Politecnico di Torino

POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Matematica

Assignement of Numerical Optimization on Unconstrained Optimization

Cristina Baitan 343428
Cecilia Bergamini 343341

Anno Accademico 2024-2025

Contents

Abstract

This report addresses the implementation and analysis of two distinct numerical methods: the Nelder-Mead simplex method and the Modified Newton method.
The Nelder-Mead method is particularly useful for non-differentiable functions as it does not require gradient information. On the other hand, the Modified Newton method represents a gradient-based approach, relying on second-order information.
Both methods have been implemented and tested on the Rosenbrock function using two different initial conditions, in order to evaluate their performance and robustness. Additionally, the methods have been applied to a set of three test problems from the literature, considering various dimensions and starting points, as well as exact and approximated gradients (via finite differences).
The performance of each method has been assessed and compared through several metrics, including the distance from optimum, the number of successful runs, iteration count, execution time and convergence rate.
It is worth noting, however, that the two methods are applied under different experimental conditions. In particular, the Modified Newton method was tested in significantly higher-dimensional settings and the stopping criteria with the associated tolerances differed between the two methods. Consequently, it is difficult to extrapolate general conclusions about the comparative performance of these two methods.

Chapter 1

We chose to study two methods of unconstrained optimization: the Nelder-Mead method and the Modified Newton Method.

1.1 Nelder-Mead

The **Nelder-Mead method**'s implementation presented is developed from the basic version provided in the course: it involves updating a simplex that moves through the solution space in search of the minimum of a given function. This process is carried out in different phases: contraction, expansion, reflection, and shrinking. The additional features of this implementation include two stopping criteria, in addition to maximum number of iteration.

1.1.1 Stopping Criterion Based on the simplex size

The first stopping criterion is based on the size of the simplex. The algorithm stops when the maximum distance between the centroid of the simplex and its vertices is less than a specified tolerance, named *tol1*. This condition indicates that the simplex has shrunk, and its points have become very close to each other. This can be interpreted as the symplex converging to a single point, suggesting that a solution has been found and the algorithm terminates, with a flag, "1", signaling convergence to a point.
This tolerance has been set to 10^{-7}. So if the maximum distance between the centroid of the simplex and its vertices is less than 10^{-7} we consider the simplex as it has converge to a single point and the method stops.
As we will see in the next chapters, with the functions tested and the initial points tested, this stopping criterion was hardly ever triggered in high dimensions. We observe that the simplex eventually doesn't reduce his dimension any more (see Figure 1.1 and 1.2 as an example). Probably this derives from the fact that the simplex starts to oscillate without significantly reducing the distances between its vertices. This is likely due to the fact that the simplex is "trapped" in a region of the function where the reflection, expansion, or contraction operations do not produce substantial improvements.
This situation can occur because there is not enough difference in the function values within the simplex: the simplex becomes small enough that the curve defining the function appears almost flat. When the function is nearly flat, the simplex can oscillate because the reflection, expansion, and contraction operations cannot find directions of improvement.
In order to avoid wasting computational effort, we decided to add an additional stopping criterion to prevent stagnation when the function shows very little variation at the vertices of the simplex. The following paragraph provides further explanation.

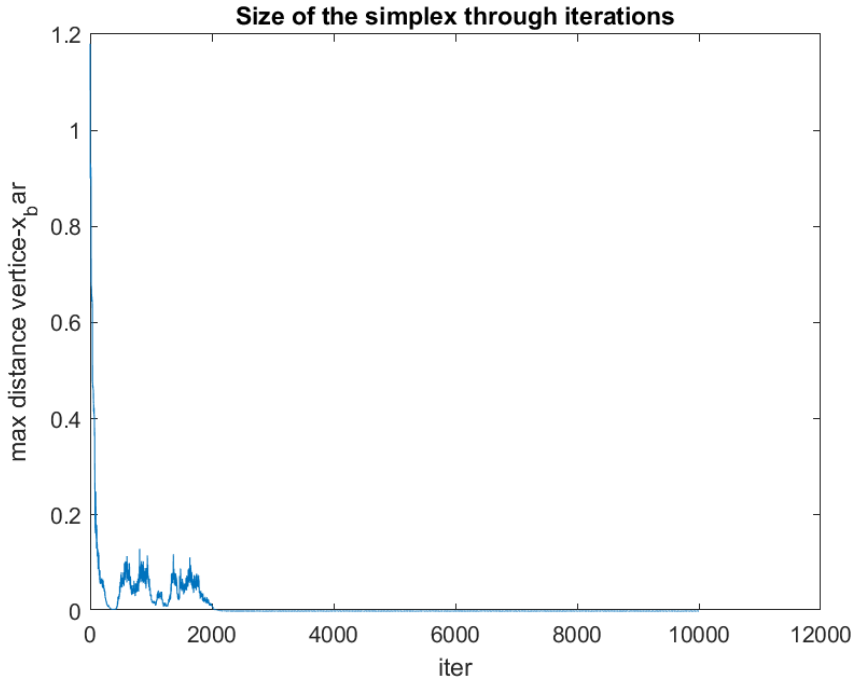

Figure 1.1: Size of the simplex in dim = 10 through iterations

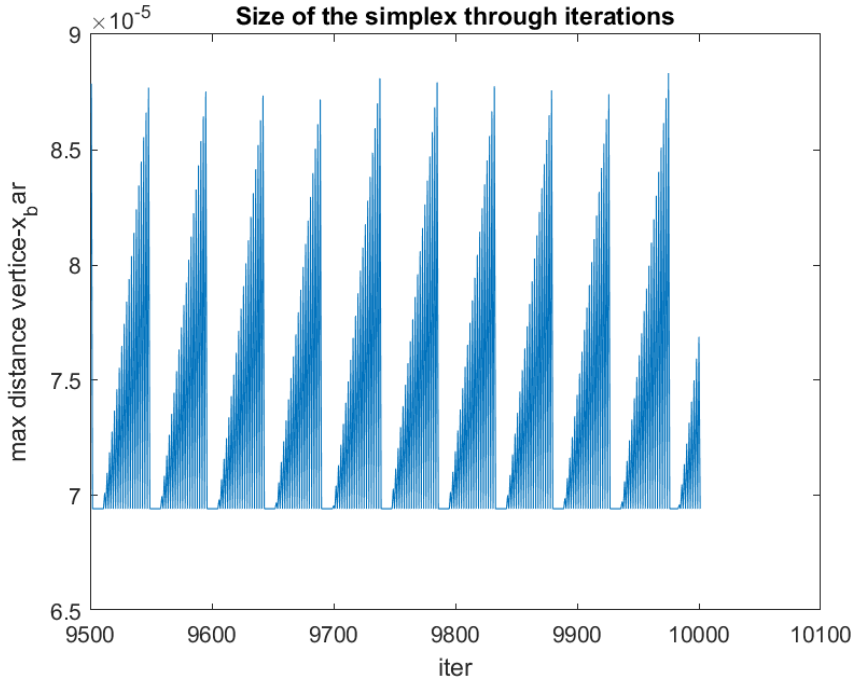

Figure 1.2: Size of the simplex in dim = 10 through iterations, last iterations

1.1.2 Stopping Criterion Based on the function value

The second stopping criterion is based on the change in the function values between the best value in the simplex and the worst one. If this change is smaller than a specified tolerance, *tol2*, it could indicates that the function has reached sort of stationary region in the objective function where the method doesn't make reasonable improvements as it is explained in the previous paragraph. This criterion avoid unnecessary computational costs. Indeed in some cases, without this criterion, the method would exhaust the maximum number of iterations without making significant progress. The tolerance vary depending from the context: 10^{-15} (negligible) for dimension 2 and 10^{-7} for higher dimension. This choice could be not the best one in general indeed the oscillation of the simplex seen in Figure 1.2 could occur in different situation depending from the function, the initial point and the dimension.
Another fundamental object for the method is an initial symplex.

1.1.3 Initial symplex

The initial symplex has one vertex that coincides with the initial point, and the remaining vertices correspond to the canonical basis. It is built in the function *NelderMead_ simplex*.

1.2 Modified Newton Method

The **Modified Newton method** is a method used when at a given $x^{(k)}$, the hessian matrix $Hf(x^{(k)})$ is not positive definite, so it is replaced with a matrix B_k defined as

$$B_k = Hf(x^{(k)}) + E_k$$

where E_k is built as B_k is positive definite and it is as small as possible to preserve the second order information contained in $Hf(^{(k)})$. We use Cholesky with Added Multiple of the Identity in order to obtain matrix B_k positive definite, next is shown the pseudo-code.

Algorithm 1 Cholesky with Added Multiple of the Identity

Data: $\beta > 0$
Result: Matrix L
if $\min_i a_{ii} > 0$ **then**
  Set $\tau_0 \leftarrow 0$
end
else
  $\tau_0 = -\min_i a_{ii} + \beta$
end
for $k = 0, 1, 2, \ldots$ **do**
  Attempt to apply the Cholesky algorithm to obtain $LL^T = A + \tau_k I$ **if** *The factorization is completed successfully* **then**
    Stop and return L
  end
  else
    $\tau_{k+1} \leftarrow \max(2\tau_k, \beta)$
  end
end

where A is the hessian of the matrix.
In particular, linear system is solved with Cholesky factorization $B_k = LL^T$:

$$\begin{cases} Ly = -\nabla f_k \\ L^T p_k = y \end{cases} \quad \Rightarrow \quad \begin{cases} y = -L^{-1}\nabla f_k \\ p_k = (L^T)^{-1} y \end{cases}$$

1.2.1 Stopping criterion and warnings

As stopping criterion we check that the number of iterations does not exceed a fixed value *kmax* and at each step we also control that the norm of the gradient computed at previous step is greater or equal to a tolerance named *tolgrad*.
Many warnings are present. If the number *bt*, which represents the number of iterations required to satisfy Armijio condition, is equal to the fixed maximum value *btmax* and the evaluation of the function f in the new point $x_{new} = x^{(k)} + \alpha^{(k)}\rho^{(k)}$ does not satisfy the Armijio condition, a warning is displayed showing that it is impossible to find an optimal α value within the maximum number of iterations and a failure is declared.
We state a *failure* if there is stagnation in case in which the difference between two next iterations $x^{(k)}, x^{(k+1)}$ differ in norm 2 less than 10^{-12}.
We also declare a failure in case the number of iterations $k = k_{max}$ or *failure* = True and the norm 2 of the gradient evaluated in the last point is greater then *tolgrad*.

1.2.2 Finite differences

The code is applied both for the case with exact derivatives and for the case where derivatives are computed by using finite differences with respect to different h values. In particular, we implement finite differences for all functions approximating the gradient and the hessian matrix exploiting the function evaluation at each point in order to avoid numerical cancellation. We use *centered difference* to compute an approximation of the gradient and the hessian matrix. What can be seen in the formulas presented in the report is that the expressions found with finite differences are the exact derivative plus h in some terms and, when $h \to 0$, finite differences tend to exact derivatives.
In exercise 3, when we use finite differences with h in the form $h_i = 10^{-k}|\hat{x}_i|$ with $k = 2, 4, 6, 8, 10, 12$ and $i = 1, \ldots, n$ we compute the hessian matrix as specified before and as the Jacobian of the exact gradient reducing the number of function evaluations taking advantage of the specific structure of each hessian matrix.

1.2.3 Experimental rate of convergence

We compute the experimental rate of convergence of the method using the last 4 x_k computed until convergence. Particularly, we aim to obtain the order of convergence $q \approx 2$ were q is computed as follows

$$q \approx \frac{log\left(\frac{||e^{(k+1)}||}{||e^{(k)}||}\right)}{log\left(\frac{||e^{(k)}||}{||e^{(k-1)}||}\right)}$$

where $e^{(k)} := x^{(k)} - x^*$.

Chapter 2

In this section we show the test of the previous implementations illustrated in Chapter 1 on the Rosenbrock function

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

with the starting points $\mathbf{x}^{(0)} = (1.2, 1.2)$ and $\mathbf{x}^{(0)} = (-1.2, 1)$, reporting the behaviour as follows.

2.3 Nelder-Mead

2.3.1 Stopping criteria tolerance

1. first stopping criteria $\longrightarrow$ tol_simplex = 1e-07;
2. second stopping criteria $\longrightarrow$ tol_varf = 1e-15 (irrelevant due to numerical precision limits);
3. kmax = 10000

2.3.2 Tuning parameters

In Nelder Mead algorithms 4 parameters are used:

- ρ: Reflection parameter.
- σ: Shrinking parameter.
- γ: Contraction parameter.
- χ: Expansion parameter.

In this section we discuss how to find the parameters which leads to the best performance and we study the sensitivity regard parameters for the specific case.
The script is in *nelderMeadComparingPar2*.

How we tuned parameters

In order to determine the optimal configuration of parameters $(\rho, \sigma^2, \chi, \gamma)$ for the Nelder-Mead method applied to the Rosenbrock function for the two suggested initial points, we decided to evaluate both the distance of the convergence point from the optimal point and the speed of convergence. Indeed, the ideal configuration of parameters should minimize both these quantities. However, it is rare to find a single configuration that is optimal from both perspectives. In fact, there may be configurations that do not approximate the optimal point accurately but converge more quickly. This is especially important in higher dimensions, where the speed of convergence helps reducing computational costs. To address this tradeoff, we decided to measure

the effectiveness of a configuration by minimizing a function that maps the parameter space to a score. This score is defined as a convex combination of the two quantities: the distance from the optimum and the total number of iterations.

$$Q(\rho_i, \sigma_i^2, \chi_i, \gamma_i) := w \cdot n_k(\rho_i, \sigma_i^2, \chi_i, \gamma_i) + (1 - w) \cdot d(\rho_i, \sigma_i^2, \chi_i, \gamma_i)$$

$n_k(\rho_i, \sigma_i^2, \chi_i, \gamma_i)$ = number of total iteration of configuration i.
$d(\rho_i, \sigma_i^2, \chi_i, \gamma_i)$ = distance from optimum of configuration i.
$w \in [0, 1]$ weight.

The weight value depends on whether there is more interest in faster convergence with possibly less accuracy or higher precision with possibly slower convergence.
In the code, we used $w = 10^{-5}$ because we decided that a distance of 0.005 from the optimum can be exchanged for 200 iterations:

$$w \cdot 200 = (1 - w) \cdot 0.005 \longrightarrow w \propto 10^{-5}.$$

In conclusion, finding the configuration that minimize Q allows to take into account both speed of convergence and number of iteration and in case there are several configuration that leads to the optimum point the faster one is chosen.

