POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Matematica

Report NO4LSCP

Assignment on Unconstrained Optimisation

Chiodo Martina - 343310
Vigè Sophie - 339268

Anno Accademico 2024-2025

INTRODUCTION

This assignment aims to evaluate and compare the results of two numerical methods for unconstrained optimization applied to various problems. The method we decided to implement are the Nealder Mead method and the Modified Newton method. In particular for each problem we are going to run 11 experiments for both methods, the first one is run with a given initial point and the others with perturbations of it. These experiments will contribute to compute some statistics that will help us to compare the method, such as the minimum value found, the number of successful runs, the number of iterations needed by the method to converge and the experimental rate of convergence.

We can now proceed giving some details on how the algorithms have been implemented.

Nealder Mead Method

The Nelder-Mead Method is a derivative-free optimization technique that minimizes the objective function by evaluating it at the vertices of a simplex, which it uses to navigate the multidimensional space. The method employs four main operations: reflection, expansion, contraction, and shrinking, each of them is controlled by a parameter ρ for reflection, χ for expansion, γ for contraction, and σ for shrinking. These parameters are provided as inputs to the algorithm to optimize performance for each specific problem.

The initial simplex is constructed by perturbing the initial point, as we are not given an initial simplex. Specifically, the i^{th} vertex is generated by perturbing one component of the initial point, formulated as $x_i = x_0 + ae_i$. These perturbations are not fixed; for each vertex, we evaluate few values of a in order to minimize the objective function computed in x_i, aiming to get closer to the minimum point and obtaining a faster convergence to it.

The stopping criterion for the method is based on the tolerance of the current simplex size. Ideally, the simplex shrinks as it approaches the minimum, so once the simplex has significantly reduced in size, we consider the method to have converged.

This algorithm is heuristic, meaning that convergence to a global minimum is not guaranteed, as it may get stuck in a non-stationary point.

Modified Newton Method

The second method we have implemented is the Modified Newton Method, which leverages the gradient and the Hessian matrix of the function to move along descent directions at each iteration to reach the minimum.

This method faces two main challenges: performing an efficient line search at each iteration and ensuring the Hessian matrix is positive definite to compute the descent direction.

To address the first issue, we use a backtracking strategy for the line search. The parameters for the Armijo conditions and the maximum number of backtracking steps varies for each problem to optimize performance.

For the second issue, we regularize the Hessian matrix by adding a positive quantity to its diagonal until it becomes positive definite. We determine the positive definiteness by attempting a Cholesky factorization, which, if successful, is also used to solve the linear system to find the descent direction.

The stopping criterion is bases on the tolerance of the norm of the gradient.

Rate of Convergence

As said before, a way to evaluate the performance of a numerical method is to compute the rate of convergence. In some cases, this could not be possible because in its definition appears the minimum of the found which is not known a priori. Then what we will compute is the experimental rate of convergence which is defined as

$$q \approx \frac{\log\left(\frac{\|\hat{e}^{(k+1)}\|}{\|\hat{e}^{(k)}\|}\right)}{\log\left(\frac{\|\hat{e}^{(k)}\|}{\|\hat{e}^{(k-1)}\|}\right)} \quad \text{for } k \text{ large enough} \tag{1}$$

where $\hat{e}^{(k)} = x^{(k)} - x^{(k-1)}$ approximates the error at the k-th iteration (if the exact solution is unknown). This approximation is valid for k large enough, so it is possible that the value we compute may not be entirely reliable.

ROSENBROCK FUNCTION

We consider the Rosenbrock function for $\boldsymbol{x} \in \mathbb{R}^2$

$$f(\boldsymbol{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

with two different starting point $\boldsymbol{x}^{(0)} = (1.2, 1.2)$ and $\boldsymbol{x}^{(0)} = (-1.2, 1)$.
The global minima for the function is zero and the global minimum point is $\boldsymbol{x}^* = (1, 1)$, as shown in the following figure.

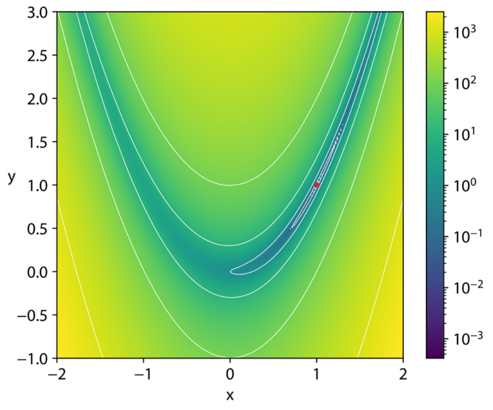


Figura 1: Rosenbrock function, top view.

For the Nelder-Mead method we use 400 as maximum number of iterations and the default parameters for reflection, expansion, contraction and shrinking; which are respectively

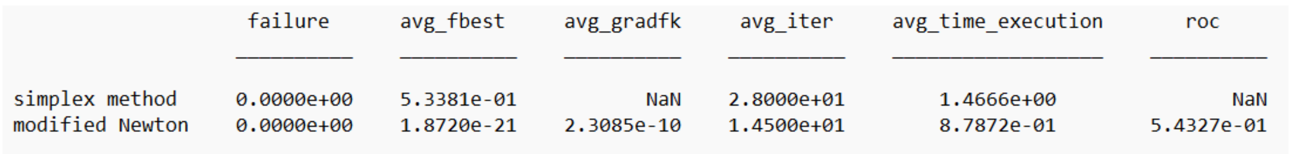
$$\begin{aligned} \rho &= 1 \\ \chi &= 2 \\ \gamma &= 0.5 \\ \sigma &= 0.5 \end{aligned}$$

For the Modified Newton method, 5000 maximum outer iterations, 100 maximum iterations allowed to compute tra matrix B_k at every step k. For backtracking we used the following values of the parameters

$$\begin{aligned} \rho &= 0.5 \\ c_1 &= 10^{-4} \\ \texttt{btmax} &= 40 \end{aligned}$$

where `btmax` is the maximum number of iterations allowed for backtracking and is chosen in such a way that stagnation is not allowed ($\rho^{\texttt{btmax}} \approx 10^{-13} > \epsilon_m$ where $\epsilon_m \approx 10^{-16}$ is the machine precision).
For both methods the tolerance used is 10^{-7}.

Computing the average for each method on the two runs, we obtain the following results:



```
                   failure     avg_fbest    avg_gradfk    avg_iter    avg_time_execution      roc
                 ----------   ----------   ----------   ----------   ------------------   ----------

simplex method   0.0000e+00   5.3381e-01          NaN   2.8000e+01           1.4666e+00          NaN
modified Newton  0.0000e+00   1.8720e-21   2.3085e-10   1.4500e+01           8.7872e-01   5.4327e-01
```

Figura 2: Results obtained by running both methods on the Rosenbrock function.

Both method were able to find a good approximation of the solution within the maximum number of iterations allowed. The Modified Newton method is in this case faster because it exploits the information contained in the gradient and the Hessian matrix, without the burden of costly operations that would occur in bigger dimension. However the rate of convergence is lower than the theoretical one. For the Nelder-Mead method it is `NaN` because this methods allows for consecutive iterations to have the same value of the current solution $x^{(k)}$, resulting in a division by zero when applying the formula (1).

PROBLEM 25

Model

The function described in this problem is the following

$$
\begin{aligned}
F(\mathbf{x}) &= \frac{1}{2}\sum_{k=1}^{n} f_k^2(x) \\
f_k(\mathbf{x}) &= 10\left(x_k^2 - x_{k+1}\right), \qquad && \bmod\ (k,2) = 1 \\
f_k(\mathbf{x}) &= x_{k-1} - 1, && \bmod\ (k,2) = 0
\end{aligned}
$$

where n denotes the dimensionality of the input vector $\mathbf{x}$. As convention, we set $x_{n+1} = x_1$ when it is necessary, that is when the dimensionality n is an odd number.
The starting point for the minimization is the vector $\mathbf{x}_0 = [-1.2, 1, -1.2, 1, \ldots]$.

In order to compute the derivatives of this problem we have to consider separately the cases when n is even or odd. In the first case, the gradient of the function is given by

$$
\begin{aligned}
\frac{\partial F}{\partial x_k}(\mathbf{x}) &= \frac{\partial}{\partial x_k}\left[\frac{1}{2}f_{k-1}^2(\mathbf{x})\right] = -100(x_{k-1}^2 - x_k) && \bmod\ (k,2) = 0 \\
\frac{\partial F}{\partial x_k}(\mathbf{x}) &= \frac{\partial}{\partial x_k}\left[\frac{1}{2}f_k^2(\mathbf{x}) + \frac{1}{2}f_{k+1}^2(\mathbf{x})\right] = 200x_k(x_k^2 - x_{k+1}) + (x_k - 1) && \bmod\ (k,2) = 1
\end{aligned}
$$

If the dimensionality n is odd, the only change is in the first component of the gradient, which becomes

$$
\frac{\partial F}{\partial x_1}(\mathbf{x}) = \frac{\partial}{\partial x_k}\left[\frac{1}{2}f_k^2(\mathbf{x}) + \frac{1}{2}f_{k+1}^2(\mathbf{x}) + \frac{1}{2}f_n^2(\mathbf{x})\right] = 200x_1(x_1^2 - x_2) + (x_1 - 1) - 100(x_n^2 - x_1)
$$

Looking at the structure of the problem we are considering, it is obvious that the Hessian matrix is a sparse matrix whose particular structure depends again on whether n is even or odd. In the first case, the Hessian is a block trigonal matrix with the following non-zero terms

$$
\begin{aligned}
&\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) = 100, && \frac{\partial^2 F}{\partial x_k \partial x_{k+1}}(\mathbf{x}) = 0, && \frac{\partial^2 F}{\partial x_k \partial x_{k-1}}(\mathbf{x}) = -200x_{k-1} && \bmod\ (k,2) = 0 \\
&\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) = 600x_k^2 - 200x_{k+1} + 1, && \frac{\partial^2 F}{\partial x_k \partial x_{k+1}}(\mathbf{x}) = -200x_k, && \frac{\partial^2 F}{\partial x_k \partial x_{k-1}}(\mathbf{x}) = 0 && \bmod\ (k,2) = 1
\end{aligned}
$$

If n is odd, there are two changes in the Hessian matrix: the derivative $\frac{\partial^2 F}{\partial x_1^2}(\mathbf{x})$ is affected by the presence of x_1 in the term $f_n()$ and the external diagonals are not zero anymore. We report the terms of the Hessian matrix that differs from the previous case

$$
\begin{aligned}
\frac{\partial^2 F}{\partial x_1^2}(\mathbf{x}) &= 600x_k^2 - 200x_{k+1} + 101 \\
\frac{\partial^2 F}{\partial x_n \partial x_1}(\mathbf{x}) &= \frac{\partial^2 F}{\partial x_1 \partial x_n}(\mathbf{x}) = -200x_n
\end{aligned}
$$

By analyzing the derivatives of the problem, we can deduce that the gradient of the function is nullified by the vector composed of ones which also nullifies the value of $F(\mathbf{x})$. As a matter of fact, we can notice that the terms $f_k()$ for an odd value of k are nullified only when $x_{k-1} = 1$, while the terms $f_k()$ for an even value of k are nullified when $1 = x_k^2 = x_{k+1}$. This leads to the conclusion that the function $F(\mathbf{x})$ assume is lower value, in other words is nullified, when for the vector $\mathbf{x} = \mathbb{1}$, which, indeed, is a global minimum.
To better understand the behavior of the function $F(\mathbf{x})$ in the neighborhood of the minimizer we have found, we now report a plot of the function in 2D for $n = 2$. From the plot, we can notice that the function is almost flat near the minimum; this may affect the performances of the algorithms implemented because flat regions may be the cause of slow convergence or even stagnation for optimization solvers.

Nealder Mead Method

We run the experiments with Nealder Mea method using the following parameters:

reflection $\rho = 1$ expansion $\chi = 2$ contraction $\gamma = 0.5$ shrinking $\sigma = 0.5$

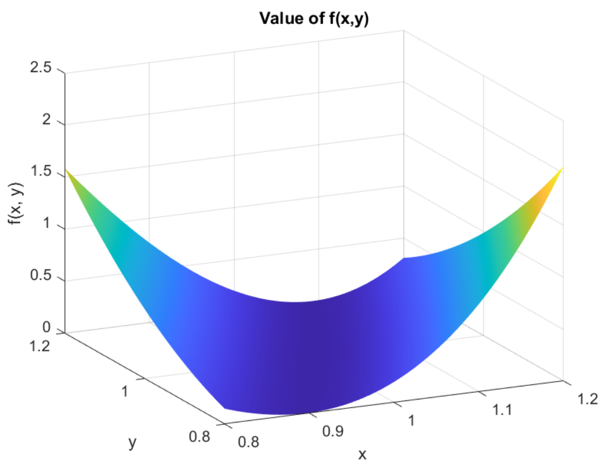
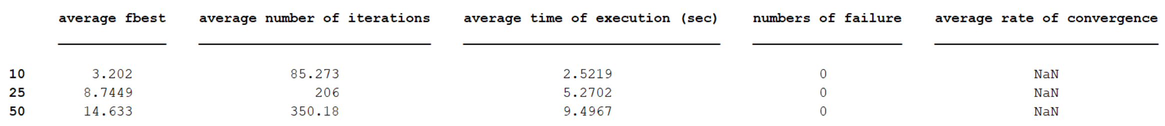


Figura 3: 2D plot of the function $F(x_1, x_2)$.

and fixing the tolerance to 10^{-4}.
For each dimensionality, $n \in \{10; 25; 50\}$, 11 experiments have been run, each of them starting from a different initial point; in particular, one of these initial point is the one declared at the beginning, the others have been obtained as perturbations of the first initial guess.

The table (4) shows some general results of these experiments, such as the value of the minimum point found, the average number of iterations used by the method, the average time of execution, the number of failures declared for each dimensionality and the average rate of convergence.



	average fbest	average number of iterations	average time of execution (sec)	numbers of failure	average rate of convergence
10	3.202	85.273	2.5219	0	NaN
25	8.7449	206	5.2702	0	NaN
50	14.633	350.18	9.4967	0	NaN

Figura 4: Results obtained by running the simplex method on the problem 25.

Looking at the table, it is clear that even if the method satisfies the stopping criterion for all dimensionalities it does not reach the point we declared be a global minimum, actually the minimum value the algorithm finds increases with the dimensionality. We are not too surprised by the por performance of the simplex method because the algorithm solely relies on function evaluation and does not take advantage of the information contained in the derivatives of the objective function and thus it is possible that it got stuck in the flat zone of the function.

The column that should contain the average rate of convergence is filled with `NaN` values. This occurs because the best value found by the method remains almost stationary in the last few iterations, usually during the shrinking phase. As a result, in the formula (1), we encounter an indeterminate form $0/0$ which leads to a `NaN` value.

Modified Newton Method - Exact Derivatives

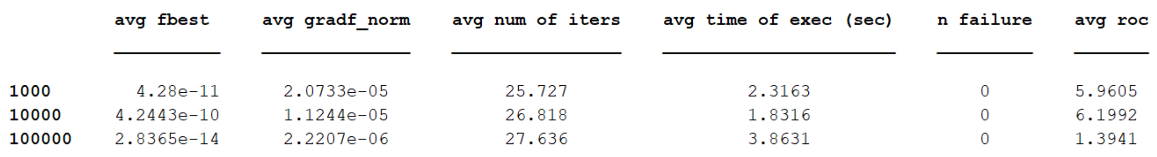
We can now proceed by applying the Modified Newton Method to the problem we are considering using the exact derivatives we computed before. As already said, it is important to store the Hessian matrix in a sparse structure because we are running experiments for dimensionality that vary in the set $\{10^3, 10^4, 10^5\}$. As we have already done previously, for each dimension we run 11 experiments, the first one with the declared initial guess and the others with perturbations of it.

For each dimensionality we have used the same set of parameters, which are:

$$\text{tolerance } tol = 10^{-4} \quad \rho = 0.5 \quad c_1 = 10^{-4} \quad \text{maximum number of backtracking step } = 48$$

The table (5) is analogous to the previous one and shows some general results obtained by running the Modified Newton method on the problem.

As expected from the theoretical background we have about these methods, the Modified Newton method performs significantly better than the Nealder Mead method. The table shows that the method converges to a point in which the norm of the gradient is below the fixed tolerance for all dimensionalities tested, and the



	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	4.28e-11	2.0733e-05	25.727	2.3163	0	5.9605
10000	4.2443e-10	1.1244e-05	26.818	1.8316	0	6.1992
100000	2.8365e-14	2.2207e-06	27.636	3.8631	0	1.3941

Figura 5: Results obtained by running the Modified Newton method on the problem 25 using the exact derivatives.

minimum value found is consistently close to zero. This is because the Modified Newton method uses the gradient and Hessian information, allowing it to make more informed steps towards the minimum and thus to converge in fewer iterations.

However, we can notice that the ratio between the average time of execution and the average number of iterations is smaller for the simplex method. This means that each iteration performed by the Modified Newton method is more high-performance but also more costly in terms of computational effort.

Modified Newton Method - Approximated Derivatives

Approximating the derivatives of the function $F(\mathbf{x})$ using finite differences is more challenging than it appears due to potential numerical cancellation issues, which can occur when subtracting two nearly equal quantities. Additionally, we aim to derive a formula that minimizes computational cost.

As done previously we will first consider the case in which the dimensionality n is an even integer and then we will specify what changes if n is an odd number.

Let's begin by approximating the first order derivatives by using the centered finite difference formula with increment h_k. We keep track of the subscript k in order to derive formula which are valid both for the case with constant increments and the case in which the increments depend on the components respect to which we are differentiating. The general formula is

$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{F(\mathbf{x}+h_k\vec{e}_k) - F(\mathbf{x}-h_k\vec{e}_k)}{2h_k} = \frac{\sum_{i=1}^n f_i(\mathbf{x}+h_k\vec{e}_k)^2 - \sum_{i=1}^n f_i(\mathbf{x}-h_k\vec{e}_k)^2}{4h_k}$$

but it would not be much wise to apply it directly to out problem because it would be unnecessary to evaluate all the terms $f_i^2(\mathbf{x})$ which are not affected by the variation of the k-th component of the vector $\mathbf{x}$. In particular, we can notice that if we are differentiating with respect to an even component the only term we need to compute is $f_{k-1}^2()$, while if we are differentiating with respect to an odd component we only need to expand the terms $f_k^2()$ and $f_{k+1}^2()$. Omitting the calculus, we obtain the following formula

$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{f_{k-1}^2(\mathbf{x}+h_k\vec{e}_k) - f_{k-1}^2(\mathbf{x}-h_k\vec{e}_k)}{4h_k} = \frac{-40h_k(10x_{k-1}^2 - 10x_k)}{4h_k} \qquad \bmod(k,2)=0$$

$$\begin{aligned}\frac{\partial F}{\partial x_k}(\mathbf{x}) &\approx \frac{f_k^2(\mathbf{x}+h_k\vec{e}_k) - f_k^2(\mathbf{x}-h_k\vec{e}_k) + f_{k+1}^2(\mathbf{x}+h_k\vec{e}_k) - f_{k+1}^2(\mathbf{x}-h_k\vec{e}_k)}{4h_k} \\ &= \frac{80x_kh_k(10x_k^2+10h_k^2-10x_{k+1}) - 4h_k(x_k-1)}{4h_k} \qquad \bmod(k,2)=1\end{aligned}$$

If n is an odd number, the approximation of $\frac{\partial F}{\partial x_1}(\mathbf{x})$ will slightly change into

$$\begin{aligned}\frac{\partial F}{\partial x_1}(\mathbf{x}) &\approx \frac{f_1^2(\mathbf{x}+h_1\vec{e}_1) - f_1^2(\mathbf{x}-h_1\vec{e}_1) + f_2^2(\mathbf{x}+h_1\vec{e}_1) - f_2^2(\mathbf{x}-h_1\vec{e}_1) + f_n^2(\mathbf{x}+h_1\vec{e}_1) - f_n^2(\mathbf{x}-h_1\vec{e}_1)}{4h_1} \\ &= \frac{80x_1h_1(10x_1^2+10h_1^2-10x_2) - 4h_1(x_1-1) - 40h_1(10x_n^2-10x_1)}{4h_1}\end{aligned}$$

For what concerns the second order derivatives, we can apply a similar reasoning based on neglecting the terms $f_i^2(\mathbf{x})$ which are not affected by the variation of the k-th component of $\mathbf{x}$ but starting from the general formula

$$\frac{\partial^2 F}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{F(\mathbf{x}+h_i\vec{e}_i+h_j\vec{e}_j) - F(\mathbf{x}+h_i\vec{e}_i) - F(\mathbf{x}-h_j\vec{e}_j) + F(\mathbf{x})}{h_ih_j}$$

Due to the particular structure of the problem we are considering, many second order derivatives are zero, thus we are going to approximate solely the ones we know are not null.

$$\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) \approx \frac{f_{k-1}^2(\mathbf{x}+2h_k\vec{e}_k) - 2f_{k-1}^2(\mathbf{x}+h_k\vec{e}_k) + f_{k-1}(\mathbf{x})}{2h_k^2} = \frac{200h_k^2}{2h_k^2}, \qquad \bmod (k,2) = 0$$

$$\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) \approx \frac{f_k^2(\mathbf{x}+2h_k\vec{e}_k) - 2f_k^2(\mathbf{x}+h_k\vec{e}_k) + f_k(\mathbf{x}) + f_{k+1}^2(\mathbf{x}+2h_k\vec{e}_k) - 2f_{k+1}^2(\mathbf{x}+h_k\vec{e}_k) + f_{k+1}(\mathbf{x})}{2h_k^2}$$
$$= \frac{40h_k^2(10x_k^2 - 10x_{k+1}) + 1400h_k^4 + 2400h_k^3x_k + 800x_kh_k^2 + 2h_k^2}{2h_k^2}, \qquad \bmod (k,2) = 1$$

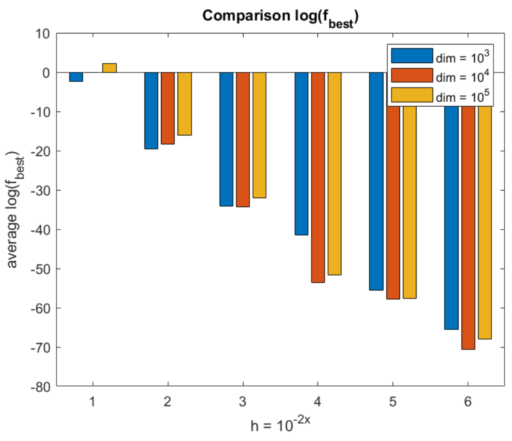
$$\frac{\partial^2 F}{\partial x_k \partial x_{k+1}}(\mathbf{x}) \approx \frac{f_k^2(\mathbf{x}+h_k\vec{e}_k + h_{k+1}\vec{e}_{k+1}) - f_k^2(\mathbf{x}+h_k\vec{e}_k) - f_k^2(\mathbf{x}+h_{k+1}\vec{e}_{k+1}) + f_k(\mathbf{x})}{2h_kh_{k+1}}$$
$$= \frac{20h_{k+1}(-10h_k^2 - 20h_kx_k)}{2h_kh_{k+1}}, \qquad \bmod (k,2) = 1$$

We explicitly approximated just the superior diagonal, the inferior one is obtained by imposing the symmetry of the hessian matrix.

