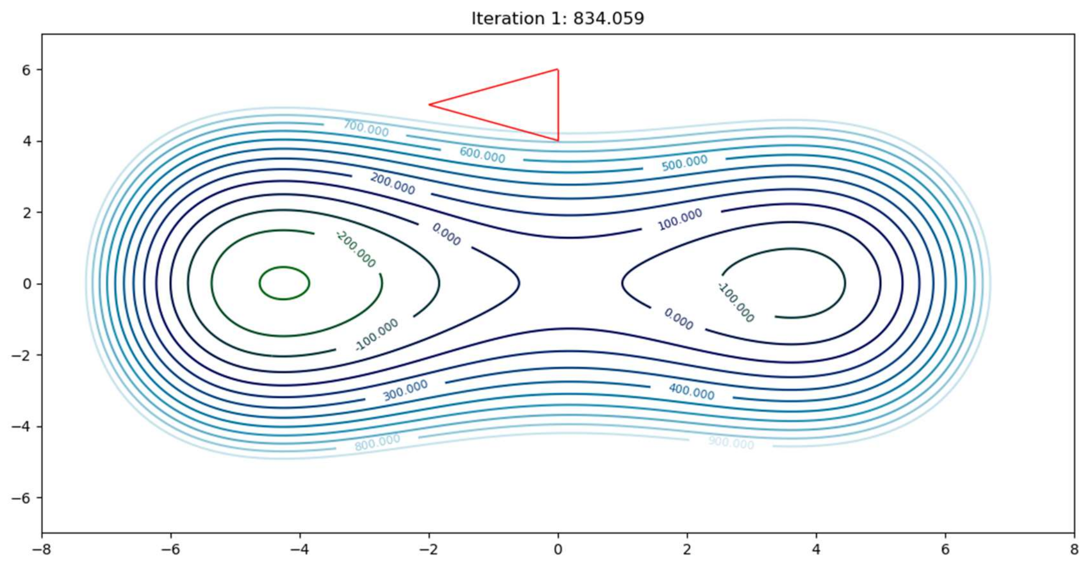


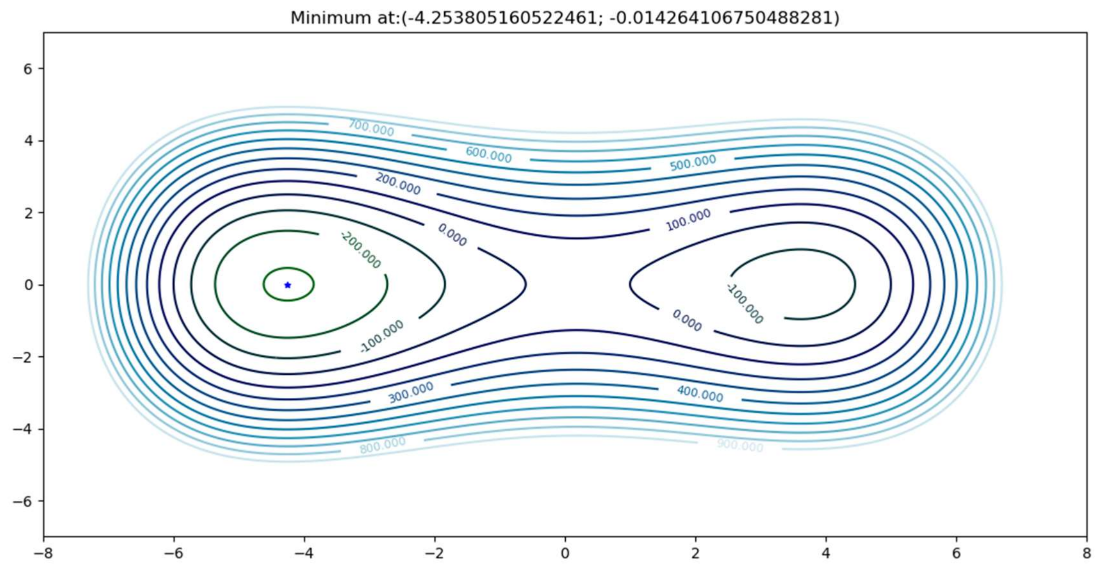
Summarizing, *in order to increase the probability of convergence*, the starting simplex **has not to be**:

- **into** the **saddle points**;
- **near** to **local minima**;
- with a **too streched shape**;
- with a too **small surface**.