In the algorithm we form all possible configuration from sets of values:

```
rho_vec = [0.1, 0.3, 0.5, 0.7, 1, 1.3, 1.5, 1.7, 1.9];
sigma_vec = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9];
gamma_vec = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9];
chi_vec = [1.1, 1.3, 1.5, 1.7, 2, 2.1, 2.3, 2.5, 2.7];
```

Resulting parameters:

The best configuration of parameters for point $(1.2, 1.2)$ is:

$$\rho = 0.7, \quad \sigma^2 = 0.1, \quad \chi = 1.5, \quad \gamma = 0.2$$

The best configuration of parameter for point $(-1.2, 1)$ is:

$$\rho = 0.7, \quad \sigma^2 = 0.1, \quad \chi = 2, \quad \gamma = 0.6$$

Study of the sensitivity to the parameters:

We aim to answer the question: how much does the choice of parameters influence the performance of the Nelder-Mead method applied to the Rosenbrock function?
The impact of the parameters can be significant, especially on the number of iterations. For example, for the initial point $(1.2, 1.2)$, the best configuration in terms of minimizing the number of iterations required only 16 iterations to reach convergence, while the worst configuration took 432 iterations. The disparity becomes even greater when considering the method applied to the other initial point: the worst configuration reached the maximum allowed number of iterations, whereas the best configuration converged in just 15 iterations.
Also, regarding the quality of the convergence point, the parameters make the difference for both initial points. The maximum distance from the optimum point recorded for both initial points is around 2, while the minimum distance is on the order of 10^{-9}.

Initial Points	min #iter	max #iter	min dist	max dist
(1.2,1.2)	16	10000	2.262 e-09	1.7015
(-1.2,1)	15	10000	5.233 e-10	1.905

Table 2.1: Table demonstrating the sensitivity of the method to the parameters. The second and third columns show the number of iteration of configuration with the number of iteration of the configuration which leads to the minimum and maximum number of iterations, respectively, while the fourth and fifth columns contains the minimum and maximum distance of the convergence point from the optimum

Initial Points	#iter best configuration	distance best configuration
(1.2,1.2)	72	1.9390e-08
(-1.2,1)	111	1.5738e-08

Table 2.2: Table regarding the number of iteration and distance of convergence point from optimum of the best configuration found

We can state that the selection of the parameters is a crucial determinant of the Nelder-Mead algorithm's performance.

2.3.3 Results

We used Nelder-Mead method with tuned parameters and studied the results in term of:

i) Point of convergence and type of convergence,

ii) Number of iteration before convergence and computational costs,

iii) Features of convergence

i) Point of convergence and type of convergence:

Initial Point	$\bar{x_1}$	$\bar{x_2}$	Stopping Criteria met	distance from optimum
$x^{(0)} = (1.2, 1.2)$	1.0000	1.0000	1	1.939 e-08
$x^{(0)} = (-1, 1.2)$	1.0000	1.0000	1	1.574 e-08

Table 2.3: Table reporting coordinates of convergence point, simplex's best point when Nelder Mead method has reached convergence, for the two fixed starting point. The stopping criteria 1 is based on the dimension of simplex.

The steps of the method are shown in Figures 2.5, 2.6, 2.7 and 2.8.

ii) Number of iterations and computational costs:

Initial point	Time to converge	Number of iterations
$\mathbf{x}^{(0)} = (1.2, 1.2)$	10^{-2} s	72
$\mathbf{x}^{(0)} = (-1.2, 1)$	10^{-2} s	111

Table 2.4: Table reporting the order of magnitude of computational costs and number of iterations necessary for the Nelder Mead method to converge starting from the two fixed starting points.

It is observed that there is a dependence on the chosen initial point, particularly with regard to the number of iterations: the point $(-1, 1.2)$ requires about twice the number of iterations to converge compared to the other point, probably due to the fact that $(-1, 1.2)$ is farther from the optimal solution than the other one.

iii) Feature of convergence
Another important measure for evaluating a method is the rate of convergence, which, in case of convergence to the optimum, refers to how quickly the method finds the optimal solution.
For this task, experimental rates of convergence are used. However, we often encounter a problem with Nelder Mead method: our calculated experimental rates frequently turn out to be 'NaN'. This typically happens when the method's progress is very minimal.
We instead analyzed how the distance from the optimum changes across iterations (as shown in Figures 2.3 and 2.4). These figures reveal a strong decay in distance during the initial iterations, followed by a flattening of the curve, illustrating diminishing progress.

Last 5 exponential rate of convergence for each point:

$$\begin{aligned} \text{Initial Point } [1.2, 1.2]: &\quad -\infty \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\ \text{Initial Rates } [-1, 1.2]: &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \end{aligned}$$

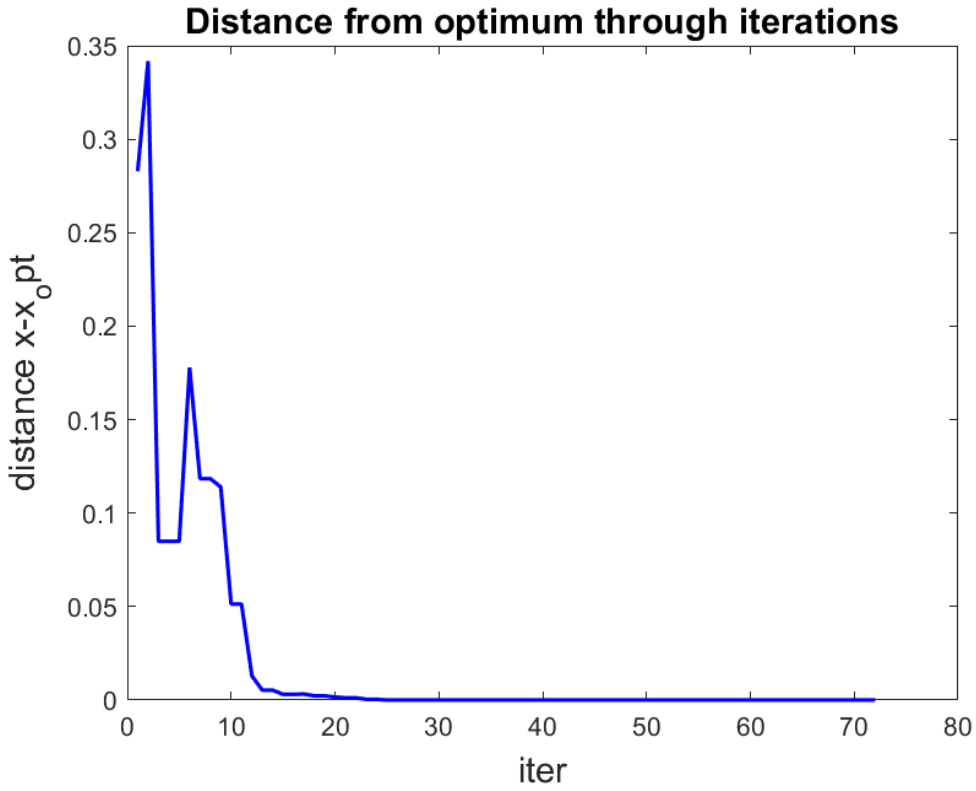

Figure 2.3: Distance from optimum, first initial point

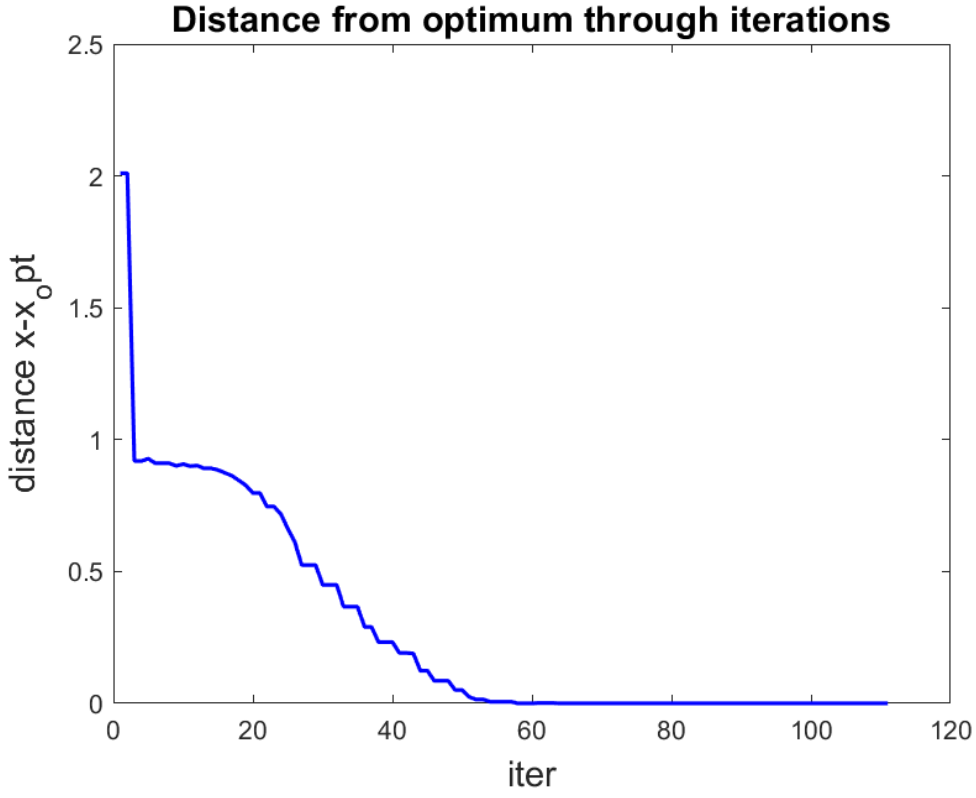

Figure 2.4: Distance from optimum, second initial point

Colour	Type of Move
Magenta	Enough reduction but not exceptional
Green	Expansion: exceptional reduction
Yellow	Contraction without shrinking: absence or very little reduction
Red	Shrinking: worst scenario

Table 2.5: Legend: each colors means a specific move of the simplex

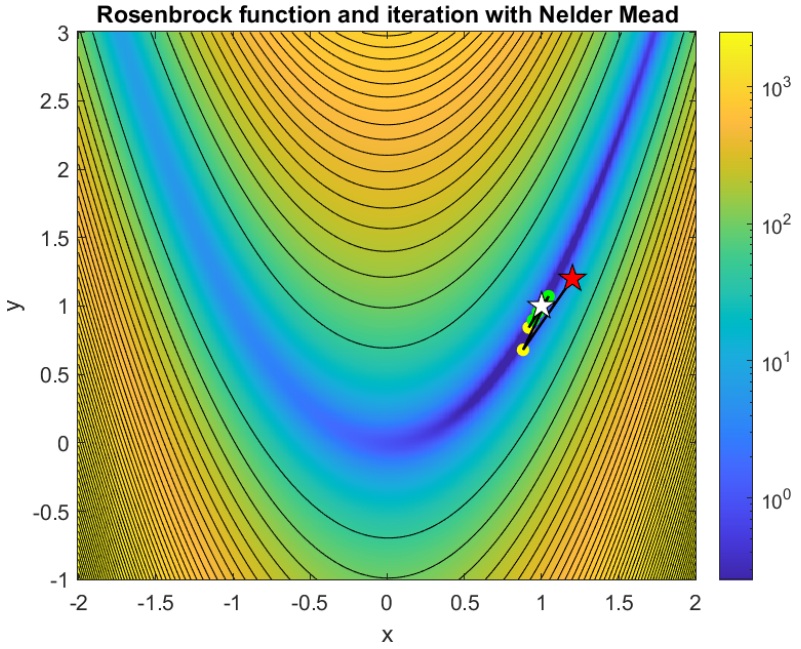

Figure 2.5: Picture showing convergence of Nelder Mead method for initial point (1.2,1.2), white star = optimum and red star = initial point

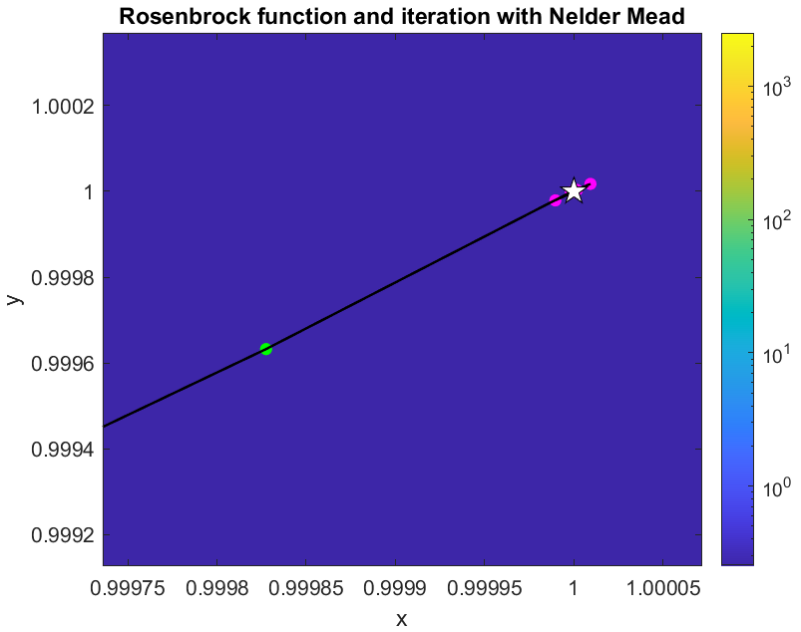

Figure 2.6: Zoom in picture showing convergence of Nelder Mead method for initial point (1.2,1.2), white star = optimum and red star = initial point

Colour	Type of Move
Magenta	Enough reduction but not exceptional
Green	Expansion: exceptional reduction
Yellow	Contraction without shrinking: absence or very little reduction
Red	Shrinking: worst scenario