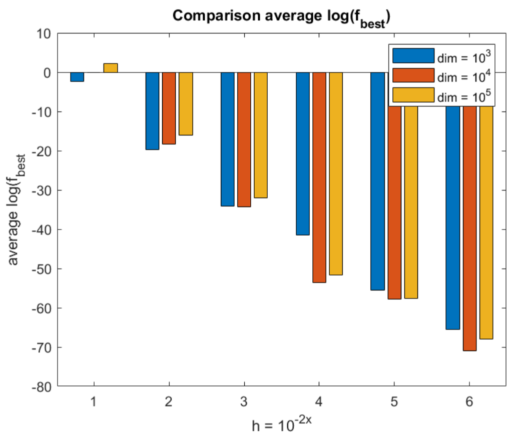
If the dimensionality n is an odd number, the changes only concern the term $\frac{\partial^2 F}{\partial x_1^2}(\mathbf{x})$ and the external diagonal which are not null anymore. In particular, these terms become

$$\frac{\partial^2 F}{\partial x_1^2}(\mathbf{x}) \approx \frac{40h_1^2(10x_1^2 - 10x_2) + 1400h_1^4 + 2400h_1^3x_1 + 800x_1h_1^2 + 202h_1^2}{2h_k1^2}$$
$$\frac{\partial^2 F}{\partial x_n \partial x_1} = \frac{\partial^2 F}{\partial x_1 \partial x_n} \approx \frac{20h_1(-20x_nh_n - 10h_n^2)}{2h_1h_n}$$

According to what we expect, seeking for the minimum using the approximations of the derivatives affects the performance of the Modified Newton method, especially for larger values of the increment h. In fact, from the theory, we know that the finite difference formula approximates the analytical derivative with an error that depends on the value of the increment h. Specifically, the error diminishes as h approaches 0.



(a) Costant Increment h



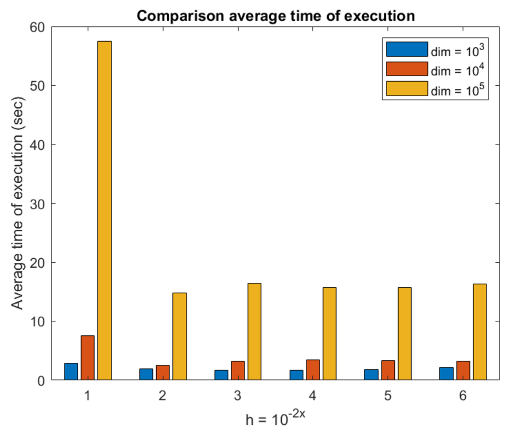
(b) Specific Increment

Figura 6: Values of the average $\log(f_{best})$ in function of the increment while running the Modified Newton Method with approximated derivatives on the problem 25.

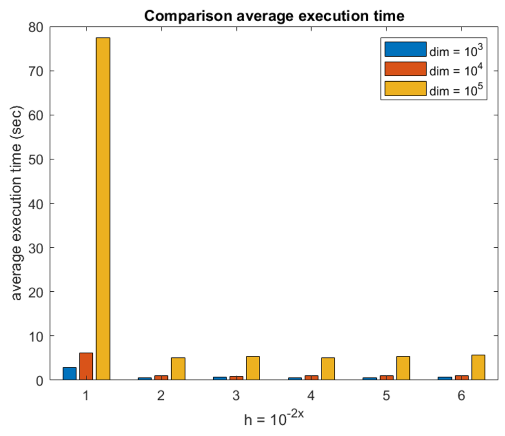
Therefore, it is not surprising that for $h = 10^{-2}$, the algorithm often converges to a point whose value is not very close to 0 and requires significantly more iterations to meet the stopping criterion. This is clearly shown in the following bar plots (Figure 6), which display the average value of $\log(f_{best})$, where f_{best} is the minimum found by the Modified Newton method, as a function of the increment h used to approximate the derivatives. Notice that we applied a logarithmic transformation to the value of f_{best} for clarity, as the values were close to 0. As we can see from the barplot, as the value of h diminish the minimum found is smaller (i.e. $\log(f_{best})$ increases in absolute value while being a negative quantity) as a consequence of the more precise approximations of the derivatives the Modified Newton method uses.

It can be interesting notice from the barplots comparing the average number of iterations needed by the Modified Newton method (Figure (8)) that a specific increment based on the point in which we are approximating the derivative seems to make the method perform better than using the constant increment.

Lastly we can also notice that the average rate of convergence, shown in the barplots (**??**)$25roc$), is still close to the expected value 2 and it is on average higher when the increment is constant for each component

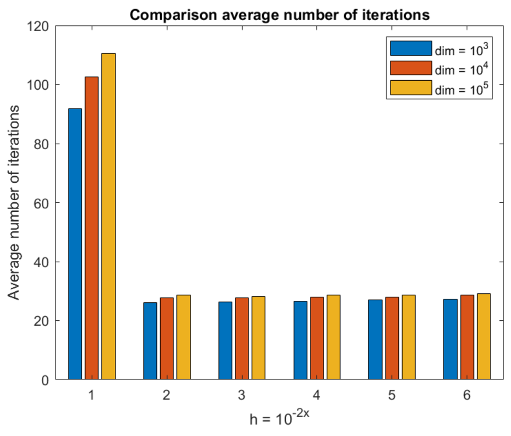


(a) Constant Increment h

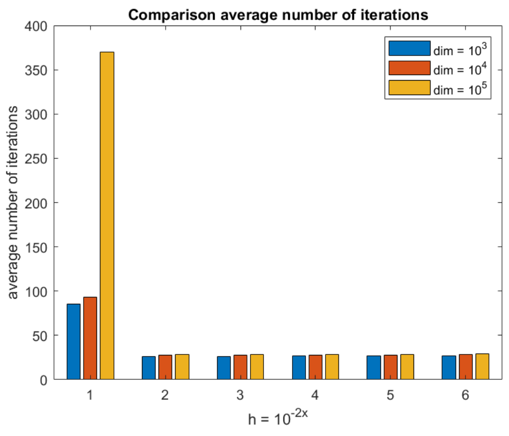


(b) Specific Increment

Figura 7: Average time of execution in function of the increment h while running the Modified Newton Method with approximated derivatives on the problem 25.

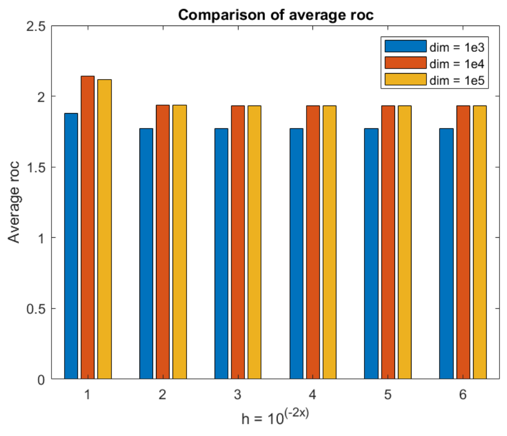


(a) Constant Increment h

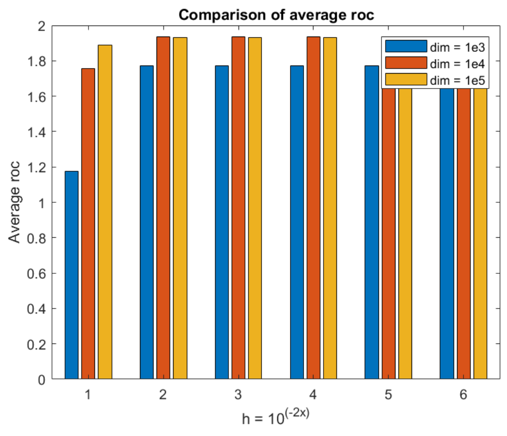


(b) Specific Increment

Figura 8: Average number of iterations in function of the increment h while running the Modified Newton Method with approximated derivatives on the problem 25.



(a) Constant Increment h



(b) Specific Increment

Figura 9: Average values of the experimental rate of convergence in function of the increment h while running the Modified Newton Method with approximated derivatives on the problem 25.

of the point in which we are approximating the derivatives. This means that all the efforts we put in to avoid numerical cancellation and to derive a more efficient formula for the finite differences have paid off. The results show that the Modified Newton method with approximated derivatives can still achieve a good performance, especially when using a specific increment based on the point in which we are approximating the derivative.

PROBLEM 75

Model

The function described in this problem is the following

$$
\begin{aligned}
F(\boldsymbol{x}) &= \frac{1}{2}\sum_{k=1}^{n} f_k^2(\boldsymbol{x}) \\
f_k(\boldsymbol{x}) &= x_k - 1, & k = 1 \\
f_k(\boldsymbol{x}) &= 10(k-1)(x_k - x_{k-1})^2, & 1 < k \leq n
\end{aligned}
$$

where n is the length of the input vector $\boldsymbol{x}$. With the given starting point for minimization being

$$\boldsymbol{x_0} = [-1.2, -1.2, ..., -1.2, -1]' \in \mathbb{R}^n.$$

The gradient of $F(\boldsymbol{x})$ is the following (note that, besides the first and last components, all the others have the same structure).

$$
\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial F}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial F}{\partial x_k}(\mathbf{x}) \\ \vdots \\ \frac{\partial F}{\partial x_n}(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1}\frac{1}{2}(f_1^2 + f_2^2)(\boldsymbol{x}) \\ \vdots \\ \frac{\partial F}{\partial x_k}\frac{1}{2}(f_k^2 + f_{k+1}^2)(\boldsymbol{x}) \\ \vdots \\ \frac{\partial F}{\partial x_n}\frac{1}{2}f_n^2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} x_1 - 1 - 200 \cdot (x_2 - x_1)^3 \\ \vdots \\ 200 \cdot \Big((k-1)^2(x_k - x_{k-1})^3 - k^2(x_{k+1} - x_k)^3\Big) \\ \vdots \\ 200 \cdot (n-1)^2(x_n - x_{n-1})^3 \end{bmatrix}
$$

The Hessian matrix of $F(\boldsymbol{x})$ is sparse since only on three diagonals elements different from zeros are present. They are the following:

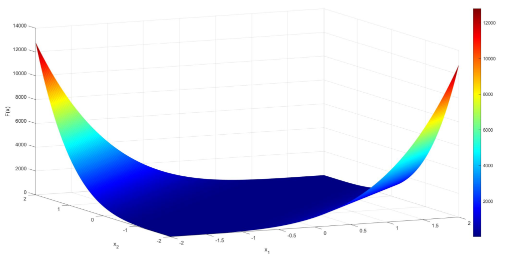
$$
\begin{aligned}
\frac{\partial^2 F}{\partial x_1^2}(\boldsymbol{x}) &= 1 + 600 \cdot (x_2 - x_1)^2 \\
\frac{\partial^2 F}{\partial x_k^2}(\boldsymbol{x}) &= 600 \cdot \Big((k-1)^2(x_k - x_{k-1})^2 + k^2(x_{k+1} - x_k)^2\Big), \quad 1 < k < n \\
\frac{\partial^2 F}{\partial x_n^2}(\boldsymbol{x}) &= 600 \cdot (n-1)^2(x_n - x_{n-1})^2 \\
\frac{\partial^2 F}{\partial x_k \partial x_{k-1}}(\boldsymbol{x}) &= -600 \cdot (k-1)^2(x_k - x_{k-1})^2, \quad 1 < k \leq n.
\end{aligned}
$$

It is easy to notice that F, being the sum of squared functions, is always non negative. Furthermore, $F(\boldsymbol{x}) = 0$ if and only if $\boldsymbol{x} = [1, 1, ..., 1]'$ since $f_1(\boldsymbol{x})^2 = 0$ if and only if $x_1 = 1$ and, for every $1 < k \leq n$, $f_k(\boldsymbol{x}) = 0$ if and only if $x_k = x_{k-1}$.

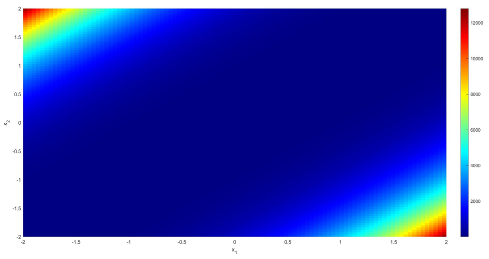
More formally, we can see that $\boldsymbol{x} = [1, 1, ..., 1]'$ solves the equation $\nabla F(\boldsymbol{x}) = \mathbf{0}$. Since F is convex (being the sum of convex functions) and differentiable, we know that any stationary point is a global minimum point for F.

Then $\boldsymbol{x} = [1, 1, ..., 1]'$ is the only global minimum point for F.

We plot the function for $n = 2$ in a neighborhood of the minimum point.



(a)



(b)

Figura 10: function $F(\boldsymbol{x})$ for $n = 2$

We can easily see that the central area, where the minimum point is located, is quite flat. This means that the minimization methods used might have some troubles when reaching this area because they might get stuck before reaching the minimizer.

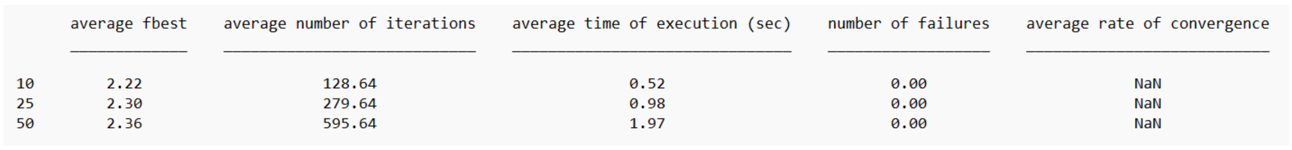
Nelder Mead Method

We runned the minimisation problem with Nelder Mead method using the following parameters:

$$\begin{aligned} \text{reflection } \rho &= 1.1 \\ \text{expansion } \chi &= 2.5 \\ \text{contraction } \gamma &= 0.6 \\ \text{shrinking } \sigma &= 0.5. \end{aligned}$$

The aim, with this choice, is to try to keep the simplex big enough so that the method will not get stuck too easily in the almost flat areas of the function's graph.

We now report a table summarizing the results obtained by running the Nelder-Mead method on the considered problem for dimensions n =10, 25, 50 and for a total of 11 starting points for each dimension, obtained as perturbations of the given one.



```
      average fbest    average number of iterations    average time of execution (sec)    number of failures    average rate of convergence
      _____________    ____________________________    _______________________________    __________________    ___________________________

10        2.22                    128.64                              0.52                       0.00                       NaN
25        2.30                    279.64                              0.98                       0.00                       NaN
50        2.36                    595.64                              1.97                       0.00                       NaN
```

Figura 11: Results obtained by running the Nelder Mead method on problem 75.

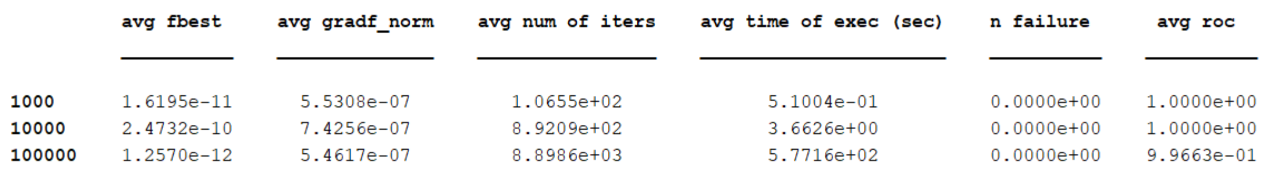
We notice that the method reports zero failures, which means that it never stopped because of the maximum number of iterations allowed (in this case $200 \cdot n$) had been reached. However, the best value of the function F that has been found is not so close to the expected value (which as observed before should be 0). The problem is in the starting point. In fact, even with the random perturbations, it always falls in the flat area of the function. Tuning the parameters in a way that encourages the expansion of the simplex's area is not enough to prevent the method from getting stuck here. It can be seen that, if we use as a staring point for example $[0, 0, ..., 0]'$, the results are a bit closer to the exact one (around 0.35).

The Nelder Mead method does not guarantee convergence and it is sensitive to the starting point and this becomes evident in this optimisation problem.

Concerning the rate of convergence, the fact that `Nan` is always reported is due to the construction of the method itself. In fact not necessarily at every iteration the best point of the current simplex is updated; expecially when contraction and shrinking phases are reached, it means that new promising points were not found, so it is quite likely that the current best point does not change among consecutive iterations. This is a problem when it comes to apply the formula for the experimental rate of convergence (1), since it leads the denominator to be 0.

Modified Newton Method - Exact Derivatives

As previously shown, we can easily compute the exact derivatives for the gradient and the Hessian matrix of $F(x)$. The Hessian should be stored as a sparse matrix due to its large dimension. We can then apply the Modified Newton Method to the considered problem, obtaining the following results.



```
          avg fbest     avg gradf_norm     avg num of iters     avg time of exec (sec)     n failure       avg roc
          __________    ______________     ________________     ______________________     __________     __________

1000      1.6195e-11      5.5308e-07          1.0655e+02              5.1004e-01           0.0000e+00     1.0000e+00
10000     2.4732e-10      7.4256e-07          8.9209e+02              3.6626e+00           0.0000e+00     1.0000e+00
100000    1.2570e-12      5.4617e-07          8.8986e+03              5.7716e+02           0.0000e+00     9.9663e-01
```

Figura 12: Results obtained by running the Modified Newton method on problem 75 using exact derivatives.

As expected, this method is performing much better due to the exploitation of the information contained in the gradient and the Hessian. In every tested dimension n it reaches the exact solution within the maximum number of iterations fixed for the corresponding dimension (in this case n has been used). For the others parameters the following values have been used:

- tolerance for the norm of the gradient: 10^{-6} for every dimension

- parameter $\rho \in (0,1)$ for the reduction of the steplength in backtracking: $\rho = \begin{cases} 0.4 & n = 10^3 \\ 0.3 & n = 10^4 \\ 0.4 & n = 10^5 \end{cases}$

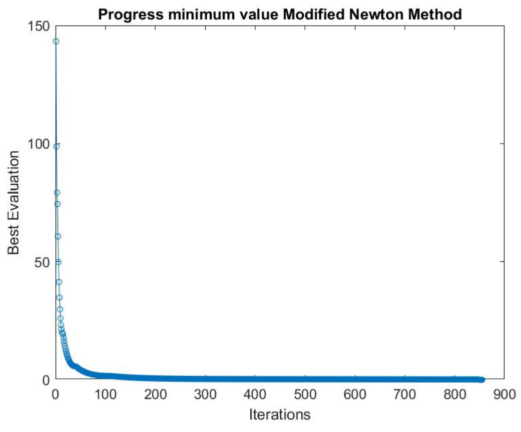
- parameter $c_1 \in (0,1)$ for the Armijo condition: $c_1 = \begin{cases} 10^{-4} & n = 10^3 \\ 10^{-4} & n = 10^4 \\ 10^{-3} & n = 10^5 \end{cases}$

- maximum number of backtracking steps allowed: `btmax` $= \begin{cases} 36 & n = 10^3 \\ 28 & n = 10^4 \\ 36 & n = 10^5 \end{cases}$

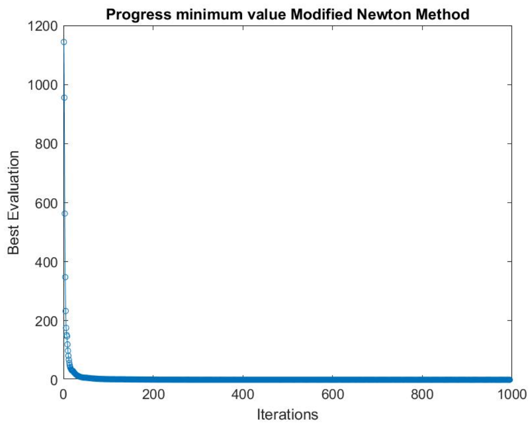
The values of ρ and `btmax` for every dimension has been chosen in such a way that stagnation is not allowed. In fact $\rho^{\texttt{btmax}} > \epsilon_m$, where the machine precision is $\epsilon_m \approx 10^{-16}$.