➢ Good starting simplex (using algorithm in part 1)

The output of the algorithm, starting from an initial simplex from which the algorithm is likely to converge (i.e. `[(0,4), (-2,5), (0,6)]`) is the following:

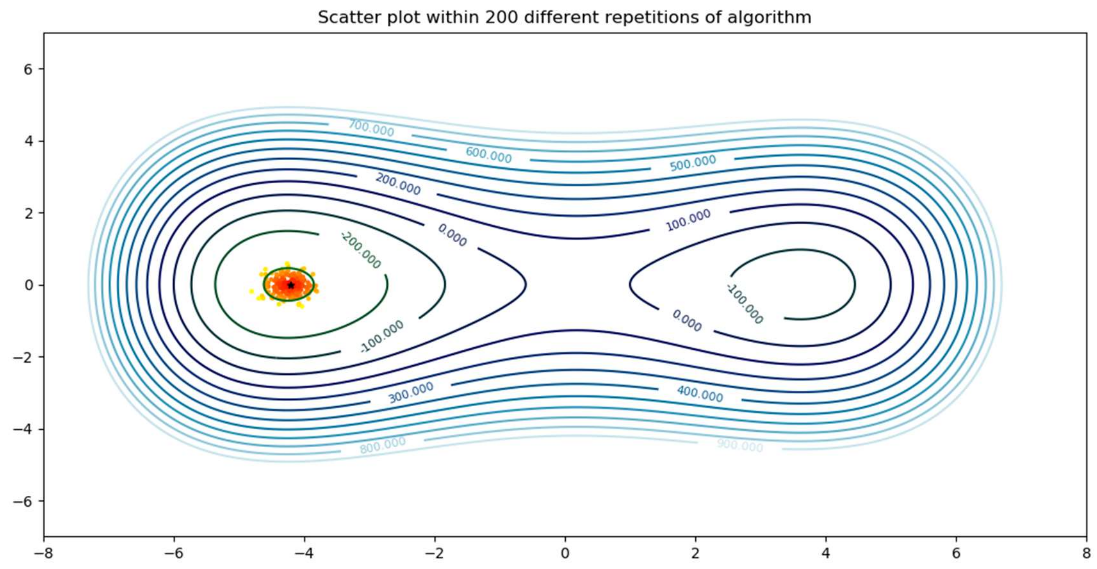




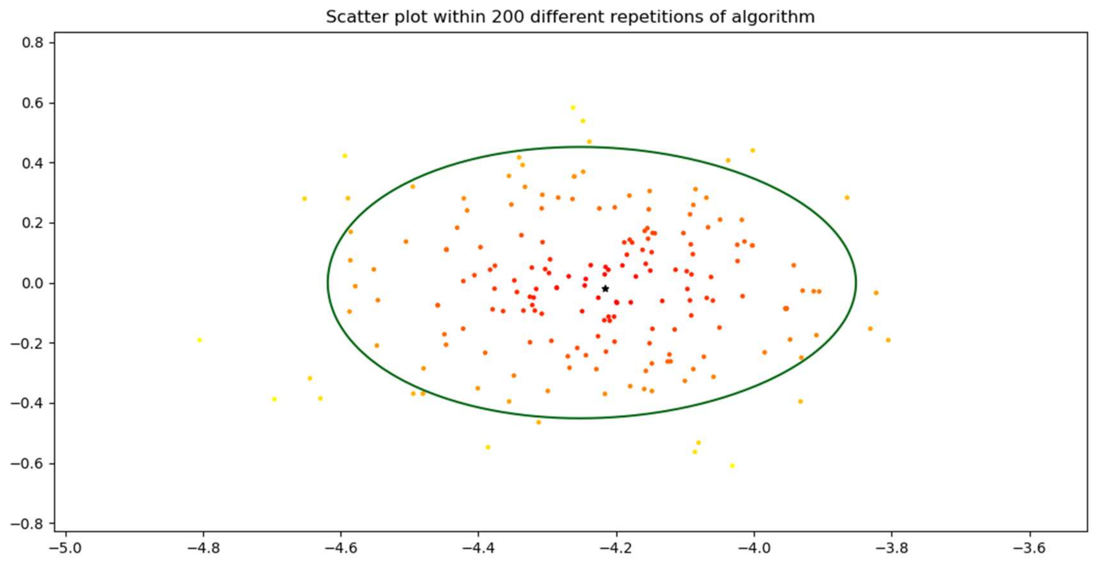
with minimum found in `(-4.254, -0.014)`.

