POLITECNICO DI TORINO

Master's degree
Data Science and Engineering

Numerical Optimization for Large Scale Problems

Implementation and validation of Nelder-Mead method and Modified Newton method

Giuseppe Mallo 346884

Academic Year 2024-2025

Indice

Chapter 1

Nelder-Mead Method

The Nelder-Mead method is a numerical optimization algorithm used to minimize functions without constraints. It is particularly useful for functions that are not differentiable or whose properties are unknown, as it does not require gradient computation. The method operates on a geometrical structure called simplex, which consists of $n + 1$ points in a n-dimensional space. Through iterative steps, the simplex is adjusted to move closer to the minimum point of the objective function.

1.1 How the Method Works

The Nelder-Mead method is structured into four possible phases: reflection, expansion, contraction, and shrinkage. At each iteration, the algorithm applies one or more of these steps (following the order), depending on the function values obtained, rather than necessarily executing all four. The process is repeated until convergence to a solution or until the maximum number of iterations is reached.

The method uses several control parameters for each distinct phase:

- ρ: used in the reflection phase. It determines how far the worst point is reflected.
- χ: used in the expansion phase. It allows the simplex to explore promising areas.
- γ: used in the contraction phase. It reduces the size of the simplex when necessary.
- σ: used in the shrinkage phase. It compresses the simplex in extreme cases.

Initialization

The initial simplex is built around a starting point x_0 and each vertex of the simplex is generated as:

$$x_1 = x_0,$$

$$x_i = x_0 + e_{i-1}, \quad i = 2, \dots, n+1$$

where x_0 is the initial point, and e_i the i-th vector of the canonical basis of $\mathbb{R}^n$.

Function Evaluation

We then proceed with the evaluation of the function. The function values are used to determine the quality of the vertices.

Sorting

The vertices are indexed according to their function values, such that

$$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1}),$$

where x_1 denotes the vertex with the best (smallest) function value, and x_{n+1} the one with the worst.

Reflection

This method starts with the first phase, which is the Reflection phase. First, the baricenter of the best n points, excluding the worst one (which is x_{n+1}), is computed as follows:

$$x_{bar} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Then, the worst point is reflected with respect to the baricenter:

$$x_r = x_{bar} + \rho(x_{bar} - x_{n+1})$$

where $\rho > 0$ is the reflection coefficient. In the code $\rho = 1$.

If $f(x_1) <= f(x_r) < f_n$, the method accepts x_r. But, if $f(x_r) < f_1$, the method proceeds to the expansion phase.

Expansion

The algorithm attempts to expand the simplex:

$$x_e = x_{bar} + \chi(x_r - x_{bar})$$

where $\chi > 1$; in the code χ is set to 2.

If $f(x_e) < f(x_r)$, x_e is accepted; otherwise, x_r is used and will substitute the worst point x_{n+1} for the next simplex.

Contraction

If reflection does not yield a good improvement (i.e. $f(x_r) >= f_n$), contraction is performed and it can be of two types, depending on the best point among x_{n+1} and x_r:

- If x_r is the best: $x_c = x_{bar} - \gamma(x_{bar} - x_r)$
- If x_{n+1} is the best: $x_c = x_{bar} - \gamma(x_{bar} - x_{n+1})$

where $0 < \gamma < 1$; in this case $\gamma = 0.5$.

If $f(x_c) < f_{n+1}$, x_c is accepted; otherwise, shrinkage is applied.

Shrinkage

If all the previous steps fail, the simplex is shrunk toward the best point:

$$x_i = x_1 + \sigma(x_i - x_1), \quad i = 2, \ldots, n+1$$

where $0 < \sigma < 1$ and here $\sigma = 0.5$. After this transformation, the new simplex vertices are re-evaluated, and then re-ordered according to their function values, i.e.

$$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1}).$$

Stopping criterion

The algorithm checks for convergence by evaluating the variation in function values. If

$$\max_i \Big(|f_i - \bar{f}| \Big) < \text{tol}$$

or the maximum number of iterations is reached, the algorithm stops.
Here, $\bar{f}$ is the mean of the function values obtained on the current simplex.

Results

At the end, the method returns:

- x^*: the best point found (approximate minimum point)
- f^*: the function value at that approximated minimum point
- The number of iterations performed (iter)
- The evolution of the simplexes over iterations

The Nelder-Mead method is a powerful and flexible algorithm for unconstrained optimization. Its simplicity makes it ideal for a wide range of applications, especially when some function information (such as derivatives) are unknown. However, for high-dimensional or complex problems, its performance might degrade, and alternative optimization techniques should be considered.

Chapter 2

Modified Newton Method

The Newton method is a numerical algorithm used in this report to solve unconstrained optimization problems, aiming to minimize a function $f : \mathbb{R}^n \to \mathbb{R}$. The main strength of this method lies in the use of both the gradient and the Hessian matrix of the objective function to compute descent directions that can be more effective than those ones based solely on the gradient.

While the Newton method demonstrates quadratic convergence near the solution, it has limitations in its standard form:

1. Positive definiteness of the Hessian matrix: the descent direction is only valid if the Hessian matrix $H(x)$ is positive definite; otherwise, the algorithm might diverge or fail.
2. Step length selection: taking steps that are too large can result in instability, while steps that are too small slow down convergence.

The modified Newton method overcomes these problems by:

1. Verifying the positive definiteness of the Hessian matrix and modifying it when necessary.
2. Using a backtracking strategy to select the step length, improving stability and convergence.

2.1 How the Modified Newton Method Works

Initialization

The following parameters are set at the beginning:

- x_0: the initial point provided by the user
- $f(x)$: the objective function to be minimized
- $\nabla f(x)$: the gradient of $f(x)$

- $H(x)$: the Hessian matrix
- $k_{\max}$: the maximum number of iterations
- tol_{grad}: the tolerance of the stopping criterion based on the gradient norm

Digression on the Newton Method

The standard Newton method computes the next iterate x_{k+1} using the formula:

$$x_{k+1} = x_k + \alpha_k p_k$$

where:

- p_k is the descent direction, computed as the solution of $H_k p_k = -\nabla f(x_k)$,
- α_k is the step length

This standard method assumes that the Hessian H_k is always positive definite, which ensures that p_k points in a direction that decreases the objective function.

Verification and Modification of the Hessian

If the Hessian matrix is not positive definite, p_k may not represent a descent direction. For this reason, at each iteration, the Hessian matrix H_k is checked to ensure that it is positive definite.

Checking Positive Definiteness

The verification is performed using the Cholesky factorization. If the factorization succeeds, H_k is positive definite. Otherwise, it needs to be modified.

Modifying the Hessian

To adjust H_k at step k and make it positive definite, we use the function `check_positive_definiteness`. If necessary, it computes a modified matrix:

$$B_k = H_k + \tau I$$

where τ is a positive value progressively increased until the Cholesky factorization succeeds.

In other words, a small positive value τ is added to the diagonal of H_k, and the factorization is attempted again. If the modification still fails, τ is progressively increased until the factorization succeeds or τ exceeds a predefined maximum. This approach ensures that the modified Hessian B_k can always be used to compute a valid descent direction, without drastically altering the inherent properties of the original Hessian.

Computing the Descent Direction

If the Hessian matrix is modified (its modified version is B_k), the descent direction p_k is computed by solving the linear system:

$$B_k p_k = -\nabla f(x_k)$$

Otherwise, the linear system is the same of the one mentioned above:

$$H_k p_k = -\nabla f(x_k)$$

This ensures that p_k is a valid direction for improving the objective function.

Backtracking Strategy

The step length α_k is determined using a backtracking strategy, which satisfies the Armijo condition:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^T p_k$$

If this condition is not satisfied, α is reduced by multiplying it by a factor $\rho \in (0,1)$, until the Armijo condition is satisfied or a maximum number of backtracks bt_{max} is reached.

Update

After determining α_k, the new point is updated as:

$$x_{k+1} = x_k + \alpha_k p_k$$

The function value, gradient, and gradient norm are also updated.

Stopping Criterion

The method stops when:

- The gradient norm $\|\nabla f(x_k)\|$ is below the threshold tol, or
- The maximum number of iterations $k = k_{\text{max}}$ is reached.

Results

At the end, the method returns:

- x_k: the point found at the end of the optimization.
- f_k: the value of the objective function at x_k.
- $\|\nabla f(x_k)\|$: the gradient norm at the optimal point.

- k: the number of iterations performed.
- the sequence of visited points.
- bt_{seq}: the number of backtracking iterations for each step.

The verification and modification of the Hessian ensure valid descent directions, while backtracking provides stable and effective step lengths. Despite the additional computational cost (calculating and modifying the Hessian can be expensive, especially for high-dimensional problems), these improvements make the method highly reliable for well-defined optimization problems and provide a reliable and efficient approach for solving unconstrained optimization problems, particularly when the objective function exhibits strong curvature

Chapter 3

Testing Nelder Mead and Modified Newton Method on Rosenbrock function

In this part of the assignment, we are asked to test our codes on the Rosenbrock function, defined as:

$$f(x) = 100 \cdot (x_2 - x_1^2)^2 + (1 - x_1)^2$$

It is a well-known reference test for optimization algorithms and represents a significant test for numerical methods. This function features a narrow and curved valley leading to the global minimum located at $x^* = (1, 1)$. The strong nonlinearity and the steep gradient make this function particularly challenging to minimize.

This report analyzes the results obtained by applying the Nelder-Mead and the Modified Newton methods to the Rosenbrock function. The two starting points used for testing are:

- $x_1^{(0)} = (1.2, 1.2)$
- $x_2^{(0)} = (-1.2, 1)$

The goal is to compare the performance of these methods in terms of iterations, accuracy, and behavior when applied to this test function.

Nelder-Mead Method

The Nelder-Mead method is a derivative-free optimization algorithm that uses a geometric simplex composed of $n + 1$ points in n-dimensional space. The simplex is iteratively updated.

The Nelder-Mead method was applied starting from the two initial points, yielding the following simulated results:

Initial Point	**Iterations**	**Execution Time (s)**	f_{opt}	**Failure**
$x_1^{(0)} = (1.2, 1.2)$	52	0.197647	$1.847012960248056 \times 10^{-13}$	No
$x_2^{(0)} = (-1.2, 1)$	111	0.021016	$3.929078942335394 \times 10^{-14}$	No

Table 3.1: Results of the Nelder-Mead method applied to the Rosenbrock function.

Analysis

Case 1: $x_1^{(0)} = (1.2, 1.2)$

The algorithm quickly converges to the global minimum point (1, 1) on which the value assumed by the function is 0.

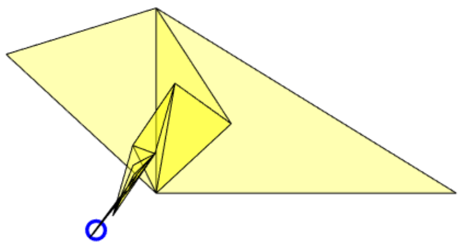


Figure 3.1: Evolution of the simplex across successive iterations starting from $x_1^{(0)}$

Figure (3.1) illustrates the evolution of the simplex, showing how the simplex (represented by the yellow triangles) progressively adapts to the minimum point.

Figure (3.2) displays the 3D surface of the Rosenbrock function

Figure (3.3) represents the evolution of the experimental rate of convergence, starting from the initial point $x_1^{(0)}$. The convergence begins with some noticeable variations; this suggests significant corrections in the approximation during the early iterations as the algorithm attempts to move closer to the optimal region.

After the initial adjustment, the sequence stabilizes, indicating that the Nelder-Mead algorithm has reached a promising region near the minimum.

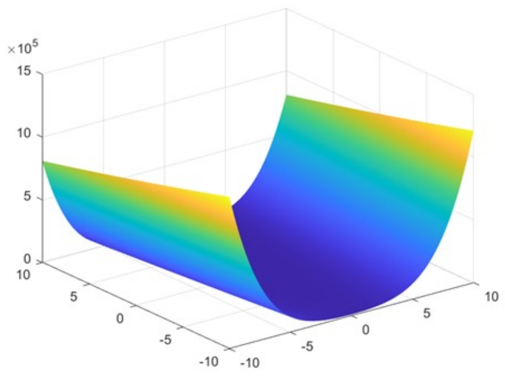


Figure 3.2

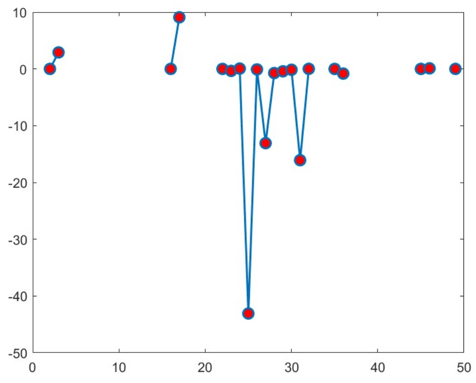


Figure 3.3

Case 2: $x_2^{(0)} = (-1.2, 1)$

Starting further from the minimum, the algorithm requires more iterations and time to converge.

Figure (3.4) depicts the simplex evolution, demonstrating its ability to navigate the complex landscape of the Rosenbrock function.

The graph in Figure (3.5) shows a different behaviour of the experimental rate; the first iteration displays a sharp increase in the error, indicating that the starting point $x_2^{(0)}$ is farther from the optimal region than $x_1^{(0)}$.
The algorithm needs to explore more extensively at the beginning.

Similar to the first case, the algorithm converges over subsequent iterations; however, stabilization begins slightly later, likely due to an unfavorable initial guess. This shows that a good starting point can significantly influence optimization performance.

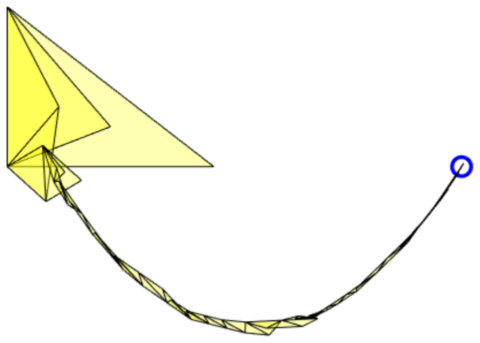


Figure 3.4: Evolution of the simplex across successive iterations starting from $x_2^{(0)}$

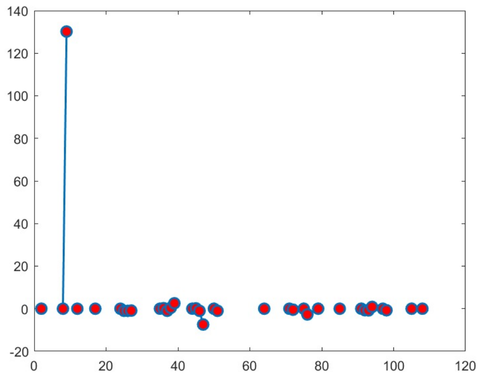


Figure 3.5

Modified Newton Method

The Modified Newton method uses both the gradient and the Hessian matrix to compute the descent direction p_k. The Modified Newton method was tested on the same initial points, producing the following simulated results:

Initial Point	**Iterations**	**Execution Time (s)**	f_{opt}	**grad_norm**	**Failure**
$x_1^{(0)} = (1.2, 1.2)$	8	0.035430	1.0882×10^{-25}	1.4360×10^{-11}	No
$x_2^{(0)} = (-1.2, 1)$	21	0.010747	3.7440×10^{-21}	4.4733×10^{-10}	No

Table 3.2: Results of optimization for two initial points.