That experimental rate of convergence is approximately 1, so we are losing some of the strength of pure Newton method.

In the following figure we can see two examples of the progress of the minimum value of $F(x)$. Notice how



(a) $n = 10^3$



(b) $n = 10^4$

Figura 13: Example of convergence to zero of the value of $F(x)$.

fast it decreases in the first iterations, while the convergence becomes smaller when entering the almost flat area of the function.

Modified Newton Method - Approximated Derivatives

Let us now analyze what happens if we suppose not to be able to compute the exact derivatives of function F.

Using forward difference with step h_k (where h_k can either be constant or $h_k = h|\hat{x}_k|$ where $k = 1, ..., n$ and $\hat{x}$ is the point where the approximation is calculated), we can obtain an approximation of the gradient of F. We denote as $\vec{e_k} \in \mathbb{R}^n$ the k-th vector of the canonic basis. Note that we can exploit the structure of the function in order to avoid the evaluation of the whole F. This makes the evaluation much faster.

$$\begin{aligned} \frac{\partial F}{\partial x_k}(x) &\approx \frac{F(x + h_k\vec{e_k}) - F(x)}{h_k} = \frac{f_k^2(x + h_k\vec{e_k}) + f_{k+1}^2(x + h_k\vec{e_k}) - f_k^2(x) - f_{k+1}^2(x)}{2h_k} \quad 1 \le k < n \\ \frac{\partial F}{\partial x_n}(x) &\approx \frac{f_n^2(x + h_n\vec{e_n}) - f_n^2(x)}{2h_n}. \end{aligned}$$

The same reasoning can be applied to approximate the Hessian using the general formula

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \approx \frac{f(x + h_i\vec{e_i} + h_j\vec{e_j}) - f(x + h_i\vec{e_i}) - f(x + h_j\vec{e_j}) + f(x)}{h_i h_j}$$

and recalling that each f_k for $1 < k \leq n$ only depends on x_k ad x_{k-1}. We obtain the following

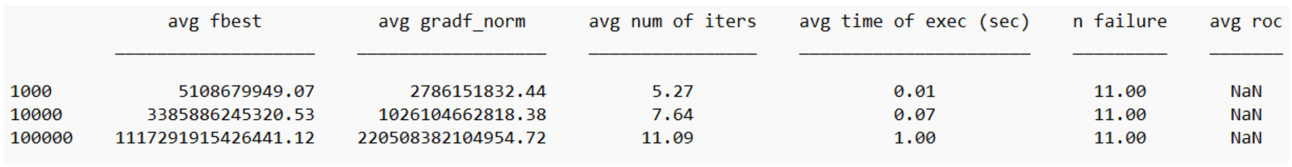
$$\begin{aligned}
\frac{\partial^2 F}{\partial x_k^2} &\approx \frac{f_k^2(x+2h_k\vec{e_k}) + f_{k+1}^2(x+2h_k\vec{e_k}) - 2f_k^2(x+h_k\vec{e_k}) - 2f_{k+1}^2(x+h_k\vec{e_k}) + f_k^2(x) + f_{k+1}^2(x)}{2h_k^2}, \quad 1 \leq k < n \\
\frac{\partial^2 F}{\partial x_n^2} &\approx \frac{f_n^2(x+2h_n\vec{e_n}) - 2f_n^2(x+h_n\vec{e_n} + f_n^2(x))}{2h_n^2} \\
\frac{\partial^2 F}{\partial x_k \partial x_{k-1}} &\approx \frac{f_k^2(x+h_k\vec{e_k}+h_{k-1}\vec{e_{k-1}}) + f_{k+1}^2(x+h_k\vec{e_k}) - f_k^2(x+h_k\vec{e_k}) - f_{k+1}^2(x+h_k\vec{e_k})}{2h_k h_{k-1}} + \\
&\quad + \frac{-f_k^2(x+h_{k-1}\vec{e_{k-1}}) - f_{k+1}^2(x) + f_k^2(x) + f_{k+1}^2(x)}{2h_k h_{k-1}}, \quad 1 < k < n \\
\frac{\partial^2 F}{\partial x_n \partial x_{n-1}} &\approx \frac{f_n^2(x+h_n\vec{e_n}+h_{n-1}\vec{e_{n-1}}) - f_n^2(x+h_n\vec{e_n}) - f_n^2(x+h_{n-1}\vec{e_{n-1}}) + f_n^2(x)}{2h_n h_{n-1}}
\end{aligned}$$

These are the only elements we have to compute to approximate the Hessian, since it is tridiagonal and we exploit its simmetry to avoid computing the elements of the upper diagonal.

Running the Modified Newton method using these approximations, with the parameters

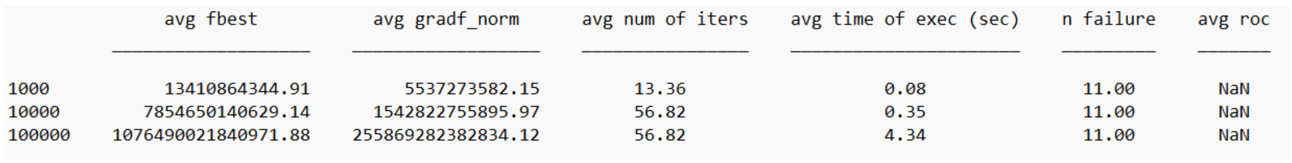
$$\begin{aligned}
\rho &= 0.8 \\
c_1 &= 10^{-5} \\
\texttt{btmax} &= 90
\end{aligned}$$

we obtain the following results:



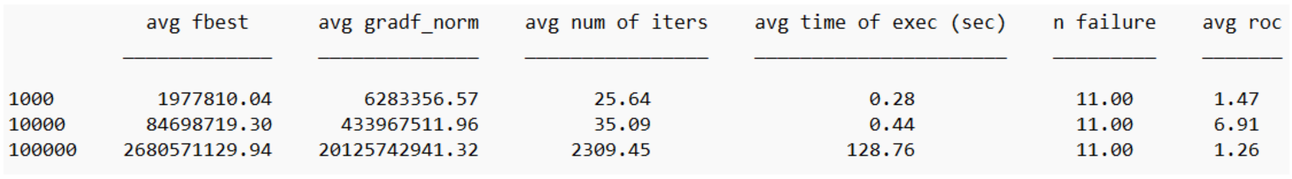
	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	5108679949.07	2786151832.44	5.27	0.01	11.00	NaN
10000	3385886245320.53	1026104662818.38	7.64	0.07	11.00	NaN
100000	1117291915426441.12	220508382104954.72	11.09	1.00	11.00	NaN

Figura 14: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-2}|\hat{x_i}|$.



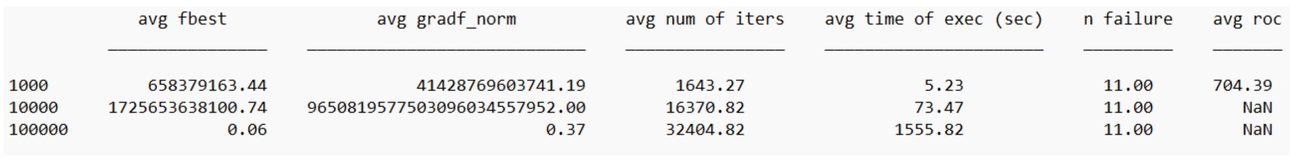
	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	13410864344.91	5537273582.15	13.36	0.08	11.00	NaN
10000	7854650140629.14	1542822755895.97	56.82	0.35	11.00	NaN
100000	1076490021840971.88	255869282382834.12	56.82	4.34	11.00	NaN

Figura 15: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-4}|\hat{x_i}|$.



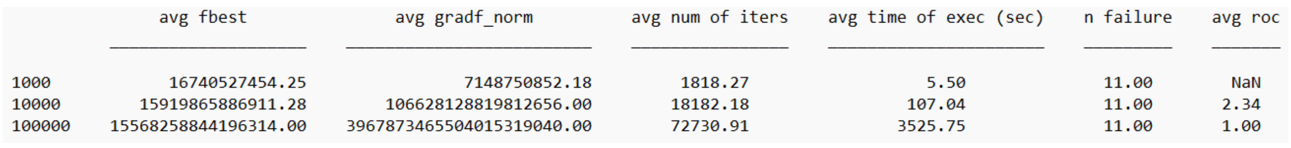
	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	1977810.04	6283356.57	25.64	0.28	11.00	1.47
10000	84698719.30	433967511.96	35.09	0.44	11.00	6.91
100000	2680571129.94	20125742941.32	2309.45	128.76	11.00	1.26

Figura 16: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-6}|\hat{x_i}|$.



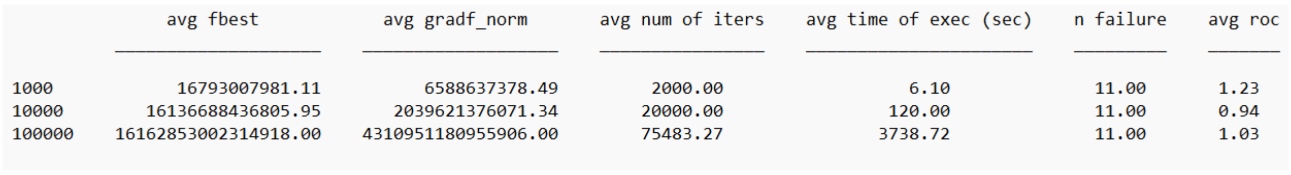
	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	658379163.44	41428769603741.19	1643.27	5.23	11.00	704.39
10000	1725653638100.74	9650819577503096034557952.00	16370.82	73.47	11.00	NaN
100000	0.06	0.37	32404.82	1555.82	11.00	NaN

Figura 17: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-8}|\hat{x_i}|$.



	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	16740527454.25	7148750852.18	1818.27	5.50	11.00	NaN
10000	15919865886911.28	106628128819812656.00	18182.18	107.04	11.00	2.34
100000	15568258844196314.00	3967873465504015319040.00	72730.91	3525.75	11.00	1.00

Figura 18: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-10}|\hat{x_i}|$.



	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	16793007981.11	6588637378.49	2000.00	6.10	11.00	1.23
10000	16136688436805.95	2039621376071.34	20000.00	120.00	11.00	0.94
100000	16162853002314918.00	4310951180955906.00	75483.27	3738.72	11.00	1.03

Figura 19: Results obtained by running the Modified Newton method on problem 75 using approximated derivatives with $h_i = 10^{-12}|\hat{x_i}|$.

The rate of convergence `NaN` that is sometimes shown is due to failures that occur before having performed the minimum number of iterations needed to compute it (which is three since we know the exact value of the solution).

We can identify two different kind of failures that occur: those due to backtracking and those due to the reach of the maximum number of iterations. In particular, for $h_i = 10^{-k}|\hat{x}_i|$ for $k = 2, 4, 6$ all the failures are because of backtracking (they can occur even for smaller h, but less frequently). This happens because, since h is too large, the approximation of the gradient and the Hessian are not good enough to guarantee that a good descent direction is found. For all the points, we obtain something similar to the following pathological case:

```
**** REL with h = 1e-06 FOR THE PB 75 (point 8, dimension 10000):  *****
Time:  0.37252 seconds
Backtracking parameters (rho, c1):  0.8 1e-05
**** MODIFIED NEWTON METHOD : RESULTS *****
************************************
f(xk):  39458126.9661
norma di gradf(xk):  414870519.4969
N. of Iterations:  18/20000
Rate of Convergence:  8.7683
************************************
FAIL
Failure due to backtracking
cos of the angle between the last computed direction and the gradient:  -1.6049e-05
************************************
```

where we can note that after few iterations, becktracking fails because the cosine of the angle between the computed descent direction and the gradient is so close to zero that this direction is almost perpendicular to the gradient and so tangent to the contour lines of F.

The second kind of failure occur for $h_i = 10^{-k}|\hat{x}_i|$ for $k = 8, 10, 12$. The best results are obtained with $k = 8$, because, with most starting points, when the method reaches the maximum number of iterations allowed, the current solution is quite close to zero. One example is the following:

```
**** REL with h = 1e-08 FOR THE PB 75 (point 10, dimension 100000):  *****
Time:  4376.7099 seconds
Backtracking parameters (rho, c1):  0.8 1e-05
**** MODIFIED NEWTON METHOD : RESULTS *****
************************************
f(xk):  0.0060851
norma di gradf(xk):  0.0012706
N. of Iterations:  80000/80000
Rate of Convergence:  0.058907
************************************
FAIL
Failure not due to backtracking
```

```
************************************
```

Still, the behaviour is very irregular and we can see it for example from the experimental rate of convergence, which is completely different from the theoretical one.

For $k = 10, 12$ we have very poor results: the method fails in most cases because of the maximum number of iterations, but the solution found is very far from the exact one. This is due to numerical cancellation that occurs when computing the approximations of the derivatives since we are subtracting two numbers very close to each other. So the approximations obtained are completely inaccurate and cause the method to perform this poorly. One example is the following:

```
**** REL with h = 1e-12 FOR THE PB 75 (point 8, dimension 100000):  *****
Time:  3679.4492 seconds
Backtracking parameters (rho, c1):  0.8 1e-05
**** MODIFIED NEWTON METHOD : RESULTS *****
************************************
f(xk):  1.756805324787388e+16
norma di gradf(xk):  698990881722798.6
N. of Iterations:  80000/80000
Rate of Convergence:  0.86925
************************************
FAIL
Failure not due to backtracking
************************************
```

When constant values of h are used, the behaviour is similar: for $h = 10^{-k}$, $k = 2, 4, 6$ we always have failures due to backtracking and for $h = 10^{-k}$, $k = 10, 12$ most failures occur because of the reach of the maximum number of iterations, producing a solution very far from the exact one.

For $h = 10^{-8}$ the behaviour is again the best, with no failures occurring for dimension $n = 10^3$ and $n = 10^4$ and some failures due to backtracking for $n = 10^5$, but happening when the method is close to the exact solution. We report here two examples of this:

```
**** COST with h = 1e-08 FOR THE PB 75 (point 1, dimension 1000):  *****
Time:  4.7991 seconds
Backtracking parameters (rho, c1):  0.8 1e-05
**** MODIFIED NEWTON METHOD : RESULTS *****
************************************
f(xk):  1.361e-05
norma di gradf(xk):  9.9781e-05
N. of Iterations:  1699/2000
Rate of Convergence:  0.9956
************************************
SUCCESS
************************************


**** COST with h = 1e-08 FOR THE PB 75 (point 11, dimension 100000):  *****
Time:  155.1714 seconds
Backtracking parameters (rho, c1):  0.8 1e-05
**** MODIFIED NEWTON METHOD : RESULTS *****
************************************
f(xk):  0.012298
norma di gradf(xk):  0.32072
N. of Iterations:  4748/80000
Rate of Convergence:  1.5507
************************************
FAIL
Failure due to backtracking
cos of the angle between last computed direction and the gradient:  -0.00022156
************************************
```

PROBLEM 76

Model

The function described in this problem is the following

$$F(\mathbf{x}) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

$$f_k(\mathbf{x}) = x_k - \frac{x_{k+1}^2}{10}, \quad 1 \le k < n$$

$$f_n(\mathbf{x}) = x_n - \frac{x_1^2}{10}$$

where n denotes the dimensionality of the input vector $\mathbf{x}$.
The starting point for the minimization is the vector $\mathbf{x}_0 = [2, 2, \dots, 2]$.

To be able to say something more about the behavior of the problem is useful to look at the gradient of the function $F(\mathbf{x})$ and at its Hessian matrix.

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial F}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial F}{\partial x_k}(\mathbf{x}) \\ \vdots \\ \frac{\partial F}{\partial x_n}(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1}\frac{1}{2}\left[f_n^2 + f_1^2\right](\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_k}\frac{1}{2}\left[f_{k-1}^2 + f_k^2\right](\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n}\frac{1}{2}\left[f_{n-1}^2 + f_n^2\right](\mathbf{x}) \end{bmatrix} = \begin{bmatrix} -\frac{x_1}{5}\left(x_n - \frac{x_1^2}{10}\right) + \left(x_1 - \frac{x_1^2}{10}\right) \\ \vdots \\ -\frac{x_k}{5}\left(x_{k-1} - \frac{x_k^2}{10}\right) + \left(x_k - \frac{x_{k+1}^2}{10}\right) \\ \vdots \\ -\frac{x_n}{5}\left(x_{n-1} - \frac{x_n^2}{10}\right) + \left(x_n - \frac{x_1^2}{10}\right) \end{bmatrix}$$

Due to the particular structure of the function, the Hessian matrix as a sparse structure, with only 3 diagonals different from zero. The non-zero elements are the following:

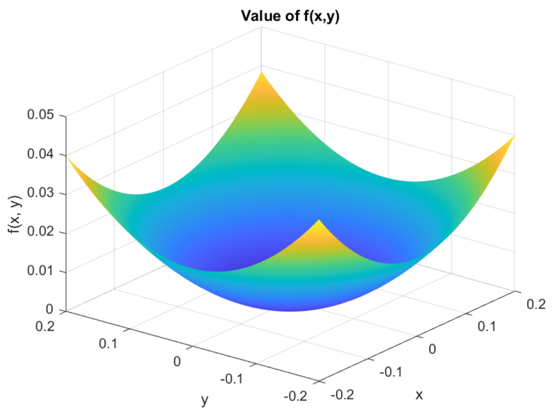
$$\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) = -\frac{1}{5}x_{k-1} - \frac{3}{50}x_k^2 + 1, \quad 1 < k \le n \qquad \frac{\partial^2 F}{\partial x_1^2}(\mathbf{x}) = -\frac{1}{5}x_n - \frac{3}{50}x_1^2 + 1,$$

$$\frac{\partial^2 F}{\partial x_k \partial x_{k+1}}(\mathbf{x}) = -\frac{1}{5}x_{k+1}, \quad 1 \le k < n \qquad \frac{\partial^2 F}{\partial x_n \partial x_1}(\mathbf{x}) = -\frac{1}{5}x_1$$

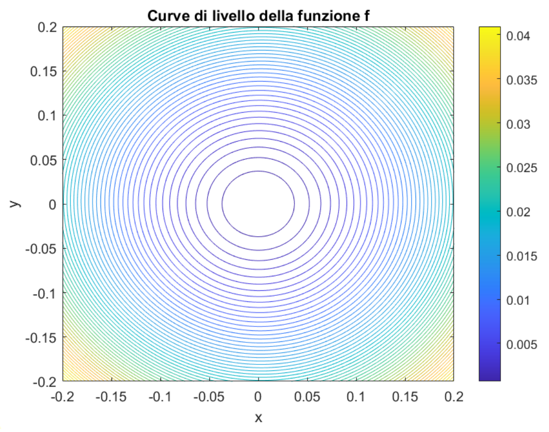
$$\frac{\partial^2 F}{\partial x_k \partial x_{k-1}}(\mathbf{x}) = -\frac{1}{5}x_k, \quad 1 < k \le n \qquad \frac{\partial^2 F}{\partial x_1 \partial x_n}(\mathbf{x}) = -\frac{1}{5}x_n$$

We can now easily notice that the gradient of the function is null when all the components of the vector $\mathbf{x}$ are equal to 0, in this case the Hessian matrix is positive definite, so the point $\mathbf{x} = \mathbf{0}$ is a minimum of the function $F(\mathbf{x})$. Because of the definition of the function, 0 is the lowest value the function can assume, so the minimum found is a global one.

We now report some plots of the function for $n = 2$ to better understand the behavior of it in a neighborhood of the minimum value.



(a)



(b)

Figura 20: function $F(\boldsymbol{x})$ for $n = 2$

«<s

Differently from the previous problems, the plots do not show flat regions in the neighborhood of the minim value for this problem; that is why we expect both method to well perform and easily found the minimum value in few iterations.