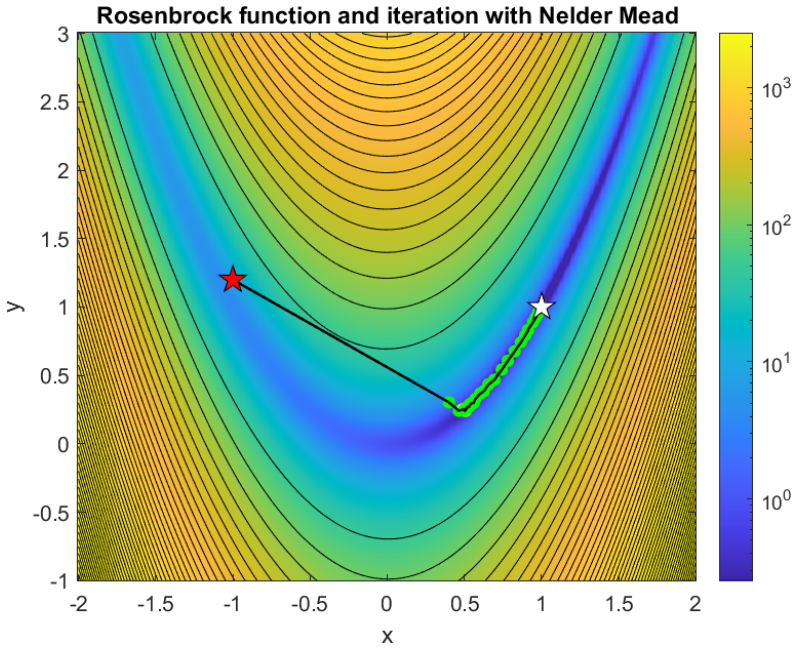

Figure 2.7: Picture showing convergence of Nelder Mead method for initial point (-1,1.2), white star = optimum and red star = initial point

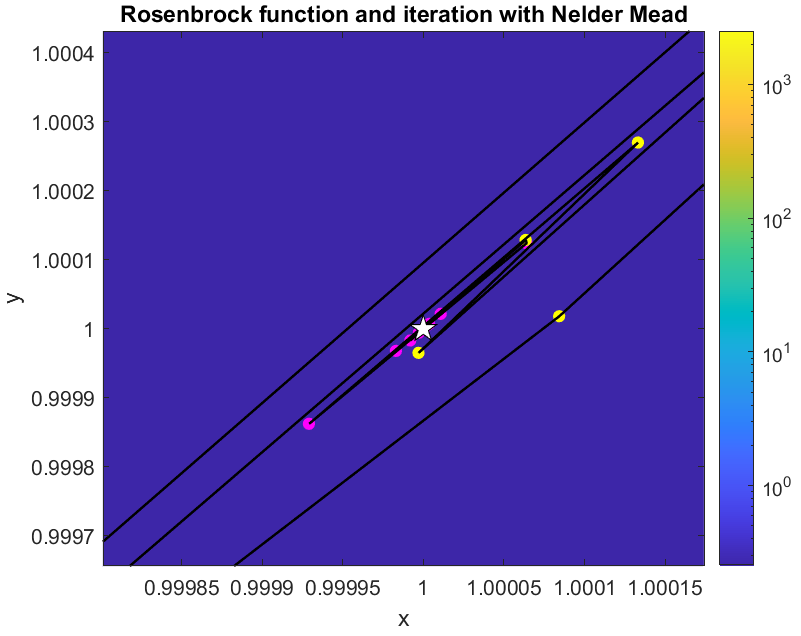

Figure 2.8: Zoom in picture showing convergence of Nelder Mead method for initial point (-1,1.2), white star = optimum and red star = initial point

2.4 Modified Newton method

The gradient and the hessian matrix of Rosenbrock function f are:

$$\nabla f = \begin{bmatrix} 400(x_1^3 - x_1 x_2) + 2(x_1 - 1) \\ 200(x_2 - x_1^2) \end{bmatrix} \quad Hf = \begin{bmatrix} 200(6x_1^2 - 2x_2) + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

While the gradient and the hessian matrix computed with finite differences are

$$\nabla f = \begin{bmatrix} 400(x_1^3 - x_1 x_2 + h_1^2 x_1) + 2(x_1 - 1) \\ 200(x_2 - x_1^2) \end{bmatrix} \quad Hf = \begin{bmatrix} 200(6x_1^2 - 2x_2 + h_1^2) + 2 & -200(2x_1 - h_1) \\ -200(2x_1 - h_1) & 200 \end{bmatrix}$$

where we exploit h_i because we pass to the functions implemented in Matlab a vector h whose components are equal to each other when we apply constant increment, and they are different when h depends to the corresponding x_i component.
The first results were obtained using exact derivatives and setting the following parameters

$$\begin{cases} \rho = 0.5 \\ c1 = 10^{-4} \\ tol\_grad = 10^{-7} \\ btmax = 40 \\ kmax = 500 \end{cases}$$

where $btmax$ is the maximum number of iterations allowed for backtracking and is chosen in a way that stagnation is not allowed, specifically $\rho^{btmax} \approx 10^{-13} > \epsilon_m$ where $\epsilon_m \approx 10^{-16}$ is the machine precision.

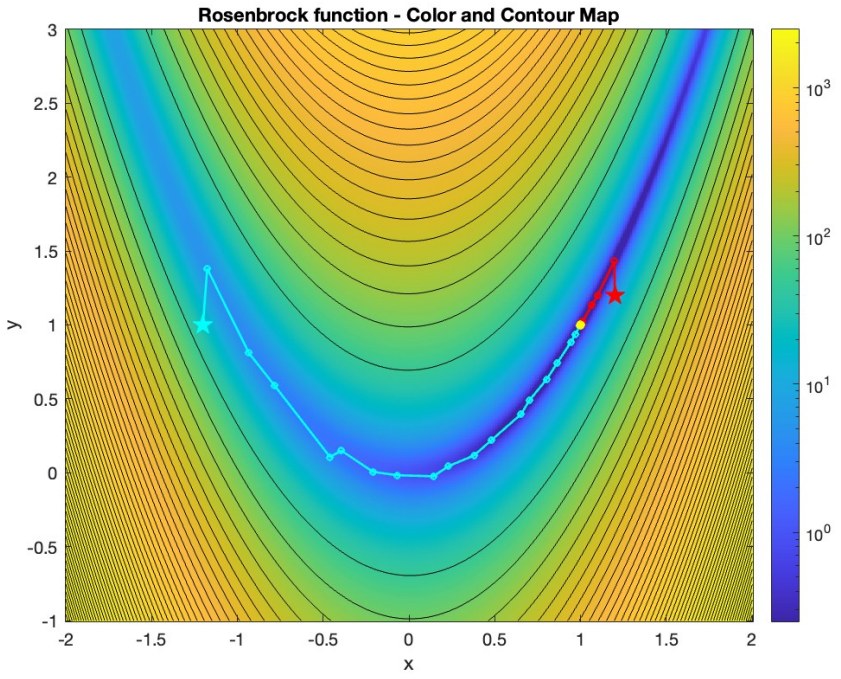

Figure 2.9: Picture showing convergence of Modified Newton method.

In Fig. 2.9 is shown the convergence of Modified Newton method for the Rosenbrock function starting from initial points $\mathbf{x}^{(0)} = (1.2, 1.2)$ colored in red, and $\mathbf{x}^{(0)} = (-1.2, 1)$ in light blue and

in the same color is the trajectory from the specific initial point to the real minimum of the function $\mathbf{x} = (1, 1)$, shown as a yellow point.
Looking at Fig. 2.9 is evident that the method converges to the real minimum $\mathbf{x} = (1, 1)$ starting from the given initial points; in particular the convergence from $\mathbf{x}^{(0)} = (1.2, 1.2)$, colored in red, is faster than the one from $\mathbf{x}^{(0)} = (-1.2, 1)$, which requires 21 iterations compared to 8 of the first. The time required to the method to converge and the number of iterations needed to satisfy the stopping criterion are reported below:

initial point	time	iter	roc	failure
$\mathbf{x}^{(0)} = (1.2, 1.2)$	0.0053 s	8	1.5948	0
$\mathbf{x}^{(0)} = (-1.2, 1)$	0.0030 s	21	1.8981	0

Table 2.6: Table reporting the time necessary to the Modified Newton method to converge starting from the two fixed starting points.

2.4.1 Variation of backtracking parameters

In this section are reported the results obtained varying ρ and c parameters for bactracking strategy

$$f(x^{(k)} + \alpha p^{(k)}) \leq f(x^{(k)}) + c\alpha \nabla f(x^{(k)})^T p^{(k)}$$

where ρ is the parameter used to set at iteration j the new value of $\alpha_{j+1}^{(k)} = \rho \alpha_j^{(k)}$ if the Armijio condition is not satisfied by $\alpha_j^{(k)}$. For each value of ρ are tested all the values proposed for c. Particularly we used

$$\rho = [0.1, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9]$$
$$c = [10^{-7}, 10^{-5}, 10^{-4}, 10^{-3}, 0.01, 0.1]$$

In Fig.(2.10) and Fig.(2.11) are reported the number of iterations. Starting from initial point 1, the best values are the pairs with $\rho = 0.6, 0.8, 0.9$ and all the values proposed for c. Starting from the second point, the best parameters are the pairs with $\rho = 0.9$ with all c values.

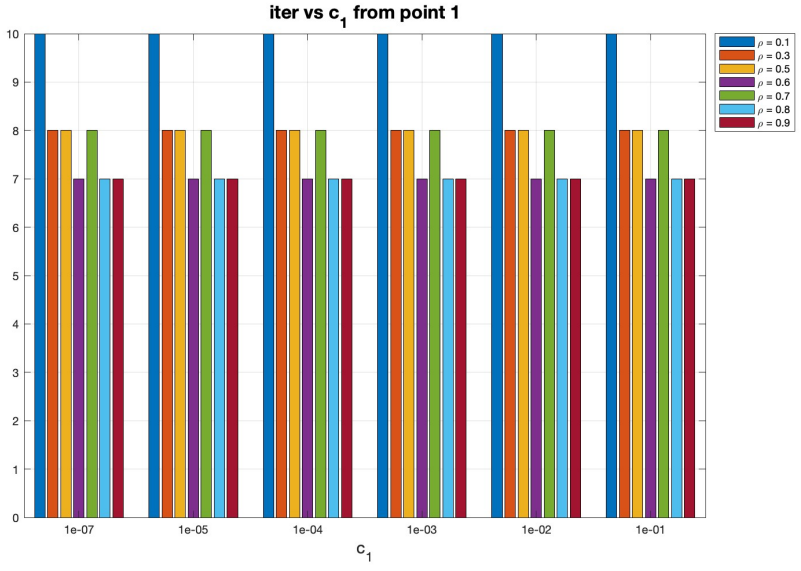

Figure 2.10: Barplot showing the different number of iterations required to converge starting from initial point 1.

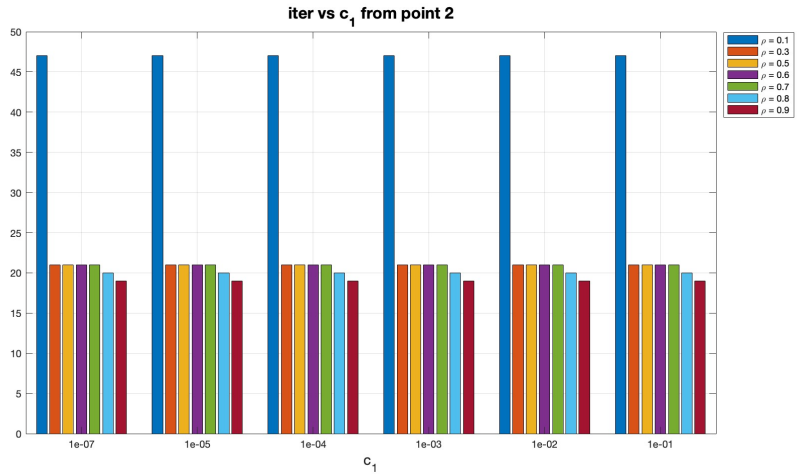

Figure 2.11: Barplot showing the different number of iterations required to converge starting from initial point 2.

2.4.2 Results with derivatives computed using finite differences

First we have tested them with different h-values, specifically $h = 10^{-k}$ with $k = 2, 4, 6, 8, 10, 12$. Then we have tested the finite difference also using a specific increment h_i when differentiating with respect to the variable x_i, according to the values $h_i = 10^{-k}|\hat{x}_i|$, with $k = 2, 4, 6, 8, 10, 12$ and $i = 1, \ldots, n$, where $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_n) \in \mathbb{R}^n$ is the point at which the derivatives have to be approximated.
The result obtained with $\rho = 0.5$ and $c = 10^{-4}$ are reported in the following table

h value	time	iter	roc	failure
10^{-2}	0.0004	23	1.0426	0
10^{-4}	0.0002	9	1.2756	0
10^{-6}	0.0001	8	1.6032	0
10^{-8}	0.0005	8	1.5949	0
10^{-10}	0.0002	8	1.5948	0
10^{-12}	0.0003	8	1.5948	0

Table 2.7: Table showing results of the method applied to the first initial point.

h_i value	time	iter	roc	failure
$10^{-2}\|x_i\|$	0.0005	29	1.0450	0
$10^{-4}\|x_i\|$	0.0002	8	1.4709	1
$10^{-6}\|x_i\|$	0.0001	8	1.6032	0
$10^{-8}\|x_i\|$	0.0003	8	1.5949	0
$10^{-10}\|x_i\|$	0.0006	8	1.5948	0
$10^{-12}\|x_i\|$	0.0002	8	1.5948	0

Table 2.8: Table showing results of the method applied to the first initial point with h variable.

h value	time	iter	roc	failure
10^{-2}	0.0008	39	1.0430	0
10^{-4}	0.0002	21	1.4199	1
10^{-6}	0.0002	21	1.8687	0
10^{-8}	0.0002	21	1.8979	0
10^{-10}	0.0006	21	1.8979	0
10^{-12}	0.0002	21	1.8979	0

Table 2.9: Table showing results of the method applied to the second initial point.

h_i value	time	iter	roc	failure
$10^{-2}\|x_i\|$	0.0004	29	1.5972	0
$10^{-4}\|x_i\|$	0.0002	8	1.5401	1
$10^{-6}\|x_i\|$	0.0002	8	1.8601	0
$10^{-8}\|x_i\|$	0.0002	8	1.8979	0
$10^{-10}\|x_i\|$	0.0002	8	1.8981	0
$10^{-12}\|x_i\|$	0.0002	8	1.8981	0

Table 2.10: Table showing results of the method applied to the second initial point with h variable.