Analysis

Case 1: $x_1^{(0)} = (1.2, 1.2)$

The algorithm converges in a very small number of iterations due to the precision of the descent direction calculation.

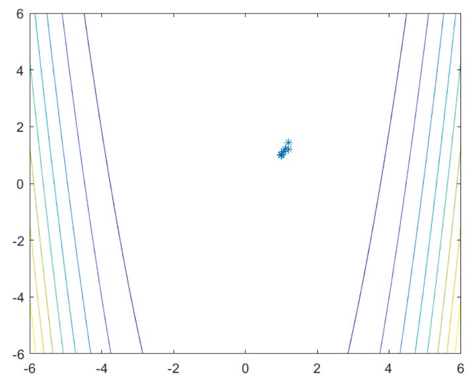


Figure 3.6

Figure (3.6) shows the optimization path on the contour plot; this graph shows the level curves of the function being minimized, with the trajectory of the optimization process starting from $x_1^{(0)}$. The trajectory indicates how the method progresses toward the minimum of the function. The iterates remain clustered near the solution, suggesting that the method converges efficiently and the corrections made at each step keep the optimization path stable.

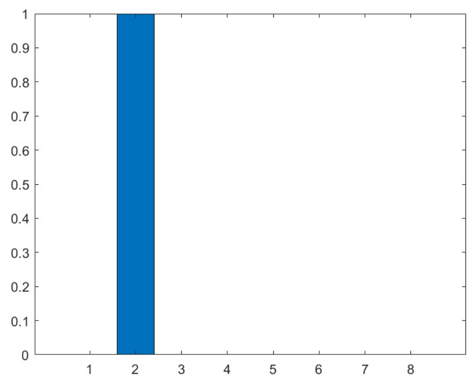


Figure 3.7

Figure (3.7) presents a bar graph representing the number of backtracking iterations required at each step. In particular, only one backtracking is needed at iteration 2, i.e., at

$iter = 2$, the backtracking was applied to ensure the step length met the required Armijp condition. For all subsequent iterations, no further backtracking was necessary.

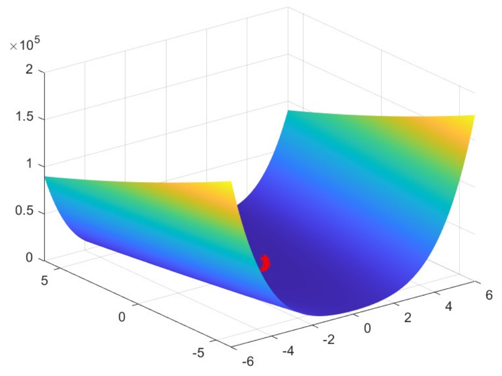


Figure 3.8

Figure (3.8) is a graph that represents the objective function on a three-dimensional surface. The red points (iterations) move toward the bottom of the valley, which corresponds to the minimum of the function.

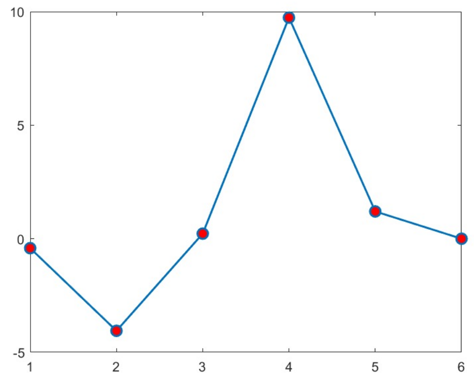


Figure 3.9: Caption

In Figure (3.9), we can see the behaviour of the experimental rate of convergence; the trajectory indicates that, starting from the initial point $x_1^{(0)}$, the modified method converges quickly after an initial phase of oscillations.

The convergence stabilizes after a few iterations, suggesting the method is effective for this starting point.

Case 2: $x_2^{(0)} = (-1.2, 1)$

Although the initial point is further from the minimum point, the algorithm converges efficiently, with some backtracks executed in some steps, as we see in Figure (3.10). The method converges with more iterations than the previous case, but it is very fast.

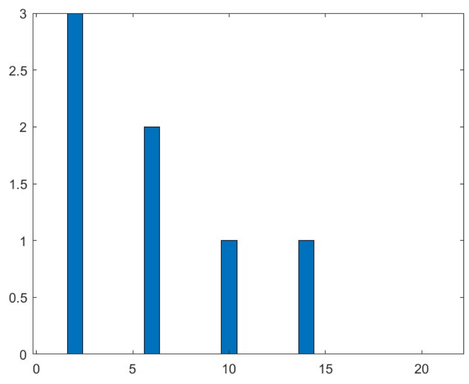


Figure 3.10: Caption

Figures (3.11) and (3.12) highlight the trajectory toward the minimizer and the corresponding 3D visualization.

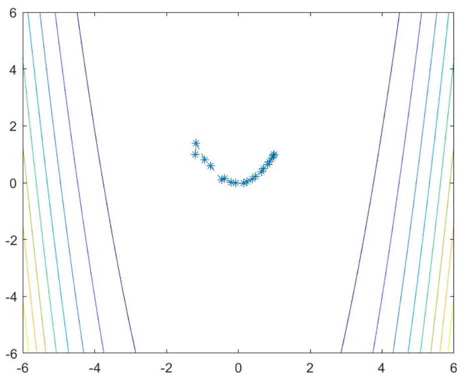


Figure 3.11

In this case in Figure (3.13) a similar behaviour of the rate of convergence of the previous case is observed, but with even faster convergence toward a steady state.

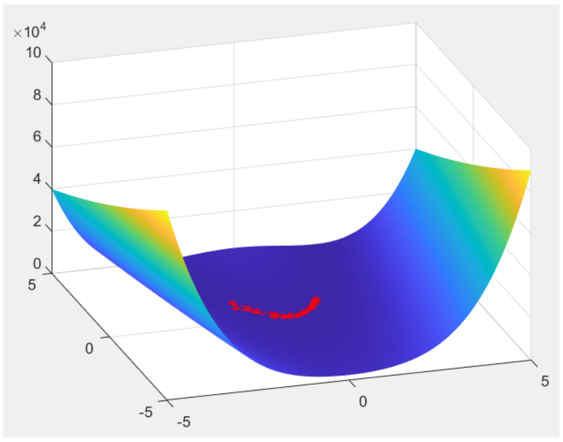


Figure 3.12

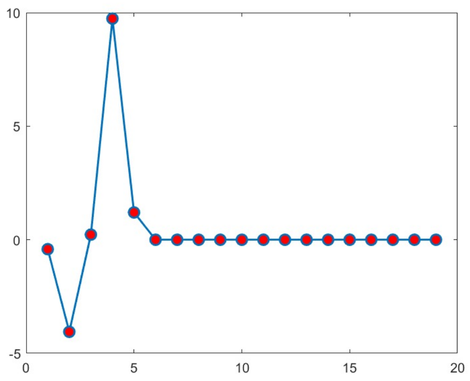


Figure 3.13

Comparison of the Methods

Both methods successfully minimized the Rosenbrock function:

- The Nelder-Mead method demonstrated robustness and effectiveness in the absence of derivative information but required more iterations to converge.
- The Modified Newton method exhibited significantly faster convergence due to the use of gradient and Hessian information, making it ideal for problems where these derivatives are available. Indeed, we can notice that this method converges in much fewer iterations than the Nelder-Mead method.

This analysis highlights the importance of selecting an optimization method based on the problem's characteristics, the availability of derivative information, and computational constraints.

Chapter 4

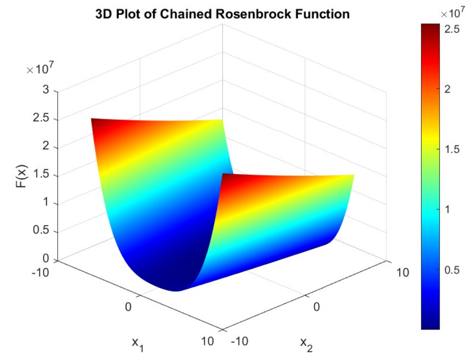
Testing Nelder Mead method on 3 test functions

In this chapter, we will test the Nelder Mead method implemented previously on three problems taken from [1], analyzing them separately by varying the dimension n.

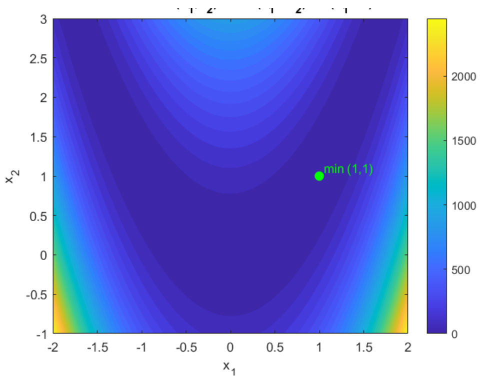
The functions chosen as test functions, along with their suggested initial points, are as follows:

1. **FUNCTION 1**. Corresponds to PROBLEM 1 from [1] and is the Chained Rosenbrock function. Its minimum is 0.

$$F(x) = \sum_{i=2}^{n} \left[100 \left(x_{i-1}^2 - x_i\right)^2 + (x_{i-1} - 1)^2\right]$$
$$\bar{x}_i = -1.2, \quad \text{if } \mathrm{mod}(i,2) = 1,$$
$$\bar{x}_i = 1.0, \quad \text{if } \mathrm{mod}(i,2) = 0.$$



(a) FUNCTION 1, Chained Rosenbrock Function in $\mathbb{R}^2$

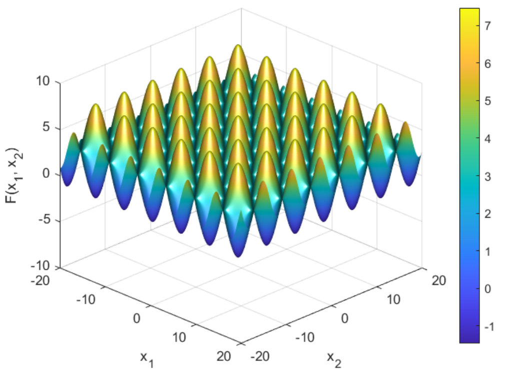


(b) FUNCTION 1, contour plot

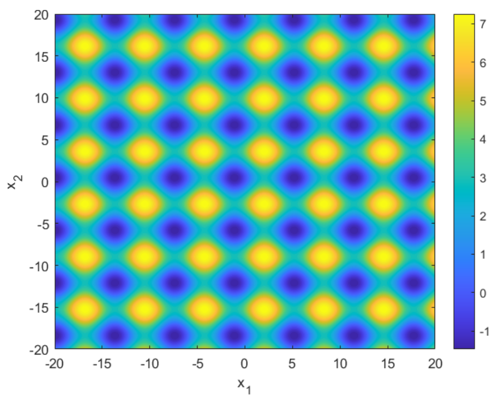
Figure 4.1

2. **FUNCTION 2**. Corresponds to PROBLEM 16 from [1] and is the Banded Trigonometric Problem. Its minimum depends on the dimension n of the problem.

$$F(x) = \sum_{i=1}^{n} i\left[(1 - \cos x_i) + \sin x_{i-1} - \sin x_{i+1}\right],$$
$$x_0 = x_{n+1} = 0,$$
$$\bar{x}_i = 1, \quad i \geq 1.$$



(a) FUNCTION 2, Banded Trigonometric Problem in $\mathbb{R}^2$



(b) FUNCTION 2, contour plot

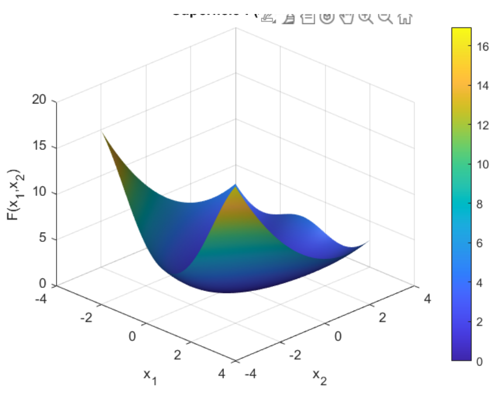
Figure 4.2

3. **FUNCTION 3**. Corresponds to PROBLEM 82 from [1]. Its minimum is 0.

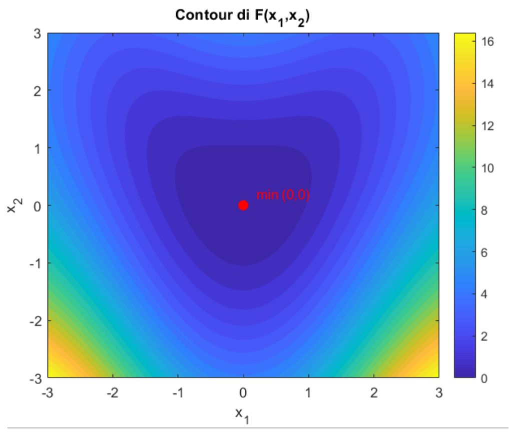
$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x),$$

$$f_k(x) = \begin{cases} x_k, & k = 1, \\ \cos(x_{k-1}) + x_k - 1, & 1 < k \le n, \end{cases}$$

$$\bar{x_l} = \frac{1}{2}, \quad l \ge 1.$$



(a) FUNCTION 3 in $\mathbb{R}^2$



(b) FUNCTION 3, contour plot, with its minimum point (0,0)

Figure 4.3

4.1 Use of the suggested starting point

The tables below show the results obtained for each test function, varying with the value of n. Initially, we considered the starting point as provided by the problem definition for each test function. The reported results include:

- The iteration index at which the method stops (*iter*).
- Execution time.
- The occurrence of a failure (Yes/No).

- The value f_{opt} assumed by the function the minimum point returned by the method

TEST FUNCTION 1	***iter***	**Execution time**	**f_opt**	**Failure**
$n = 10$	4717	0.1197127	1.630794800576799e-13	No
$n = 25$	72258	1.303473	15.4122289	Yes
$n = 50$	561018	11.6668661	44.852577182340760	Yes

Table 4.1: Results for Test Function 1 with varying n.

In table (4.1) are reported the results obtained by applying the Nelder-Mead algorithm to the Chained Rosenbrock function for various dimensions n. The function is designed to be a challenging test for optimization algorithms. This function is also used because, in any dimension, the minimum is 0:

$$\min = \begin{cases} n = 2, & \to f(1{,}1) = 0, \\ n = 3, & \to f(1{,}1{,}1) = 0, \\ n > 3, & \to f(1, \cdots, 1) = 0 \end{cases}$$

We can observe that the number of iterations increases significantly with n: from 4717 for $n = 10$ to 561018 for $n = 50$. This is expected, as the Nelder-Mead algorithm does not use gradient information and relies on heuristic search in a space of dimension n. Therefore, the complexity of the function's landscape increases with n, requiring more iterations to find a satisfactory solution. We deduce that the algorithm is effective for smaller dimensions ($n = 10$) but becomes computationally expensive for larger n.

The execution time increases rapidly with n, from 0.12 seconds for $n = 10$ to over 11.67 seconds for $n = 50$. This increase is nonlinear and reflects both the growing number of iterations and the computational cost associated with updating the simplex in a space of dimension n.

f_{opt} is very close to 0 for $n = 10$, indicating that the algorithm effectively converges to the global minimum. However, f_{opt} increases significantly with n, reaching 15.41 for $n = 25$ and 44.85 for $n = 50$. This could be due to the algorithm's difficulty in following the narrow valley of the function for higher n. For this reason, failure was declared in these latter two cases.