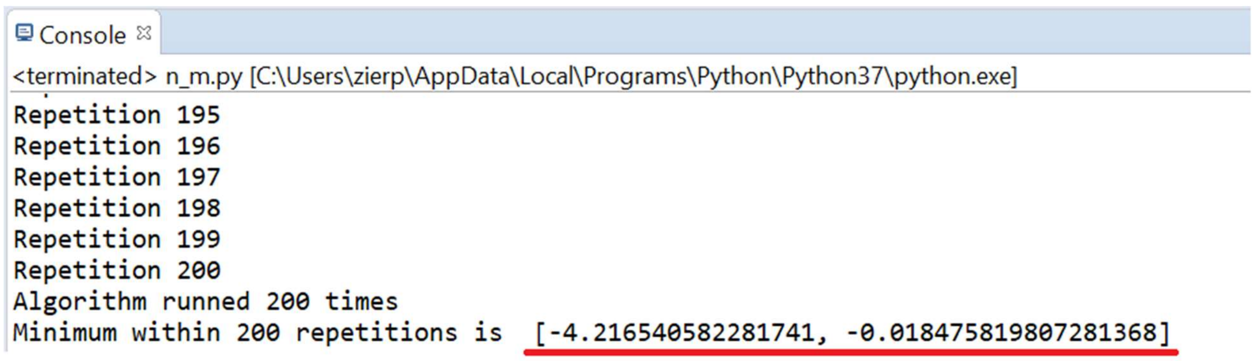
Obviously, even in this situation, due to the noise, the result of the algorithm **cannot be too much accurate**.

In order to try to get rid of the noise, we can run multiple times the algorithm from the "good" simplex and take the mean value of the found minima as **best result** (**best minimum**).
The result of these steps is the following:



zooming this plot, we can see:





```
<terminated> n_m.py [C:\Users\zierp\AppData\Local\Programs\Python\Python37\python.exe]
Repetition 195
Repetition 196
Repetition 197
Repetition 198
Repetition 199
Repetition 200
Algorithm runned 200 times
Minimum within 200 repetitions is  [-4.216540582281741, -0.018475819807281368]
```

However, the result reached after **200** repetitions of the Nelder-Mead algorithm is `(-4.216, -0.018)`, that is not very satisfyng, but we have to notice that the previous result was *lucky* with respect to these repetitions, that are often further from the real minimum.
Indeed, the error (**Euclidian distance**) of the centroid is `0.01456`, while the average error over the 200 repetitions is `0.26064`.

➢ Final considerations

Indeed, the Nelder-Mead algorithm is a **fast**, **simple** and **smart** method to perform minimization.
It is fast and simple because, being a non-derivative method, **doest not to perform heavy calculations**, actually, except in case of shrinking, it only has to generate few points and to evaluate the function at these points.
Moreover, it is smart because, even in **unfavorable situation**(pag.8), such as being too near to local minima, if the simplex is big enough it can manage to converge to the global minimum.
Its **sole problem** is to find an initial simplex from which it is likely to converge, *expecially in case of noise* that can mislead the algorithm.

Students
Cavagnero Niccolò
Mascarello Andrea
Serra Alessio