Nealder Mead Method

To optimize the function using the Nealder Mead method the following parameters have been fixed:

$$\text{reflection } \rho = 1 \quad \text{expansion } \chi = 2 \quad \text{contraction } \gamma = 0.5 \quad \text{shrinking } \sigma = 0.5$$

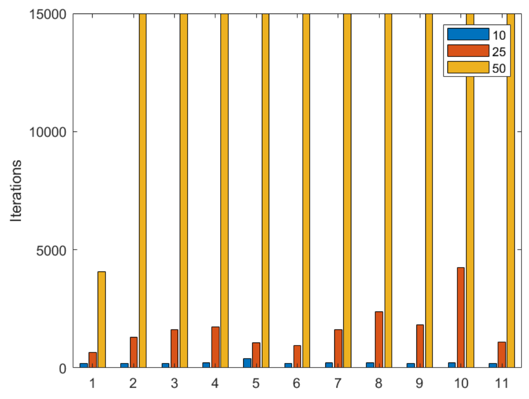
Table (21) contains some general results obtained by running the Nealder Mead method on the function $F(\mathbf{x})$.

	avg fbest	avg num of iters	avg time of exec (sec)	n failure	avg roc
10	5.555e-05	218.64	4.5116	0	NaN
25	4.1038e-05	1680.4	31.524	0	NaN
50	29.039	14007	269.17	10	NaN

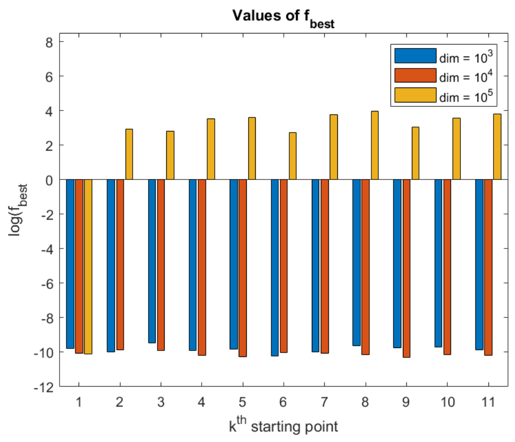
Figura 21: Results obtained by running the simplex method on the problem 76.

First thing we can notice is that for smaller dimensionalities the simplex method is able to find the minimum in a reasonable amount of time, but when the dimensionality becomes higher the method starts failing. From the plot in figure (22a), we can see that for most points belonging to $\mathbb{R}^{50}$, the method keeps iterating until the maximum number of iterations is reached without satisfying the stopping criterion. This behavior can probably be explained by the fact that when the dimensionality increases the starting point is more far from the minimum due to its definition, so the method would needs to perform more iterations to reach the minimum.

However, as expected due to the shape of the function, the algorithm consistently approximates the minimum point close to 0 (as we can see from the barplot (22b)), even for the only starting point in dimension 50 from which the method is able to converge.



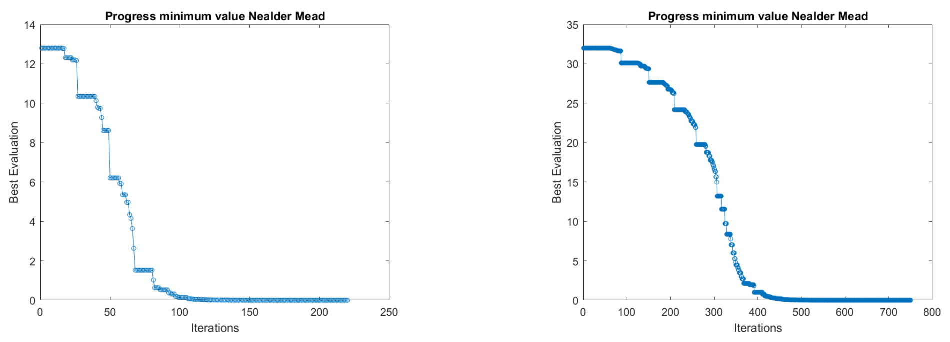
(a) Number of iterations used by the method for each starting point.



(b) $\log(f_{best})$ found by the method for each starting point.

Figura 22

From the previous table, we can notice that the experimental rate of convergence is always `Nan`: this is due to the fact that in the last iterations the value of $\mathbf{x}^{(k)}$ does not change much and thus it yields a division by zero in the formula (1) which defines the experimental rate of convergence. This can be seen in the plots (23), showing that, in the last iterations, the approximated value of the minimum seems to be stationary.



(a) dimension 10 (b) dimension 25

Figura 23: Plots of the progresses of the Nealder Mead method for different dimensionalities for the problem 76.

Modified Newton Method - Exact Derivatives

The experiments have been performed also using the Modified Newton method and the parameters have been fixed to the following values:

$$\rho = 0.4, \quad c_1 = 10^{-4}, \quad bt\_max = 36 \quad \text{for dimension } 10^3$$
$$\rho = 0.3, \quad c_1 = 10^{-4}, \quad bt\_max = 28 \quad \text{for dimension } 10^4$$
$$\rho = 0.4, \quad c_1 = 10^{-3}, \quad bt\_max = 36 \quad \text{for dimension } 10^5$$

Table (24) contains some general results obtained by running the Modified Newton method on the function $F(\mathbf{x})$. We obviously expect the method to perform better than the simplex method because of the exact derivatives used in the computation of the descent direction.

	avg fbest	avg gradf_norm	avg num of iters	avg time of exec (sec)	n failure	avg roc
1000	2.9818e-10	1.1915e-05	5.4545	0.028048	0	1.7721
10000	2.9521e-16	2.369e-08	4.9091	0.025717	0	1.9344
100000	3.2292e-15	7.6604e-08	5	0.24656	0	1.9326

Figura 24: Results obtained by running the Modified Newton Method on the problem 76 using the exact derivatives.

This time, the method always converges to the minimum point in very few iterations, even for higher dimensionalities. We can also appreciate the fact that the approximated rate of convergence is close to 2, as expected for a Newton method. Comparing this table with the previous one (showing the results obtained by running the simplex method), we can see that the Modified Newton method identifies as minimum a point in which the evaluation of the function is much smaller. This behavior aligns with theoretical expectations, as the Modified Newton method leverages the exact derivatives of the function $F(\mathbf{x})$ to determine the descent direction, while the simplex method depends only on function evaluations.

Modified Newton Method - Approximated Derivatives

Approximating the derivatives of the function $F(\mathbf{x})$ using finite differences is more challenging than it appears due to potential numerical cancellation issues, which can occur when subtracting two nearly equal quantities. Additionally, we aim to derive a formula that minimizes computational cost.

Let's begin by approximating the first-order derivatives of the function $F(\mathbf{x})$ using the centered finite difference formula with step h_k. The subscript k is specified because the following formula are valid both with a constant increment, $h_k = k$ for all $h = 1, \ldots, n$, and with a specific increment $h_k = h|\hat{x}_k|$ $k = 1, \ldots, n$, where $\hat{\mathbf{x}}$ is the point at which we approximate the derivatives.

$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{F(\mathbf{x} + h_k \vec{e}_k) - F(\mathbf{x} - h_k \vec{e}_k)}{2h_k} = \frac{\sum_{i=1}^{n} f_i(\mathbf{x} + h_k \vec{e}_k)^2 - \sum_{i=1}^{n} f_i(\mathbf{x} - h_k \vec{e}_k)^2}{4h_k}$$

We can observe that each term f_i^2 only depends on x_i and x_{i+1}, so $f_i(\mathbf{x} + h_k \vec{e}_k)^2 - f_i(\mathbf{x} - h_k \vec{e}_k)^2 = 0$ for all $i \neq k-1, k$ (or $i \neq 1, n$ if we are considering $k = 1$). This allows to simplify the formula, even in order to decrease the computational cost, as follows

$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{f_{k-1}(\mathbf{x} + h_k \vec{e}_k)^2 - f_{k-1}(\mathbf{x} - h_k \vec{e}_k)^2 + f_k(\mathbf{x} + h_k \vec{e}_k)^2 - f_k(\mathbf{x} - h_k \vec{e}_k)^2}{4h_k} \qquad 1 < k \leq n$$

$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{f_n(\mathbf{x}+h_k\vec{e}_k)^2 - f_n(\mathbf{x}-h_k\vec{e}_k)^2 + f_k(\mathbf{x}+h_k\vec{e}_k)^2 - f_k(\mathbf{x}-h_k\vec{e}_k)^2}{4h_k} \qquad k=1$$

In order to avoid numerical cancellation, the numerator has been expanded obtaining the following formula

$$\frac{\partial F}{\partial x_1}(\mathbf{x}) \approx \frac{4h_k x_1 - 2/5 h_k x_2^2 - 4/5 h_k x_n x_1 + 8/100 h_k x_1 (x_1^2 + h_k^2)}{4h_k}$$
$$\frac{\partial F}{\partial x_k}(\mathbf{x}) \approx \frac{4h_k x_k - 2/5 h_k x_{k+1}^2 - 4/5 h_k x_{k-1} x_k + 8/100 h_k x_k (x_k^2 + h_k^2)}{4h_k}$$
$$\frac{\partial F}{\partial x_n}(\mathbf{x}) \approx \frac{4h_k x_n - 2/5 h_k x_1^2 - 4/5 h_k x_{n-1} x_n + 8/100 h_k x_n (x_n^2 + h_k^2)}{4h_k}$$

We can now proceed to approximate the second order derivatives of the function $F(\mathbf{x})$ using the centered finite difference formula; this time we need to use two different increments h_i and h_j based on the two components with respect to which we are differentiating. The general formula is the following

$$\frac{\partial^2 F}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{F(\mathbf{x}+h_i\vec{e}_i+h_j\vec{e}_j) - F(\mathbf{x}+h_i\vec{e}_i) - F(\mathbf{x}-h_j\vec{e}_j) + F(\mathbf{x})}{h_i h_j}$$

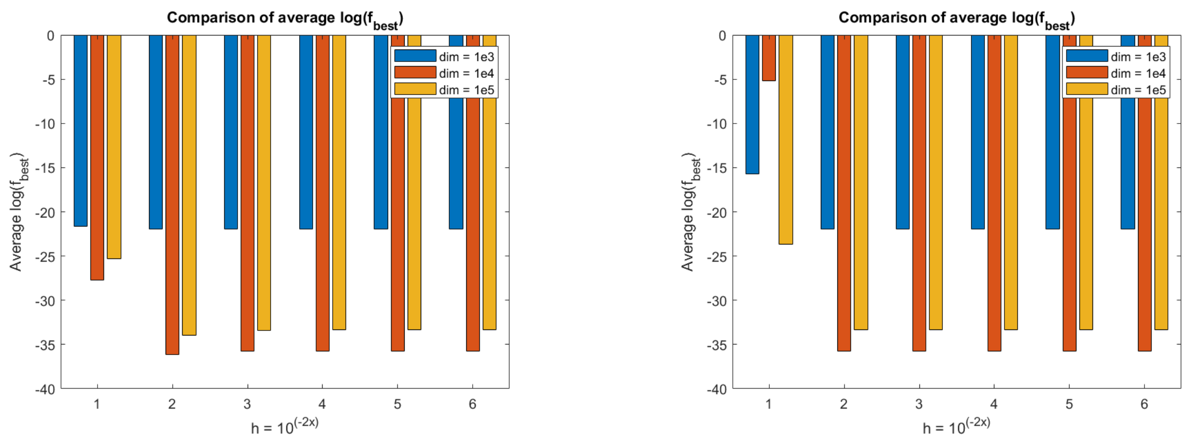
The approximation of the Hessian matrix has to be approached taking into account its sparsity in order to reduce the computational cost, indeed in the Matlab script we have implemented a function that approximates the Hessian matrix just by computing the non-null terms which are the following

$$\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) \approx 2h_k - \frac{2}{5}x_{k-1}h_k + \frac{12}{100}x_k^2 h_k^2 + \frac{24}{100}x_k h_k^3 + \frac{14}{100}h_k^2 \qquad 1 < k \le n$$
$$\frac{\partial^2 F}{\partial x_k^2}(\mathbf{x}) \approx 2h_k - \frac{2}{5}x_n h_k + \frac{12}{100}x_k^2 h_k^2 + \frac{24}{100}x_k h_k^3 + \frac{14}{100}h_k^2 \qquad k = 1$$
$$\frac{\partial^2 F}{\partial x_k \partial x_{k+1}}(\mathbf{x}) \approx -\frac{2}{5}h_k h_{k+1} x_{k+1} - \frac{1}{5}h_k^2 h_{k+1} \qquad 1 \le k < n$$

The values of the inferior diagonal are obtained by exploiting the symmetry of the Hessian matrix.

The terms have been computed following the same approach described above: the numerator has been expanded neglecting the $f_i^2()$ that are not affected by the variation of the components with respect to which we are differentiating.

We now report some barplots showing the results obtained by running the Modified Newton method on the function $F(\mathbf{x})$ using the approximated derivatives.

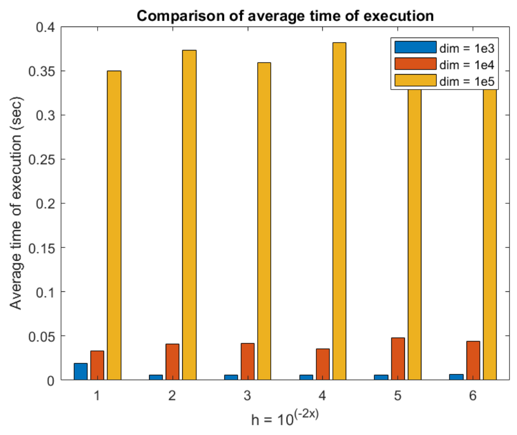


(a) Constant Increment h (b) Specific Increment

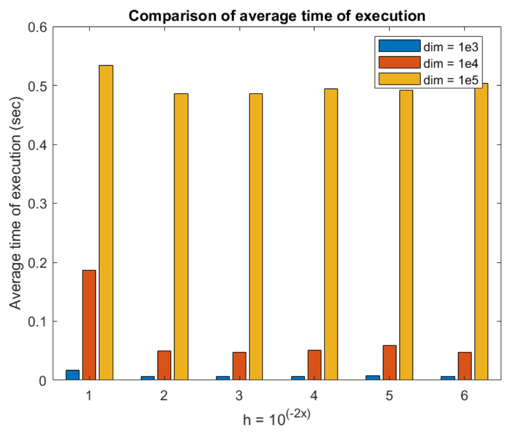
Figura 25: Values of the average $\log(f_{best})$ in function of the increment while running the Modified Newton Method with approximated derivatives on the problem 76.

As we can see from the plots (25), especially for larger values of the increments, the algorithm converges to a point such that the value of the function is higher accordingly to the fact that the approximated derivatives are less accurate. Nonetheless, the method succeeds to find an acceptable approximation of the minimum value even when computing the descent direction with just an approximation of the derivatives.

In general, we can notice that for this specific problem the exact derivatives or the approximated ones while running the Modified Newton method does not significantly affect the results. As a matter of fact we can see



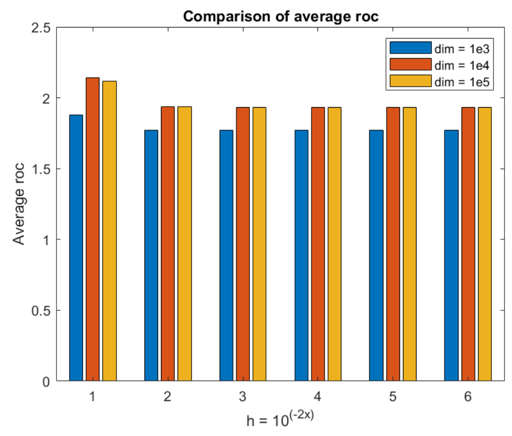
(a) Constant Increment h



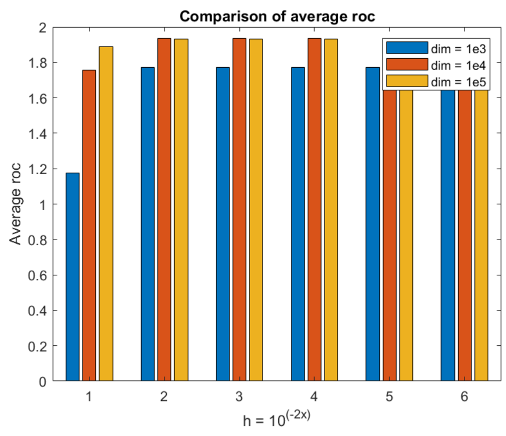
(b) Specific Increment

Figura 26: Average time of execution in function of the increment h while running the Modified Newton Method with approximated derivatives on the problem 76.

that the average minimum point found is consistently close to 0 for each choice of the increment h (as shown by the barplots (25)) and also the average time of execution remains almost the same as we can see from the barplots (26). Lastly, also the average rate of convergence, shown in barplots (27), computed while running the method with approximated derivatives is close to 2 for each increment.



(a) Constant Increment h



(b) Specific Increment

Figura 27: Average values of the experimental rate of convergence in function of the increment h while running the Modified Newton Method with approximated derivatives on the problem 76.

In conclusion, as deducted at the beginning, for this specific problem, which does not have a flat neighborhood of the minimum, optimizing the function using a method which takes advantage of the information contained in the derivatives is a great choice because it can easily converge in few iterations.

CONCLUSION

APPENDIX

Nelder Mead

```
function [xbest,xseq,iter,fbest, flag, failure]= nelderMead(f,x0,rho,chi,
    gamma,sigma,kmax,tol)

close all

% [xbest,iter,fbest]= nelderMead(f,x0,rho,chi,gamma,sigma,kmax,tol)
%
% Functiopn that finds the minimizer of the function f using the Nealder
% Method.
%
% INPUTS:
% f = function handle that return the value of the function we want to
    minimize f : R^n --> R
% x0 = either the initial point (x in R^n) of the method or the initial
% symplex (x in R^(n,n+1), the columns of x are the vertices of the symplex)
% rho = reflection factor
% chi = expansion factor
% gamma = contraction factor
% sigma = shrinking factor
% kmax = maximum number of iterations
% tol = tollerance on the absolute value of f(xN) - f(x1)
%
% OUTPUTS:
% xbest = the last xk computed by the function
% xseq = matrix n x 3, the k-th col contains the best point of the symplex
% of the last 3 iterates
% which minimize the function f
% iter = number of iterations
% fbest = approximation of the minimum
% flag = if true, the given symplex was degenere
% failure = if true, we are declaring a failure
%

% we are verifying that all the parameters are passed as inputs, eventually
% we set rho, chi, gamma and sigma with default valuess
if isempty(rho)
    rho=1;
end
if isempty(chi)
    chi=2;
end
if isempty(gamma)
    gamma=0.5;
end
if isempty(sigma)
    sigma=0.5;
end
if isempty(kmax)
    kmax=200*size(x0,1);
end
if isempty(tol)
    tol=1e-6;
end


n=size(x0,1); %dimension of the space we are working
flag = false;
```

```
failure = false;

if rank(x0) < n && size(x0,2) > 1
    % se simplesso degenere ritorniamo flag = true
    flag = true;
    xbest = nan; iter = 0; fbest = nan;
    return
end


% preliminary analysis of the function in order to compute a smart complex
eval_pt = [1, 50, -50, 100, -100, 300, -300, 600, -600, 1000, -1000, 3000,
   -3000, 6000, -6000 10000, -10000];
best_direction = zeros(n,1);
for comp = 1:n
    eval = zeros(length(eval_pt),1);
    id = 1;
    for pt = eval_pt
        x = x0;
        x(comp) = x(comp) + pt;
        eval(id) = f(x);
        id = id + 1;
    end
    [~, id_pt] = min(eval);
    best_direction(comp) = eval_pt(id_pt);
end

disp("ho finito la valutazione di funzione")


if size(x0,2)==1
    %se in input un solo punto costruiamo il simplesso di partenza
    simplex0=zeros(n,n+1);
    simplex0(:,1)=x0;
    for i=1:n
        ei=zeros(n,1);
        ei(i) = best_direction(i);
        % ei(i) = 0.05*x(i); % according to Matlab implementation
        simplex0(:,i+1)= x0 + ei;
    end
    x0=simplex0;
end


fk=zeros(n+1,1);
comp=0;

% sorting the point based on the evaluation of the function in the point
for i=1:n+1
    fk(i)=f(x0(:,i));
end
[fk_sorted,indices]=sort(fk);


xseq = zeros(n,4);
cont = 1;
xseq(:,cont) = x0(:,indices(1));
best_values = []; % list I will use to plot the convergence of the method


while comp<kmax && (fk_sorted(n) - fk_sorted(1)) > tol
```

```
shrinking=false; %false se devo aggiornare solo un punto, true se ho
    fatto shrink

% indices last element
np1 = indices(end);

% we are keeping the n best vertices to compute the centroid
x0_best_n=x0;
x0_best_n(:,indices(end))=[];
centroid=sum(x0_best_n(:, 1:n))/n;
centroid = centroid';

% REFLECTION PHASE
xR= (1+rho) * centroid - rho * x0(:,np1);
fxR=f(xR);
if fxR>fk_sorted(1) && fxR<fk_sorted(n)
    xnew=xR;
    %x0(:,indices(end))=xnew; %se non lo metto non lo aggiorna (con il
        continue passa subito all'iterazione successiva?)
    %continue
elseif fxR<=fk_sorted(1)
    % EXPANSION PHASE
    xE= (1+rho*chi) * centroid - rho*chi*x0(:, np1);
    if f(xE)<fxR
        xnew=xE;
        %x0(:,indices(end))=xnew; %se non lo metto non lo aggiorna (con
            il continue passa subito all'iterazione successiva?)
        %continue
    else
        xnew=xR;
        %x0(:,indices(end))=xnew; %?
        %continue
    end
else
    % CONTRACTION PHASE
    if fxR > fk_sorted(n+1)
        % inside contraction
        xC=(1-gamma) * centroid + gamma*x0(:,np1);
    else
        % outside contraction
        xC= (1 + gamma*rho)*centroid - gamma * rho*x0(:, np1);
    end
    if f(xC)<fk_sorted(end)
        xnew=xC;
    else
        % SHRINKING PHASE
        shrinking=true;
        x=zeros(n,n+1);
        x(:,1:n+1)=x0(:,indices(1))+sigma.*(x0(:,1:n+1)-x0(:,indices(1))
            );
        x(:,indices(1))=x0(:,indices(1));
        x0=x;
    end
end

% if we have not shrunk the symplex, we have to update the symplex by
% replacing the worst vertice with the new one
if ~shrinking
    x0(:,np1)=xnew;
end
```

```
    % PREPARATION for next iterations
    comp=comp+1;

    % sorting the point based on the evaluation of the function in the point
    for i=1:n+1
        fk(i)=f(x0(:,i));
    end
    [fk_sorted,indices]=sort(fk);

    % updating xseq
    if cont == 4
        cont = 1;
    else
        cont = cont + 1;
    end
    xseq(:,cont) = x0(:,indices(1));

    best_values(end+1) = fk_sorted(1);
    % plot
    if mod(comp, 10) == 0
        figure(1);
        plot(best_values, '-o', 'MarkerSize', 4);
        xlabel('Iterations');
        ylabel('Best Evaluation');
        title('Progress minimum value Nealder Mead');
        drawnow;
    end

end

% computing the minimizer and the minimum found
xbest = x0(:,indices(1));
iter = comp;
fbest = fk_sorted(1);

% cutting xseq and ordering in in such a way that the last column is the
% most recent solution
m = min(iter,4); %number of iterations available in xseq
xseq = xseq(:,1:m);
shift = mod(cont,m);
xseq = circshift(xseq,-shift,2);


if iter == kmax && (fk_sorted(n) - fk_sorted(1)) > tol
    failure = true;
end
```