TEST FUNCTION 2	***iter***	**Execution time**	**f_opt**	**Failure**
$n = 10$	978	0.094372	-8.051392105063043	No
$n = 25$	20656	0.4082884	-16.137926968911447	No
$n = 50$	348087	7.9238946	-27.891655804352403	No

Table 4.2: Results for Test Function 2 with varying n.

Before analyzing Table (4.2), let us study the structure of the Banded Trigonometric function, which exhibits complex behavior because the value of $F(x)$ depends on:

- The weighted sum i, which increases with the index i.
- Nonlinear components $\cos x_i, \sin x_{i-1}, \sin x_{i+1}$, which introduce oscillations.

The number of iterations increases significantly with the dimension n: for $n = 10$, the iterations are 978, while for $n = 50$, the iterations rise to 348087. This behavior is consistent because the number of variables grows linearly with n, and the complexity of the function increases with the introduction of new oscillatory terms.

The execution time increases rapidly with n: it rises from 0.094 seconds for $n = 10$ to nearly 8 seconds for $n = 50$. This increase is correlated with the growth in the number of iterations.

The value of f_{opt} becomes more negative as n increases. This behavior is expected, as $F(x)$ includes a weighted sum i that grows with n. Since increasing n also increases the number of terms to sum, the optimal value of the function will change depending on the problem's dimension. Although the Nelder-Mead method does not make use of the gradient or the Hessian, we nevertheless evaluated the gradient norm at the point x_{opt} where the function attains the values f_{opt} reported in the table. We found that, in those points, the gradient norm is of the order of 10^{-6}, which can be considered sufficiently small to regard x_{opt} as a stationary point. In this case, we chose to report also the gradient norm values, because for this function the minimum value is not known a priori, since it depends on the size of the problem.

The Banded Trigonometric function is particularly challenging for Nelder-Mead, likely due to its oscillatory nature. Despite this issue, the method appears to converge, and it does so very quickly.

The Function 3 is constructed as a sum of squares of terms $f_k(x)$. The theoretical global minimum is achieved when $f_k(x) = 0$ for all k, and the optimal value is therefore: $f_{\text{opt}} = 0$.

TEST FUNCTION 3	***iter***	**Execution time**	**f_opt**	**Failure**
$n = 10$	737	0.1273765	1.778630616932833e-13	No
$n = 25$	10065	0.3103339	2.948592720109134e-12	No
$n = 50$	205501	4.3817103	1.793995850292785e-07	No

Table 4.3: Results for Test Function 3 with varying n.

Let us now analyze the table (4.3). As in the previous cases, the number of iterations increases significantly with the dimension n. This behavior is expected, as the number of terms $f_k(x)$ grows linearly with n, and the complexity of the problem increases.

As we also expect, the execution time increases with n, and this directly reflects the rise in the number of iterations.

Unlike the other two functions, the values of f_{opt} are very close to 0, the theoretically expected minimum.

Since Nelder-Mead consistently found a valid solution within the imposed limits for all dimensions, the *Failure* column reports *No*. In this case, the Nelder-Mead algorithm converges effectively.

It is worth noting, however, that the method is relatively simple as it does not require the computation of gradients or Hessians, making it suitable for non-differentiable functions or functions with difficult-to-compute gradients. Therefore, based on the results obtained and the observed behavior of the three different functions, we consider the method to be efficient.

4.2 Use of random starting points

After testing the methods with the proposed points, we randomly generated 10 initial points by sampling from a hypercube where, for each component, we defined:

- *Lower bound*: $\mathbf{x}_{0i} - \mathbf{1}$
- *Lower bound*: $\mathbf{x}_{0i} + \mathbf{1}$

Where $\mathbf{x}_{0i}$ is the suggested starting point used for the i-th test function.
To obtain the 10 random points, we executed a loop from 1 to 10, using the following expression:

$$x_{0\_\text{random}} = \text{lower\_bound} + (\text{upper\_bound} - \text{lower\_bound}) \cdot \text{rand}(n, 1)$$

Below, we present the data obtained for the three functions using different initial vectors.
We start by observing Table (4.4), where the number of iterations ranges from 2452 to 5432. This indicates that, for relatively small dimensions, the algorithm converges quickly. Execution times are short, reflecting the low computational complexity for $n = 10$. These results had already been noticed in the initial vector. However, compared to that vector, the values of f_{opt} fall into two categories: some are very small values (on the order of 10^{-13}), indicating an excellent approximation of the global minimum, while others ($f_{\text{opt}} = 3.9866$) are larger and suggest that the algorithm may have stopped at a local minimum. This behavior of f_{opt} suggests that the function is sensitive to the choice of the initial point.

As expected, in Table (4.5), the number of iterations varies significantly from 70099 to 151553, and the execution time also increases. This behavior compared to the case $n = 10$

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	0.0737	3.9866	2532
2	0.0502	3.9866	2874
3	0.0587	1.7179e-13	2649
4	0.0594	1.2506e-13	2452
5	0.0558	2.9461e-13	3059
6	0.0581	1.5925e-13	4340
7	0.0441	3.9333e-13	3376
8	0.0760	3.2392e-13	5432
9	0.0430	3.9866	3351
10	0.0377	1.6308e-13	2856

Table 4.4: Simulation results with $n = 10$ for Function 1 using different random vectors.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	1.3516	17.9903	94802
2	1.4319	14.2569	79170
3	1.7096	10.6183	117929
4	2.0803	15.8042	151553
5	1.6048	13.6336	114409
6	0.9974	19.3145	70099
7	1.2241	15.7678	85625
8	1.3220	14.3134	93052
9	1.0781	17.3630	78507
10	0.9631	16.6066	72258

Table 4.5: Simulation results with $n = 25$ for Function 1 using different random vectors.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	12.6601	42.1116	511258
2	13.8283	41.8556	582274
3	5.6417	43.7462	255896
4	14.7816	42.9069	600000
5	14.8186	42.8128	600000
6	10.0532	44.7770	429267
7	10.8863	40.1625	500672
8	8.3089	47.9109	388345
9	14.5756	48.0052	421106
10	20.6393	45.4088	572907

Table 4.6: Simulation results with $n = 50$ for Function 1 using different random vectors.

is anticipated, as the complexity of the problem grows with n. However, despite the fact that the function performs many more iterations and takes significantly more time, the values of f_{opt} lie between 10.6183 and 19.3145. We thus deduce that, while the algorithm sometimes improves, the solution remains far from being optimal.

In Table (4.6) concerning Function 1 with $n = 50$, the number of iterations is very high, ranging from 255896 to 600000. The maximum iteration limit was reached in some cases (e.g id 4 and 5). Execution times are significant, ranging between 5.6417 and 20.6393 seconds. The values of f_{opt} range from 40.1625 to 48.0052, indicating that the algorithm struggles even more to find optimal solutions for larger dimensions. We conclude that for $n = 50$, the algorithm exhibits clear limitations in efficiency and accuracy, as the results are often far from the global minimum. This also happens because the Rosenbrock function is ill-conditioned, since around the minimum it forms a narrow, curved valley.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	0.0504	-8.0514	1171
2	0.0316	-8.0514	805
3	0.0442	-8.0514	1334
4	0.0466	-8.0514	1295
5	0.0293	-8.0514	961
6	0.0279	-8.0514	841
7	0.0319	-8.0514	1061
8	0.0374	-8.0514	1183
9	0.0261	-8.0514	918
10	0.0298	-8.0514	974

Table 4.7: Simulation results with $n = 10$ for Function 2 using different random vectors.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	0.5662	-16.1379	22596
2	0.5152	-16.1379	15333
3	0.4996	-16.1379	14844
4	0.4997	-16.1379	18351
5	0.6932	-16.1379	29876
6	0.4399	-16.1379	21796
7	0.5404	-16.1379	22255
8	0.4452	-16.1379	23815
9	0.3042	-16.1379	14173
10	0.2618	-16.1379	13663

Table 4.8: Simulation results with $n = 25$ for Function 2 using different random vectors.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	16.0908	-27.8474	557011
2	8.1506	-27.8948	303051
3	9.9908	-27.8940	377106
4	5.4091	-27.8914	222884
5	12.7786	-27.8663	411180
6	11.9600	-27.8892	349919
7	20.4893	-24.8844	600000
8	8.1038	-27.8818	335059
9	7.6320	-27.8948	285823
10	9.4386	-27.8941	377903

Table 4.9: Simulation results with $n = 50$ for Function 2 using different random vectors.

Let us now analyze the results obtained for Function 2. The following tables show very interesting findings.

In all the tables, we can observe that for each perturbed starting point, both the iteration patterns and execution times reflect the results obtained with the suggested initial point. The surprising fact is that the value of f_{opt} remains unchanged. We might conclude that the algorithm converges to the minimum point.

Due to space constraints, the x_{opt} vectors are not included, as they differ only slightly from one another. However, for Function 2, it is useful to examine some of these vectors. Below, we present two solutions, found in the case $n = 10$, that the method identifies.

The first:

$$\mathbf{v} = \begin{bmatrix} 5.1760 \\ 5.4978 \\ 5.6952 \\ -0.4636 \\ -0.3805 \\ -0.3218 \\ -0.2783 \\ -0.2450 \\ -0.2187 \\ 0.7328 \end{bmatrix}$$

And the second:

$$\mathbf{v} = \begin{bmatrix} -1.1071 \\ -0.7854 \\ 5.6952 \\ -0.4636 \\ 5.9027 \\ -0.3218 \\ -0.2783 \\ -0.2450 \\ -0.2187 \\ 0.7328 \end{bmatrix}$$

We conclude that the function has a single minimum, but there are many minimum points that minimize it, as Figure 4.2 shows.

For Function 3, with larger dimensions, many relationships can be deduced between the parameters and the perturbed vectors. For example, in the cases with $n = 10$ and $n = 25$, the difference in iterations between two different vectors was not significant enough to be correlated to the different perturbed starting points.

Id	**Execution Time (s)**	f_{opt}	**Iterations**
1	1.0218e-02	1.9456e-13	706
2	8.8215e-03	2.1786e-13	725
3	7.4519e-03	3.3563e-13	712
4	1.3263e-02	1.8466e-13	1255
5	8.2512e-03	5.5490e-13	826
6	9.5497e-03	1.3179e-13	885
7	8.0469e-03	1.8662e-13	842
8	6.4149e-03	2.2104e-13	781
9	5.5565e-03	2.3467e-13	719
10	5.7372e-03	1.1351e-13	781

Table 4.10: Simulation results with $n = 10$ for Function 3 using different random vectors.

As we can notice in Table 4.12, the number of iterations varies significantly between different initial points, with a minimum of 18,714 (Vector 5) and a maximum of 259,766 (Vector 8). This variability suggests that the algorithm's convergence strongly depends on the initial vector. Less favorable initial vectors appear to cause a higher number of iterations, resulting in increased computation time (0.4107 seconds for Vector 5 and 5.6886 seconds for Vector 8).

Id	Execution Time (s)	f_{opt}	Iterations
1	8.0095e-02	2.3454e-12	10510
2	9.5107e-02	9.8838e-13	11741
3	6.5981e-02	2.2151e-12	8230
4	9.1348e-02	8.7932e-12	11633
5	7.6421e-02	1.3188e-11	10090
6	5.2129e-02	1.7758e-12	6422
7	6.6457e-02	3.9309e-12	8835
8	7.4024e-02	1.8909e-12	9877
9	5.4491e-02	1.0896e-12	7110
10	8.1774e-02	3.0720e-12	10856

Table 4.11: Simulation results with $n = 25$ for Function 3 using different random vectors.

Id	Execution Time (s)	f_{opt}	Iterations
1	3.6420	4.5852e-08	160636
2	0.4260	4.4076e-12	19232
3	2.1273	7.3318e-10	93266
4	1.5363	2.0562e-10	69936
5	0.4107	7.1750e-12	18714
6	1.0337	5.9613e-11	47159
7	1.0263	7.3836e-11	47442
8	5.6886	3.4103e-06	259766
9	0.5040	1.8127e-11	22857
10	0.5096	2.7939e-12	23204

Table 4.12: Simulation results with $n = 50$ for Function 3 using different random vectors.

In table 4.12 the values of f_{opt} are distributed across different orders of magnitude:

- The best values (e.g., for Vectors 2, 5, and 10) are very close to zero ($\sim 10^{-12}$), indicating an excellent approximation of the theoretical minimum.

- However, some values (e.g., Vector 8 with $f_{\text{opt}} = 3.4103 \times 10^{-6}$) deviate significantly, suggesting that in some cases the algorithm attains a lower accuracy in the solution (e.g., around 10^{-6} instead of 10^{-11}).

This dispersion in the f_{opt} results confirms the algorithm's dependence on the initial vector, which can significantly affect the quality of the solution.

In general, cases with a low number of iterations (e.g., Vector 5 with 18,714 iterations) tend to produce an f_{opt} value that is very small (on the order of magnitude 10^{-10} or 10^{-12}), suggesting that the starting point was closer to the optimal solution. Conversely, cases with a high number of iterations do not always correspond to precise solutions (e.g., Vector 8, which has an order of magnitude 10^{-6}).

4.3 Experimental rate of convergence

To evaluate the behavior of the method, we calculated the experimental rate of convergence, obtained using the formula

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

where $e^{(k)} := x^{(k)} - x^{(k-1)}$ for k sufficiently large.

From the plots of the three functions, it is difficult to understand the trend of the rates. This happens because Nelder-Mead is a heuristic method: very often, the new optimal value found may be far from the previous one or very close, depending on the phase of the method.

We observed the following behaviors:

- If the residual $\|e^{(k)}\|$ becomes very small, a value numerically approximated to zero can be obtained.
- NaN values also appear due to undefined mathematical operations: this often occurs when the method is converging slowly because the iterations are very close, leading to residual vectors being similar. As a result, the denominator becomes approximately $\log(1) = 0$.
- There are also a few values equal to ∞. If $\|e^{(k)}\| \to 0$ but $\|e^{(k-1)}\| > 0$, the ratio $\frac{\|e^{(k)}\|}{\|e^{(k-1)}\|}$ can result in 0, and $\log(0)$ is undefined.
- If the computation of one of the two logarithms results is a negative number or a value close to zero due to numerical instability, the result can become undefined or negative.