In general, the number of iterations are greater for the first value of h, this is due to the way in which we compute finite differences, they are built as, when $h \to 0$, they tend to exact derivatives. The only failure registered corresponds to $h = 10^{-4}$ both in the case with h constant and adapted to each component of x, except for the first initial point with h constant.
During the phase of tuning we do not observe particular changes. We fix $c = 10^{-4}$ and test $\rho \in \{0.1, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9\}$.
We report the tuning of ρ showing the invariance of the number of iterations through different h value starting from the first point:

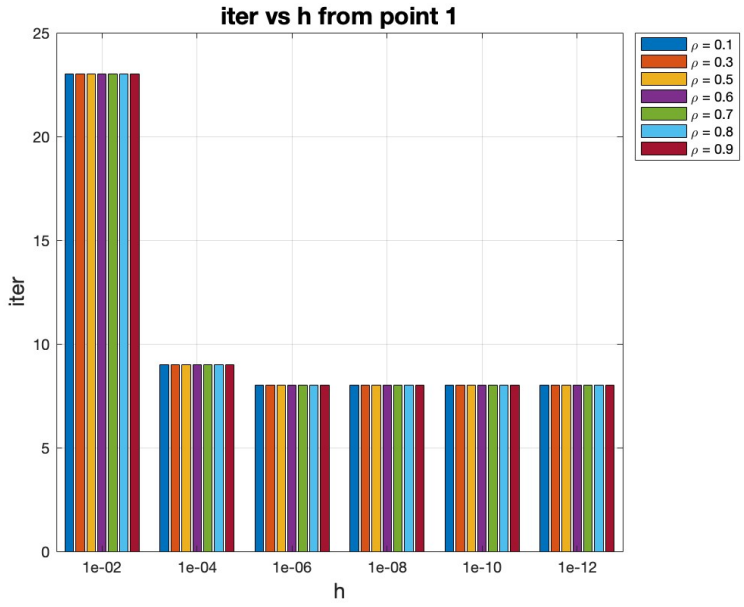

Figure 2.12: Barplot showing the different number of iterations required to converge starting from initial point 1 varying ρ with finite difference and h constant.

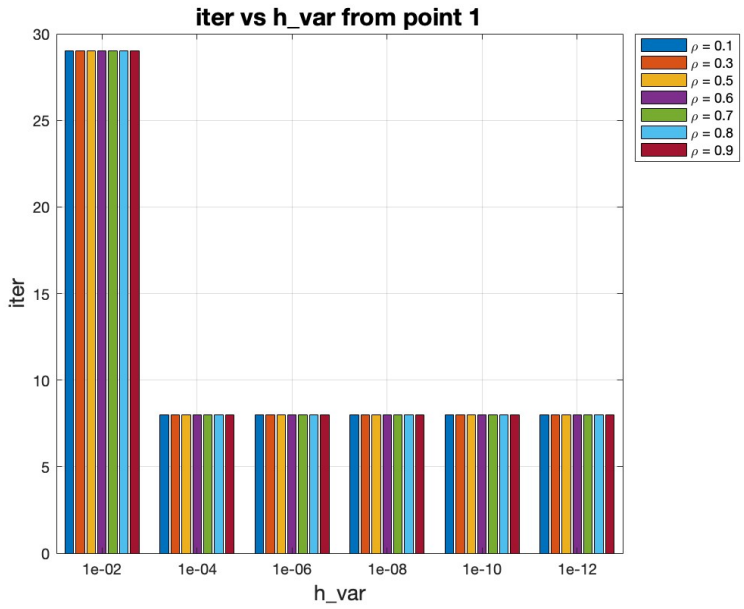

Figure 2.13: Barplot showing the different number of iterations required to converge starting from initial point 1 varying ρ with finite difference and h variable.

2.5 Comparison Nelder-Mead - Modified Newton method (exact)

Regarding the performance comparison, both methods, the Modified Newton Method and the Nelder-Mead Method, demonstrated successful convergence for the two distinct starting points: 0 failures reported and the stopping criteria being met in both test cases, respectively.
However, a significant difference emerges in the number of iterations required to satisfy the stopping criterion.
The Modified Newton Method achieved convergence in a considerably fewer number of iterations (8 and 21 iterations for the two starting points) compared to the Nelder-Mead Method (72 and 111 iterations).
This efficiency is further highlighted by the experimental rate of convergence observed for the Modified Newton Method, which showed superlinear convergence with rates of 1.5948 and 1.8981.
Consequently, the execution times also reflect this disparity, with the Modified Newton Method exhibiting much shorter convergence times (0.0053 s and 0.0030 s) compared to the Nelder-Mead Method (0.01 s for both starting points).

Chapter 3

In this section we report the results of the applications of Nelder-Mead and Modified Newton method to the following functions:

1. Rosenbrock chained

$$F(x) = \sum_{i=2}^{n} [100(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2] \tag{3.1}$$

   Minimum point: $x^* = \mathbf{1}$.

2. Chained Wood function

$$F(x) = \sum_{j=1}^{k} [100(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2 + 90(x_{i+1}^2 - x_{i+2})^2 + (x_{i+1} - 1)^2 + 10(x_i + x_{i+2} - 2)^2 + (x_i - x_{i+2})^2/10] \tag{3.2}$$

   where $i = 2j$, $k = (n-2)/2$. Minimum point: $x^* = \mathbf{1}$.

3. Chained Powell singular function

$$F(x) = \sum_{j=1}^{k} [(x_{i-1} + 10x_i)^2 + 5(x_{i+1} - x_{i+2})^2 + (x_i - 2x_{i+1})^4 + 10(x_{i-1} - x_{i+2})^4] \tag{3.3}$$

   where $i = 2j$, $k = (n-2)/2$ Minimum point: $x^* = \mathbf{0}$.

The study is organized as follow:

- In the first step we fixed as initial point[1]:

  1. $\bar{x}_i = -1.2$, $mod(i, 2) = 1$, $\bar{x}_i = 1$, $mod(i, 2) = 0$
  2. $\bar{x}_i = -3$, $mod(i, 2) = 1$, $i \leq 4$, $\bar{x}_i = -2$, $mod(i, 2) = 1$, $i > 4$,
     $\bar{x}_i = -1$, $mod(i, 2) = 0$, $i \leq 4$, $\bar{x}_i = 0$, $mod(i, 2) = 0$, $i > 4$
  3. $\bar{x}_i = 3$, $mod(i, 4) = 1$, $\bar{x}_i = -1$, $mod(i, 4) = 2$
     $\bar{x}_i = 0$, $mod(i, 4) = 3$, $\bar{x}_i = 1$, $mod(i, 4) = 0$

  and we study the problem with Nelder-Mead for the dimension $n = 10, 25, 50$, while with Modified Newton method we use $n = 10^d$, $d = 3, 4, 5$.

- Next we use 10 new starting points $\mathbf{x}^{(0)} = \bar{\mathbf{x}} \in \mathbb{R}^n$ randomly generated with uniform distribution in a hyper-cube $[\bar{x}_1 - 1, \bar{x}_1 + 1] \times \ldots \times [\bar{x}_n - 1, \bar{x}_n + 1] \subset \mathbb{R}^n$.

[1] https://www.researchgate.net/publication/325314497 Test Problems for Unconstrained Optimization

In particular, for Modified Newton method we change the parameters for the backtracking strategy in order to impose sufficient decrease condition. Then, we apply the codes both for the case with exact derivatives and for the case where derivatives are computed by using finite differences with respect to the following values for the increment h for each differentiation:

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

Moreover, we test the finite differences also using a specific increment h_i when differtiating with respect to the variables x_i, according to the following values:

$$h_i = 10^{-k}|\hat{x}_i|, \quad k = 2, 4, 6, 8, 10, 12, \quad i = 1, \ldots, n$$

where $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_n) \in \mathbb{R}^n$ is the point at which the derivatives have to be approximated.

3.6 Chained Rosenbrock function

3.6.1 Nelder-Mead

Results for the Nelder Mead method with chained Rosenbrock function.

Default tolerance for stopping criteria

1. first stopping criteria $\longrightarrow$ tol_simplex = 1e-07;
2. second stopping criteria $\longrightarrow$ tol_varf = 1e-07;
3. kmax = 10000

Tuned parameters
We found the best configuration of parameters for the suggested initial point as explained in Section 2.4.2. In the algorithm we form all possible configuration from values:

```
rho_vec = [0.25, 0.5, 1, 1.35, 1.75];
sigma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
gamma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
chi_vec = [1.1, 1.5, 2, 2.5, 3];
```

The best configuration of parameters we found is:

- for dimension = 10:

$$\rho = 1, \quad \sigma^2 = 0.1, \quad \chi = 1.5, \quad \gamma = 0.75$$

- for dimension = 25:

$$\rho = 1, \quad \sigma^2 = 0.1, \quad \chi = 2, \quad \gamma = 0.5$$

- for dimension = 50:

$$\rho = 0.5, \quad \sigma^2 = 0.1, \quad \chi = 2.5, \quad \gamma = 0.9$$

Before presenting the results, a few comments are needed.
Tables 3.11, 3.12, 3.13 summarize the main outcomes of the algorithm when applied to the method using the 11 initial points (the first being the suggested one, and the others being random points). The first column shows the distance of the solution from the optimum. The third column indicates the reason why the method stopped, corresponding to the stopping criterion that was satisfied:

- flag = 1: The method reached the first stopping criterion (related to the size of the simplex).
- flag = 2: The method reached the second stopping criterion (related to the stationary region).
- flag = 3: The method reached the maximum number of iterations without satisfying one of the two previous stopping criteria.

The fourth column gives information about the order of magnitude of computational costs.

Results for dimension = 10:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\| \cdot 10^{-3}$	# iterations	flag	time (s)
$\mathbf{p}_1$	0.14	1964	2	0.1
$\mathbf{p}_2$	0.028	2233	2	0.1
$\mathbf{p}_3$	0.021	2062	2	0.1
$\mathbf{p}_4$	0.1	1860	2	0.1
$\mathbf{p}_5$	0.076	2326	2	0.1
$\mathbf{p}_6$	0.17	1775	2	0.1
$\mathbf{p}_7$	0.16	10000	3	0.1
$\mathbf{p}_8$	0.28	2098	2	0.1
$\mathbf{p}_9$	0.14	2197	2	0.1
$\mathbf{p}_{10}$	0.077	10000	3	0.1
$\mathbf{p}_{11}$	0.12	2143	2	0.1
sample average	0.12	4175,8	/	0.1

Table 3.11: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random points)

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	0	-11.7	-0.47	2.45	∞
Point 5:	NaN	NaN	NaN	NaN	0
Point 6:	0.12	$-\infty$	NaN	0	∞
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	4.53	0.31	$-\infty$	NaN	0
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results: It can be observed that the final distance from the solution doesn't change so much with the different starting point and in general point of convergence is relatively good. Some points reach the maximum iteration limit of 10000, indicating that the algorithm didn't converge.
The convergence rate for each point is shown, calculated from the last five steps. However, the recorded values are mostly NaN (Not a Number) or infinite, suggesting that in many iterations, the method did not produce significant improvements.

Results for dimension = 25:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	4.1925	10000	3	1
$\mathbf{p}_2$	4.4925	10000	3	1
$\mathbf{p}_3$	4.3748	10000	3	1
$\mathbf{p}_4$	4.4361	10000	3	1
$\mathbf{p}_5$	4.2046	10000	3	1
$\mathbf{p}_6$	4.1087	10000	3	1
$\mathbf{p}_7$	4.3074	10000	3	1
$\mathbf{p}_8$	4.1371	10000	3	1
$\mathbf{p}_9$	4.3029	10000	3	1
$\mathbf{p}_{10}$	4.3508	10000	3	1
$\mathbf{p}_{11}$	4.2397	10000	3	1
sample average	4.2782	10000	/	1

Table 3.12: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point)

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	NaN	NaN	NaN	NaN	NaN
Point 5:	NaN	NaN	NaN	NaN	NaN
Point 6:	NaN	NaN	NaN	NaN	NaN
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	NaN	NaN	NaN	NaN	NaN
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results: While in the case of dimension 10, a significant variation in the final distances is observed. Furthermore, for all the initial points, the method terminates after reaching the maximum allowed number of iterations.
This suggests that the method does not converge, meaning the simplex does not reduce sufficiently and continues to oscillate, probably because it fails to find effective directions.
This is confirmed in Figure 3.15, where the size of the simplex in the last iterations is actually increasing, and in Figure 3.16, where it is shown that the distance between the best simplex points and the optimum doesn't decrease.
Moreover, the region where the simplex is located in the last iterations is quite far from the optimum and is similar for all points.