Modified Newton method

```
function [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck,
    failure, cos_pk_gradf] ...
     = modified_Newton(f,gradf, Hessf, x0, itermax, rho, c1, btmax, tolgrad,
        tau_kmax, alg_modificare_hess, x_esatto)

close all

% [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure] =
    modified_Newton(f,gradf, Hessf, x0, itermax, rho, c1, btmax, tolgrad,
    tau_kmax)
%
% Function the minimizer of the function f by using the Modified Newton
% Method and implementing backtracking
%
% INPUTS:
% f = function handle that return the values of function we want to minimize
     f_R^n --> R;
% gradf = function handle that compute the gradient of the function f in a
    given point;
% Hessf = function handle that compute the Hessian of the function f in a
    given point;
% x0 = starting point in R^n;
% itermax = maximum number of outter iterations;
% rho = fixed factor, lesser than 1, used for reducing alpha0;
% c1 = the factor of the Armijo condition that must be a scalar in (0,1);
% btmax =  maximum number of steps for updating alpha during the
    backtracking strategy;
% tolgrad = value used as stopping criterion w.r.t. the norm of the gradient
    ;
% tau_kmax = maximum number of iterations permitted to compute Bk at each
    step;
% alg_modificare_hess = 'ALG' or 'EIG'
% x_esatto = used to display he dstance from the minimum point, -1 if min
% is not know, 0 if we don't want to disp anything
%
% OUTPUTS:
% xbest = the last xk computed by the function;
% xseq = matrix nx3 where we stored just the last 3 xk;
% iter = index of the last iteration performed;
% fbest = the value of f(xbest);
% gradfk_norm = value of the norm of gradf(xbest);
% btseq =  1-by-iter vector where elements are the number of backtracking
    iterations at each optimization step;
% flag_bcktrck = returns true if the method stopped because the backtracking
     failed;
% failure = returns true if the method stopped with iter = itermax and
    without satisfying the stopping criterion w.r.t- the norm of the gradient
    ;
%


% we are verifying that all the parameters are passed as inputs
if isempty(rho)
    rho=0.5;
end
if isempty(c1)
    c1 = 1e-4;
end
if isempty(itermax)
```

```
    itermax=500;
end
if isempty(tolgrad)
    tolgrad=1e-6;
end
if isempty(btmax)
    btmax=45;
end

if isempty(tau_kmax)
    tau_kmax=50;
end
if isempty(x_esatto)
    x_esatto=-1;
end
if isempty(alg_modificare_hess)
    alg_modificare_hess='ALG';
end







% Function handle for the armijo condition
farmijo = @(fk, alpha, c1_gradfk_pk) fk + alpha * c1_gradfk_pk;

% initializing quantities
n = length(x0); %dimension
xseq = zeros(n,4);
cont = 1;
xseq(:,cont) = x0;
btseq = zeros(itermax,1);
failure = false;
flag_bcktrck = false;

fk = f(x0);
gradfk = gradf(x0);
Hessfk = Hessf(x0);
k = 0;

best_values = zeros(itermax,1);
best_values(1) = fk;
best_gradf = zeros(itermax,1);
best_gradf(1) = norm(gradfk);

while k < itermax && sum(gradfk.^2) > tolgrad^2

    switch alg_modificare_hess
        case 'ALG'

            % Calcolo di Bk secondo l'Algoritmo 3.3
            beta = 1e-3;
            min_diag = min(diag(Hessfk));

            % Inizializzazione di tau_0 (aggiustamento per la definizione
               positiva)
            if min_diag > 0
                tau_0 = 0;
            else
                tau_0 = -min_diag + beta;
            end

            failure_chol = false; % Inizializza il flag per il fallimento
               del Cholesky

            % Loop per la regolarizzazione
            k_tau = 0;
            p=7;
            while p > 0 && k_tau < tau_kmax
                Bk = Hessfk + tau_0 * speye(n); % Incrementa il termine
                   diagonale
                [R, p] = chol(Bk);

                % mi preparo per un eventual step successivo
                k_tau = k_tau+1;
                tau_0 = max(beta, 5*tau_0);
            end

            if k_tau == tau_kmax && p > 0
                failure_chol = true;
            end

            % Controllo finale del successo del Cholesky
            if failure_chol
                disp("ALGORITMO 3.3 HA FALLITO: Hessiana non regolarizzabile
                   ");
                disp(["minimo e massimo autovalore di HessF:", num2str(min(
                   eig(Hessfk + tau_0 * speye(n)))),...
                    num2str(max(eig(Hessfk + tau_0 * speye(n))) )] );
                xbest = x0; fbest = fk; iter = k; gradfk_norm = norm(gradfk)
                   ; failure = true;
                return;
            end

            % Calcolo della direzione pk sfruttando la fattorizzazione di
               Cholesky
            y = -R' \ gradfk;
            pk = R \ y;
            cos_pk_gradf = (pk' * gradfk)/(norm(pk) * norm(gradfk));

        case 'EIG'
        % calcolo Bk secondo la definizione
        autovett_min = eigs(Hessfk, 3, 'smallestreal', 'FailureTreatment','
           keep', 'MaxIterations', 500);
        tau_k = max([0, 1e-6 - min(autovett_min)]);
        Bk = Hessfk + tau_k * speye(n);
        pk = -Bk\ gradfk;
    end


    % BACKTRACKING
    % Reset the value of alpha
    alpha = 1;

    % Compute the candidate new xk
    xnew = x0 + alpha * pk;
    % Compute the value of f in the candidate new xk
    fnew = f(xnew);
    c1_gradfk_pk = c1 * (gradfk' * pk);
    bt = 0;
    % Backtracking strategy:
    % 2nd condition is the Armijo condition not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
        % Reduce the value of alpha
        alpha = rho * alpha;
        % Update xnew and fnew w.r.t. the reduced alpha
        xnew = x0 + alpha * pk;
        fnew = f(xnew);

        % Increase the counter by one
        bt = bt + 1;
    end
    if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
        btseq(k+1, 1) = bt;
        flag_bcktrck = true;
        x0 = xnew;
        k = k+1;
        break
    end

    x0 = xnew;


    % preparing for the next iteration
    k = k+1;
    fk = f(x0);
    btseq(k,1) = bt;
    gradfk = gradf(x0);
    Hessfk = Hessf(x0);

    % updating xseq
    if cont == 4
        cont = 1;
    else
        cont = cont + 1;
    end
    xseq(:,cont) = x0;

    best_values(k) = fk;
    best_gradf(k) = norm(gradfk);
    if mod(k, 10) == 0
        figure(1);
        plot(best_values(6:k), '-o', 'MarkerSize', 4);
        xlabel('Iterations');
        ylabel('Best Evaluation');
        title('Progress minimum value Modified Newton Method');
        drawnow;

        % figure(2);
        % plot(best_gradf(5:k), '-o', 'MarkerSize', 4);
        % xlabel('Iterations');
        % ylabel('Best Evaluation');
        % title('Progress gradient value Modified Newton Method');
        % drawnow;

    end

%     if x_esatto == -1
%         testo = ['norm gradiente = ', num2str(norm(gradfk)), ' alla
   iterazione ',  num2str(k)];
%         disp(testo)
%     elseif length(x_esatto) > 1
%          testo = ['distanza alla ', num2str(k), ' iterazione = ', num2str(
    norm(x_esatto-x0)), ' e norm gradiente = ', num2str(norm(gradfk))];
%          disp(testo)
%      end


end

% declaring failure if this is the case
if (k == itermax || flag_bcktrck) && sum(gradfk.^2) > tolgrad^2
    failure = true;
end

xbest = x0;
fbest = fk;
iter = k;
m = min(iter,4); %number of iterations available in xseq
xseq = xseq(:,1:m);
shift = mod(cont,m);
xseq = circshift(xseq,-shift,2);
btseq = btseq(1:iter,1);
gradfk_norm = norm(gradf(x0));

end
```

Rosenbrock function

```
% ESERCIZIO 2

clear all
clc

% function that compute the rate of convergence
function rate_of_convergence = compute_roc(x_esatto, xseq)
if size(xseq,2) >=3
    rate_of_convergence = log(norm(x_esatto - xseq(:,end))/norm(x_esatto -
       xseq(:, end-1)))/log(norm(x_esatto - xseq(:,end-1))/norm(x_esatto -
       xseq(:, end-2)));
else
    rate_of_convergence = nan;
end
end



% Parametric Rosenbrock function in dimension n
function f = parametric_rosenbrock(x, alpha)
    f = 0;
    n = length(x);
    for i = 2:n
        f = f + alpha * (x(i) - x(i-1)^2)^2 + (1 - x(i-1))^2;
    end
end

function gradf = grad_parametric_rosenbrock(x,alpha)
    n = length(x);
    gradf = zeros(n,1);

    for k = 2:n-1
        gradf(k,1) = -2*alpha*(x(k-1)^2 - x(k)) + 2*(x(k) -1) +4*alpha*x(k)
           *(x(k)^2 - x(k+1));
    end

    gradf(1,1) = 2*(x(1) -1) + 4*alpha*x(1)*(x(1)^2 - x(2));
    gradf(n,1) = -2*alpha*(x(n-1)^2 - x(n)) ;

end

function Hessf = hess_parametric_rosenbrock(x,alpha)
    n = length(x);
    diags = zeros(n,3);
    % diags(:,1) is the principal one, diags(:,2) is the superior one and
    % diags(:,3) is the inferior one

    diags(1,1) = 2 + 12*alpha*x(1)^2 - 4*alpha*x(2);
    diags(n,1) = 2*alpha;
    diags(n-1,3) = -4*alpha*x(n-1);
    diags(n,2) = -4*alpha*x(n-1);

    for k = 2:n-1
       diags(k,1) = 2*alpha + 12*alpha*x(k)^2 - 4*alpha*x(k+1) +2;
       diags(k-1,3) = -4*alpha*x(k-1); %diag inferior: k is the first
          derivative
       diags(k,2)= -4*alpha*x(k-1); %diag superior: k id the first
          derivative
    end
    Hessf = spdiags(diags, [0, +1, -1], n, n);

end

% the excercice asks to fix alpha = 100
f = @(x) parametric_rosenbrock(x, 100);
gradf = @(x) grad_parametric_rosenbrock(x,100);
Hessf = @(x) hess_parametric_rosenbrock(x,100);

%%

% initial points for the algorithms
x0_a = [1.2; 1.2];
x0_b = [-1.2; 1];
x_esatto = [1;1];
n = 2;

tol = 1e-7;
rho = 0.5; c1 = 1e-4; btmax = 40; tau_kmax = 100;
iter_max = 200;

time_SX = 0;
time_MN = 0;

% we run each model twice with different initial conditions and we compute
% some summary
t1 = tic;
[xbest_SX_a,xseq_SX_a,iter_SX_a,fbest_SX_a, flag_SX_a, failure_SX_a] =
   nelderMead(f,x0_a,[],[],[],[],iter_max*n,tol);
time_SX_a =toc(t1);
disp('**** SIMPLEX METHOD FOR THE PB 1 (point [1.2; 1.2] ):  *****');
disp(['Time: ', num2str(time_SX_a), ' seconds']);

disp('**** SIMPLEX METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest_SX_a)])
disp(['norma di gradf(xk): ', num2str(norm(gradf(xbest_SX_a)))])
disp(['N. of Iterations: ', num2str(iter_SX_a),'/',num2str(iter_max*n)])
disp('************************************')

if (failure_SX_a)
    disp('FAIL')
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

t1 = tic;
[xbest_MN_a, xseq_MN_a, iter_MN_a, fbest_MN_a, gradfk_norm_MN_a, btseq_MN_a,
    flag_bcktrck_MN_a, failure_MN_a, ~] = modified_Newton(f,gradf, Hessf,
   x0_a, 5000, rho, c1, btmax, tol, tau_kmax, 'ALG', 0);
time_MN_a =  toc(t1);
disp('**** MODIFIED NEWTON METHOD FOR THE PB 1 (point [1.2; 1.2] ):  *****')
   ;
disp(['Time: ', num2str(time_MN_a), ' seconds']);
disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ', num2str(c1)
   ]);
```

```
disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest_MN_a)])
disp(['norma di gradf(xk): ', num2str(gradfk_norm_MN_a)])
disp(['N. of Iterations: ', num2str(iter_MN_a),'/',num2str(iter_max)])
disp('************************************')

if (failure_MN_a)
    disp('FAIL')
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')


t1 = tic;
[xbest_SX_b,xseq_SX_b,iter_SX_b,fbest_SX_b, flag_SX_b, failure_SX_b] =
   nelderMead(f,x0_b,[],[],[],[],400,tol);
time_SX_b =toc(t1);
disp('**** SIMPLEX METHOD FOR THE PB 1 (point [-1.2; 1] ):  *****');
disp(['Time: ', num2str(time_SX_b), ' seconds']);

disp('**** SIMPLEX METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest_SX_b)])
disp(['norma di gradf(xk): ', num2str(norm(gradf(xbest_SX_b)))])
disp(['N. of Iterations: ', num2str(iter_SX_b),'/',num2str(iter_max*n)])
disp('************************************')

if (failure_SX_b)
    disp('FAIL')
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

t1 = tic;
[xbest_MN_b, xseq_MN_b, iter_MN_b, fbest_MN_b, gradfk_norm_MN_b, btseq_MN_b,
    flag_bcktrck_MN_b, failure_MN_b, ~] = modified_Newton(f,gradf, Hessf,
   x0_b, 5000, rho, c1, btmax, tol, tau_kmax, 'ALG', 0);
time_MN_b =  toc(t1);
disp('**** MODIFIED NEWTON METHOD FOR THE PB 1 (point [-1.2; 1] ):  *****');
disp(['Time: ', num2str(time_MN_b), ' seconds']);
disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ', num2str(c1)
   ]);

disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest_MN_b)])
disp(['norma di gradf(xk): ', num2str(gradfk_norm_MN_b)])
disp(['N. of Iterations: ', num2str(iter_MN_b),'/',num2str(iter_max)])
disp('************************************')

if (failure_MN_b)
    disp('FAIL')
    disp('************************************')
```

```
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

% creation of the table
time_SX = time_SX_a + time_SX_b;
time_MN = time_MN_a + time_MN_b;
failure = [failure_SX_a + failure_SX_b;  failure_MN_a + failure_MN_b];
avg_iter = [(iter_SX_b+iter_SX_a)/2; (iter_MN_b + iter_MN_a)/2];
avg_time_execution = [time_SX/2; time_MN/2];
roc = [(compute_roc(x_esatto, xseq_SX_b)+ compute_roc(x_esatto, xseq_SX_a))
   /2; (compute_roc(x_esatto, xseq_MN_b)+ compute_roc(x_esatto, xseq_MN_a))
   /2];
avg_gradfk = [NaN; (gradfk_norm_MN_a +gradfk_norm_MN_b)/2];
avg_fbest =[(fbest_SX_b+fbest_SX_a)/2; (fbest_MN_b+fbest_MN_a)/2];

T = table( failure, avg_fbest, avg_gradfk, avg_iter, avg_time_execution, roc
   , 'RowNames', {'simplex method'; 'modified Newton'});
display(T)
```