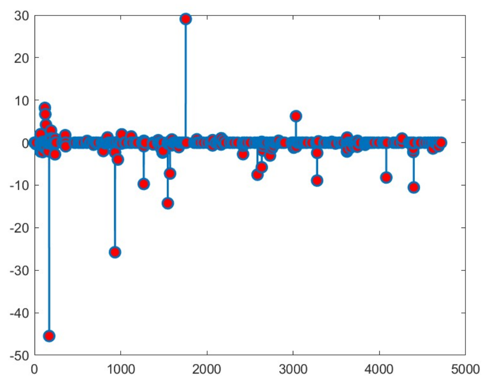


Figure 4.4: Experimental rate of convergence for Funtion 1 with $n = 10$

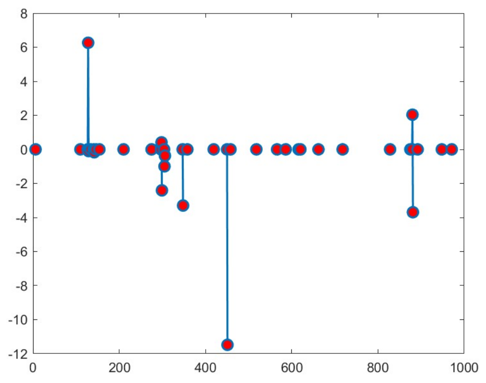


Figure 4.5: Experimental rate of convergence for Funtion 2 with $n = 10$

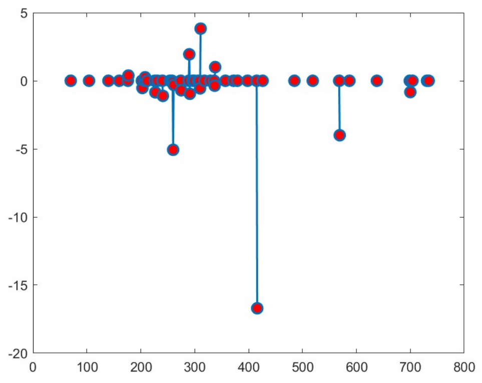


Figure 4.6: Experimental rate of convergence for Funtion 3 with $n = 10$

Chapter 5

Testing Modified Newton Method on 3 test functions

5.1 Use of the suggested starting point

Let us now analyze the Newton method in the same way we analyzed the Nelder-Mead method. The functions on which it was tested are the same as those used in the previous chapter.

TEST FUNCTION 1	***iter***	**Execution time**	**f_k**	**grad_norm**	**Failure**
$n = 1000$	1476	16.525	3.9866	9.8757e-09	Yes
$n = 10000$	14669	3641.636	3.9866	1.8678e-08	Yes
$n = 100000$	inf	inf	inf	inf	Yes

Table 5.1: Results for Test Function 1 with varying n.

In Table (5.1), it can be observed that the number of iterations increases significantly with n. For $n = 1000$, the algorithm requires approximately 1476 iterations, while for $n = 10000$, nearly 10 times more iterations are needed (14,669). For $n = 100000$, the number of iterations is theoretically much higher, but the execution was halted before completion as the execution time was becoming excessively large.

The execution time increases dramatically. From 16.53 seconds for $n = 1000$, it rises to over an hour for $n = 10000$. For $n = 100000$, the required time would have been so high as to make it impractical.

For $n = 1000$ and $n = 10000$, The value of the objective function is not zero, but it is also not excessively large. However, it was decided to declare failure. This also happens for the ill-conditioning of the Test Function 1, as explained previously.

For $n = 100000$, the method reached its practical limits. The handling of the Hessian matrix becomes too computationally expensive, highlighting that the modified Newton method is not suitable for problems of such high dimensionality without further optimizations, so it was determined that the method fails.

TEST FUNCTION 2	***iter***	**Execution time**	**f_k**	**grad_norm**	**Failure**
$n = 1000$	8	0.0244	-427.4045	3.0150e-13	No
$n = 10000$	8	0.1874	-4.1599e+03	1.8403e-12	No
$n = 100000$	8	8.5185	-4.1443e+04	2.8158e-13	No

Table 5.2: Results for Test Function 2 with varying n.

In Table (5.2) the number of iterations is constant for all values of n, equal to 8. This behavior is significant, as it indicates that, with this test function, the modified Newton method can maintain rapid convergence regardless of the dimension n. We can observe that the structure of the function and the efficiency of the method, which utilizes both the gradient and the Hessian matrix, allow for a quick identification of the minimum in a very small number of iterations.

Although the time increases with n, the method remains extremely efficient even for very large dimensions. This behavior is a distinctive feature compared to more complex functions, such as the Rosenbrock function.

The value of f_k scales with n, in line with the structure of the function, which depends on a factor i that grows with the dimension, as observed more thoroughly in the previous chapter. This behavior demonstrates that the algorithm has correctly identified the minimum in each case.

The gradient norm is extremely small in all cases, and the algorithm achieves excellent accuracy in every instance, regardless of the dimension n. This result indicates that the structure of the function and the implementation of the modified Newton method are particularly robust for the problem under consideration.

TEST FUNCTION 3	***iter***	**Execution time**	**f_k**	**grad_norm**	**Failure**
$n = 1000$	7	0.155173	3.944413905238746e-31	8.8819e-16	No
$n = 10000$	8	1.373784	1.384913245764988e-29	5.2630e-15	No
$n = 100000$	8	381.157248	1.384933483710488e-29	5.2630e-15	No

Table 5.3: Results for Test Function 3 with varying n.

Even for the Function 3, the method converges in a very low number of iterations, regardless of the value of n. This suggests that the structure of the function is well-suited to the Newton method, which quickly identifies the solution.

With $n = 100000$ the execution time is 381.1572 seconds. There is a significant increase compared to the previous cases, reflecting the computational cost associated with handling the Hessian matrix for very large n. Although the time increases drastically for

$n = 100000$, it is still acceptable, considering the enormous number of variables. This demonstrates good scalability compared to less sophisticated methods.

We note that $f_k = 0$ corresponds to the theoretical minimum of the Test Function 3. The method is able to precisely identify the global minimum in all cases, regardless of the dimension n. This confirms the effectiveness of the algorithm in solving the problem.

The gradient norm is consistently small in all cases, confirming that the algorithm achieves solutions very close to the global minimum.

Test Function 3 represents an example where the modified Newton method demonstrates high efficiency and precision, converging quickly and successfully even for high-dimensional problems.

5.2 Use of random starting points

For the Test Function 1, it was decided to report only the table with $n = 1000$, as for larger dimensions, we have obtained as well failures that are explained below.

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	0.021532	5.5411e+04	5.6560e+05	1
2	0.024197	5.6550e+04	5.8652e+05	1
3	0.021052	5.7519e+04	6.1082e+05	1
4	0.032567	5.4012e+04	5.5462e+05	1
5	0.020426	5.6252e+04	5.8989e+05	1
6	0.020766	5.2659e+04	5.3475e+05	1
7	0.024715	5.3971e+04	5.4711e+05	1
8	0.023418	5.4626e+04	5.7079e+05	1
9	0.023493	5.3443e+04	5.3785e+05	1
10	0.024921	5.7126e+04	6.0730e+05	1

Table 5.4: Results for Function 1 with n= 1000

The Table (5.4) shows a particular behavior. It can be observed that all cases perform only one iteration before stopping, and the gradient norm is excessively high. With all 10 initial vectors, the method fails because it is not possible to make the Hessian positive definite. Therefore, the method is highly sensitive to the initial point.

The Table (5.5) shows that for Function 2, the method always works: it requires little time and consistently achieves a very low gradient norm. It can be observed that with certain starting points, the method requires more iterations compared to others. In all cases, the method finds the same minimum value. With regard to Tables (5.6) and (5.7), the results obtained are satisfactory. It can be observed that all runs reach the maximum number of iterations, since the algorithm converges to a point where the gradient norm is on the order of 10^{-6} and 10^{-4} in the respective cases, and then remains there until the

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	0.076576	3.0150e-13	-427.4045	8
2	0.027146	3.0146e-08	-427.4045	19
3	0.021951	2.2166e-08	-427.4045	19
4	0.025394	1.9479e-08	-427.4045	20
5	0.019669	1.9127e-08	-427.4045	15
6	0.017582	1.4156e-07	-427.4045	16
7	0.019437	5.7091e-08	-427.4045	19
8	0.020428	1.8554e-07	-427.4045	17
9	0.018469	7.3122e-09	-427.4045	14
10	0.017750	1.5447e-08	-427.4045	14

Table 5.5: Results Funtion 2 with $n = 1000$

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	7.978122	5.3584e-06	-4.1599e+03	600
2	17.180461	7.9964e-06	-4.1599e+03	600
3	17.283041	8.1560e-06	-4.1599e+03	600
4	10.291489	8.3958e-06	-4.1599e+03	600
5	7.610149	3.5321e-06	-4.1599e+03	600
6	7.390846	3.7105e-06	-4.1599e+03	600
7	10.795300	8.7251e-06	-4.1599e+03	600
8	12.735664	6.2903e-06	-4.1599e+03	600
9	7.810752	3.3660e-06	-4.1599e+03	600
10	0.275056	9.0889e-07	-4.1599e+03	22

Table 5.6: Results Funtion 2 with $n = 10000$

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	108.502317	1.3051e-04	-4.1444e+04	60
2	111.102457	1.6485e-04	-4.1444e+04	60
3	115.316453	7.2101e-04	-4.1444e+04	60
4	100.572140	6.7997e-04	-4.1444e+04	60
5	114.193031	2.4921e-04	-4.1444e+04	60
6	109.503808	1.7115e-04	-4.1444e+04	60
7	113.668384	6.4914e-04	-4.1444e+04	60
8	114.291770	3.7764e-04	-4.1444e+04	60
9	115.678061	1.2190e-04	-4.1444e+04	60
10	109.613813	7.5748e-05	-4.1444e+04	60

Table 5.7: Results Funtion 2 with $n = 100000$

end of the iterations. Nevertheless, the function values achieved are satisfactory.

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	0.044346	1.2137e-09	7.3654e-19	10
2	0.042924	3.3874e-10	5.7373e-20	10
3	0.037214	2.3235e-15	2.6994e-30	10
4	0.040283	2.4825e-16	3.0815e-32	9
5	0.039009	6.7088e-09	2.2504e-17	9
6	0.047730	1.8003e-10	1.6205e-20	12
7	0.045533	1.6921e-11	1.4317e-22	11
8	0.057051	4.1792e-15	8.7329e-30	14
9	0.044425	7.7444e-14	2.9988e-27	10
10	0.049829	2.2204e-16	2.4652e-32	10

Table 5.8: Results Funtion 3 with $n = 1000$

Id	Execution Time (s)	gradfk_norm	fk	Iterations
1	1.363357	4.6911e-14	1.1003e-27	10
2	1.385928	2.7195e-16	3.6978e-32	10
3	0.427273	46.1612	641.7634	3
4	1.373040	4.7778e-15	1.1414e-29	10
5	1.825115	2.3575e-10	2.7788e-20	14
6	0.922651	6.2878e-14	1.9768e-27	10
7	0.927460	2.4826e-14	3.0816e-28	10
8	1.010934	2.9374e-16	4.3141e-32	11
9	1.202969	1.9151e-14	1.8337e-28	13
10	0.281549	36.7379	437.3019	3

Table 5.9: Results Funtion 3 with $n = 10000$

It can be observed that, for Function 3, the method does not always converge to the minimum, which is close to zero. Occasionally, the method finds solutions that are far from the optimal value, and this happens when $n = 10000$ or $n = 100000$. In these particular cases, besides failing to reach the optimum, the function performs very few iterations and, as a consequence, doesn't take a significant amount of time.

Vector	Execution Time (s)	gradfk_norm	fk	Iter
1	622.958128	1.8116e-12	1.6409e-24	17
2	125.914937	74.2469	1.6866e+03	3
3	586.524479	6.2884e-09	1.9772e-17	16
4	375.583425	3.5639e-09	6.3508e-18	10
5	155.475264	78.8377	1.3304e+03	4
6	857.244115	1.2194e-09	7.4344e-19	22
7	372.164052	1.9230e-16	1.8489e-32	11
8	297.310821	9.4079e-14	4.4255e-27	10
9	95.577462	92.9824	2.2431e+03	3
10	364.663200	1.1122e-14	6.1845e-29	11

Table 5.10: Results Funtion 3 with $n = 100000$

In this case, Table (5.10), the reason for the failure is always that the method is unable to make the Hessian positive definite.

5.3 Experimental rate of convergence

For Functions 2 and 3, we calculated the order for each k, as there were only a few iterations. With Function 1, the method requires significantly more iterations; however, we observed that the values of $e^{(k)}$ become close to zero after a few iterations. For this reason, we considered only the first k iterations where $e^{(k)}$ is non-zero, as k is sufficiently large.

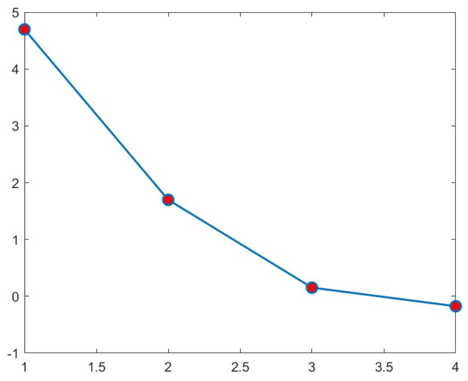


Figure 5.1: Experimental Rate of convergence Funtion 1

We can observe that for Functions 2 and 3, the method has a convergence order very close to 2, which corresponds to the convergence order of the Modified Newton method. In contrast, Function 1 initially has a convergence order of 5, which is very high, and then it tends to decrease because the subsequent points it finds are very close to each other,

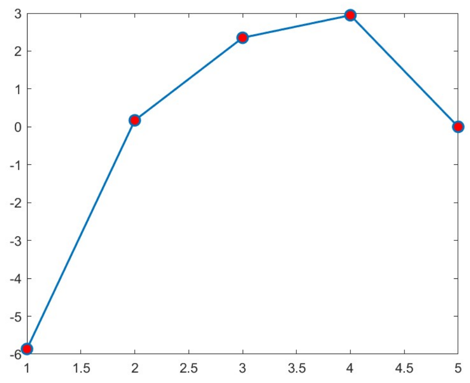


Figure 5.2: Experimental Rate of convergence Function 2

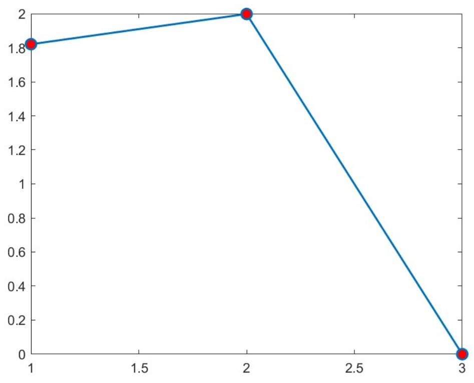


Figure 5.3: Experimental Rate of convergence Funtion 3

resulting in insignificant movements.

5.4 Comparisons Between the Two Methods

The two reported methods are using different approaches, that's why when we have tested them with different functions, we obtained different results.

The convergence behavior of the Nelder-Mead method demonstrates its effectiveness for small-dimensional problems and smooth landscapes, such as cases with $n = 10$. However, as the dimensionality increases, particularly for $n \geq 50$, the method encounters significant challenges due to its reliance on heuristic searches in large spaces. But in general, Nelder-Mead often converges to the minimum point.

On the other hand, the Modified Newton method exhibits quadratic convergence for smooth and well-conditioned functions, such as Functions 2 and 3, where it consistently outperforms Nelder-Mead. The Modified Newton method has limitations especially when the Hessian matrix is ill-conditioned or difficult to stabilize to make it positive definite. Indeed, for most of the failures, the method fails to achieve convergence because it is required a substantial modification of the Hessian.

Regarding execution time, Nelder-Mead experiences a rapid increase in computational demand as the problem dimension n grows, performing sometimes poorly when $n \geq 50$ (as it can be seen in Function 1, for instance), but the execution times are not high. The Modified Newton method, even if it generally requires much less iterations than Nelder-Mead for smooth and differentiable functions, faces an increase of the computational costs and execution time as the dimension grows. The need to build, evaluate and modify the Hessian matrix sometimes makes the method impractical for very large problems (e.g., $n = 100000$), where the overhead outweighs its theoretical advantages, as can be observed in Function 1. However, it performs pretty well in Function 2 and 3 when $n = 100000$.