This region is around zero:

$$\mathbf{p_1} = \begin{bmatrix} 0.9890 & 0.9758 & 0.9593 & 0.9215 & 0.8449 & 0.7085 & 0.4989 & 0.2445 & 0.0519 & 0.0072 \\ -0.0029 & 0.0078 & 0.0207 & 0.0021 & 0.0299 & 0.0146 & 0.0096 & -0.0044 & -0.0042 & 0.0079 \\ 0.0173 & 0.0179 & -0.0103 & 0.0033 & 0.0159 & & & & & \end{bmatrix}$$

[2]

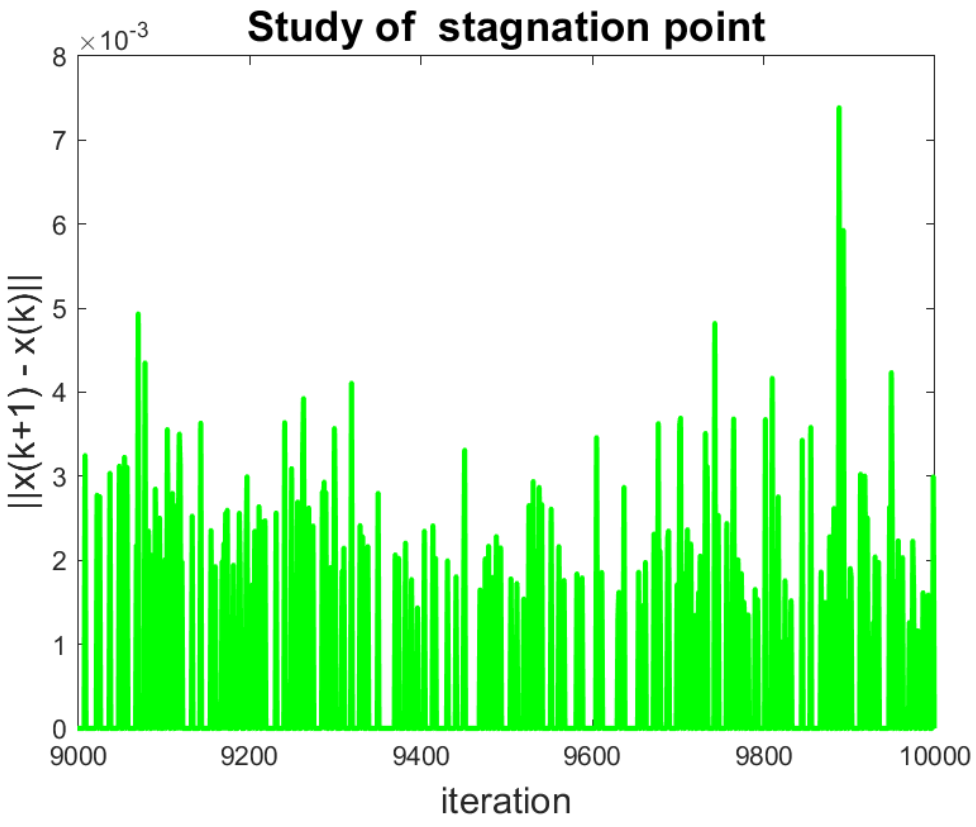

Figure 3.14: How the solution varies through iterations (p_1)

[2]Solutions of the method starting from point 1

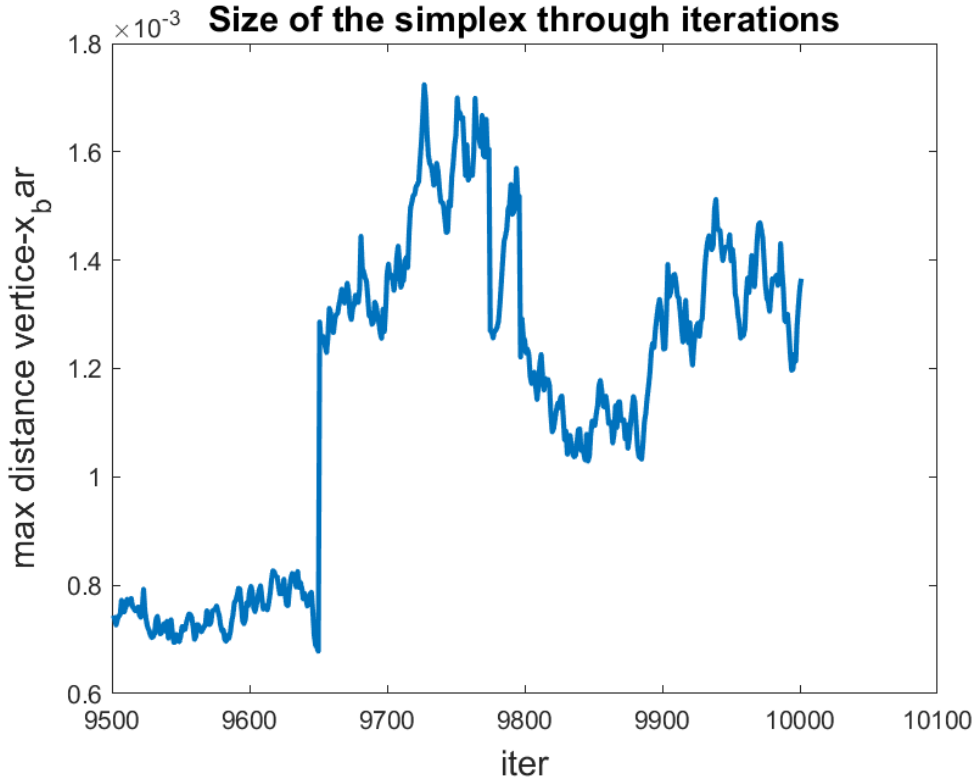

Figure 3.15: Size of the simplex in last iterations (p_1)

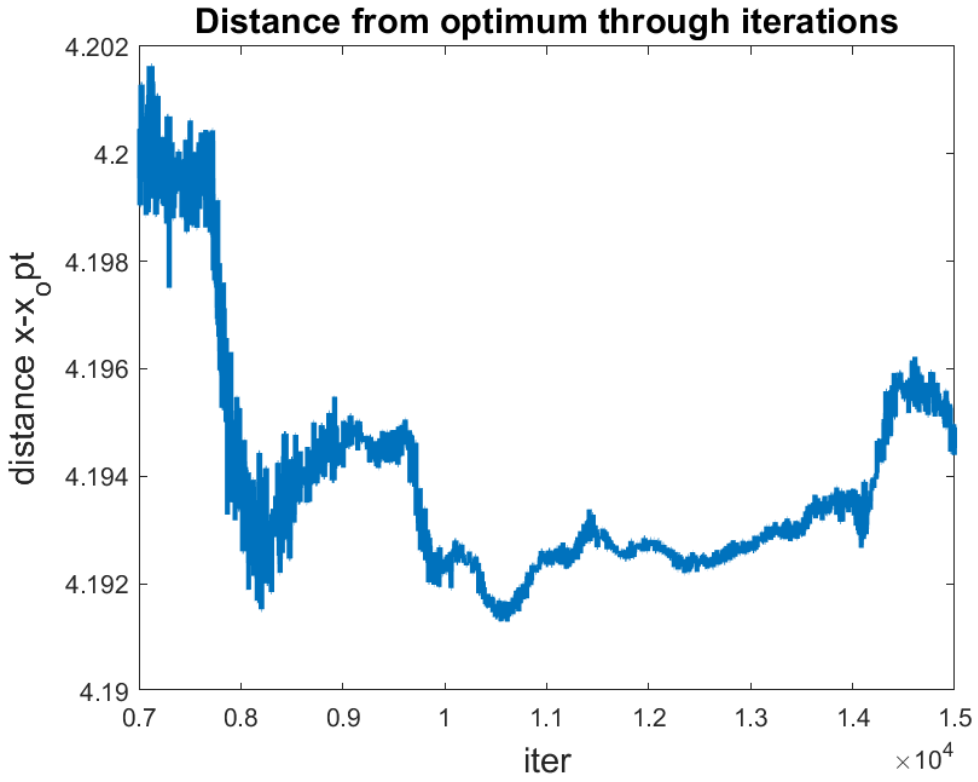

Figure 3.16: Distance solution from optimum through iterations (p_1)

Results for dimension = 50:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	6.9273	1249	2	0.1
$\mathbf{p}_2$	6.9350	1337	2	0.1
$\mathbf{p}_3$	6.9279	1377	2	0.1
$\mathbf{p}_4$	6.9286	1390	2	0.1
$\mathbf{p}_5$	6.9245	1394	2	0.1
$\mathbf{p}_6$	6.9236	1788	2	0.1
$\mathbf{p}_7$	6.9279	1283	2	0.1
$\mathbf{p}_8$	6.9259	1344	2	0.1
$\mathbf{p}_9$	6.9291	1302	2	0.1
$\mathbf{p}_{10}$	6.9278	1373	2	0.1
$\mathbf{p}_{11}$	6.9244	1297	2	0.1
sample average	6.9275	1340	/	0.1

Table 3.13: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point)

Last 5 exponential rate of convergence for each point:

$$
\begin{aligned}
&\text{Point 1:} \quad \text{NaN} \quad \text{NaN} \quad 0 \quad -\infty \quad \text{NaN} \\
&\text{Point 2:} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
&\text{Point 3:} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
&\text{Point 4:} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
&\text{Point 5:} \quad \text{NaN} \quad 0 \quad -0.17 \quad -0.93 \quad 5.93 \\
&\text{Point 6:} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
&\text{Point 7:} \quad -\infty \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad 0 \\
&\text{Point 8:} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad 0 \\
&\text{Point 9:} \quad 0 \quad \infty \quad \text{NaN} \quad 0 \quad -0.72 \\
&\text{Point 10:} \quad \text{NaN} \quad 0 \quad 0.1861 \quad 0.92 \quad 4.87 \\
&\text{Point 11:} \quad \infty \quad \text{NaN} \quad 0 \quad -2.05 \quad -\infty
\end{aligned}
$$

Comments on results: The method is convergent for the second convergence criterion: the simplex has found a stationary region or is small enough that the variation of the function is indistinguishable. Nevertheless, the convergence point, similar for all points, is far from the optimum (on average, the distance is about 7).

An overview:

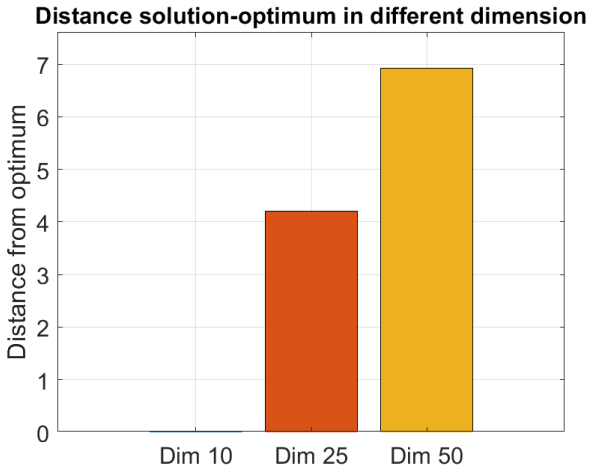

Figure 3.17: Chained Rosenbrock: distance from optimum

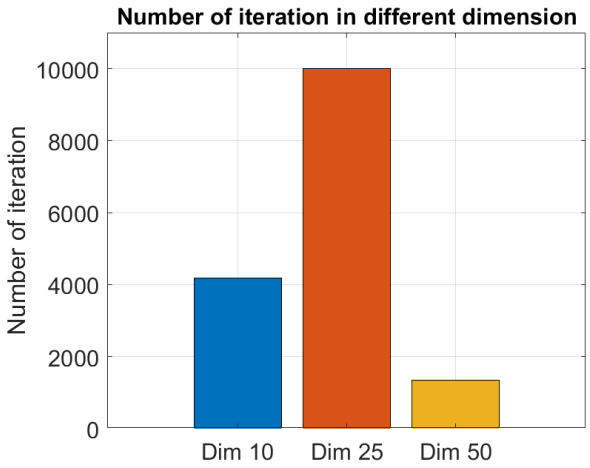

Figure 3.18: Chained Rosenbrock: number of iterations

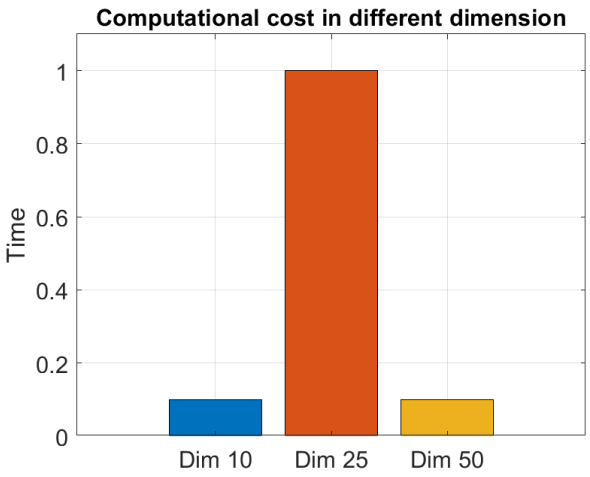

Figure 3.19: Chained Rosenbrock: computational cost

3.6.2 Modified Newton method

Gradient and hessian matrix of the Rosenbrock chained function.

$$\nabla f = \begin{cases} 2(x_1 - 1) + 400x_1(x_1^2 - x_2) \\ 2(101x_i - 1) + 400x_i(x_i^2 - x_{i+1}) - 200x_{i-1}^2 \quad \forall i = 2, \ldots, n-1 \\ 200(x_n - x_{n-1}^2) \end{cases}$$

$$\nabla^2 f = \begin{bmatrix} 2 + 1200x_1^2 - 400x_2 & -400x_1 & 0 & \cdots & 0 \\ -400x_1 & 202 + 1200x_2^2 - 400x_3 & -400x_2 & \cdots & 0 \\ 0 & -400x_2 & 202 + 1200x_3^2 - 400x_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 200 \end{bmatrix}$$

In order to avoid numerical cancellation while approximating the gradient and the hessian matrix of the chained Rosenbrock function we used the following expressions for the centered finite differences:

$$\begin{aligned} \frac{\partial f}{\partial x_1} &\approx \frac{f(x + h_1 e_1) - f(x - h_1 e_1)}{2h_1} = 100(4x_1^3 - 4x_1 x_2 + 4h_1^2 x_1) + 2x_1 - 2 \\ \frac{\partial f}{\partial x_i} &\approx \frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i} = 2(101x_i - 1) + 400x_i(x_i^2 - x_{i+1} + h_i^2) - 200x_{i-1}^2 \quad \forall i = 2, \ldots, n-1 \\ \frac{\partial f}{\partial x_n} &\approx \frac{f(x + h_n e_n) - f(x - h_n e_n)}{2h_n} = 200(x_n - x_{n-1}^2) \end{aligned}$$

while the hessian matrix becomes

$$\begin{aligned} H_{i+1,i} &\approx \frac{f(x + h_i e_i + h_{i+1} e_{i+1}) - f(x + h_{i+1} e_{i+1}) - f(x + h_i e_i) + f(x)}{h_i h_{i+1}} = -400x_i - 200h_i \quad \forall i = 1, \ldots, n-1 \\ H_{1,1} &\approx \frac{f(x + h_1 e_1) - 2f(x) + f(x + h_1 e_1)}{h_1^2} = 1200x_1^2 + 200h_1^2 - 400x_2 + 2 \\ H_{i,i} &\approx \frac{f(x + h_i e_i) - 2f(x) + f(x + h_i e_i)}{h_i^2} = 1200x_i^2 - 400x_{i+1} + 200h_1^2 + 202 \quad \forall i = 2, \ldots, n-1 \\ H_{n,n} &\approx \frac{f(x + h_n e_n) - 2f(x) + f(x + h_n e_n)}{h_n^2} = 200 \\ H_{i,i+1} &\approx \frac{f(x + h_i e_i + h_{i+1} e_{i+1}) - f(x + h_{i+1} e_{i+1}) - f(x + h_i e_i) + f(x)}{h_i h_{i+1}} = -400x_i - 200h_i \quad \forall i = 1, \ldots, n-1 \end{aligned}$$

We have implemented the gradient and hessian matrix of the function exploiting the component of the step-length h, when it is constant and equal to $h = 10^{-k}$, with $k = 2, 4, 6, 8, 10, 12$ we pass to the function a vector h whose components are all the same and equal to 10^{-k}.
We have also computed the hessian matrix implemented as the Jacobian of the exact gradient using centered finite differences working with the sparse structure of the hessian:

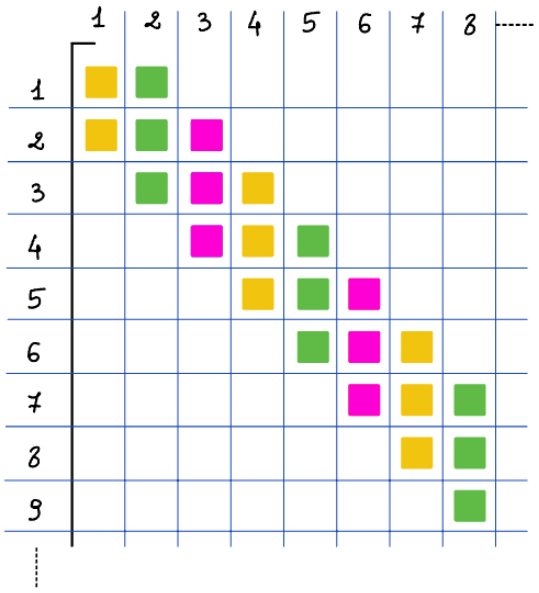

Figure 3.20: Hessian matrix of Rosenbrock chained function underling elements in the same column.