Problem 25

```
%% PROBLEMA 25
% non capisco dove abbia minimo
close all
clear all
clc

% setting the seed
seed = min(339268, 343310);

% function to compute the rate of convergence
function rate_of_convergence = compute_roc(xseq)
if size(xseq,2) >=4
    k = size(xseq,2) -1;
    norm_ekplus1 = norm(xseq(:, k+1) - xseq(:,k));
    norm_ek = norm(xseq(:, k) - xseq(:,k-1));
    norm_ekminus1 = norm(xseq(:, k-1) - xseq(:,k-2));
    rate_of_convergence = log(norm_ekplus1/norm_ek) / log(norm_ek/
       norm_ekminus1);
else
    rate_of_convergence = nan;
end
end


% implementing the function, the gradient and the hessiano for problem 64
function val = function_pb25(x)
    n = length(x);
    val = 0;

    for k= 1:2:n-1
        val = val + 10*(x(k)^2 - x(k+1))^2;
    end
    for k=2:2:n-1
        val = val + (x(k-1) -1)^2;
    end

    if mod(n,2) == 1
        val = val + (10*x(n)^2 - x(1))^2;
    else
        val = val + (x(n-1) -1)^2;
    end

    val = 0.5*val;
end

f = @(x) function_pb25(x);

function grad = grad_pb25(x)
    n = length(x);
    grad = zeros(n,1);

    if mod(n,2) == 0
        grad(1:2:n-1) = 200*x(1:2:n-1).^3 - 200*x(1:2:n-1).*x(2:2:n) + x
           (1:2:n-1) -1;
        grad(2:2:n) = - 100*(x(1:2:n-1).^2 - x(2:2:n));
    else
        grad(1, 1) = 200*x(1)^3 - 200*x(1)*x(2) + x(1) -1 - 100*(x(n)^2 - x
           (1));
        grad(3:2:n-1) = 200*x(3:2:n-1).^3 - 200*x(3:2:n-1).*x(4:2:n) + x
```

```
            (3:2:n-1) -1;
        grad(2:2:n-2) = - 100*(x(1:2:n-3).^2 - x(2:2:n-2));
        grad(n,1) = 200*x(n).^3  - 200*x(n)*x(1) + x(n) -1;
    end
end

gradf = @(x) grad_pb25(x);


function val = hessian_pb25(x)
    n = length(x);
    diags = zeros(n,5); %1st column is the principal diag, 2nd column is the
        superior diag and 3rd column is the inferior

    % principal diag
    if mod(n,2) == 0
        diags(2:2:n,1) = 100;
        diags(1:2:n-1,1) = 600*x(1:2:n-1).^2 - 200 * x(2:2:n) +1;
    else
        diags(1,1) = 600*x(1)^2  - 200*x(2) +101;
        diags(2:2:n,1) = 100;
        diags(3:2:n-1,1) = 600*x(3:2:n-1).^2 - 200 * x(4:2:n) +1;
        diags(n,1) = 600*x(n).^2 - 200*x(1) +1;
    end

    % inferior diagonal
    diags(1:2:n-1,3) = -200*x(1:2:n-1);
    diags(2:2:n-2, 3) = 0;

    %superior diagonal
    diags(3:2:n-1,2) = 0;
    diags(2:2:n, 2) = -200*x(1:2:n-1);

    % these diagonals exists only if n is odd
    if mod(n,2) == 1
        diags(1,5) = - 200*x(n);
        diags(n,4) = - 200*x(n);
    end


    val = spdiags(diags, [0,1,-1, n-1, - (n-1)], n,n);
end


Hessf = @(x) hessian_pb25(x);

tol = 1e-4;


%% RUNNING THE EXPERIMENTS ON NEALDER MEAD
format short e
clc

% setting the dimensionality
dimension = [10 25 50];
iter_max = 400;
rng(seed);


% initializing the structures to store some stats
execution_time_SX = zeros(length(dimension),11);
```

```
failure_struct_SX = zeros(length(dimension),11); %for each dimension we
    count the number of failure
iter_struct_SX = zeros(length(dimension),11);
fbest_struct_SX = zeros(length(dimension),11);
roc_struct_SX = zeros(length(dimension),11);

for dim = 1:length(dimension)
    n = dimension(dim);

    % defining the given initial point
    x0 = ones(n,1);
    x0(1:2:n) = -1.2;

    % in order to generate random number in [a,b] I apply the formula r = a
        + (b-a).*rand(n,1)
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);

    % SOLVING SIMPLEX METHOD
    % first initial point
    t1 = tic;
    [~, xseq,iter,fbest, ~, failure] = nelderMead(f,x0,[],[],[],[],iter_max*
        size(x0,1),tol);
    execution_time_SX(dim,1) = toc(t1);
    fbest_struct_SX(dim,1) = fbest;
    iter_struct_SX(dim,1) = iter;
    roc_struct_SX(dim,1) = compute_roc(xseq);
    disp(['**** SIMPLEX METHOD FOR THE PB 25 (point ', num2str(1), ',
        dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_SX(dim,1)), ' seconds']);

    disp('**** SIMPLES METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(x0
        ,1))])
    disp(['Rate of Convergence: ', num2str(roc_struct_SX(dim,1))])
    disp('************************************')

    if (failure)
        disp('FAIL')
        disp('************************************')
    else
        disp('SUCCESS')
        disp('************************************')
    end
    disp(' ')


    % if failure = true (failure == 1), the run was unsuccessful; otherwise
    % failure = 0
    failure_struct_SX(dim,1) = failure_struct_SX(dim,1) + failure;

    for i = 1:10
        t1 = tic;
        [~,xseq,iter,fbest, ~, failure] = nelderMead(f,x0_rndgenerated(:,i)
            ,[],[],[],[],iter_max*size(x0,1),tol);
        execution_time_SX(dim,i+1) = toc(t1);
        fbest_struct_SX(dim,i+1) = fbest;
        iter_struct_SX(dim,i+1) = iter;
        failure_struct_SX(dim,i+1) = failure_struct_SX(dim,i+1) + failure;
        roc_struct_SX(dim,i+1) = compute_roc(xseq);

        disp(['**** SIMPLEX METHOD FOR THE PB 25 (point ', num2str(i+1), ',
           dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_SX(dim,i+1)), ' seconds']);

        disp('**** SIMPLES METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(
           x0,1))])
        disp(['Rate of Convergence: ', num2str(roc_struct_SX(dim,i+1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')

    end
end


varNames = ["average fbest", "average number of iterations", "average time
   of execution (sec)", "numbers of failure", "average rate of convergence
   "];
rowNames = string(dimension');
TSX = table(sum(fbest_struct_SX,2)/11, sum(iter_struct_SX,2)/11, sum(
   execution_time_SX,2)/11, sum(failure_struct_SX,2),sum(roc_struct_SX,2)/11
    ,'VariableNames', varNames, 'RowNames', rowNames);
display(TSX)



%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD
format short e
clc

% setting the values for the dimension
dimension = [1e3 1e4 1e5];
iter_max = 3000;

param = [0.5, 1e-4, 48; 0.5, 1e-4, 48; 0.5, 1e-4, 48];

rng(seed);

% initializing structures to store some stats
execution_time_MN = zeros(length(dimension),11);
failure_struct_MN = zeros(length(dimension),11); %for each dimension we
   count the number of failure
iter_struct_MN = zeros(length(dimension),11);
fbest_struct_MN = zeros(length(dimension),11);
gradf_struct_MN = zeros(length(dimension),11);
roc_struct_MN = zeros(length(dimension),11);
ultima_direz_discesa = zeros(length(dimension), 11);
```

```
for dim = 1:length(dimension)
    n = dimension(dim);

    [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3));


    %defining the given initial point
    x0 = ones(n,1);
    x0(1:2:n) = -1.2;

    % in order to generate random number in [a,b] I apply the formula r = a
       + (b-a).*rand(n,1)
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);


    % SOLVING MODIFIED NEWTON METHOD METHOD
    % first initial point
    t1 = tic;
    [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure,
       pk_scalare_gradf] = modified_Newton(f,gradf, Hessf, x0, iter_max, rho
       , c1, btmax, tol, [], 'ALG', 0);
    execution_time_MN(dim,1) = toc(t1);
    fbest_struct_MN(dim,1) = fbest;
    iter_struct_MN(dim,1) = iter;
    gradf_struct_MN(dim,1) = gradfk_norm;
    roc_struct_MN(dim,1) = compute_roc(xseq);
    ultima_direz_discesa(dim,1) = pk_scalare_gradf;
    disp(['**** MODIFIED NEWTON METHOD FOR THE PB 25 (point ', num2str(1), '
       , dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
    disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ', num2str(
       c1)]);

    disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
    disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
    disp('************************************')

    if (failure)
        disp('FAIL')
        if (flag_bcktrck)
            disp('Failure due to backtracking')
        else
            disp('Failure not due to backtracking')
        end
        disp('************************************')
    else
        disp('SUCCESS')
        disp('************************************')
    end
    disp(' ')

    % if failure = true (failure == 1), the run was unsuccessful; otherwise
    % failure = 0
    failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;
    for i = 1:10
        t1 = tic;
        [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
           , pk_scalare_gradf] = modified_Newton(f,gradf, Hessf,
           x0_rndgenerated(:,i), iter_max, rho, c1, btmax, tol, [], 'ALG',
           0);
        execution_time_MN(dim,i+1) = toc(t1);
        fbest_struct_MN(dim,i+1) = fbest;
        iter_struct_MN(dim,i+1) = iter;
        failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) + failure;
        gradf_struct_MN(dim,i+1) = gradfk_norm;
        roc_struct_MN(dim,i+1) = compute_roc(xseq);
        ultima_direz_discesa(dim,i+1) = pk_scalare_gradf;

        disp(['**** MODIFIED NEWTON METHOD FOR THE PB 25 (point ', num2str(i
           +1), ', dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds']);
        disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
           num2str(c1)]);

        disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
        disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,i+1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            if (flag_bcktrck)
                disp('Failure due to backtracking')
            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')
    end

end

% plotto pk_scalar_gradfk
bar(ultima_direz_discesa')
ylabel('cos(angolo)')
title('Ultimo valore assunto da t(pk)*gradfk/(norm_pk * norm_gradfk)')
legend({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'}, "Box", 'on', 'Location', '
   best')

varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time of
   exec (sec)", "n failure", "avg roc"];
rowNames = string(dimension');
TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 ,sum(
   iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(failure_struct_MN
   ,2), sum(roc_struct_MN,2)/11,'VariableNames', varNames, 'RowNames',
   rowNames);
```

```
display(TMN)



%% FINITE DIFFERENCES
clc

function grad_approx = findiff_grad_25(x, h, type_h)
    n = length(x);
    grad_approx = zeros(n,1);



    if mod(n,2) == 0
        % CASO n PARI
        for k = 1:n
            switch type_h
                case 'REL'
                    passok = h * abs(x(k));
                case 'COST'
                    passok = h;
            end

            if mod(k,2) == 0
                grad_approx(k,1) = (-40*passok*(10*x(k-1)^2 - 10*x(k)))/(4*
                    passok);
            else
                grad_approx(k,1) = (80 * x(k)*passok*(10*x(k)^2 + 10*passok
                    ^2 - 10*x(k+1)) + 4*passok*(x(k)-1))/(4*passok);
            end
        end
    else
        % CASO n DISPARI
        for k = 1:n
            switch type_h
                case 'REL'
                    passok = h * abs(x(k));
                case 'COST'
                    passok = h;
            end

            if k == 1
                grad_approx(1,1) = (80 * x(k)*passok*(10*x(k)^2 + 10*passok
                    ^2 - 10*x(k+1)) + 4*passok*(x(k)-1) - 40*passok*(10*x(n)
                    ^2 - 10*x(1)))/(4*passok);
            elseif k == n
                 grad_approx(k,1) = (80 * x(k)*passok*(10*x(k)^2 + 10*passok
                     ^2 -10*x(1)))/(4*passok);
            elseif mod(k,2) == 0
                grad_approx(k,1) = (-40*passok*(10*x(k-1)^2 - 10*x(k)))/(4*
                    passok);
            else
                grad_approx(k,1) = (80 * x(k)*passok*(10*x(k)^2 + 10*passok
                    ^2 - 10*x(k+1)) + 4*passok*(x(k)-1))/(4*passok);
            end
        end
    end
end
```

```
function hessian_approx = findiff_hess_25(x, h, type_h)
    % Calcola la matrice Hessiana sparsa per la funzione f(x)
    % Input:
    %   - x: vettore colonna (punto in cui calcolare l'Hessiana)
    %   - h: passo
    % Output:
    %   - H: matrice Hessiana sparsa

    n = length(x); % Dimensione del problema

    % Preallocazione per la struttura sparsa
    i_indices = [];
    j_indices = [];
    values = [];
    cont = 1;

    % Loop su k (dalla definizione della funzione)
    for k = 1:n

        % ELEMENTI DIAGONALI
        if k == 1 && mod(n,2) == 1
            % caso k == 1 con n dispari

            switch type_h
                case 'COST'
                    hk = h;
                case 'REL'
                    hk = h*abs(x(k));
            end

            H_kk = (40*hk^2*(10*x(k)^2 -10*x(k+1)) + 1400*hk^4 + 2400*hk^3*x
                (k) + 800*x(k)^2*hk^2 + 2*hk^2 + 200*hk^2)/(2*hk^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;


        elseif k == n && mod(n,2) == 1
            % caso k pari con n pari

            switch type_h
                case 'COST'
                    hk = h;
                case 'REL'
                    hk = h*abs(x(k));
            end

            H_kk = (40*hk^2*(10*x(k)^2 -10*x(1)) + 1400*hk^4 + 2400*hk^3*x(k
                ) + 800*x(k)^2*hk^2)/(2*hk^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;


        elseif mod(k,2) == 1
            % caso k dispari (se n dispari, non entra qui ma nella
                condizione sopra)
            switch type_h
                case 'COST'
                    hk = h;
                case 'REL'
                    hk = h*abs(x(k));
            end

            H_kk = (40*hk^2*(10*x(k)^2 -10*x(k+1)) + 1400*hk^4 + 2400*hk^3*x
               (k) + 800*x(k)^2*hk^2 + 2*hk^2)/(2*hk^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;

        elseif mod(k,2) == 0
            % caso k pari con n pari
            switch type_h
                case 'COST'
                    hk = h;
                case 'REL'
                    hk = h*abs(x(k));
            end

            H_kk = (200*hk^2)/(2*hk^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;

        end

        % ELEMENTI EXTRA DIAG
        if mod(n,2) == 1 && k == n
            % ho le due diagonali estremali

            switch type_h
                case 'COST'
                    h1 = h;
                case 'REL'
                    h1 =h*abs(x(1));
            end

            H_n1 = (20*h1 *(-20*x(k)* hk - 10*hk^2))/(2*hk*h1);
            i_indices(cont) = n;
            j_indices(cont) = 1;
            values(cont) = H_n1;
            cont = cont +1;

            % impongo la simmetria
            i_indices(cont) = 1;
            j_indices(cont) = n;
            values(cont) = H_n1;
            cont = cont +1;

        elseif mod(k,2) == 1 && k < n
            % ho solo le derivate k, k+1 con k dispari

            switch type_h
                case 'COST'
                    hk1 = h;
                case 'REL'
                    hk1 =h*abs(x(k+1));
            end
```

```
            H_k_k1 = (20*hk1*(-10 * hk^2 - 20*hk*x(k)))/(2*hk*hk1);
            i_indices(cont) = k;
            j_indices(cont) = k+1;
            values(cont) = H_k_k1;
            cont = cont +1;

            % impongo la simmetria
            i_indices(cont) = k+1;
            j_indices(cont) = k;
            values(cont) = H_k_k1;
            cont = cont +1;
        end

    end


    % Creazione della matrice Hessiana sparsa
    hessian_approx = sparse(i_indices, j_indices, values, n, n);
end


h = 1e-2;
type_h = 'COST';
gradf_approx = @(x) findiff_grad_25(x,h, type_h);
Hessf_approx = @(x) findiff_hess_25(x,h, type_h);

vec = [1; 0.5*ones(13,1); 1];
% vec = [0.2; 0.4; -0.2; 0.5; 0; 0.3; 0];

gradf(vec)
gradf_approx(vec)

format short
full(Hessf(vec))
tic
full(Hessf_approx(vec))
time = toc

%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD WITH FIN DIFF
format short e
clc

iter_max = 3000;
tol = 1e-3;



% setting the values for the dimension
h_values = [1e-2 1e-4 1e-6 1e-8 1e-10 1e-12];
dimension = [1e3 1e4 1e5];
param = [0.5, 1e-4, 48; 0.5, 1e-4, 48; 0.5, 1e-4, 48;];
type_h = 'REL';

tables = struct;

% initializing structures to store some stats
execution_time_MN_h = zeros(length(dimension),6);
failure_struct_MN_h = zeros(length(dimension),6);
iter_struct_MN_h = zeros(length(dimension),6);
fbest_struct_MN_h = zeros(length(dimension),6);
```

```
gradf_struct_MN_h = zeros(length(dimension),6);
roc_struct_MN_h = zeros(length(dimension),6);


for id_h = 1:length(h_values)

   h = h_values(id_h);
   tol = min(1e-3, h);

    gradf_approx = @(x) findiff_grad_25(x,h, type_h);
    hessf_approx = @(x) findiff_hess_25(x,h, type_h);

    % initializing structures to store some stats
    execution_time_MN = zeros(length(dimension),11);
    failure_struct_MN = zeros(length(dimension),11); %for each dimension we
       count the number of failure
    iter_struct_MN = zeros(length(dimension),11);
    fbest_struct_MN = zeros(length(dimension),11);
    gradf_struct_MN = zeros(length(dimension),11);
    roc_struct_MN = zeros(length(dimension),11);


    for dim = 1:length(dimension)
        n = dimension(dim);

        [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3))
           ;


        %defining the given initial point
        x0 = ones(n,1);
        x0(1:2:n) = -1.2;

        % in order to generate random number in [a,b] I apply the formula r
           = a + (b-a).*rand(n,1)
        rng(seed);
        x0_rndgenerated = zeros(n,10);
        x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);


        % SOLVING MODIFIED NEWTON METHOD METHOD
        % first initial point
        t1 = tic;
        [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
           , ~] = modified_Newton(f,gradf_approx, hessf_approx, x0, iter_max
           , rho, c1, btmax, tol, [], 'ALG', 0);
        execution_time_MN(dim,1) = toc(t1);
        fbest_struct_MN(dim,1) = fbest;
        iter_struct_MN(dim,1) = iter;
        gradf_struct_MN(dim,1) = gradfk_norm;
        roc_struct_MN(dim,1) = compute_roc(xseq);
        disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, ' with
            h = ', num2str(h), ') FOR THE PB 25 (point ', num2str(1), ',
           dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
        disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
           num2str(c1)]);

        disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
        disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            if (flag_bcktrck)
                disp('Failure due to backtracking')
            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')

        % if failure = true (failure == 1), the run was unsuccessful;
           otherwise
        % failure = 0
        failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;

        for i = 1:10
            t1 = tic;
            [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck,
               failure, ~] = modified_Newton(f,gradf_approx, hessf_approx,
               x0_rndgenerated(:,i), iter_max, rho, c1, btmax, tol, [], 'ALG
               ', 0);
            execution_time_MN(dim,i+1) = toc(t1);
            fbest_struct_MN(dim,i+1) = fbest;
            iter_struct_MN(dim,i+1) = iter;
            failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) +
               failure;
            gradf_struct_MN(dim,i+1) = gradfk_norm;
            roc_struct_MN(dim,i+1) = compute_roc(xseq);

            disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, '
               with h = ', num2str(h), ') FOR THE PB 25 (point ', num2str(i
               +1), ', dimension ', num2str(n), '):  *****']);

            disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds'
               ]);
            disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
               num2str(c1)]);

            disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
            disp('************************************')
            disp(['f(xk): ', num2str(fbest)])
            disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
            disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)
               ])
            disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,i+1))])
            disp('************************************')

            if (failure)
                disp('FAIL')
                if (flag_bcktrck)
                    disp('Failure due to backtracking')
```

```
                else
                    disp('Failure not due to backtracking')
                end
                disp('************************************')
            else
                disp('SUCCESS')
                disp('************************************')
            end
            disp(' ')
        end
    end



    varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time
       of exec (sec)", "n failure", "avg roc"];
    rowNames = string(dimension');
    TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 ,sum(
       iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(
       failure_struct_MN,2), sum(roc_struct_MN,2)/11,'VariableNames',
       varNames, 'RowNames', rowNames);
    format short e
    display(TMN)

    tables.(['Table' num2str(id_h)]) = TMN;

    execution_time_MN_h(:,id_h) = sum(execution_time_MN,2)/11;
    failure_struct_MN_h(:,id_h) = sum(failure_struct_MN,2);
    iter_struct_MN_h(:,id_h) = sum(iter_struct_MN,2)/11;
    fbest_struct_MN_h(:,id_h) = sum(fbest_struct_MN,2)/11;
    gradf_struct_MN_h(:,id_h) = sum(gradf_struct_MN,2)/11;
    roc_struct_MN_h(:,id_h) = sum(roc_struct_MN,2)/11;

end
```