For functions 2 and 3, the Nelder-Mead and Modified Newton methods differ significantly in terms of the number of iterations. Nelder-Mead requires many iterations to converge because this approach relies only on function evaluations. Newton's method, on the other hand, uses much more information, such as the gradient and Hessian, and converges to the minimum with fewer iterations (at the expense of execution time). Thanks to this additional information, it can identify faster directions to reach the minimum.

Chapter 6

Finite Difference Approximations for derivatives

The Modified Newton method presents some issues in terms of computational cost. As can be seen in the explanation of the method, finding the direction p_k along which to move requires solving a linear system, whose coefficient matrix is precisely the Hessian. Building the Hessian with exact second derivatives requires a significant computational cost. For this reason, the modified Newton method has also been tested in cases where the Hessian is built by computing each entry using finite differences. These are approximations of derivatives that make their calculation much more straightforward, mitigating the cost of computation.

From an implementation point of view, the strategy used is as follows: first, it is ensured that the test function has specific characteristics that make the Hessian a matrix with an advantageous structure, reducing the computational cost of constructing it. Subsequently, the following formula is applied to each entry of the matrix:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(\bar{x} + he_i + he_j) - f(\bar{x} + he_i) - f(\bar{x} + he_j) + f(\bar{x})}{h^2}$$

Note that the Hessians obtained are tridiagonal (test functions 1 and 3) and diagonal (test function 2), which theoretically allows for a significant saving in terms of computation time.

Regarding the gradient, each component has been computed using the centered finite difference approximation for first order derivatives, since it grants a good precision. It is explained by the following formula:

$$\frac{\partial f}{\partial x_i} = \frac{f(\bar{x} + he_i) - f(\bar{x} - he_i)}{2h}$$

The increment h can take different values: Let's begin with the case where the increment h for each differentiation is:

$$h = 10^{-k}, \quad k = 2, 4, 6, 8, 10, 12$$

For each test function X, a dedicated function `finite_difference_functionX` was created, where the formulas mentioned above are applied to calculate the gradient and Hessian at each step.

The Newton method was run on the Test Functions 2 and 3, and for each of them we use as starting point: the suggested starting point and 10 random points. For the first test function, there was no convergence across the configurations examined, and those results are therefore not included here. Let us analyze each case.

NB: It is worth noting that the tables with the results we will see below refer to the cases where the problem size is 10^3 and 10^4, as for $n = 10^5$ the execution times of the code are very long.

6.1 Runs and comments with $h = 10^{-k}$

The tables below show the results obtained for each test function, varying with the value of k. The reported results include:

- The norm of the gradient calculated at the point where the method stops (*gradfk_norm*)
- The iteration index at which the method stops (*iter*)
- Execution time
- The optimal value reached f_k
- The occurrence of a failure (Yes/No)

6.1.1 Use of the suggested starting point

TEST FUNCTION 2	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	6.1635e-09	7	0.5237	-421.1429	No
$k = 4$	1.4290e-08	7	0.5032	-427.4039	No
$k = 6$	3.3262e-06	178	11.0482	-427.4045	No
$k = 8$	3.2241e-04	3258	326.0313	-427.4045	No
$k = 10$	0.0223	-	-	-	Yes
$k = 12$	1.5533e+04	0	0.1336	2.3092e+05	Yes

Table 6.1: Finite difference on test function 2 with $h = 10^{-k}$ and $n = 10^3$

TEST FUNCTION 2	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	4.1579e-09	12	89.5409	-4.1599e+03	No
$k = 4$	4.3504e-07	6	59.1512	-4.1599e+03	No
$k = 6$	5.2970e-05	52	506.4868	-4.1599e+03	No
$k = 8$	-	-	-	-	-
$k = 10$	-	-	-	-	-
$k = 12$	5.3184e+05	0	11.2647	2.2996e+07	Yes

Table 6.2: Finite difference on test function 2 with $h = 10^{-k}$ and $n = 10^4$

As we can see in the Table (6.1), the algorithm converges for most values of k, and the execution times remain limited. However, for $k = 10$, although the algorithm initially makes progress towards the minimum point, it eventually stagnates at a point where the gradient norm is 0.0223. For $k = 12$, on the other hand, the algorithm stops immediately since it cannot find a sufficiently positive definite Hessian matrix. For $n = 10^4$, in Table (6.2), the algorithm converges for the first three values of k. However, for larger values of k it either fails to converge for being not able to find a sufficient positive definite Hessian, as in the case of $k = 12$, or requires an excessively long time to reach satisfactory values, as in the case of $k = 8, 10$. Therefore, the results for those values of k are not reported.

TEST FUNCTION 3	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	1.2739e-09	6	0.7851	8.1129e-15	No
$k = 4$	1.0893e-08	6	0.6934	5.9331e-17	No
$k = 6$	7.9657e-07	5	0.6778	3.1726e-13	No
$k = 8$	6.2292	0	0.1548	71.3380	Yes
$k = 10$	6.2290	0	0.1528	71.3380	Yes
$k = 12$	11.6322	0	0.2777	55.3339	Yes

Table 6.3: Finite difference on test function 3 with $h = 10^{-k}$ and $n = 10^3$

TEST FUNCTION 3	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	1.2820e-07	6	71.8544	8.2174e-15	No
$k = 4$	5.2558e-08	7	55.7817	1.3812e-15	No
$k = 6$	4.1613e-08	7	72.8907	8.6583e-16	No
$k = 8$	26.2373	1	22.7287	591.7147	Yes
$k = 10$	19.6721	0	11.4469	712.8967	Yes
$k = 12$	12.6175	3	45.3172	266.2742	Yes

Table 6.4: Finite difference on test function 3 with $h = 10^{-k}$ and $n = 10^4$

Concerning the test function 3, in tables (6.3), (6.4), in general, an excellent convergence can be observed in many cases. However, there are situations in which the algorithm stops prematurely because it fails to find a sufficiently positive definite Hessian matrix.

Use of random starting points

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	7.1867e-06	11	0.7024	-421.1429	No
2	0.0289	200	10.9117	-421.1430	No
3	0.0041	200	12.6262	-421.1429	No
4	1.6213e-04	200	13.8723	-421.1429	No
5	6.2670e-08	13	0.7733	-421.1429	No
6	3.9542e-04	200	10.8773	-421.1429	No
7	0.0638	200	11.9514	-421.1431	No
8	5.5037e-06	10	1.1710	-421.1429	No
9	2.8498e-05	200	17.7066	-421.1429	No
10	0.0921	200	11.1045	-421.1431	No

Table 6.5: 10 random points. Finite difference on test function 2 with $h = 10^{-2}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	4.3055e-08	13	0.8182	-427.4039	No
2	4.2886e-08	15	0.8692	-427.4039	No
3	6.3610e-06	16	0.9503	-427.4039	No
4	1.8378e-07	14	0.8251	-427.4039	No
5	3.6027e-08	14	0.8320	-427.4039	No
6	6.1478e-08	14	0.8307	-427.4039	No
7	4.1215e-06	16	0.9445	-427.4039	No
8	4.9948e-08	15	0.8790	-427.4039	No
9	2.7969e-08	14	0.8331	-421.1429	No
10	4.6096e-08	14	0.8284	-421.1431	No

Table 6.6: 10 random points. Finite difference on test function 2 with $h = 10^{-4}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	1.4996e-06	16	0.9817	-427.4045	No
2	6.0415e-06	15	0.8785	-427.4045	No
3	3.8389e-06	16	0.9223	-427.4045	No
4	6.2212e-06	14	0.8250	-427.4045	No
5	1.1978e-06	14	0.8227	-427.4045	No
6	2.3874e-06	20	1.1291	-427.4045	No
7	1.3012e-06	18	1.0444	-427.4039	No
8	2.4495e-06	19	1.0904	-427.4045	No
9	1.5886e-06	14	0.8290	-427.4045	No
10	7.9231e-06	17	0.9753	-427.4045	No

Table 6.7: 10 random points. Finite difference on test function 2 with $h = 10^{-6}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	2.2372e-04	5000	458.2038	-427.4045	No
2	2.1846e-04	5000	539.1419	-427.4045	No
3	2.2459e-04	5000	582.5118	-427.4045	No
4	3.0127e-04	5000	479.5610	-427.4045	No
5	2.9890e-04	5000	457.4784	-427.4045	No
6	2.9976e-04	5000	334.5901	-427.4045	No
7	3.1056e-04	5000	533.3229	-427.4039	No
8	3.0148e-04	5000	445.3660	-427.4045	No
9	2.1337e-04	5000	342.7685	-427.4045	No
10	2.9190e-04	5000	366.2737	-427.4045	No

Table 6.8: 10 random points. Finite difference on test function 2 with $h = 10^{-8}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	9.1770e-06	22	258.2537	-3.5348e+03	No
2	0.0092	30	778.0182	-3.5348e+03	No
3	0.0196	30	935.0418	-3.5348e+03	No
4	1.2524e-06	21	660.6884	-3.5348e+03	No
5	8.3609e-06	20	646.8754	-3.5348e+03	No
6	4.1815e-06	21	680.4982	-3.5348e+03	No
7	0.0150	30	938.7883	-3.5348e+03	No
8	8.3732e-06	19	611.6618	-3.5348e+03	No
9	0.0031	30	932.7636	-3.5348e+03	No
10	3.5946	30	955.9024	-3.5335e+03	Yes

Table 6.9: 10 random points. Finite difference on test function 2 with $h = 10^{-2}$ and $n = 10^4$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	2.3203e-05	100	697.2217	-4.1599e+03	No
2	1.0136e-05	100	698.2897	-4.1599e+03	No
3	1.2193e-05	100	694.6385	-4.1599e+03	No
4	9.7226e-06	72	509.4444	-4.1599e+03	No
5	4.9811e-06	60	424.2098	-4.1599e+03	No
6	6.8551e-06	63	445.4437	-4.1599e+03	No
7	2.2110e-05	100	695.0969	-4.1599e+03	No
8	7.9569e-06	76	530.3784	-4.1599e+03	No
9	4.5448e-05	100	694.0957	-4.1599e+03	No
10	3.4187e-06	57	400.8155	-4.1599e+03	No

Table 6.10: 10 random points. Finite difference on test function 2 with $h = 10^{-4}$ and $n = 10^4$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	3.3850e-04	68	785.3821	-4.1599e+03	No
2	6.8303e-04	100	1.0849e+03	-4.1599e+03	No
3	3.6358e-04	58	565.6754	-4.1599e+03	No
4	2.9893e-04	22	170.8640	-4.1599e+03	No
5	2.7056e-04	30	243.1941	-4.1599e+03	No
6	1.6150e-04	23	181.9558	-4.1599e+03	No
7	3.0764e-04	27	226.8176	-4.1599e+03	No
8	4.1423e-04	23	192.0567	-4.1599e+03	No
9	2.9784e-04	30	256.8978	-4.1599e+03	No
10	1.7302e-04	26	215.5498	-4.1599e+03	No

Table 6.11: 10 random points. Finite difference on test function 2 with $h = 10^{-6}$ and $n = 10^4$

For the test function 2, with $n = 10^3$, as we can see from the tables (6.5), (6.6), (6.7), the results obtained with $h = 10^{-k}$ for $k = 2{,}4{,}6$ are satisfactory. It is interesting to note that for $h = 10^{-2}$ at table (6.5), there are situations in which, although the gradient norm reaches the order of 10^{-2} or 10^{-3} (well above the threshold of 10^{-5} imposed for the run of this increment) and remains stuck at those values staying constant until the end of the iterations (in this run the maximum number of iterations is 200), the function value returned is still close to the optimal one obtained with a much smaller gradient norm. Therefore, we decided not to classify these specific cases as failures. The same reasoning has been applied on Table (6.9), except for the 10-th random point. Regarding Table (6.8), a lower threshold for the gradient norm has been introduced, since several runs converged to values of the order of 10^{-4}. It can be observed that, even in this case, all runs converge to the optimal minimum until the end of the iterations (in this run the maximum number of iterations is 5000). Concerning $h = 10^{-10}$ and $h = 10^{-12}$, all the 10 runs with the random starting points fail at the first iteration, since it is not possible to find a sufficient positive definite Hessian. So, it has been decided to not report the relative table.

Regarding the case of $n = 10^4$, for $k = 8$ and $k = 10$, the algorithm takes too much time to converge, while for $k = 12$ it is not able to find a sufficient positive definite hessian. So, for those cases, it has been decided to not report the relative table.

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	1.8986e-08	9	1.0106	1.8021e-16	No
2	4.6339e-07	8	0.9646	1.0736e-13	No
3	6.9096e-07	8	0.9131	2.3869e-13	No
4	1.2493e-08	8	0.9086	7.8024e-17	No
5	3.1615e-07	8	0.9630	4.9972e-14	No
6	3.7873e-07	9	1.0418	7.1712e-14	No
7	3.3690e-07	10	1.1311	5.6745e-14	No
8	5.3885e-08	9	0.9592	1.4517e-15	No
9	1.1067e-07	9	1.0448	6.1231e-15	No
10	3.7219e-08	13	1.5120	6.9257e-16	No

Table 6.12: 10 random points. Finite difference on test function 3 with $h = 10^{-2}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	2.0338e-09	9	1.0024	2.0681e-18	No
2	1.6505e-09	9	0.9397	1.3620e-18	No
3	3.3897e-08	8	0.8234	5.7452e-16	No
4	4.5024e-09	7	0.7992	1.0136e-17	No
5	2.9328e-09	8	0.8708	4.3007e-18	No
6	2.5102e-07	11	1.1591	3.1506e-14	No
7	9.7758	3	0.4887	34.9874	Yes
8	3.9607e-09	11	1.2036	7.8435e-18	No
9	1.7432e-09	10	1.0069	1.5194e-18	No
10	4.2888e-09	8	0.8622	9.1968e-18	No

Table 6.13: 10 random points. Finite difference on test function 3 with $h = 10^{-4}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	3.5521e-10	14	1.5668	6.3087e-20	No
2	7.3516e-07	7	0.8325	2.7023e-13	No
3	8.3016e-10	10	1.0420	3.4458e-19	No
4	1.4441e-08	7	0.8125	1.0427e-16	No
5	1.0677e-12	8	0.8788	5.7003e-25	No
6	7.1116e-07	9	1.0428	2.5288e-13	No
7	3.6357e-10	10	1.0478	6.6091e-20	No
8	1.5711e-07	8	0.7995	1.2342e-14	No
9	6.3697e-09	8	0.9459	2.0287e-17	No
10	1.6768e-08	8	0.8643	1.4058e-16	No

Table 6.14: 10 random points. Finite difference on test function 3 with $h = 10^{-6}$ and $n = 10^3$

For test function 3, with $n = 10^3$, the algorithm converges for $k \in \{2,4,6\}$. It also exhibits a repeated need to modify the Hessian to render it sufficiently positive definite. Indeed, in the case of $h = 10^{-4}$, with the 7-th random point (Table (6.13)), the algoritm stops since it isn't able to find a sufficient positive definite Hessian. For $k \geq 8$, the algorithm stops at iteration 0 in all runs, as it fails to obtain a sufficiently positive-definite modification of the Hessian without drastically altering the second-order information contained in the original matrix.