By analyzing the derivatives of the function, we can deduce that all gradient's components are nullified by $x = \mathbf{1}$. This leads to conclusion that Rosenbrock chained function assumes its global minimum in that point.
To better understand the behavior of the function $f(x)$ near the minimizer we report a plot of the function in 2D for $n = 2$.

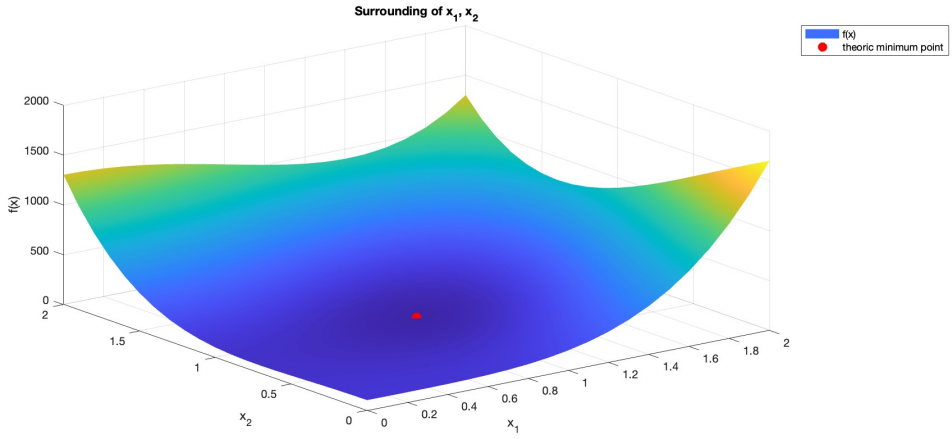

Figure 3.21: 2D plot of the function $f(x_1, x_2)$

From the plot, we can notice that the function is a little bit flat near the minimum; this may affect the performance of modified newton method because flat region may cause slow convergence or even stagnation for optimization solvers.

Results for modified Newton method with chained Rosenbrock function.
In this section are reported solutions from the application of Modified Newton Method to the Rosenbrock chained function considering exact derivatives. We store the hessian matrix in a sparse

and three-diagonal structure due to the high dimension of the input vector $\{10^3, 10^4, 10^5\}$. For each dimension we run the method starting from 11 different points, where the first is suggested and the other 10 are a random perturbation of it.
For each dimension we have set the following parameters:

$$tol\_grad = 10^{-3} \quad kmax = \begin{cases} 15 \times 10^2 \text{ for dim} = 10^3 \\ 15 \times 10^3 \text{ for dim} = 10^4 \\ 15 \times 10^4 \text{ for dim} = 10^5 \end{cases} \quad \rho = 0.9 \quad btmax = 100 \quad c = 10^{-4}$$

Where ρ and $btmax$ were chosen in order to avoid stagnation because $\rho^{btmax} \approx 10^{-5}$.
Next are reported some bar plots that compare the results obtained by modified newton method applied to Rosenbrock chained function in dimensions $n = 10^3, 10^4, 10^5$ in terms of average time, average number of iterations, average difference between minimum point and minimum found and average rate of convergence.
Looking at Fig.(3.22) can be seen that increasing the input size by one order of magnitude results in an approximately one order of magnitude increase in the number of iterations required for convergence. This impact the average time required, for dimension 10^5 it is approximately $1200s$.
In Fig.(3.24) are reported the average rate of convergence, it can be noted that there is a growing trend as the size increases starting from a value $roc \approx 1.2$ for dimension 10^3 until $roc \approx 1.6$ for dimension 10^5. This is exploited by theory, the experimental rate of convergence approximates the order of convergence as the number of iterations are sufficiently large.
The average difference in norm between the real minimizer and the last point found is quite the same for dimension 10^3 and 10^5, a little bit greater for the last dimension.
Finally, we have not obtained any failures.

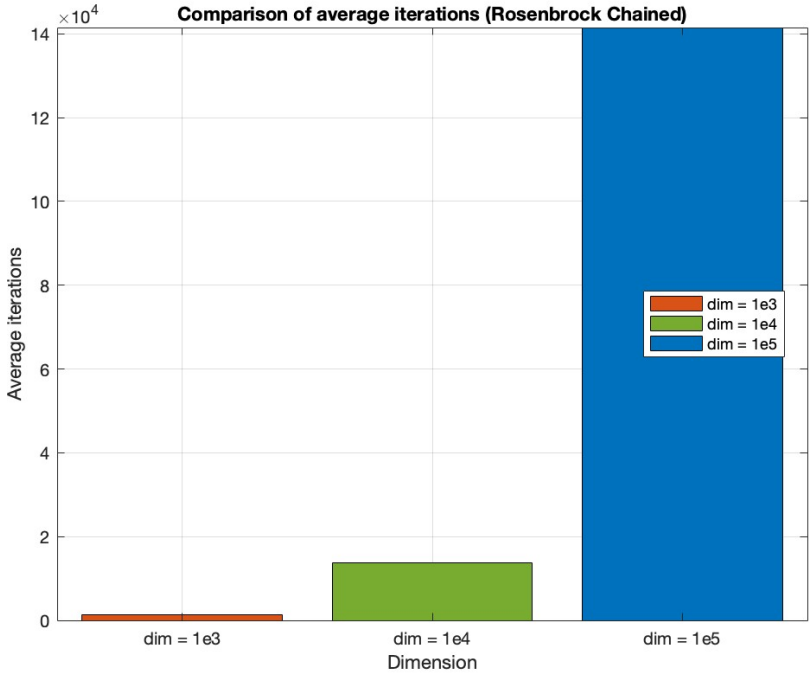

Figure 3.22: Average iterations need to converge compared the three different dimensions.

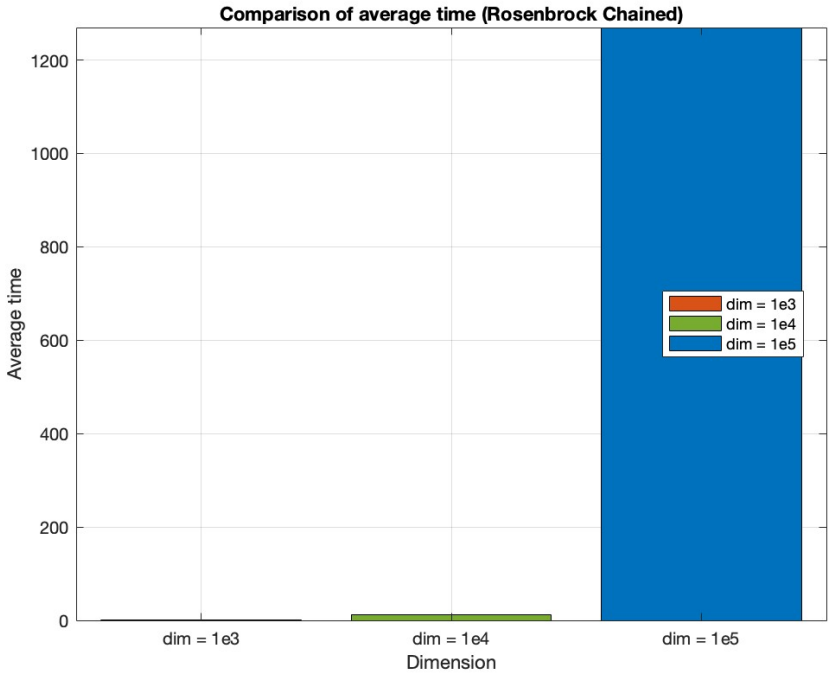

Figure 3.23: Average time need to converge compared the three different dimensions.

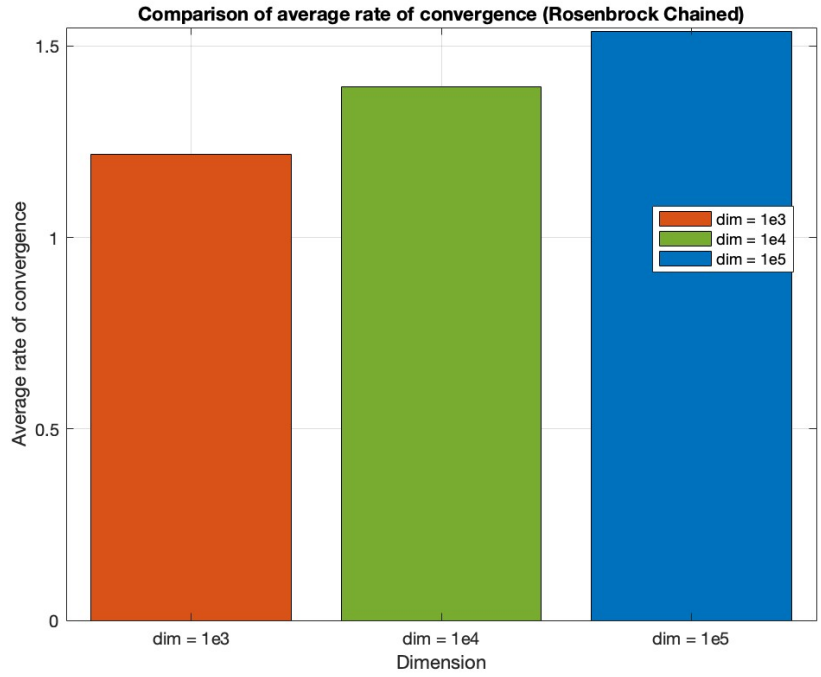

Figure 3.24: Average rate of convergence needed to converge compared the three different dimensions.

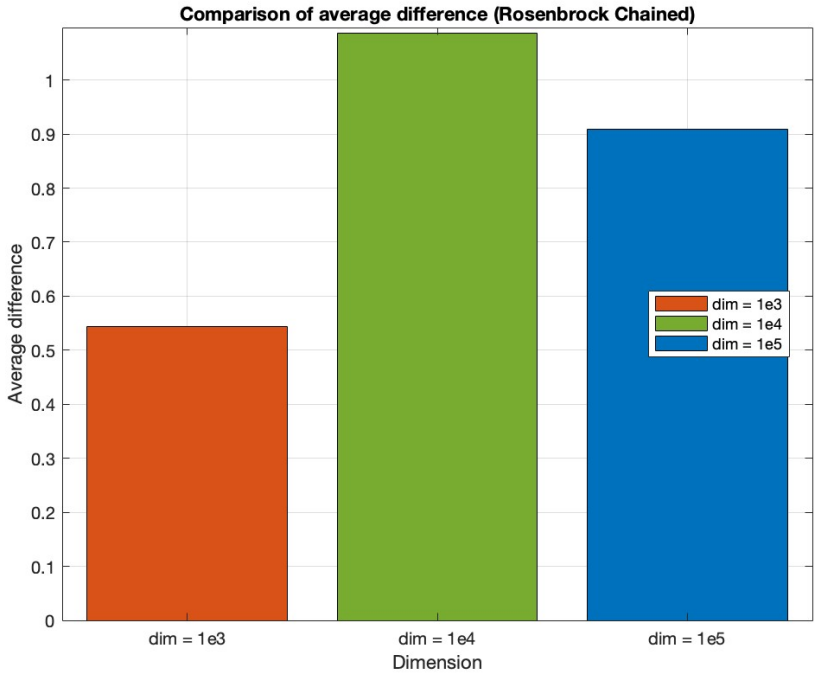

Figure 3.25: Average difference between last point found by the method and real minimizer compared the three different dimensions.

As the region near the minimizer appears significantly flat, we have decided to investigate the behavior of the function evaluation and the gradient into the sequence of points found by the method until convergence, in order to understand which was the cause of the high number of iterations; in the following figures we test the behavior in dimension 10^3 starting from the first point suggested and setting parameters as shown before.
In Fig.(3.26) is shown the trend of the best function evaluations, the starting point is quite distant from the minimizer, so it is registered a significant decrease in the first iterations. In Fig.(3.27) is presented a zoom of the previous image, it can be noticed that in the last 200 iterations the decrease is slower than the beginning, but still evident.
What is significant is the trend of gradient into the sequence of points. In Fig.(3.28) and in Fig.(3.29) can be observed that after few iterations the gradient starts to zig-zagging through two values between two consecutive iterations, nearly 20 and 5, until the last five iterations, in which it decreases until stopping because it reaches the gradient tolerance tol_grad.