Problem 75

```
%% problem 75
clc
clear
close all

seed=min(339268,343310);
rng(seed);

% Definition of the problem
%n=1e4; %togli
F_75= @(x) 0.5*((x(1)-1)^2 + sum( (10*(1:length(x)-1)'.*(x(2:end)-x(1:end-1)
    ).^2).^2 ) );
gradF_75= @(x) gradient_pb_75(x);
hessF_75= @(x) hessian_pb_75(x);
%h = 1e-2; %togli
approx_gradF_75= @(x) approxgradient_pb_75(x,h,'COST');
approx_hessF_75= @(x) approxhessian_pb_75(x,h,'COST');

%% RUNNING THE EXPERIMENTS ON NEALDER MEAD
format short e

iter_max = 200;
tol = 1e-12;
[rho, chi, gamma, sigma] = deal(1.1, 2.5, 0.6, 0.5);

% setting the dimensionality
dimension = [10 25 50];


% initializing the structures to store some stats for every dimension and
% starting point
execution_time_SX = zeros(length(dimension),11);
failures_SX = zeros(length(dimension),11); %we count the number of failures
iter_SX = zeros(length(dimension),11);
fbest_SX = zeros(length(dimension),11);
roc_SX = zeros(length(dimension),11);

for dim = 1:length(dimension)
    n = dimension(dim);
    x_esatto = ones(n,1);

    % defining the given initial point
    x0 = -1-2*ones(n,1);
    x0(end) = -1;

    % in order to generate random number in [a,b] I apply the formula r = a
       + (b-a).*rand(n,1)
    % where [a, b] = [x0-1, x0+1]
    rng(seed);
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);

    % SOLVING SIMPLEX METHOD
    % first initial point
    t1 = tic;
    [~, xseq,iter,fbest, ~, failure] = nelderMead(F_75,x0,rho,chi,gamma,
       sigma,iter_max*size(x0,1),tol);
    execution_time_SX(dim,1) = toc(t1);
    fbest_SX(dim,1) = fbest;
```

```
iter_SX(dim,1) = iter;
roc_SX(dim,1) = compute_roc(xseq,x_esatto);
disp(['**** SIMPLEX METHOD FOR THE PB 75 (point ', num2str(1), ',
    dimension ', num2str(n), '):  *****']);

disp(['Time: ', num2str(execution_time_SX(dim,1)), ' seconds']);

disp('**** SIMPLES METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest)])
disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(x0
    ,1))])
disp(['Rate of Convergence: ', num2str(roc_SX(dim,1))])
disp('************************************')

if (failure)
    disp('FAIL')
    if (flag_bcktrck)
        disp('Failure due to backtracking')
    else
        disp('Failure not due to backtracking')
    end
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')


% if failure = true (failure == 1), the run was unsuccessful; otherwise
% failure = 0
failures_SX(dim,1) = failures_SX(dim,1) + failure;

for i = 1:10
    t1 = tic;
    [~,~,iter,fbest, ~, failure] = nelderMead(F_75,x0_rndgenerated(:,i),
       rho,chi,gamma,sigma,iter_max*size(x0,1),tol);
    execution_time_SX(dim,i+1) = toc(t1);
    fbest_SX(dim,i+1) = fbest;
    iter_SX(dim,i+1) = iter;
    failures_SX(dim,i+1) = failures_SX(dim,i+1) + failure;
    roc_SX(dim,i+1) = compute_roc(xseq, x_esatto);

    disp(['**** SIMPLEX METHOD FOR THE PB 75 (point ', num2str(i+1), ',
       dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_SX(dim,i+1)), ' seconds']);

    disp('**** SIMPLES METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(
       x0,1))])
    disp(['Rate of Convergence: ', num2str(roc_SX(dim,i+1))])
    disp('************************************')

    if (failure)
        disp('FAIL')
        if (flag_bcktrck)
            disp('Failure due to backtracking')
            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')

    end
end


varNames = ["average fbest", "average number of iterations", "average time
    of execution (sec)", "number of failures", "average rate of convergence
    "];
rowNames = string(dimension');
TSX = table(sum(fbest_SX,2)/11, sum(iter_SX,2)/11, sum(execution_time_SX,2)
    /11, sum(failures_SX,2),sum(roc_SX,2)/11 ,'VariableNames', varNames, '
    RowNames', rowNames);
format bank
display(TSX)

%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD
% with exact gradient and hessian
format short e

% setting the values for the dimension and the parameters
dimension = [1e3 1e4 1e5];
max_iter_per_dimension = [1e3, 1e4, 1e5];
tol = 1e-6;
param = [0.4, 1e-4, 36; 0.3, 1e-4, 28; 0.4, 1e-3, 36];


% initializing structures to store some stats
execution_time_MN = zeros(length(dimension),11);
failure_struct_MN = zeros(length(dimension),11); % number of failures
iter_struct_MN = zeros(length(dimension),11);
fbest_struct_MN = zeros(length(dimension),11);
gradf_struct_MN = zeros(length(dimension),11);
roc_struct_MN = zeros(length(dimension),11);
ultima_direz_discesa = zeros(length(dimension), 11);

for dim = 1:length(dimension)
    n = dimension(dim);
    iter_max = max_iter_per_dimension(dim);
    [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3));
    x_esatto = ones(n,1);

    %defining the given initial point
    x0 = -1.2*ones(n,1);
    x0(n) = -1;

    % in order to generate random number in [a,b] I apply the formula r = a
       + (b-a).*rand(n,1)
    rng(seed);
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);
```

```
% SOLVING MODIFIED NEWTON METHOD METHOD
% first initial point
t1 = tic;
[xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure,
   cos_pk_gradf] ...
    = modified_Newton(F_75,gradF_75, hessF_75, x0, iter_max, rho, c1,
        btmax, tol, [], 'ALG', x_esatto);
execution_time_MN(dim,1) = toc(t1);
fbest_struct_MN(dim,1) = fbest;
iter_struct_MN(dim,1) = iter;
gradf_struct_MN(dim,1) = gradfk_norm;
roc_struct_MN(dim,1) = compute_roc(xseq, x_esatto);
ultima_direz_discesa(dim,1) = cos_pk_gradf;
disp(['**** MODIFIED NEWTON METHOD FOR THE PB 75 (point ', num2str(1), '
   , dimension ', num2str(n), '):  *****']);

disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ', num2str(
   c1)]);

disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest)])
disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
disp('************************************')

if (failure)
    disp('FAIL')
    if (flag_bcktrck)
        disp('Failure due to backtracking')
        disp(['cosine of the angle between the last computed direction
           pk and the gradient: ', num2str(cos_pk_gradf)])
        disp(['last values of steplength alphak: ', mat2str((rho.^btseq(
           max(1:length(btseq)-10):end))')])
    else
        disp('Failure not due to backtracking')
    end
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

% if failure = true (failure == 1), the run was unsuccessful; otherwise
% failure = 0
failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;

for i = 1:10
    t1 = tic;
    [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
       , cos_pk_gradf] ...
        = modified_Newton(F_75,gradF_75, hessF_75, x0_rndgenerated(:,i),
            iter_max, rho, c1, btmax, tol, [], 'ALG', x_esatto);
    execution_time_MN(dim,i+1) = toc(t1);
    fbest_struct_MN(dim,i+1) = fbest;
    iter_struct_MN(dim,i+1) = iter;
    failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) + failure;
    gradf_struct_MN(dim,i+1) = gradfk_norm;
```

```
        roc_struct_MN(dim,i+1) = compute_roc(xseq, x_esatto);
        ultima_direz_discesa(dim,i+1) = cos_pk_gradf;

        disp(['**** MODIFIED NEWTON METHOD FOR THE PB 75 (point ', num2str(i
           +1), ', dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds']);
        disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
           num2str(c1)]);

        disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
        disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,i+1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            if (flag_bcktrck)
                disp('Failure due to backtracking')
                disp(['cosine of the angle between the last computed
                   direction pk and the gradient: ', num2str(cos_pk_gradf)])
                disp(['last values of steplength alphak: ', mat2str((rho.^
                   btseq(max(1:length(btseq)-10):end))')])


            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')
    end
end

% % plotto cos_pk_gradf
% bar(ultima_direz_discesa')
% ylabel('cos(angolo)')
% title('Ultimo valore assunto da t(pk)*gradfk/(norm_pk * norm_gradfk)')
% legend({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'}, "Box", 'on', 'Location', '
   best')
%

varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time of
   exec (sec)", "n failure", "avg roc"];
rowNames = string(dimension');
TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 ,sum(
   iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(failure_struct_MN
   ,2), sum(roc_struct_MN,2)/11,'VariableNames', varNames, 'RowNames',
   rowNames);
format bank
display(TMN)


%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD
% with approximated gradient and hessian
```

```
format short e

max_iter_per_dimension=[2*1e3, 2*1e4, 8*1e4];
tol = 1e-4;

% setting the values for the dimension
h_values = [1e-12]; %1e-2, 1e-4, 1e-6, 1e-8, 1e-10,
dimension = [1e3, 1e4, 1e5];

param = [0.8, 1e-5, 90; 0.8, 1e-5, 90; 0.8, 1e-5, 90];
type_h = 'COST';

% initializing structures to store some stats
execution_time_MN_h = zeros(length(dimension),6);
failure_struct_MN_h = zeros(length(dimension),6);
iter_struct_MN_h = zeros(length(dimension),6);
fbest_struct_MN_h = zeros(length(dimension),6);
gradf_struct_MN_h = zeros(length(dimension),6);
roc_struct_MN_h = zeros(length(dimension),6);


for id_h = 1:length(h_values)
    h = h_values(id_h);

    approx_gradF_75 = @(x) approxgradient_pb_75 (x,h,type_h);
    approx_hessF_75 = @(x) approxhessian_pb_75 (x,h,type_h);

    % initializing structures to store some stats
    execution_time_MN = zeros(length(dimension),11);
    failure_struct_MN = zeros(length(dimension),11); % number of failures
    iter_struct_MN = zeros(length(dimension),11);
    fbest_struct_MN = zeros(length(dimension),11);
    gradf_struct_MN = zeros(length(dimension),11);
    roc_struct_MN = zeros(length(dimension),11);
    ultima_direz_discesa = zeros(length(dimension), 11);


    for dim = 1:length(dimension)
        n = dimension(dim);
        x_esatto = ones(n,1);
        iter_max = max_iter_per_dimension(dim);
        [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3))
           ;

        %defining the given initial point
        x0 = -1.2*ones(n,1);
        x0(n) = -1;

        % in order to generate random number in [a,b] I apply the formula r
           = a + (b-a).*rand(n,1)
        rng(seed);
        x0_rndgenerated = zeros(n,10);
        x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);


        % SOLVING MODIFIED NEWTON METHOD METHOD
        % first initial point
        t1 = tic;
        [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
           , cos_pk_gradf] ...
            = modified_Newton(F_75,approx_gradF_75, approx_hessF_75, x0,
               iter_max, rho, c1, btmax, tol, [], 'ALG', x_esatto);
        execution_time_MN(dim,1) = toc(t1);
        fbest_struct_MN(dim,1) = fbest;
        iter_struct_MN(dim,1) = iter;
        gradf_struct_MN(dim,1) = gradfk_norm;
        roc_struct_MN(dim,1) = compute_roc(xseq,x_esatto);
        ultima_direz_discesa(dim,1) = cos_pk_gradf;
        disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, ' with
            h = ', num2str(h), ') FOR THE PB 75 (point ', num2str(1), ',
           dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
        disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
           num2str(c1)]);

        disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
        disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            if (flag_bcktrck)
                disp('Failure due to backtracking')
                disp(['cosine of the angle between the last computed
                   direction pk and the gradient: ', num2str(cos_pk_gradf)])
                disp(['last values of steplength alphak: ', mat2str((rho.^
                   btseq(max(1:length(btseq)-10):end))')])

            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')

        % if failure = true (failure == 1), the run was unsuccessful;
           otherwise
        % failure = 0
        failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;

        for i = 1:10
            t1 = tic;
            [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck,
               failure, cos_pk_gradf] ...
                = modified_Newton(F_75,approx_gradF_75, approx_hessF_75,
                   x0_rndgenerated(:,i), iter_max, rho, c1, btmax, tol, [],
                   'ALG', x_esatto);
            execution_time_MN(dim,i+1) = toc(t1);
            fbest_struct_MN(dim,i+1) = fbest;
            iter_struct_MN(dim,i+1) = iter;
            failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) +
               failure;
            gradf_struct_MN(dim,i+1) = gradfk_norm;
            roc_struct_MN(dim,i+1) = compute_roc(xseq,x_esatto);
```

```
            ultima_direz_discesa(dim,i+1) = cos_pk_gradf;

            disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, '
               with h = ', num2str(h), ') FOR THE PB 75 (point ', num2str(i
               +1), ', dimension ', num2str(n), '):  *****']);

            disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds'
               ]);
            disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
               num2str(c1)]);

            disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
            disp('************************************')
            disp(['f(xk): ', num2str(fbest)])
            disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
            disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)
               ])
            disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,i+1))])
            disp('************************************')

            if (failure)
                disp('FAIL')
                if (flag_bcktrck)
                    disp('Failure due to backtracking')
                    disp(['cosine of the angle between the last computed
                       direction pk and the gradient: ', num2str(
                       cos_pk_gradf)])
                    disp(['last values of steplength alphak: ', mat2str((rho
                       .^btseq(max(1:length(btseq)-10):end))')])

                else
                    disp('Failure not due to backtracking')
                end
                disp('************************************')
            else
                disp('SUCCESS')
                disp('************************************')
            end
            disp(' ')
        end
    end

% plotto cos_pk_gradf
bar(ultima_direz_discesa')
ylabel('cos(angolo)')
title('Ultimo valore assunto da t(pk)*gradfk/(norm_pk * norm_gradfk)')
legend({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'}, "Box", 'on', 'Location', '
   best')


    varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time
       of exec (sec)", "n failure", "avg roc"];
    rowNames = string(dimension');
    TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 ,sum(
       iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(
       failure_struct_MN,2), sum(roc_struct_MN,2)/11,'VariableNames',
       varNames, 'RowNames', rowNames);
    format bank
    display(TMN)

    execution_time_MN_h(:,id_h) = sum(execution_time_MN,2)/11;
    failure_struct_MN_h(:,id_h) = sum(failure_struct_MN,2);
    iter_struct_MN_h(:,id_h) = sum(iter_struct_MN,2)/11;
    fbest_struct_MN_h(:,id_h) = sum(fbest_struct_MN,2)/11;
    gradf_struct_MN_h(:,id_h) = sum(gradf_struct_MN,2)/11;
    roc_struct_MN_h(:,id_h) = sum(roc_struct_MN,2)/11;
end


%%
% I draw the graph for n=2
% Function definition for n=2
F = @(x1, x2) 0.5 * ((x1 - 1).^2 + (10 * (2 - 1) * (x2 - x1).^2).^2);

x1_range = linspace(-2, 2, 100);
x2_range = linspace(-2, 2, 100);
[X1, X2] = meshgrid(x1_range, x2_range);
Z = F(X1, X2);

figure;
surf(X1, X2, Z, 'EdgeColor', 'none');
colormap jet;
colorbar;
xlabel('x_1');
ylabel('x_2');
zlabel('F(x)');


%functions definition of exact gradient
function grad = gradient_pb_75 (x)
    n=length(x);
    grad=zeros(n,1);
    grad(1)= x(1)-1-200*(x(2)-x(1))^3;
    grad(2:n-1)= 200*((1:n-2)'.^2 .* (x(2:n-1)-x(1:n-2)).^3 - (2:n-1)'.^2 .*
        (x(3:n)-x(2:n-1)).^3 );
    grad(n)= 200*(n-1)^2*(x(n)-x(n-1))^3;
end

%function definition of exact hessian
function hess = hessian_pb_75(x)
    n=length(x);
    diag_princ=zeros(n,1); %d^2F/dx_k^2
    diag_princ(1)=1+600*(x(2)-x(1))^2;
    diag_princ(2:n-1)= 600* ((1:n-2)'.^2 .* (x(2:n-1) - x(1:n-2)).^2 + (2:n
       -1)'.^2 .* (x(3:n) - x(2:n-1)).^2 );
    diag_princ(n)= 600 * (n-1)^2 * (x(n)-x(n-1))^2;
    diag_upper=zeros(n,1);
    diag_upper(2:n)= -600* ( (1:n-1)'.^2 .* (x(2:n)-x(1:n-1)).^2 );
    diag_upper=diag_upper(2:n); %long n-1
    %matrix whose columns are the diags of the sparse hessian
    D=[[diag_upper;0], diag_princ, [0;diag_upper]];
    hess=spdiags(D,-1:1,n,n);
end

%function definition of approximated gradient (using forward differences)
function grad = approxgradient_pb_75 (x,h,type_h)
    n= length(x);
    grad= zeros(n,1);
    for i=1:n
        switch type_h
            case 'COST'
                hi = h;
            case 'REL'
                hi = h*abs(x(i));
        end
        if i==1
            grad(i) = ((x(1)+hi-1)^2 ...
                + (10*(x(2)-x(1)-hi)^2)^2 ...
                - (x(1)-1)^2 - (10*(x(2)-x(1))^2)^2)/(2*hi) ;
        elseif i<n
            %(F(x+he_i) - F(x))/hk =
            %f_i^2(x+he_i)+f_{i+1}^2(x+he_i)-f_i^2(x)-f_{i+1}^2(x)
            grad(i) = ((10*(i-1)*(x(i)+hi-x(i-1))^2)^2 ...
                + (10*(i)*(x(i+1)-x(i)-hi)^2)^2 ...
                - (10*(i-1)*(x(i)-x(i-1))^2)^2 - (10*i*(x(i+1)-x(i))^2)^2)
                  /(2*hi) ;
        else %i==n
            grad(i) = ((10*(n-1)*(x(n)+hi-x(n-1))^2)^2 - (10*(n-1)*(x(n)-x(n
               -1))^2)^2)/(2*hi);
        end

    end
end

%function definition of approximated hessian (using forward differences)
function hess = approxhessian_pb_75 (x,h,type_h)
    n = length(x);
    % In this case I know that the hessian is sparse (tridiagonal). The
    % total number of non-zero elements in principle is n + 2*(n-1) = 3*n-2
    indices_i = zeros(3*n-2, 1);
    indices_j = zeros(3*n-2, 1);
    values = zeros(3*n-2, 1);
    iter = 1;

    for k=1:n
        switch type_h
            case 'COST'
                hk = h;
                hkm1 = h;
            case 'REL'
                hk = h*abs(x(k));
                if k>1
                    hkm1 = h*abs(x(k-1));
                end

        end

        %diagonal element
        indices_i(iter) = k;
        indices_j(iter) = k;
        %(f(x+2*he_k)-2*f(x+he_k)+fx)/(hk^2)=(f_k^2(x+2he_k)+f_{k+1}^2(x+2
           he_k)
        %-2f_k^2(x+he_k)-2f_{k+1}^2(x+he_k)+f_k^2(x)+f_{k+1}^2(x))/(2hk^2)
        if k==1
            values(iter) = ((x(1)+2*hk-1)^2 + (10*(x(2)-x(1)-2*hk)^2)^2 ...
                -2*(x(1)+hk-1)^2 -2*(10*(x(2)-x(1)-hk)^2)^2 ...
                + (x(1)-1)^2 + (10*(x(2)-x(1))^2)^2 )/(2*hk^2);
            iter = iter+1;
        elseif k<n
            values(iter) = ((10*(k-1)*(x(k)+2*hk-x(k-1))^2)^2 + (10*(k)*(x(k
               +1)-x(k)-2*hk)^2)^2 ...
                -2*(10*(k-1)*(x(k)+hk-x(k-1))^2)^2 -2*(10*(k)*(x(k+1)-x(k)-
                  hk)^2)^2 ...
                + (10*(k-1)*(x(k)-x(k-1))^2)^2 + (10*(k)*(x(k+1)-x(k))^2)^2
                    )/(2*hk^2);
            iter = iter+1;
        else %k==n
            values(iter) = ((10*(n-1)*(x(n)+2*hk-x(n-1))^2)^2 ...
                -2*(10*(n-1)*(x(n)+hk-x(n-1))^2)^2 ...
                +(10*(n-1)*(x(n)-x(n-1))^2)^2 )/(2*hk^2);
            iter = iter+1;
        end

        %lower diagonal element (only exists if k>1)
        if k>1 && k<n
            indices_i(iter) = k;
            indices_j(iter) = k-1;
            values(iter) = ((10*(k-1)*(x(k)+hk - x(k-1)-hkm1)^2)^2 ...
                    + (10*k*(x(k+1)-x(k)-hk)^2)^2 ...
                    - (10*(k-1)*(x(k)+hk-x(k-1))^2)^2 ...
                    - (10*k*(x(k+1)-x(k)-hk)^2)^2 ...
                    - (10*(k-1)*(x(k)-x(k-1)-hkm1)^2)^2 ...
                    - (10*k*(x(k+1)-x(k))^2)^2 ...
                    + (10*(k-1)*(x(k)-x(k-1))^2)^2 ...
                    + (10*k*(x(k+1)-x(k))^2)^2)/(2*hk*hkm1);
            iter = iter+1;

            %for simmetry, upper diagonal element
            indices_i(iter) = k-1;
            indices_j(iter) = k;
            values(iter) = values(iter-1);
            iter = iter+1;

        elseif k==n
            indices_i(iter) = k;
            indices_j(iter) = k-1;
            values(iter) = ((10*(k-1)*(x(k)+hk - x(k-1)-hkm1)^2)^2 ...
                    - (10*(k-1)*(x(k)+hk-x(k-1))^2)^2 ...
                    - (10*(k-1)*(x(k)-x(k-1)-hkm1)^2)^2 ...
                    + (10*(k-1)*(x(k)-x(k-1))^2)^2)/(2*hk*hkm1);
            iter = iter+1;

            %for simmetry, upper diagonal element
            indices_i(iter) = k-1;
            indices_j(iter) = k;
            values(iter) = values(iter-1);
            iter = iter+1;
        end
    end
    hess = sparse(indices_i, indices_j, values, n, n);
end


%to compute the rate of convergence
function rate_of_convergence = compute_roc(xseq, x_esatto)
if size(xseq,2) >=4 && isempty(x_esatto)
    k = size(xseq,2) -1;
    norm_ekplus1 = norm(xseq(:, k+1) - xseq(:,k));
    norm_ek = norm(xseq(:, k) - xseq(:,k-1));
    norm_ekminus1 = norm(xseq(:, k-1) - xseq(:,k-2));
    rate_of_convergence = log(norm_ekplus1/norm_ek) / log(norm_ek/
        norm_ekminus1);
elseif size(xseq,2)>=3 && ~isempty(x_esatto)
    norm_ekplus1 = norm(x_esatto - xseq(:,end));
    norm_ek = norm(x_esatto - xseq(:,end-1));
    norm_ekminus1 = norm(x_esatto - xseq(:,end-2));
    rate_of_convergence = log(norm_ekplus1/norm_ek) / log(norm_ek/
       norm_ekminus1);
else
    rate_of_convergence = nan;
end
end
```