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	1.8350e-08	9	85.4980	1.6834e-16	No
2	8.2887e-07	8	84.4229	3.4348e-13	No
3	44.4671	2	46.4120	539.3142	Yes
4	1.9218e-08	9	85.8793	1.8465e-16	No
5	1.0646e-07	11	111.1952	5.6663e-15	No
6	3.9037e-07	10	98.0888	7.6187e-14	No
7	3.3612e-08	9	86.1705	5.6483e-16	No
8	5.0798e-08	12	120.8831	1.2901e-15	No
9	9.2160e-09	9	91.9173	4.2463e-17	No
10	37.0385	2	35.9365	741.9929	Yes

Table 6.15: 10 random points. Finite difference on test function 3 with $h = 10^{-2}$ and $n = 10^4$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	2.0703e-07	8	102.2699	2.1430e-14	No
2	5.6439e-09	8	93.7614	1.5927e-17	No
3	46.0092	2	43.6804	634.0625	Yes
4	5.2845e-08	8	98.2443	1.3963e-15	No
5	1.8754e-09	12	111.1340	1.7586e-18	No
6	3.1417e-07	8	75.3251	4.9351e-14	No
7	1.4789e-07	8	74.8235	1.0936e-14	No
8	7.0105e-09	9	86.4766	2.4573e-17	No
9	4.6522e-09	9	86.5650	1.0821e-17	No
10	36.7881	2	35.7475	438.1279	Yes

Table 6.16: 10 random points. Finite difference on test function 3 with $h = 10^{-4}$ and $n = 10^4$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	8.8142e-09	11	109.3946	3.8845e-17	No
2	1.4885e-08	9	86.3518	1.1078e-16	No
3	35.9088	3	48.0786	510.2242	Yes
4	4.5325e-09	10	98.5794	1.0272e-17	No
5	5.7407e-08	11	109.5579	1.6478e-15	No
6	6.8001e-08	10	99.9728	2.3120e-15	No
7	18.3841	4	59.1101	97.5201	Yes
8	2.0194e-08	10	101.6040	2.0390e-16	No
9	7.9648e-08	10	109.0701	3.1719e-15	No
10	1.7636e-08	9	87.0694	1.5551e-16	No

Table 6.17: 10 random points. Finite difference on test function 3 with $h = 10^{-6}$ and $n = 10^4$

For $n = 10^4$, the algorithm converges in most runs; the occasional failures occur because, even after modifying the Hessian, it is not possible to obtain a matrix that is sufficiently positive definite without altering the second-order information it carries. Consequently, the procedure stops after a few iterations. The same behavior is observed for $k \geq 8$, where the algorithm always stops in principle for the same reason.

6.1.2 Runs and comments with $h_i = 10^{-k}\,|\hat{x}_i|$

The finite differences were also performed using a specific increment for each component $\hat{x}_i$ of $\hat{x}$ ($\hat{x} \in \mathbb{R}^n$ is the point at which the derivatives have to be approximated), according to the following formula:

$$h_i = 10^{-k} \left|\hat{x}_i\right|, \quad k = 2, 4, 6, 8, 10, 12$$

The functions that compute the gradient and the Hessian with this type of increment for the Test Function `X` are called: `finite_difference_1_test_functionX`.

Use of the suggested starting point

TEST FUNCTION 2	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	0.7965	8	0.7184	–426.9006	No
$k = 4$	6.9377e-06	320	27.6361	-427.4045	No
$k = 6$	4.1245e-04	9124	1.0591e+03	-427.4045	No
$k = 8$	3.0355e+04	31	4.9056	2.2576e+05	Yes
$k = 10$	3.0219e+04	7	1.2412	2.2389e+05	Yes
$k = 12$	3.0753e+04	0	0.1716	2.3092e+05	Yes

Table 6.18: Finite difference on test function 2 with $h = 10^{-k} \left|\hat{x}_i\right|$ and $n = 10^3$

TEST FUNCTION 2	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	0.1010	6	121.1606	-4.1581e+03	No
$k = 4$	190.9003	50	802.9772	-4.1053e+03	Yes
$k = 6$	2.9507e+05	13	364.6479	8.6107e+06	Yes
$k = 8$	4.8329e+05	12	377.8132	2.2826e+07	Yes
$k = 10$	4.8338e+05	7	231.3138	2.2849e+07	Yes
$k = 12$	5.0411e+05	0	28.8180	2.2996e+07	Yes

Table 6.19: Finite difference on test function 2 with $h = 10^{-k} \left|\hat{x}_i\right|$ and $n = 10^4$

As for the second test function, we observe several cases in which the algorithm fails to converge and instead stops after a few iterations. This occurs because, here as well, it is not possible to modify the Hessian to obtain a matrix that is sufficiently positive definite without excessively distorting the second-order information it carries. In Table (6.19), with $k = 2$, it has been decided to not declare failure since the value of f_k reached is very close to the one obtained in the previous successful runs. However, with $k = 4$, it has been decided to declare failure since the norm of the gradient is too high and the value f_k is not properly close to the one obtained in the successful runs.

TEST FUNCTION 3	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	1.2756e-07	5	0.6709	8.1354e-15	No
$k = 4$	0.0012	5	0.7620	7.1080e-07	No
$k = 6$	13.1495	1	0.2803	66.4708	Yes
$k = 8$	6.2292	0	0.1536	71.3380	Yes
$k = 10$	6.2284	0	0.1504	71.3380	Yes
$k = 12$	6.3033	0	0.1524	71.3380	Yes

Table 6.20: Finite difference on test function 3 with $h = 10^{-k} \left|\hat{x}_i\right|$ and $n = 10^3$

TEST FUNCTION 3	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
$k = 2$	4.0813e-07	5	59.2922	8.3284e-14	No
$k = 4$	1.7253	3	45.0527	0.8896	Yes
$k = 6$	12.9156	1	22.6628	253.3648	Yes
$k = 8$	19.6603	0	11.3850	712.8967	Yes
$k = 10$	26.3209	1	25.1090	642.3219	Yes
$k = 12$	20.1905	0	11.3901	712.8967	Yes

Table 6.21: Finite difference on test function 3 with $h = 10^{-k}\,|\hat{x_i}|$ and $n = 10^4$

For the third test function, the failures occur because we cannot obtain a Hessian that is sufficiently positive definite. In Table (6.20), for $k = 4$ we did not mark the run as a failure, even though $\|\nabla f(x_k)\|$ is not very small, because the objective value f_k is small enough to be treated as zero.

Use of random starting points

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	3.5860	11	0.9114	-213.6000	Yes
2	0.3369	300	21.0371	206.4096	Yes
3	0.1167	300	21.2749	389.5205	Yes
4	1.3851	57	4.1088	-161.5287	Yes
5	128.5622	7	0.6073	3.0049e+03	Yes
6	2.0914	300	21.1722	-301.2698	Yes
7	22.0799	300	21.7744	3.7718e+03	Yes
8	1.0239	300	21.7342	336.8953	Yes
9	23.3930	300	27.9926	2.7733e+03	Yes
10	14.9060	9	0.7475	3.7329e+03	Yes

Table 6.22: 10 random points. Finite difference on test function 2 with $h = 10^{-2}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	50.2535	300	21.6452	278.3795	Yes
2	145.8264	300	23.3413	-230.5606	Yes
3	853.1365	300	21.7939	1.0367e+03	Yes
4	11.5882	300	20.9940	-407.8217	Yes
5	31.7074	300	24.6835	-198.9984	Yes
6	0.0032	300	37.7896	-427.4045	No
7	17.8597	300	40.2145	-252.9724	Yes
8	270.5001	300	37.0112	850.2036	Yes
9	24.9146	300	23.3988	-161.3709	Yes
10	11.2913	9	31.5716	-284.1044	Yes

Table 6.23: 10 random points. Finite difference on test function 2 with $h = 10^{-4}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	5.5706e-06	1681	215.6112	-427.4045	No
2	9.3235e-06	2172	302.0655	-427.4045	No
3	9.8493e-06	2346	301.3558	-427.4045	No
4	12.6054	3000	332.6265	-426.7351	Yes
5	0.0628	3000	482.8633	-427.4044	No
6	3.5067e-06	2141	427.9737	-427.4045	No
7	9.7169e-06	2521	281.4531	-427.4045	No
8	3.8682e+03	38	3.1715	1.9638e+04	Yes
9	7.3416e-06	2118	260.8164	-427.4045	No
10	2.5654e-06	1639	226.1875	-427.4045	No

Table 6.24: 10 random points. Finite difference on test function 2 with $h = 10^{-6}$ and $n = 10^3$

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	1.4097e+04	144	14.2553	2.7219e+05	Yes
2	1.3946e+04	14	1.4339	2.7037e+05	Yes
3	1.3943e+04	269	26.4213	2.6265e+05	Yes
4	1.4238e+04	125	11.9638	2.7870e+05	Yes
5	1.4479e+04	40	3.8782	2.8888e+05	Yes
6	1.4093e+04	59	5.7774	2.7047e+05	Yes
7	1.3755e+04	28	2.6857	2.5646e+05	No
8	1.4071e+04	71	6.7337	2.7122e+05	Yes
9	1.3833e+04	94	8.8982	2.6966e+05	Yes
10	1.3849e+04	197	19.2243	2.5229e+05	Yes

Table 6.25: 10 random points. Finite difference on test function 2 with $h = 10^{-8}$ and $n = 10^3$

For the second test function, it is observed that for $k = 6$ the algorithm converges for almost all random starting points. In some cases (see Table (6.24), the 5-th run), the gradient norm is not exactly zero; however, the run was not labeled as a failure because the objective value f_k is very close to the optimal value. This is not the case of the 4-th run, where the gradient norm is too high.

In general, failures occur either because a Hessian that is sufficiently positive definite cannot be obtained, or because the algorithm gets stuck near a point until the iteration limit is reached. For $k = 10$ and $k = 12$, the results are not reported, as they are not very informative: the algorithm fails at the first iteration due to the inability to obtain a sufficiently positive definite Hessian.

Id	***gradfk_norm***	iter	**Execution time**	f_k	**Failure**
1	0.0098	8	1.1103	4.8015e-05	No
2	0.3942	7	0.9665	0.0699	Yes
3	0.0057	9	1.1925	1.5985e-05	No
4	0.4182	4	0.6103	0.0805	Yes
5	0.1984	5	0.7287	0.0184	Yes
6	0.3290	7	0.9812	0.0485	Yes
7	1.5332	10	1.3230	0.8040	Yes
8	8.9588	7	0.9739	17.4978	Yes
9	0.6120	8	1.0927	0.1499	Yes
10	0.3744	6	0.8499	0.0634	Yes

Table 6.26: 10 random points. Finite difference on test function 3 with $h = 10^{-2}$ and $n = 10^3$

Id	***gradfk__norm***	iter	**Execution time**	f_k	**Failure**
1	3.0847	6	80.7765	2.9825	Yes
2	8.0398	3	47.1599	16.1388	Yes
3	38.2466	2	43.7587	624.1553	Yes
4	17.1882	5	89.5673	61.8233	Yes
5	21.3379	2	47.4075	152.0639	Yes
6	24.2899	2	48.0397	579.3103	Yes
7	80.2168	0	17.5951	2.3292e+03	Yes
8	36.5411	3	62.0361	356.3375	Yes
9	17.7339	3	61.3482	65.7059	Yes
10	80.8270	0	17.5720	2.3371e+03	Yes

Table 6.27: 10 random points. Finite difference on test function 3 with $h = 10^{-2}$ and $n = 10^4$

For the third test function, we do not observe any notable results. The failures occur because it is not possible to modify the Hessian to make it sufficiently positive definite without overly changing the second–order information it contains. We report only the runs with $k = 2$, since for the other values of k the results are not particularly informative.

Chapter 7

Codes

7.0.1 Test functions

```
function F = test_function1(x)
n = length(x);

if n < 2
    error('The point x must have at least two components');
end

F = 0;
for i = 2:n
    F = F + 100 * (x(i-1)^2 - x(i))^2 + (x(i-1) - 1)^2;
end
end
```

```
function [y] = test_function2_1(x)
y=0;
for k=1:length(x)
    if k==1
        y = y + k*(1-cos(x(k))+sin(0)-sin(x(2)));
    elseif k==length(x)
        y = y + k*(1-cos(x(k))+sin(x(k-1))-sin(0));
    else
        y = y + k*(1-cos(x(k))+sin(x(k-1))-sin(x(k+1)));
    end
end
end
```

```
function [F] = test_function3(x)
n = length(x);
F=0;
for k = 1:n
    if k == 1
        fk = x(k); % fk(x) = xk per k = 1;
    else
        fk = cos(x(k-1)) + x(k) - 1; % fk(x) per 1 < k âĽď n;
    end
    F = F + fk^2;
end
F = 0.5 * F;
end
```

7.0.2 Nelder-Mead

Nelder Mead function

This function has been used also for the item 3. However, in that case, we have removed *simplex collection* because, with higher dimensions, it goes out of memory.

```

function [x_opt, f_opt, iter, rates, simplex_collection] = nelder_mead(f, x0)

% Parameters (fixed and not changeable)
rho = 1;          % Reflection coefficient
chi = 2;          % Expansion coefficient
gamma = 0.5;      % Contraction coefficient
sigma = 0.5;      % Shrinkage coefficient
tol = 1e-13;       % Tolerance for convergence
maxIter = 60000;   % Maximum number of iterations
rates = zeros(maxIter, 1);
X4 = zeros(length(x0), 4);
% Input validation
if nargin < 2
    error('Not enough input arguments. Provide at least a function handle and initial guess.');
end
if ~isa(f, 'function_handle')
    error('First argument must be a function handle.');
end
if ~isvector(x0)
    error('Second argument must be a vector as the initial guess.');
end

% Initialization
```

```
n = length(x0); % Dimension of the problem
simplex = zeros(n, n+1); % Preallocate simplex matrix
simplex(:, 1) = x0; % First column is the initial guess
simplex_collection = zeros(n, n+1, maxIter);


for i = 2:n+1
    perturbation = zeros(n, 1);
    perturbation(i-1) = 1; % Perturb along each axis
    simplex(:, i) = x0 + perturbation; % Perturbed vertices
end

% Evaluate the function at the vertices of the simplex
f_vals = zeros(1, n+1);
for i = 1:n+1
    f_vals(i) = f(simplex(:, i));
end
iter=1;
while iter < maxIter && max(abs(f_vals-mean(f_vals)))>=tol
    [f_vals, idx] = sort(f_vals);
    simplex = simplex(:,idx);
    if iter<=4
        X4(:, iter) = simplex(:, 1);
        if iter==4
            rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:,
               2)))/log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4
               (:, 1)));
            rates(iter)=rate;
        end
    else
        X4 = [X4(:, 2:end), simplex(:, 1)];
        rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:, 2)))/
           log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4(:, 1)));
        rates(iter)=rate;
    end

    simplex_collection(:, :, iter) = simplex;

    %reflection phase
    baricenter = mean(simplex(:, 1:n), 2);
    x_reflection = baricenter + rho*(baricenter-simplex(:,n+1));
    f_reflection = f(x_reflection);
    if f_vals(1) <= f_reflection && f_reflection < f_vals(n)
       simplex(:,n+1) = x_reflection;
       f_vals(n+1) = f_reflection;

    %expansion phase
    elseif f_reflection < f_vals(1)