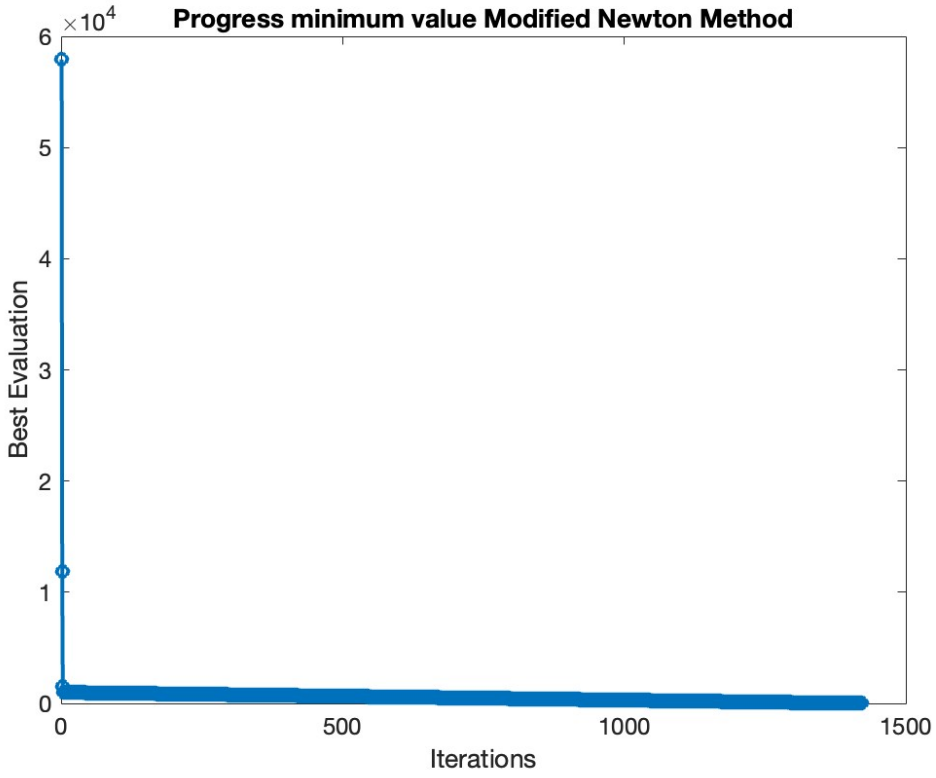

Figure 3.26: Best function evaluations in the sequence obtained by the method with $\rho = 0.9$ and other parameters as before.

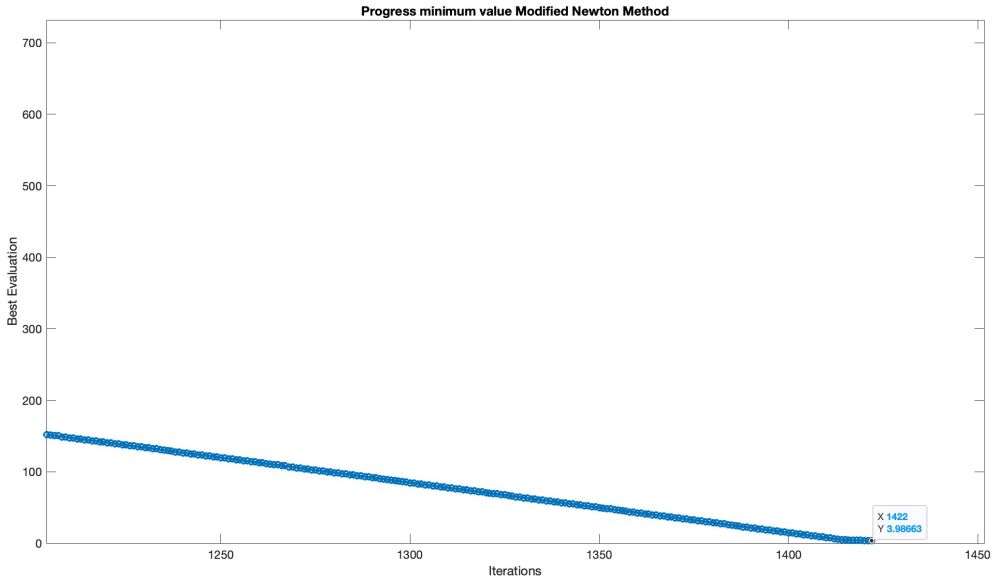

Figure 3.27: Zoom of the best function evaluations in the sequence obtained by the method with $\rho = 0.9$ and other parameters as before.

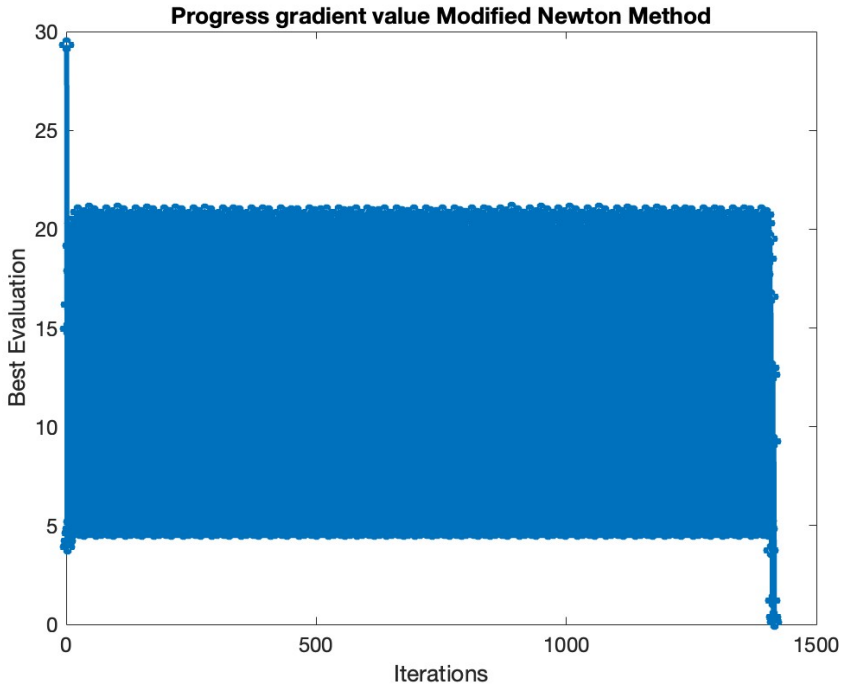

Figure 3.28: Best gradient evaluations in the sequence obtained by the method with $\rho = 0.9$ and other parameters as before.

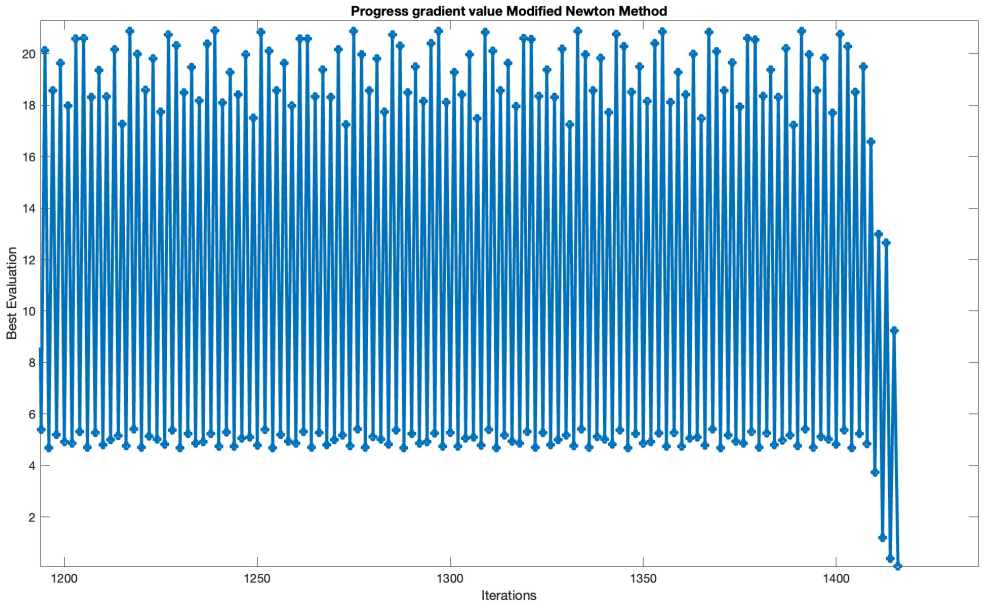

Figure 3.29: Zoom of the best function evaluations in the sequence obtained by the method with $\rho = 0.9$ and other parameters as before.

We tuned backtracking parameters in order to understand if we could improve our results. We show the results from the run of the problem in dimension $n = 10^3$ starting from the first point suggested. Specifically we have observed that fixing all the parameters as before while decreasing

$\rho = 0.2$ imply a slower but more precise convergence; it took $iter = 3261$ compared to $iter = 1422$ with $\rho = 0.9$ as presented in Fig.(3.30) and Fig.(3.31).

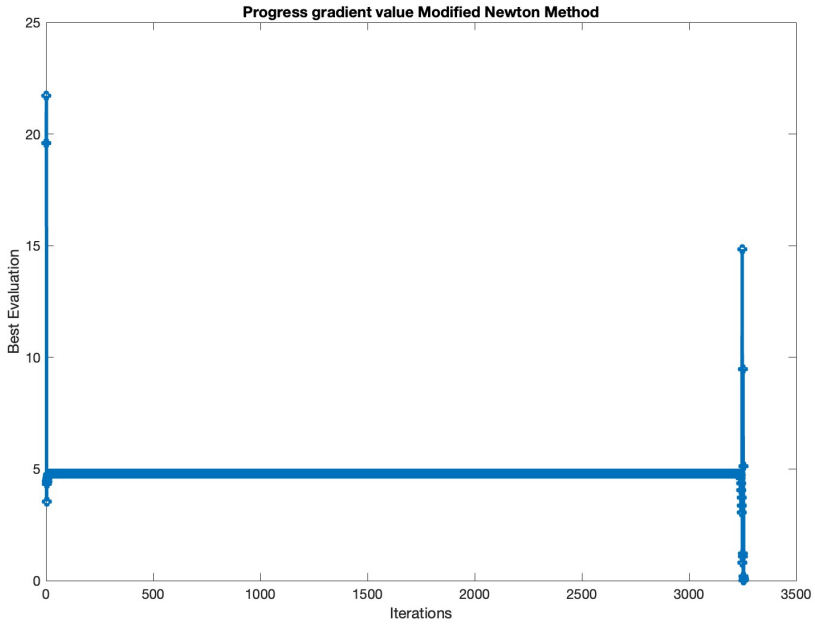

Figure 3.30: Best gradient evaluations in the sequence obtained by the method with $\rho = 0.2$ and other parameters as before.

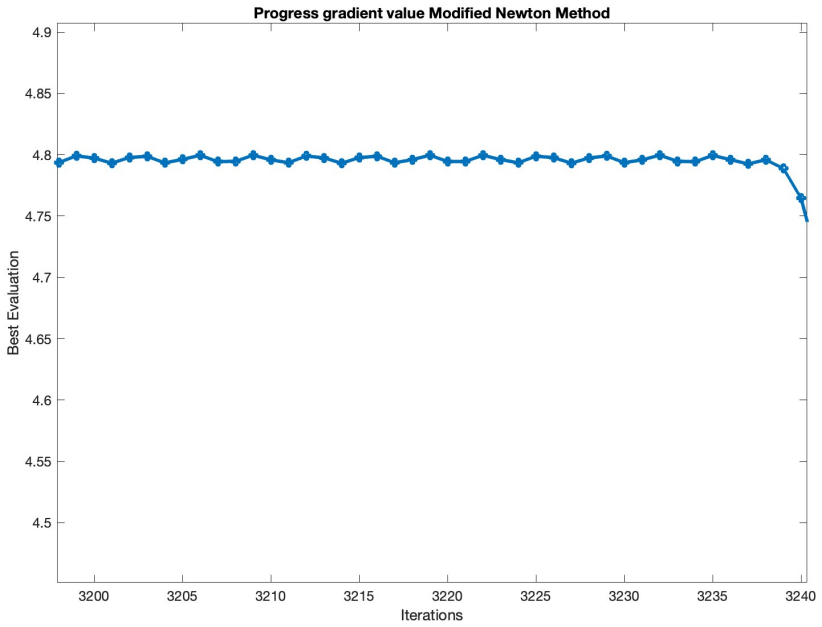

Figure 3.31: Zoom of the best function evaluations in the sequence obtained by the method with $\rho = 0.2$ and other parameters as before.

Neither decreasing $\rho = 0.5$ and $btmax = 50$ improved our results, because the final point was

a little bit less accurate than the one obtained by the original parameters and the number of iterations increase to $iter = 1473$. In addition, the gradient becomes more irregular as we move along the sequence of points as shown in Fig(3.32) and Fig.(3.33).

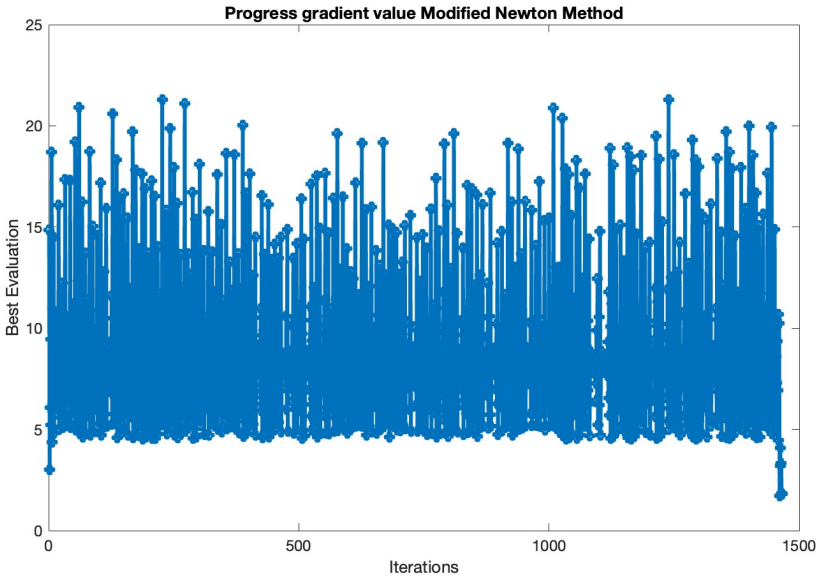

Figure 3.32: Best gradient evaluations in the sequence obtained by the method with $\rho = 0.5$, $btmax = 50$ and other parameters as before.