Problem 76

```
%% PROBLEMA 76
% ha minimo pari a 0 nell'origine
close all
clear all
clc

% setting the seed
seed = min(339268, 343310);

% function to compute the rate of convergence
function rate_of_convergence = compute_roc(xseq)
if size(xseq,2) >=4
    k = size(xseq,2) -1;
    norm_ekplus1 = norm(xseq(:, k+1) - xseq(:,k));
    norm_ek = norm(xseq(:, k) - xseq(:,k-1));
    norm_ekminus1 = norm(xseq(:, k-1) - xseq(:,k-2));
    rate_of_convergence = log(norm_ekplus1/norm_ek) / log(norm_ek/
       norm_ekminus1);
else
    rate_of_convergence = nan;
end
end


% implementing the function, the gradient and the hessiano for problem 76
function val = function_pb76(x)
    n = length(x);

    val = (x(n) - x(1)^2/10)^2;
    for k = 1:n-1
        val = val + (x(k) - x(k+1)^2/10)^2;
    end

    val = 0.5*val;

end

f = @(x) function_pb76(x);

function val = grad_pb76(x)
    n = length(x);

    val = zeros(n, 1);
    val(1,1) = (x(n) - x(1)^2/10) * (-0.2*x(1)) + (x(1) - x(2)^2/10);
    val(n, 1) = (x(n-1) - x(n)^2/10) *(-0.2*x(n)) + (x(n) - x(1)^2/10);

    for k =2:n-1
        val(k,1) = (x(k-1) - x(k)^2/10) * (-0.2*x(k)) + (x(k) - x(k+1)^2/10)
           ;
    end
end

gradf = @(x) grad_pb76(x);


function val = hessian_pb76(x)
    n = length(x);
    diags = zeros(n,5); %1st column is the principal diag, 2nd column is the
        superior diag and 3rd column is the inferior
    % principal diag
    diags(2:n,1) = -0.2*x(1:n-1) + 3/50 *x(2:n).^2 +1;
    diags(1,1) =  -0.2*x(n) + 3/50 *x(1)^2 +1;

    % inferior diagonal
    diags(1:n-1,3) = -0.2*x(2:n);

    %superior diagonal
    diags(2:n,2) =  -0.2*x(2:n);

    % 2-inf e 2-suo diag
    diags(1, 5) = -x(1)/5;
    diags(n, 4) =-x(1)/5;

    val = spdiags(diags, [0,1,-1, n-1, - (n-1)], n,n);
end


Hessf = @(x) hessian_pb76(x);

tol = 1e-4;
iter_max = 300;


%% RUNNING THE EXPERIMENTS ON NEALDER MEAD
format short e

% setting the dimensionality
dimension = [10 25 50];


% initializing the structures to store some stats
execution_time_SX = zeros(length(dimension),11);
failure_struct_SX = zeros(length(dimension),11); %for each dimension we
   count the number of failure
iter_struct_SX = zeros(length(dimension),11);
fbest_struct_SX = zeros(length(dimension),11);
roc_struct_SX = zeros(length(dimension),11);

for dim = 1:length(dimension)
    n = dimension(dim);

    % defining the given initial point
    x0 = 2*ones(n,1);

    % in order to generate random number in [a,b] I apply the formula r = a
       + (b-a).*rand(n,1)
    rng(seed);
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);

    % SOLVING SIMPLEX METHOD
    % first initial point
    t1 = tic;
    [~, xseq,iter,fbest, ~, failure] = nelderMead(f,x0,[],[],[],[],iter_max*
       size(x0,1),tol);
    execution_time_SX(dim,1) = toc(t1);
    fbest_struct_SX(dim,1) = fbest;
    iter_struct_SX(dim,1) = iter;
    roc_struct_SX(dim,1) = compute_roc(xseq);
    disp(['**** SIMPLEX METHOD FOR THE PB 76 (point ', num2str(1), ',
       dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_SX(dim,1)), ' seconds']);

    disp('**** SIMPLES METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(x0
       ,1))])
    disp(['Rate of Convergence: ', num2str(roc_struct_SX(dim,1))])
    disp('************************************')

    if (failure)
        disp('FAIL')
    else
        disp('SUCCESS')
        disp('************************************')
    end
    disp(' ')


    % if failure = true (failure == 1), the run was unsuccessful; otherwise
    % failure = 0
    failure_struct_SX(dim,1) = failure_struct_SX(dim,1) + failure;

    for i = 1:10
        t1 = tic;
        [~,~,iter,fbest, ~, failure] = nelderMead(f,x0_rndgenerated(:,i)
            ,[],[],[],[],iter_max*size(x0,1),tol);
        execution_time_SX(dim,i+1) = toc(t1);
        fbest_struct_SX(dim,i+1) = fbest;
        iter_struct_SX(dim,i+1) = iter;
        failure_struct_SX(dim,i+1) = failure_struct_SX(dim,i+1) + failure;
        roc_struct_SX(dim,i+1) = compute_roc(xseq);

        disp(['**** SIMPLEX METHOD FOR THE PB 76 (point ', num2str(i+1), ',
           dimension ', num2str(n), '):  *****']);

        disp(['Time: ', num2str(execution_time_SX(dim,i+1)), ' seconds']);

        disp('**** SIMPLES METHOD : RESULTS *****')
        disp('************************************')
        disp(['f(xk): ', num2str(fbest)])
        disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max*size(
           x0,1))])
        disp(['Rate of Convergence: ', num2str(roc_struct_SX(dim,1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')

    end
end
```

```
varNames = ["avg fbest", "avg num of iters", "avg time of exec (sec)", "n
   failure", "avg roc"];
rowNames = string(dimension');
TSX = table( round(sum(fbest_struct_SX,2)/11, 4), round(sum(iter_struct_SX
   ,2)/11, 4), round(sum(execution_time_SX,2)/11, 4), sum(failure_struct_SX
   ,2), round(sum(roc_struct_SX,2)/11, 4) ,'VariableNames', varNames, '
   RowNames', rowNames);
format short e
display(TSX)




%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD
format short e

iter_max = 5000;

% setting the values for the dimension
dimension = [1e3 1e4 1e5];

param = [0.4, 1e-4, 40; 0.3, 1e-4, 28; 0.4, 1e-3, 36];


% initializing structures to store some stats
execution_time_MN = zeros(length(dimension),11);
failure_struct_MN = zeros(length(dimension),11); %for each dimension we
   count the number of failure
iter_struct_MN = zeros(length(dimension),11);
fbest_struct_MN = zeros(length(dimension),11);
gradf_struct_MN = zeros(length(dimension),11);
roc_struct_MN = zeros(length(dimension),11);
ultima_direz_discesa = zeros(length(dimension), 11);

for dim = 1:length(dimension)
    n = dimension(dim);

    [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3));


    %defining the given initial point
    x0 = 2*ones(n,1);

    % in order to generate random number in [a,b] I apply the formula r = a
       + (b-a).*rand(n,1)
    rng(seed);
    x0_rndgenerated = zeros(n,10);
    x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);


    % SOLVING MODIFIED NEWTON METHOD METHOD
    % first initial point
    t1 = tic;
    [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure,
       pk_scalare_gradf] = modified_Newton(f,gradf, Hessf, x0, iter_max, rho
       , c1, btmax, tol, [], 'ALG', 0);
    execution_time_MN(dim,1) = toc(t1);
    fbest_struct_MN(dim,1) = fbest;
    iter_struct_MN(dim,1) = iter;
    gradf_struct_MN(dim,1) = gradfk_norm;
    roc_struct_MN(dim,1) = compute_roc(xseq);
```

```
ultima_direz_discesa(dim,1) = pk_scalare_gradf;

disp(['**** MODIFIED NEWTON METHOD FOR THE PB 76 (point ', num2str(1), '
    , dimension ', num2str(n), '):  *****']);

disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ', num2str(
    c1)]);

disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
disp('************************************')
disp(['f(xk): ', num2str(fbest)])
disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
disp('************************************')

if (failure)
    disp('FAIL')
    if (flag_bcktrck)
        disp('Failure due to backtracking')
    else
        disp('Failure not due to backtracking')
    end
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

% if failure = true (failure == 1), the run was unsuccessful; otherwise
% failure = 0
failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;

for i = 1:10
    t1 = tic;
    [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
        , pk_scalare_gradf] = modified_Newton(f,gradf, Hessf,
        x0_rndgenerated(:,i), iter_max, rho, c1, btmax, tol, [], 'ALG',
        0);
    execution_time_MN(dim,i+1) = toc(t1);
    fbest_struct_MN(dim,i+1) = fbest;
    iter_struct_MN(dim,i+1) = iter;
    failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) + failure;
    gradf_struct_MN(dim,i+1) = gradfk_norm;
    roc_struct_MN(dim,i+1) = compute_roc(xseq);
    ultima_direz_discesa(dim,i+1) = pk_scalare_gradf;

    disp(['**** MODIFIED NEWTON METHOD FOR THE PB 76 (point ', num2str(i
        +1), ', dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds']);
    disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
        num2str(c1)]);

    disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
```

```
        disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
        disp('************************************')

        if (failure)
            disp('FAIL')
            if (flag_bcktrck)
                disp('Failure due to backtracking')
            else
                disp('Failure not due to backtracking')
            end
            disp('************************************')
        else
            disp('SUCCESS')
            disp('************************************')
        end
        disp(' ')
    end
end

% plotto pk_scalar_gradfk
bar(ultima_direz_discesa')
ylabel('cos(angolo)')
title('Ultimo valore assunto da t(pk)*gradfk/(norm_pk * norm_gradfk)')
legend({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'}, "Box", 'on', 'Location', '
   best')



varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time of
   exec (sec)", "n failure", "avg roc"];
rowNames = string(dimension');
TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 , sum(
   iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(failure_struct_MN
   ,2), sum(roc_struct_MN,2)/11,'VariableNames', varNames, 'RowNames',
   rowNames);
format short e
display(TMN)




%% FINITE DIFFERENCES
clc

function grad_approx = findiff_grad_76(x, h, type_h)
    %    type h: indica se derivata calcolata con h costante o h
    %   relativo
    n = length(x);

    if isempty(type_h)
        type_h = 'COST';
    end

    switch type_h
        case 'COST'
            passo1 = h;
            passok = h;
            passon = h;
        case 'REL'
            passo1 = h*abs(x(1));
            passon = h*abs(x(n));
    end

    grad_approx = zeros(n,1);
    grad_approx(1,1) =  passo1*(4*x(1)  - 2/5 * x(2)^2 + (8*x(1)*(x(1)^2+
       passo1^2))/100 - 4/5 * x(n)*x(1))/(4*passo1);

    for k = 2:n-1
        if strcmp('REL', type_h)
            passok = h*abs(x(k));
        end
        grad_approx(k,1) = passok*(4*x(k)  - 2/5 * x(k+1)^2 + (8*x(k)*(x(k)
           ^2+passok^2))/100 - 4/5 * x(k-1)*x(k))/(4*passok);
    end
    grad_approx(n,1) = passon*(4*x(n)  - 2/5 * x(1)^2 + (8*x(n)*(x(n)^2+
       passon^2))/100 - 4/5 * x(n-1)*x(n))/(4*passon);

end


function hessian_approx = findiff_hess_76(x, h, type_h)
    % Calcola la matrice Hessiana sparsa per la funzione f(x)
    % Input:
    %   - x: vettore colonna (punto in cui calcolare l'Hessiana)
    %   - h: passo o vettore per la differenzaz rispetto ad una componente
    %   - type h: indica se derivata calcolata con h costante o h
    %   relativo
    % Output:
    %   - H: matrice Hessiana sparsa

    n = length(x); % Dimensione del problema

    if isempty(type_h)
        type_h = 'COST';
    end

    % Preallocazione per la struttura sparsa
    i_indices = zeros(3*n,1);
    j_indices = zeros(3*n,1);
    values = zeros(3*n,1);
    cont = 1;

    % Loop su k (dalla definizione della funzione)
    for k = 1:n
        % Elementi diagonali H(k, k)
        if k == 1
            switch type_h
                case 'REL'
                    passok = h*abs(x(k));
                    passok1 = h*abs(x(k+1));
                case 'COST'
                    passok = h;
                    passok1 = h;
            end
            % H_kk = (fn_quadro(x+2*he_k) + fk_quadro(x+2*he_k, 1) - 2*
               fn_quadro(x+he_k) -2*fk_quadro(x+he_k,1) + fn_quadro(x) +
               fk_quadro(x,1))/(2*h^2);
            H_kk = (2*passok^2 - 2/5 * x(n)*passok^2+ 0.12 * x(k)^2*passok
               ^2+ 0.24 * x(k)*passok^3 +0.14 * passok^4)/(2*passok^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;
        elseif k < n
            switch type_h
                case 'REL'
                    passok = h*abs(x(k));
                    passok1 = h*abs(x(k+1));
                case 'COST'
                    passok = h;
                    passok1 = h;
            end
            % H_kk = (fk_quadro(x+2*he_k, k-1) + fk_quadro(x+2*he_k, k) -
               2*fk_quadro(x+he_k, k-1) -2*fk_quadro(x+he_k,k) + fk_quadro(x
               ,k-1) + fk_quadro(x,k))/(2*h^2);
            H_kk = (2*passok^2 - 2/5 * x(k-1)*passok^2+ 0.12 * x(k)^2*passok
               ^2+ 0.24 * x(k)*passok^3 +0.14 * passok^4)/(2*passok^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;
        else
            switch type_h
                case 'REL'
                    passok = h*abs(x(n));
                    passok1 = h*abs(x(1));
                case 'COST'
                    passok = h;
                    passok1 = h;
            end
            % H_kk = (fk_quadro(x+2*he_k, k-1) + fn_quadro(x+2*he_k) -  2*
               fk_quadro(x+he_k, k-1) -2*fn_quadro(x+he_k) + fk_quadro(x,k
               -1) + fn_quadro(x))/(2*h^2);
            H_kk = (2*passok^2 - 2/5 * x(n-1)*passok^2+ 0.12 * x(k)^2*passok
               ^2+ 0.24 * x(k)*passok^3 +0.14 * passok^4)/(2*passok^2);
            i_indices(cont) = k;
            j_indices(cont) = k;
            values(cont) = H_kk;
            cont = cont +1;
        end

        % Elementi fuori diagonale H(k, k+1)
        if k < n
            % H_k_k1 = (fk_quadro(x+he_k1 +he_k,k) - fk_quadro(x+he_k, k) -
               fk_quadro(x+he_k1,k) - fk_quadro(x, k))/(2*h^2);
            H_k_k1 = (-2/5 *passok*passok1*x(k+1) - 1/5 * passok1^2*passok1)
               /(2*passok*passok1);
            i_indices(cont) = k;
            j_indices(cont) = k+1;
            values(cont) = H_k_k1;
            cont = cont +1;

            % impongo la simmetria
            i_indices(cont) = k+1;
            j_indices(cont) = k;
            values(cont) = H_k_k1;
            cont = cont+1;
        else
            % Caso circolare: H(n, 1)
            H_n1 = (-2/5 *passok*passok1*x(1) - 1/5 * passok1^2*passok1)/(2*
               passok*passok1);
            i_indices(cont) = 1;
            j_indices(cont) = n;
```

```
            values(cont) = H_n1;
            cont = cont +1;

            % impongo la simmetria
            i_indices(cont) = n;
            j_indices(cont) = 1;
            values(cont) = H_n1;
            cont = cont +1;
        end

    end

    % Creazione della matrice Hessiana sparsa
    hessian_approx = sparse(i_indices, j_indices, values, n, n);
end

h = 1e-10;
type_h = 'REL';
gradf_approx = @(x) findiff_grad_76(x,h, type_h);
Hessf_approx = @(x) findiff_hess_76(x,h, type_h);

vec = 0.5*ones(7,1);
vec = [0.2; 0.4; -0.2; 0.5; 0.1; -1; 0.1];

gradf(vec)
gradf_approx(vec)

full(Hessf(vec))
tic
full(Hessf_approx(vec))
time = toc


%% RUNNING THE EXPERIMENTS ON MODIFIED NEWTON METHOD WITH FIN DIFF
format short e
clc

iter_max = 5000;
tol = 1e-4;



% setting the values for the dimension
h_values = [1e-2 1e-4 1e-6 1e-8 1e-10 1e-12];
dimension = [1e3 1e4 1e5];
param = [0.4, 1e-4, 40; 0.3, 1e-4, 28; 0.4, 1e-3, 36];
type_h = 'REL';

tables = struct;

% initializing structures to store some stats
execution_time_MN_h = zeros(length(dimension),6);
failure_struct_MN_h = zeros(length(dimension),6);
iter_struct_MN_h = zeros(length(dimension),6);
fbest_struct_MN_h = zeros(length(dimension),6);
gradf_struct_MN_h = zeros(length(dimension),6);
roc_struct_MN_h = zeros(length(dimension),6);


for id_h = 1:length(h_values)
```

```
h = h_values(id_h);

 gradf_approx = @(x) findiff_grad_76(x,h, type_h);
 hessf_approx = @(x) findiff_hess_76(x,h, type_h);

 % initializing structures to store some stats
 execution_time_MN = zeros(length(dimension),11);
 failure_struct_MN = zeros(length(dimension),11); %for each dimension we
    count the number of failure
 iter_struct_MN = zeros(length(dimension),11);
 fbest_struct_MN = zeros(length(dimension),11);
 gradf_struct_MN = zeros(length(dimension),11);
 roc_struct_MN = zeros(length(dimension),11);


 for dim = 1:length(dimension)
     n = dimension(dim);

     [rho, c1, btmax] = deal(param(dim, 1), param(dim, 2), param(dim, 3))
        ;


     %defining the given initial point
     x0 = 2*ones(n,1);

     % in order to generate random number in [a,b] I apply the formula r
        = a + (b-a).*rand(n,1)
     rng(seed);
     x0_rndgenerated = zeros(n,10);
     x0_rndgenerated(1:n, :) = x0(1:n) - 1 + 2.*rand(n,10);


     % SOLVING MODIFIED NEWTON METHOD METHOD
     % first initial point
     t1 = tic;
     [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck, failure
        ] = modified_Newton(f,gradf_approx, hessf_approx, x0, iter_max,
        rho, c1, btmax, tol, [], 'ALG', 0);
     execution_time_MN(dim,1) = toc(t1);
     fbest_struct_MN(dim,1) = fbest;
     iter_struct_MN(dim,1) = iter;
     gradf_struct_MN(dim,1) = gradfk_norm;
     roc_struct_MN(dim,1) = compute_roc(xseq);
     disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, ' with
         h = ', num2str(h), ') FOR THE PB 76 (point ', num2str(1), ',
        dimension ', num2str(n), '):  *****']);

     disp(['Time: ', num2str(execution_time_MN(dim,1)), ' seconds']);
     disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
        num2str(c1)]);

     disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
     disp('************************************')
     disp(['f(xk): ', num2str(fbest)])
     disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
     disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)])
     disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
     disp('************************************')

     if (failure)
         disp('FAIL')
    if (flag_bcktrck)
        disp('Failure due to backtracking')
    else
        disp('Failure not due to backtracking')
    end
    disp('************************************')
else
    disp('SUCCESS')
    disp('************************************')
end
disp(' ')

% if failure = true (failure == 1), the run was unsuccessful;
   otherwise
% failure = 0
failure_struct_MN(dim,1) = failure_struct_MN(dim,1) + failure ;

for i = 1:10
    t1 = tic;
    [xbest, xseq, iter, fbest, gradfk_norm, btseq, flag_bcktrck,
       failure] = modified_Newton(f,gradf_approx, hessf_approx,
       x0_rndgenerated(:,i), iter_max, rho, c1, btmax, tol, [], 'ALG
       ', 0);
    execution_time_MN(dim,i+1) = toc(t1);
    fbest_struct_MN(dim,i+1) = fbest;
    iter_struct_MN(dim,i+1) = iter;
    failure_struct_MN(dim,i+1) = failure_struct_MN(dim,i+1) +
       failure;
    gradf_struct_MN(dim,i+1) = gradfk_norm;
    roc_struct_MN(dim,i+1) = compute_roc(xseq);

    disp(['**** MODIFIED NEWTON METHOD WITH FIN DIFF ( ', type_h, '
       with h = ', num2str(h), ') FOR THE PB 76 (point ', num2str(i
       +1), ', dimension ', num2str(n), '):  *****']);

    disp(['Time: ', num2str(execution_time_MN(dim,i+1)), ' seconds'
       ]);
    disp(['Backtracking parameters (rho, c1): ', num2str(rho), ' ',
       num2str(c1)]);

    disp('**** MODIFIED NEWTON METHOD : RESULTS *****')
    disp('************************************')
    disp(['f(xk): ', num2str(fbest)])
    disp(['norma di gradf(xk): ', num2str(gradfk_norm)])
    disp(['N. of Iterations: ', num2str(iter),'/',num2str(iter_max)
       ])
    disp(['Rate of Convergence: ', num2str(roc_struct_MN(dim,1))])
    disp('************************************')

    if (failure)
        disp('FAIL')
        if (flag_bcktrck)
            disp('Failure due to backtracking')
        else
            disp('Failure not due to backtracking')
        end
        disp('************************************')
    else
        disp('SUCCESS')
        disp('************************************')
    end
```

```
            disp(' ')
        end
    end



    varNames = ["avg fbest", "avg gradf_norm","avg num of iters", "avg time
       of exec (sec)", "n failure", "avg roc"];
    rowNames = string(dimension');
    TMN = table(sum(fbest_struct_MN,2)/11, sum(gradf_struct_MN,2)/11 ,sum(
       iter_struct_MN,2)/11, sum(execution_time_MN,2)/11, sum(
       failure_struct_MN,2), sum(roc_struct_MN,2)/11,'VariableNames',
       varNames, 'RowNames', rowNames);
    format short e
    display(TMN)

    tables.(['Table' num2str(id_h)]) = TMN;

    execution_time_MN_h(:,id_h) = sum(execution_time_MN,2)/11;
    failure_struct_MN_h(:,id_h) = sum(failure_struct_MN,2);
    iter_struct_MN_h(:,id_h) = sum(iter_struct_MN,2)/11;
    fbest_struct_MN_h(:,id_h) = sum(fbest_struct_MN,2)/11;
    gradf_struct_MN_h(:,id_h) = sum(gradf_struct_MN,2)/11;
    roc_struct_MN_h(:,id_h) = sum(roc_struct_MN,2)/11;

end
```