```

```
       x_expansion = baricenter + chi*(x_reflection-baricenter);
       f_expansion = f(x_expansion);

       if f_expansion < f_reflection
           simplex(:,n+1) = x_expansion;
           f_vals(n+1) = f_expansion;
       else
           simplex(:,n+1) = x_reflection;
           f_vals(n+1) = f_reflection;
       end

    %contraction phase
    elseif f_reflection >= f_vals(n)

        if f_vals(n+1) < f_reflection
            x_contraction = baricenter - gamma*(baricenter-simplex(:,n+1));
        else
            x_contraction = baricenter - gamma*(baricenter-x_reflection);
        end
        f_contraction = f(x_contraction);

        if f_contraction < f_vals(n+1)
            simplex(:,n+1) = x_contraction;
            f_vals(n+1) = f_contraction;
        else
            %shrinkage phase
            for j=2:n+1
                simplex(:,j) = simplex(:,1) + sigma*(simplex(:,j)-simplex(:,1));
                f_vals(j) = f(simplex(:,j));
            end
        end
    end

    iter = iter+1;

end
x_opt = simplex(:,1);
f_opt = f(x_opt);
end
```

Main Nelder Mead of item 2

```
clear all
clc
rng(346884)
% Rosenbrock function
rosenbrock = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;

% Initial points
x0_1 = [1.2; 1.2];
x0_2 = [-1.2; 1];

% Perform optimization for the first starting point
tic
[x_opt1, f_opt1, iter1, simplex_collection] = nelder_mead(
    rosenbrock, x0_1);
toc
disp(iter1)
[X1, Y1] = meshgrid(linspace(-10, 10, 500), linspace(-10, 10, 500)
    );
Z1 = arrayfun(@(x, y) rosenbrock([x; y]), X1, Y1);
figure(1);
% Contour plot with curve levels for each point in xseq
[C1, ~] = contour(X1, Y1, Z1, 50);
hold on
plot(x_opt1(1), x_opt1(2), 'bo', 'MarkerSize', 10, 'LineWidth', 2)
    ;
hold on
for i = 1:iter1
    simplex = simplex_collection(:, :, i); % Vertici del simplesso
         per l'iterazione i
    fill(simplex(1, :), simplex(2, :), 'y', 'FaceAlpha', 0.3, '
        EdgeColor', 'k'); % Disegna il simplesso
    pause(2); % Per animare l'evoluzione
end
hold off
figure(2);
surf(X1, Y1, Z1,'EdgeColor','none')
hold on

tic
[x_opt2, f_opt2, iter2, simplex_collection] = nelder_mead(
    rosenbrock, x0_2);
toc
disp(iter2)
[X1, Y1] = meshgrid(linspace(-10, 10, 500), linspace(-10, 10, 500)
    );
Z1 = arrayfun(@(x, y) rosenbrock([x; y]), X1, Y1);
figure(3);
```

```
% Contour plot with curve levels for each point in xseq
[C1, ~] = contour(X1, Y1, Z1, 50);
hold on
plot(x_opt2(1), x_opt2(2), 'bo', 'MarkerSize', 10, 'LineWidth', 2)
    ;
hold on
for i = 1:iter2
    simplex = simplex_collection(:, :, i); % Vertici del simplesso
        per l'iterazione i
    fill(simplex(1, :), simplex(2, :), 'y', 'FaceAlpha', 0.3, '
        EdgeColor', 'k'); % Disegna il simplesso
    pause(0.1); % Per animare l'evoluzione
end
hold off
```

Main Nelder Mead of item 3

```
function [x0] = create_x0(n)
x0=zeros(n,1);
for i=1:2:n
    x0(i)=-1.2;
end
for i=2:2:n
    x0(i)=1;

end
end
```

```
clear all
clc
n = 10;
rng(346884)


fprintf('First test function\n');
f1 = @test_function1;
x0_f1 = create_x0(n);
tic
[x_opt, f_opt, iter, rates] = nelder_mead(f1, x0_f1);
elapsed_time=toc;

fprintf('Ultimi 10 valori di rates: %.4f ', rates(max(1, end-9):
    end)); fprintf('\n');
plot(1:k-3, rates(4:k), '-o', 'LineWidth', 1.5, 'MarkerSize', 8, '
    MarkerFaceColor', 'r');


```

```matlab
lower_bound = x0_f1-ones(n,1);
upper_bound = x0_f1+ones(n,1);
for j=1:10
    x0_random1 = lower_bound + (upper_bound-lower_bound).*rand(n
        ,1);
    tic
    [x_opt, f_opt, iter] = nelder_mead(f1, x0_random1);
    elapsed_time=toc;
    fprintf('elapsed time = %d seconds with %d(th) random vector\n
        ', elapsed_time, j);
    disp(x_opt)
    disp(f_opt)
    disp(iter)
end

fprintf('\n')


%%
clear all
clc
rng(346884)
n = 10 ;
fprintf('\n\n');
fprintf('Second test function\n');
f2 = @test_function2_1;
x0_f2 = ones(n,1);
tic
[x_opt, f_opt, iter, rates] = nelder_mead(f2, x0_f2);
elapsed_time=toc;
fprintf('elapsed time = %d seconds with initial vector suggested
    for this function\n', elapsed_time);
fprintf('Ultimi 10 valori di rates: %.4f ', rates(max(1, end-9):
    end)); fprintf('\n');
lower_bound = x0_f2-ones(n,1);
upper_bound = x0_f2+ones(n,1);
for j=1:10
    x0_random2 = lower_bound + (upper_bound-lower_bound).*rand(n
        ,1);
    tic
    [x_opt, f_opt, iter] = nelder_mead(f2, x0_random2);
    elapsed_time=toc;
    fprintf('elapsed time = %d seconds with %d(th) random vector\n
        ', elapsed_time, j);
    disp(x_opt)
    disp(f_opt)
    disp(iter)
end
fprintf('\n')
```

```


%%
clear all
clc
n = 10;
rng(346884)
fprintf('\n\n');
fprintf('Third test function\n');
f3 = @test_function3;
x0_f3 =0.5*ones(n,1);
tic
[x_opt, f_opt, iter,rates] = nelder_mead(f3, x0_f3);
elapsed_time=toc;
fprintf('elapsed time = %d seconds with initial vector suggested
    for this function\n', elapsed_time);
fprintf('Ultimi 10 valori di rates: %.4f ', rates(max(1, end-9):
    end)); fprintf('\n');
plot(length(rates)-9:length(rates), rates(end-9:end), '-o', '
    LineWidth', 1.5), title('Grafico degli ultimi 10 valori di
    rates'), xlabel('Indice'), ylabel('Valore'), grid on;


lower_bound = x0_f3-ones(n,1);
upper_bound = x0_f3+ones(n,1);
for j=1:10
    x0_random3 = lower_bound + (upper_bound-lower_bound).*rand(n
        ,1);
    tic
    [x_opt, f_opt, iter] = nelder_mead(f3, x0_random3);
    elapsed_time=toc;
    fprintf('elapsed time = %d seconds with %d(th) random vector\n
        ', elapsed_time, j);
    disp(x_opt)
    disp(f_opt)
    disp(iter)
end
fprintf('\n')
```

7.0.3 Modified Newton Method

Modified Newton Method for item2

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rates] = ...
newton_bcktrck(x0, f, gradf, Hessf, ...
kmax, tolgrad, c1, rho, btmax)

farmijo = @(fk, alpha, c1_gradfk_pk) fk + alpha * c1_gradfk_pk;
```

```matlab

% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);
rates = zeros(kmax, 1);
X4 = zeros(length(x0), 4);
xk = x0;
fk = f(xk);
gradfk = gradf(xk);
Hessfk = Hessf(xk);
k = 1;
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    try
        Rk = check_positive_definitess(Hessfk);
    catch ME
        disp('No sufficient positive definite Hessian found')
        break
    end
    R_hessf = sparse(Rk);
    y = R_hessf'\-gradfk;
    pk = R_hessf\y;

    % Reset the value of alpha
    alpha = 1;

    % Compute the candidate new xk
    xnew = xk + 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 = xk + alpha * pk;
        fnew = f(xnew);

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

    % Update xk, fk, gradfk_norm
    xk = xnew;
    fk = fnew;
    if k<=4
        X4(:, k) = xk;
        if k==4
            rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:,
               2)))/log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4
               (:, 1)));
            rates(k)=rate;
        end
    else
        X4 = [X4(:, 2:end), xk];
        rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:, 2)))/
           log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4(:, 1)));
        rates(k)=rate;
    end
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk);
    Hessfk = Hessf(xk);
    % Increase the step by one
    k = k + 1;

    % Store current xk in xseq
    xseq(:, k) = xk;
    % Store bt iterations in btseq
    btseq(k) = bt;
end

xseq = xseq(:, 1:k);
btseq = btseq(1:k);
xseq = [x0, xseq];

end
```

Main of Modified Newton Method of item 2

```
clear all
clc
rosenbrock = @(x) 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;
gradf = @(x)[-400*x(1)*(x(2)-x(1)^2)-2*(1-x(1)); 200*(x(2)-x(1)^2)
   ];
hessf = @(x)[-400*(x(2)-3*x(1)^2)+2, -400*x(1); -400*x(1), 200];
x0_1 = [1.2; 1.2];
x0_2 = [-1.2; 1];


```

```
[xk, fk, gradfk_norm, k, xseq, btseq, rates] = ...
newton_bcktrck(x0_1, rosenbrock, gradf, hessf, ...
    1000, 1e-6, 1e-4, 0.5, 100);

plot(1:k-3, rates(4:k), '-o', 'LineWidth', 1.5, 'MarkerSize', 8, '
    MarkerFaceColor', 'r');

[X, Y] = meshgrid(linspace(-6, 6, 500), linspace(-6, 6, 500));
Z = arrayfun(@(x, y) rosenbrock([x; y]), X, Y);
figure(1);
% Contour plot with curve levels for each point in xseq
[C1, ~] = contour(X, Y, Z);
hold on
plot(xseq(1, :), xseq(2,:), '--*')
hold off

fig3 = figure();
bar(btseq)

rosenbrock_vals = arrayfun(@(i) rosenbrock(xseq(:, i)), 1:size(
    xseq, 2));
fig4 = figure();
surf(X, Y, Z,'EdgeColor','none')
hold on
plot3(xseq(1, :), xseq(2, :), rosenbrock_vals, 'r--*')
hold off


%%

x0_2 = [-1.2; 1];
[xk, fk, gradfk_norm, k, xseq, btseq] = ...
newton_bcktrck(x0_2, rosenbrock, gradf, hessf, ...
    1000, 1e-6, 1e-4, 0.5, 100);

figure(5);
% Contour plot with curve levels for each point in xseq
[C1, ~] = contour(X, Y, Z);
hold on
plot(xseq(1, :), xseq(2,:), '--*')
hold off

fig6 = figure();
bar(btseq)

rosenbrock_vals = arrayfun(@(i) rosenbrock(xseq(:, i)), 1:size(
    xseq, 2));
fig7 = figure();
surf(X, Y, Z,'EdgeColor','none')
```

```
hold on
plot3(xseq(1, :), xseq(2, :), rosenbrock_vals, 'r--*')
hold off
```

Computation of the gradient and the Hessian

```
function [H, grad] = hessian_gradient_test_function1(x)
n = length(x);

% compute the Hessian
H = sparse(n, n);
for i = 2:n
        % Main diagonal
        H(i-1, i-1) = H(i-1, i-1) + (1200 * x(i-1)^2 - 400 * x(i)
            + 2);
        H(i, i) = H(i, i) + 200;

        % Elements outside the main diagonal
        H(i-1, i) = - 400 * x(i-1);
        H(i, i-1) = H(i-1, i); % Exploit the symmetry of the
            Hessian
end

% compute the gradient
grad = zeros(n,1);


for i=1:n
    if i==1
        grad(i) = 400*x(i)*(x(i)^2-x(i+1))+2*(x(i)-1);
    elseif i==n
        grad(i) = -200*(x(i-1)^2-x(i));
    else
        grad(i) = -200*(x(i-1)^2-x(i)) + 400*x(i)*(x(i)^2-x(i+1))
            +2*(x(i)-1);
    end
end
end
```

```
    function [H, grad] = hessian_gradient_test_function2_1(x)
    n=length(x);
    H = sparse(n,n);
    H(1,1) = cos(x(1))-2*sin(x(1));
    H(n,n) = (n-1)*sin(x(n)) + n*cos(x(n));
    for k=2:n-1
        H(k,k) = -2*sin(x(k)) + k*cos(x(k));
    end

    grad = zeros(n,1);
    grad(1) = sin(x(1)) + 2*cos(x(1));
    grad(n) = -(n-1)*cos(x(n)) + n*sin(x(n));
    for k=2:n-1
        grad(k) = 2*cos(x(k)) + k*sin(x(k));
    end
end
```

```
    function [H, grad] = hessian_gradient_test_function3(x)
    n=length(x);
    H = sparse(n, n);
    H(1, 1) = 1+(sin(x(1)))^2-(cos(x(1)))^2-x(2)*cos(x(1))+cos(x
       (1));
    H(1, 2) = -sin(x(1));
    for k=2:n-1
        H(k, k) = 1+(sin(x(k)))^2-(cos(x(k)))^2-x(k+1)*cos(x(k))+
           cos(x(k));
        H(k, k-1) = -sin(x(k-1));
        H(k, k+1) = -sin(x(k));
    end
    H(n, n-1) = -sin(x(n-1));
    H(n,n) = 1;

    grad = zeros(n, 1);
    grad(1) = x(1)-cos(x(1))*sin(x(1))-sin(x(1))*x(2)+sin(x(1));
    for k=2:n-1
        grad(k) = cos(x(k-1))+x(k)-1-sin(x(k))*(cos(x(k))+x(k+1)
           -1);
    end
    grad(n) = cos(x(n-1))+x(n)-1;
    end
```

Checking positive definitess of the Hessian

```
function [R] = check_positive_definitess(Hk)
beta = 10^-3;
min_el_diag = min(diag(Hk));
already_notified = 0;
tau_max=5;
tau=0;
if min_el_diag>0
    tau=0;
else
    tau = -min_el_diag+beta;
end
while 1
    Bk = sparse(Hk+tau*speye(size(Hk)));
    try
        R = sparse(chol(Bk));
        break;
    catch ME
        if contains(ME.message, 'positive definite')
            if already_notified==0
                disp('the modifing of the Hessian was needed')
                already_notified=1;
            end
            tau = max([2*tau, beta]);
            if tau>=tau_max
                error('Tau exceeded the maximum allowed value (%e)
                    ', tau_max)
            end
        end

    end
end
end
```

Main of Modified Newton Method, item 3

```matlab
clear all
clc
rng(346884)
n = 10;
f1 = @test_function1;
x0 = create_x0(n);
[xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_function1(x0, f1, ...
    100000, 1e-6, 1e-4, 0.5, 100);
disp(gradfk_norm)
disp(k)
disp(xk)

lower_bound1 = x0-ones(n,1);
upper_bound1 = x0 + ones(n,1);
for i=1:10
    x0_random1 = lower_bound1 + (upper_bound1-lower_bound1).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_function1(x0_random1, f1, ...
    200000, 1e-6, 1e-4, 0.8, 145);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    disp(xk)
    fprintf('\n')
end


%%
clear all
clc
n = 10^3;
rng(346884)
f2 = @test_function2_1;
x0 = ones(n,1);
[xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_function2_1(x0, f2, ...
    200000, 1e-6, 1e-4, 0.8, 145);
disp(gradfk_norm)
disp(k)
disp(xk)


lower_bound2 = x0-ones(n,1);
upper_bound2 = x0 + ones(n,1);
for i=1:10
```

```
    x0_random2 = lower_bound2 + (upper_bound2-lower_bound2).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_function2_1(x0_random2, f2, ...
    200000, 1e-6, 1e-4, 0.8, 145);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    disp(xk)
    fprintf('\n')
end


%%
clear all
clc
n = 10^5;
rng(346884)
f3 = @test_function3;
x0 = 0.5*ones(n,1);
[xk, fk, gradfk_norm, k, rates] = ...
    newton_bcktrck_item3_function3(x0, f3, ...
    10000, 1e-8, 1e-4, 0.5, 47);
disp(gradfk_norm)
disp(k)
disp(xk)
%plot(1:k-3, rates(4:k), '-o', 'LineWidth', 1.5, 'MarkerSize', 8,
    'MarkerFaceColor', 'r');

lower_bound3 = x0-ones(n,1);
upper_bound3 = x0 + ones(n,1);
for i=1:10
    fprintf('\n')
    fprintf('%d-th random point', i)
    x0_random3 = lower_bound3 + (upper_bound3-lower_bound3).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_function3(x0_random3, f3, ...
    10000, 1e-8, 1e-4, 0.5, 47);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    %disp(xk)
    fprintf('\n')
end
```

Modified Newton Method of item 3

Note that for each test function, we have created a specific function for the Modified Newton Method; however, the only difference lies in the calculation of the gradient and the Hessian, which vary depending on the test function. We report only the one related to the first test function, since the other Newton backtracking functions applied to the remaining test functions differ solely in the choice of the gradient and Hessian computation.

```
function [xk, fk, gradfk_norm, k, rates] = ...
    newton_bcktrck_item3_function1(x0, f, ...
    kmax, tolgrad, c1, rho, btmax)


farmijo = @(fk, alpha, c1_gradfk_pk) fk + alpha * c1_gradfk_pk;

rates = zeros(kmax,1);
X4 = zeros(length(x0), 4);
xk = x0;
fk = f(xk);
[Hessfk, gradfk] = hessian_gradient_test_function1(xk);
k = 1;
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    try
        Rk = check_positive_definitess(Hessfk);
    catch ME
        break
    end
    R_hessf = sparse(Rk);
    y = R_hessf'\-gradfk;
    pk = R_hessf\y;

    % Reset the value of alpha
    alpha = 1;

    % Compute the candidate new xk
    xnew = xk + 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
```

```matlab
        xnew = xk + alpha * pk;
        fnew = f(xnew);

        % Increase the counter by one
        bt = bt + 1;
    end
    if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
        disp('Maximum number of backtracks has been reached')
        break
    end

    % Update xk, fk, gradfk_norm
    xk = xnew;
    fk = fnew;
    %compute experimental rate of convergence
    if k<=4
        X4(:, k) = xk;
        if k==4
            rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:, 2)))/log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4(:, 1)));
            rates(k)=rate;
        end
    else
        X4 = [X4(:, 2:end), xk];
        rate = log(norm(X4(:,4)-X4(:,3))/norm(X4(:, 3)-X4(:, 2)))/log(norm(X4(:, 3)-X4(:, 2))/norm(X4(:, 2)-X4(:, 1)));
        rates(k)=rate;
    end

    [Hessfk, gradfk] = hessian_gradient_test_function1(xk);
    gradfk_norm = norm(gradfk);

    % Increase the step by one
    k = k + 1;


end
end
```

Approximation of gradient and Hessian with finite differences

```
function [H, grad] =finite_difference_function1(x_bar, f, k)
n = length(x_bar);
grad = zeros(n, 1);
f_bar = f(x_bar);
h = 10^-k;
e_i = zeros(n, 1);
for i=1:n
    e_i(i) = 1;
    grad(i) = (f(x_bar+h.*e_i)-f(x_bar-h.*e_i))/(2*h);
    e_i(i)=0;
end
E = speye(n,n);
row = [];
col = [];
val = [];

for i = 1:n
    xi_p = x_bar + h*E(:,i);
    xi_m = x_bar - h*E(:,i);
    Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h^2);
    row = [row, i];   col = [col, i];   val = [val, Hii];
end

for i = 1:n-1
    xpp = x_bar + h*E(:,i) + h*E(:,i+1);
    xpm = x_bar + h*E(:,i);
    xmp = x_bar + h*E(:,i+1);
    Hij = ( f(xpp) - f(xpm) - f(xmp) + f_bar ) / (h^2);

    row = [row, i,   i+1];
    col = [col, i+1, i   ];
    val = [val, Hij, Hij ];
end
H = sparse(row, col, val, n, n);

end
```

```matlab
function [H, grad] =finite_difference_function2(x_bar, f, k)
n = length(x_bar);
grad = zeros(n, 1);
f_bar = f(x_bar);
h = 10^-k;
e_i = zeros(n, 1);
for i=1:n
    e_i(i) = 1;
    grad(i) = (f(x_bar + h*e_i) - f(x_bar - h*e_i))/h;
    e_i(i)=0;
end
E = speye(n,n);
row = [];
col = [];
val = [];
for i = 1:n
    xi_p = x_bar + h*E(:,i);
    xi_m = x_bar - h*E(:,i);
    Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h^2);
    row = [row, i];   col = [col, i];   val = [val, Hii];
end
H = sparse(row, col, val, n, n);

end
```

```matlab
function [H, grad] =finite_difference_function3(x_bar, f, k)
n = length(x_bar);
grad = zeros(n, 1);
f_bar = f(x_bar);
h = 10^-k;
e_i = zeros(n, 1);
for i=1:n
    e_i(i) = 1;
    grad(i) = ( f(x_bar + h*e_i) - f(x_bar - h*e_i) ) / (2*h);
    e_i(i)=0;
end
E = speye(n,n);
row = [];
col = [];
val = [];
for i = 1:n
    xi_p = x_bar + h*E(:,i);
    xi_m = x_bar - h*E(:,i);
    Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h^2);
    row = [row, i]; col = [col, i]; val = [val, Hii];
end
for i = 1:n-1
    xpp = x_bar + h*E(:,i) + h*E(:,i+1);
```

```
        xpm = x_bar + h*E(:,i);
        xmp = x_bar + h*E(:,i+1);
        Hij = ( f(xpp) - f(xpm) - f(xmp) + f_bar ) / (h^2);

        row = [row, i,   i+1];
        col = [col, i+1, i   ];
        val = [val, Hij, Hij ];
    end
    H = sparse(row, col, val, n, n);

    end
```

Approximation of gradient and Hessian with finite differences, using the special increment

```
    function [H, grad] =finite_difference_1_function1(x_bar, f, k)
    n = length(x_bar);
    grad = zeros(n, 1);
    f_bar = f(x_bar);
    h=10^-k;
    e_i = zeros(n, 1);
    for i=1:n
        e_i(i) = 1;
        grad(i) = (f(x_bar+h*abs(x_bar(i).*e_i))-f(x_bar-h*abs(
            x_bar(i).*e_i)))/(2*h*abs(x_bar(i)));
        e_i(i)=0;
    end
    E = speye(n,n);
    row = [];
    col = [];
    val = [];
    for i = 1:n
        xi_p = x_bar + h*abs(x_bar(i))*E(:,i);
        xi_m = x_bar - h*abs(x_bar(i))*E(:,i);
        Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h^2);
        row = [row, i];   col = [col, i];   val = [val, Hii];
    end

    for i = 1:n-1
        xpp = x_bar + h*abs(x_bar(i))*E(:,i) + h*abs(x_bar(i+1))*E
            (:,i+1);
        xpm = x_bar + h*abs(x_bar(i))*E(:,i);
        xmp = x_bar + h*abs(x_bar(i+1))*E(:,i+1);
        Hij = ( f(xpp) - f(xpm) - f(xmp) + f_bar ) / (h*abs(x_bar(
            i)))^2;

        row = [row, i,   i+1];
        col = [col, i+1, i   ];
        val = [val, Hij, Hij ];
    end
    H = sparse(row, col, val, n, n);

    end
```

```
    function [H, grad] =finite_difference_1_function2(x_bar, f, k)
    n = length(x_bar);
    grad = zeros(n, 1);
    f_bar = f(x_bar);
    h = 10^-k;
    e_i = zeros(n, 1);
    for i=1:n
        e_i(i) = 1;
        grad(i) = (f(x_bar+(h*abs(x_bar(i))).*e_i)-f(x_bar-(h*abs(
           x_bar(i))).*e_i))/(2*h*abs(x_bar(i)));
        e_i(i)=0;
    end
    E = speye(n,n);
    row = [];
    col = [];
    val = [];
    for i = 1:n
        xi_p = x_bar + (h*abs(x_bar(i)))*E(:,i);
        xi_m = x_bar - (h*abs(x_bar(i)))*E(:,i);
        Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h*abs(x_bar(i)))
           ^2;
        row = [row, i];
        col = [col, i];
        val = [val, Hii];
    end
    H = sparse(row, col, val, n, n);

    end
```

```
    function [H, grad] =finite_difference_1_function3(x_bar, f, k)
    n = length(x_bar);
    grad = zeros(n, 1);
    f_bar = f(x_bar);
    h=10^-k;
    e_i = zeros(n, 1);
    for i=1:n
        e_i(i) = 1;
        grad(i) = (f(x_bar+(h*abs(x_bar(i))).*e_i)-f(x_bar-(h*abs(
           x_bar(i))).*e_i))/(2*h*abs(x_bar(i)));
        e_i(i)=0;
    end
    E = speye(n,n);
    row = [];
    col = [];
    val = [];
    for i = 1:n
        xi_p = x_bar + (h*abs(x_bar(i)))*E(:,i);
        xi_m = x_bar - (h*abs(x_bar(i)))*E(:,i);
        Hii  = ( f(xi_p) - 2*f_bar + f(xi_m) ) / (h*abs(x_bar(i)))
           ^2;
        row = [row, i]; col = [col, i]; val = [val, Hii];
    end
    for i = 1:n-1
        xpp = x_bar + (h*abs(x_bar(i)))*E(:,i) + (h*abs(x_bar(i+1)
           ))*E(:,i+1);
        xpm = x_bar + (h*abs(x_bar(i)))*E(:,i);
        xmp = x_bar + (h*abs(x_bar(i+1)))*E(:,i+1);
        Hij = ( f(xpp) - f(xpm) - f(xmp) + f_bar ) / (h*abs(x_bar(
           i)))^2;

        row = [row, i,   i+1];
        col = [col, i+1, i   ];
        val = [val, Hij, Hij ];
    end

    H = sparse(row, col, val, n, n);
    end
```

Modified Newton Method of item 3, with the Hessian and Gradient computed with finite differences

Note that for each test function, we have created a specific function for the Modified Newton Method; however, the only difference lies in the calculation of the gradient and the Hessian, which vary depending on the test function. We report only the one related to the first test function.

```
function [xk, fk, gradfk_norm, k, m] = ...
newton_bcktrck_item3_1_function1(x0, f, ...
kmax, tolgrad, c1, rho, btmax)

farmijo = @(fk, alpha, c1_gradfk_pk) fk + alpha * c1_gradfk_pk;

xk = x0;
fk = f(xk);
[Hessfk, gradfk] = finite_difference_function1(xk, f, 12);
k = 0;
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    try
        Hk = check_positive_definitess(Hessfk);
    catch ME
        disp('No sufficient positive definite Hessian found')
        m=1;
        break
    end
    disp(cond(Hk))
    R_hessf = sparse(Hk);
    y = R_hessf'\-gradfk;
    pk = R_hessf\y;

    % Reset the value of alpha
    alpha = 1;

    % Compute the candidate new xk
    xnew = xk + 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 = xk + alpha * pk;
        fnew = f(xnew);

        % Increase the counter by one
        bt = bt + 1;
    end
    if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
        disp('Armijo not satisfied')
        m=2;
        break
    end

    % Update xk, fk, gradfk_norm
    xk = xnew;
    fk = fnew;

    [Hessfk, gradfk] = finite_difference_function1(xk, f, 12);
    gradfk_norm = norm(gradfk);

    % Increase the step by one
    k = k + 1;
end
end
```

7.0.4 Main of Modified Newton Method with finite differences

```
clear all
clc
rng(346884)
n = 10^3;
f1 = @test_function1;
x0 = create_x0(n);
tic
[xk, fk, gradfk_norm, k, m] = ...
    newton_bcktrck_item3_1_function1(x0, f1, ...
    1000000, 1e-6, 1e-10, 0.8, 145);
el = toc;
disp(el)
disp(gradfk_norm)
disp(k)
disp(xk)



lower_bound1 = x0-ones(n,1);
upper_bound1 = x0 + ones(n,1);
x_prev1 = zeros(n, 1);
for i=1:10
    fprintf('%d - random point', i)
    fprintf('\n')
    x0_random1 = lower_bound1 + (upper_bound1-lower_bound1).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_1_function1(x0_random1, f1, ...
    200000, 1e-6, 1e-4, 0.8, 145);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    fprintf('The difference between the (%d)-th x_opt and the
       previous one is: (%e)\n', i, norm(xk-x_prev1))
    x_prev1 = xk;
    fprintf('\n')
end




%%
clear all
clc
```

```
n = 10^3;
rng(346884)
f2 = @test_function2_1;
x0 = ones(n,1);
tic
[xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_1_function2_1(x0, f2, ...
    1000000, 1e-6, 1e-4, 0.5, 45);
disp(toc)
disp(gradfk_norm)
disp(k)
disp(xk)


lower_bound2 = x0-ones(n,1);
upper_bound2 = x0 + ones(n,1);
x_prev2 = zeros(n, 1);
for i=1:10
    fprintf('%d - random point', i)
    fprintf('\n')
    x0_random2 = lower_bound2 + (upper_bound2-lower_bound2).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_1_function2_1(x0_random2, f2, ...
    200000, 1e-6, 1e-4, 0.8, 145);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    %disp(xk)
    fprintf('The difference between the (%d)-th x_opt and the
       previous one is: (%e)\n', i, norm(xk-x_prev2))
    x_prev2 = xk;
    fprintf('\n')
end

%%

clear all
clc
n = 10^3;
rng(346884)
f3 = @test_function3;
x0 = 0.5*ones(n,1);
tic
[xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_1_function3(x0, f3, ...
    10000, 1e-8, 1e-4, 0.5, 47);
disp(toc)
disp(gradfk_norm)
disp(k)
disp(xk)

lower_bound3 = x0-ones(n,1);
upper_bound3 = x0 + ones(n,1);
x_prev3 = zeros(n,1);
for i=1:10
    fprintf('%d - random point', i)
    fprintf('\n')
    x0_random3 = lower_bound3 + (upper_bound3-lower_bound3).*rand(
       n,1);
    [xk, fk, gradfk_norm, k] = ...
    newton_bcktrck_item3_1_function3(x0_random3, f3, ...
    10000, 1e-8, 1e-4, 0.5, 47);
    disp(gradfk_norm)
    disp(fk)
    disp(k)
    %disp(xk)
    fprintf('The difference between the (%d)-th x_opt and the
       previous one is: (%e)\n', i, norm(xk-x_prev3))
    x_prev3 = xk;
    fprintf('\n')
end
```

Bibliografia

[1] Marek Molga and Cezary Smutnicki. Test problems for unconstrained optimization. https://www.researchgate.net/publication/325314497, 2018. Accessed: 2025-01-17.