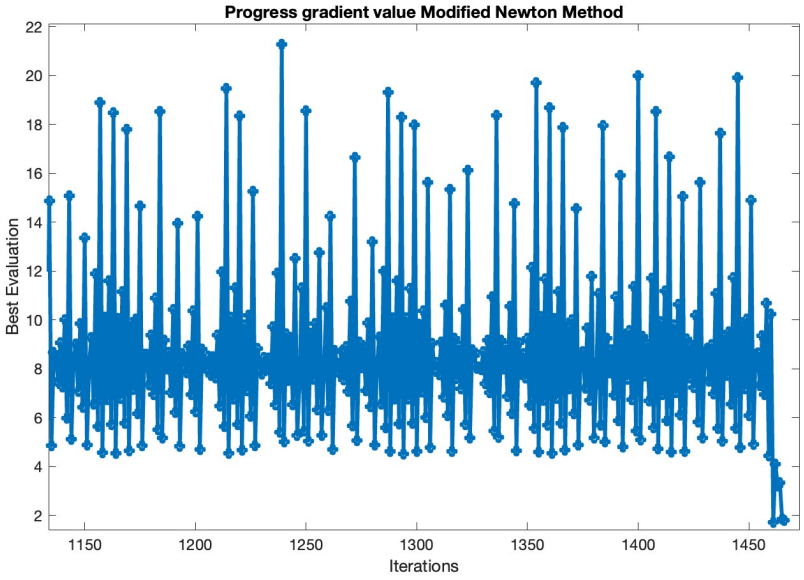

Figure 3.33: Zoom of the best function evaluations in the sequence obtained by the method with $\rho = 0.5$, $btmax = 50$ and other parameters as before.

Finite differences

In this section are reported results obtained from the application of the method to the problem with derivatives computed with finite differences. For the testing phase, we set the following

parameters

$$tol\_grad = 10^{-3} \quad kmax = \begin{cases} 15 \times 10^2 \text{ for dim} = 10^3 \\ 15 \times 10^3 \text{ for dim} = 10^4 \\ 15 \times 10^4 \text{ for dim} = 10^5 \end{cases} \quad \rho = 0.9 \quad btmax = 75 \quad c = 10^{-4}$$

Furthermore, given the high number of function evaluations and the limited computational resources available, we imposed a time limit of 1000s when running the method for dimension $n = 10^5$. We considered the method to have failed if it stopped upon reaching this time limit before achieving convergence.

Constant h

The results obtained differ a lot comparing the three dimensions. Particularly, for $n = 10^3$ we registered only three failures between the total 66 applications of the method. They were registered for $h = 10^{-2}$ because the backtracking failed to find a good step in the maximum number of iterations decided, $btmax = 75$. Increasing the dimension of one order of magnitude, $n = 10^4$, induces a significant increment of failures, about 20 over 66 and all of them due to the inability to find a good step with backtracking. Specifically, the method failed 7 times on 11 for $h = 10^{-2}$, twice for $h \in \{10^{-4}, 10^{-6}\}$ ad three times with $h \in \{10^{-8}, 10^{-10}, 10^{-12}\}$. Indeed, in dimension $n = 10^5$ nearly 80% reached the time limit imposed, while the others stopped due to stagnation. As illustrated in Fig.(3.35), the average time is quite similar to the one registered with exact derivatives, except for $n = 10^5$ when it reached on average $1000s$ for each h value.
Looking at Fig.(3.34), the number of iterations increase significantly with dimension for each step-length value.
The average rate of convergence shown in Fig.(3.36) vary a lot with dimension and h value. For example, when $n = 10^3$ the best results correspond to $h = 10^{-6}$, in dimension $n = 10^4$ it is on average more accurate for $h = 10^{-8}$, while with $n = 10^5$ it is better with $h \in \{10^{-4}, 10^{-8}, 10^{-10}, 10^{-12}\}$.

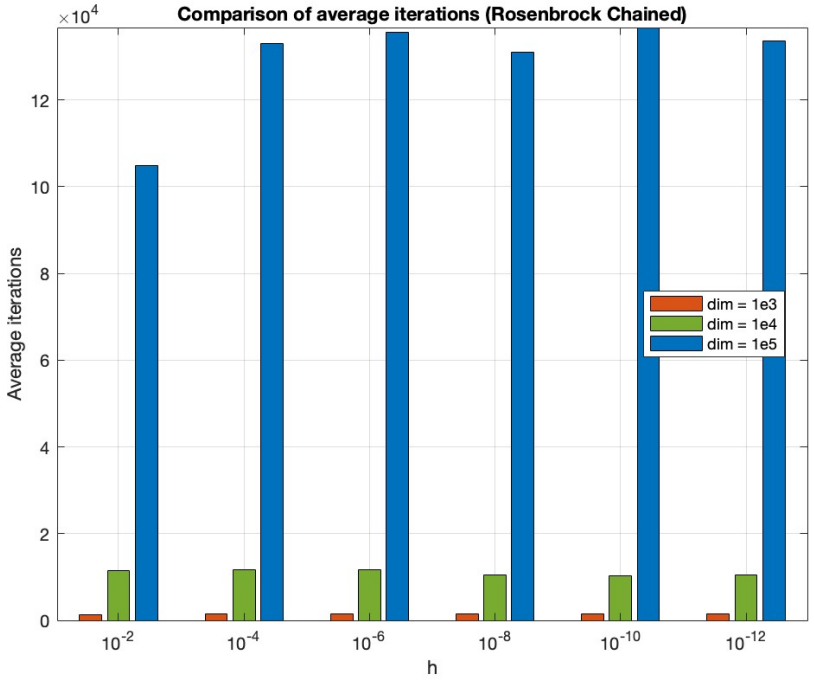

Figure 3.34: Average iterations need to converge compared the three different dimensions.

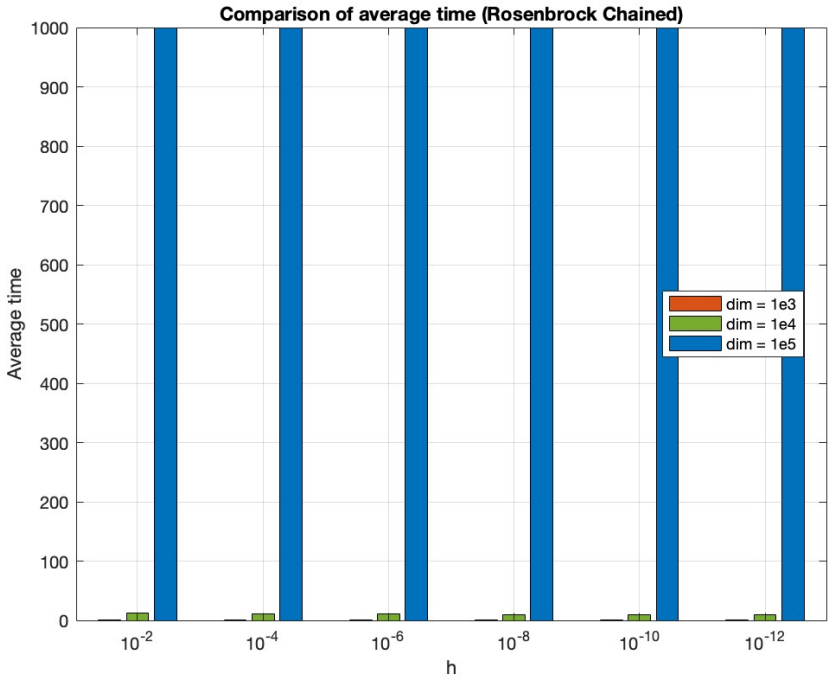

Figure 3.35: Average time need to converge compared the three different dimensions.

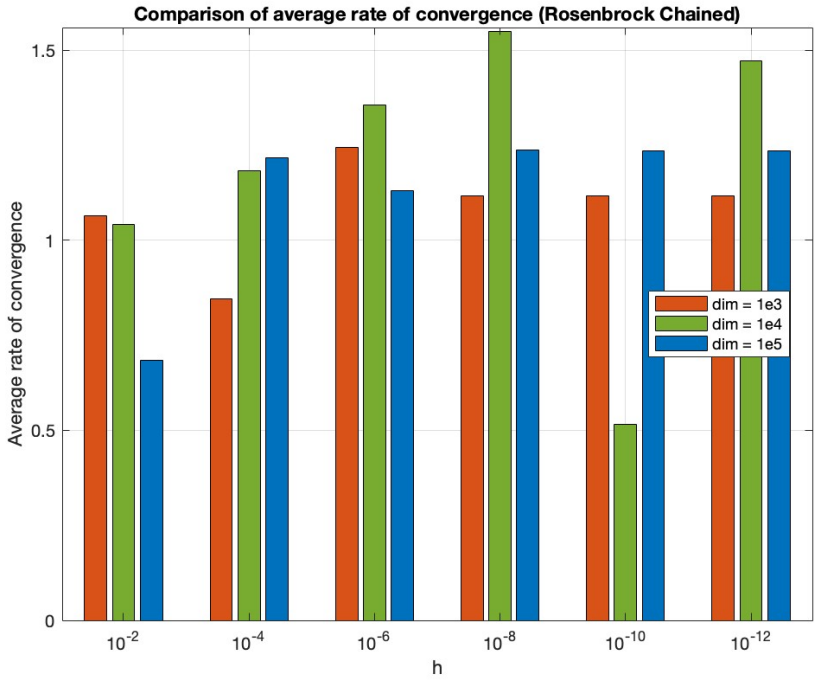

Figure 3.36: Average rate of convergence needed to converge compared the three different dimensions.

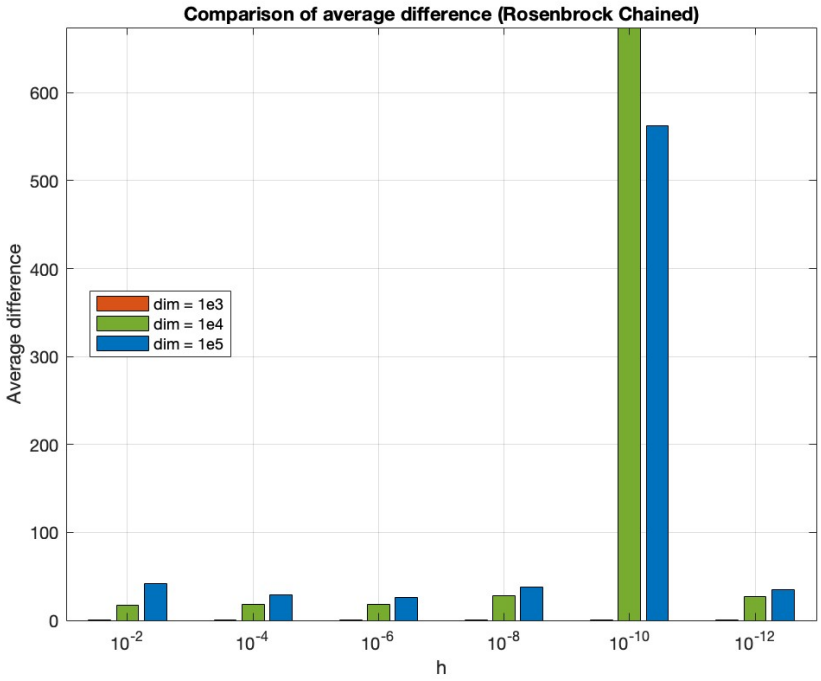

Figure 3.37: Average difference between last point found by the method and real minimizer compared the three different dimensions.

Looking at Fig.(3.37), the average difference in norm between the real minimizer and the point to which the method converge is approximately the same for each h value, increasing with dimension, with the exception of $h = 10^{-10}$ for $n \in \{10^4, 10^5\}$. This is due to the peculiar behavior of the function near the minimizer, it is significantly flat.

Variable h

In this section are reported results obtained with finite differences and a step-length h adapted to each component of the input vector x.
The number of failures depends on both h_i and dimension.

- $n = 10^3 \quad \Rightarrow \quad$ 0 failures
- $n = 10^4 \quad \Rightarrow \quad \begin{cases} 2 \text{ failures with } h_i = 10^{-k}|x_i| \text{ with } k \in \{2, 4, 6\} \\ 3 \text{ failures with } h_i = 10^{-k}|x_i| \text{ with } k \in \{8, 10, 12\} \end{cases}$
- $n = 10^5 \quad \Rightarrow \quad$ every application of the method stopped due to a failure

As for the previous tests, the average number of iterations increase with dimension for each h_i, Fig.(3.38).
Due to time limit, for each step-length value h_i, in dimension $n = 10^5$ the method stopped after $1000s$.
The average rate of convergence is approximately the same for each h_i and dimension $10^3, 10^4$ except for $h_i = 10^{-10}$ and $n = 10^5$, this is due to the fact that the method stopped before reaching the optimum point.
Finally, the norm difference between the real minimizer and the last point found is quite good for dimension $n = 10^3$ and each h_i. It increases on average in dimension $n = 10^4$ until varying a lot for all h_i in dimension $n = 10^5$, due to the early stopping because the method reached time limit.

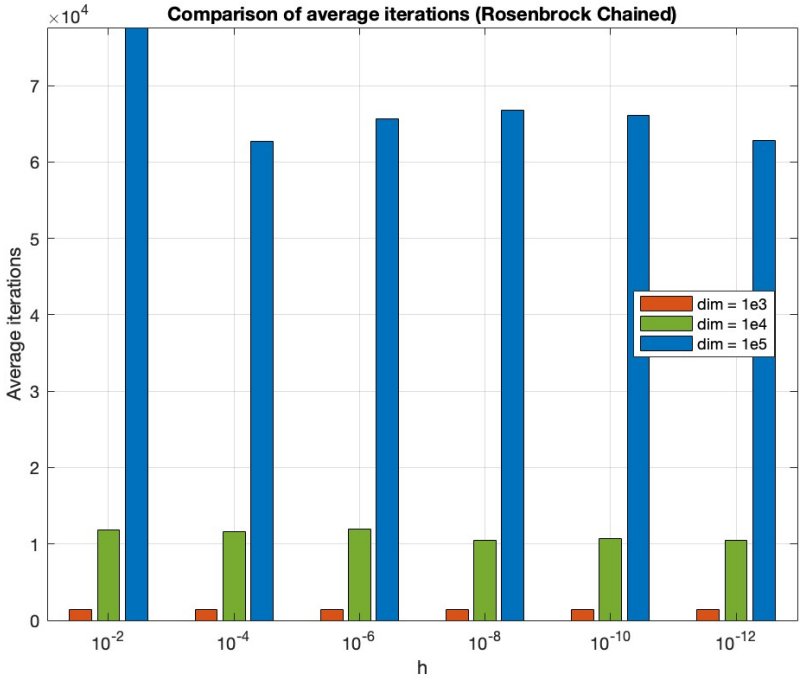

Figure 3.38: Average iterations need to converge compared the three different dimensions.

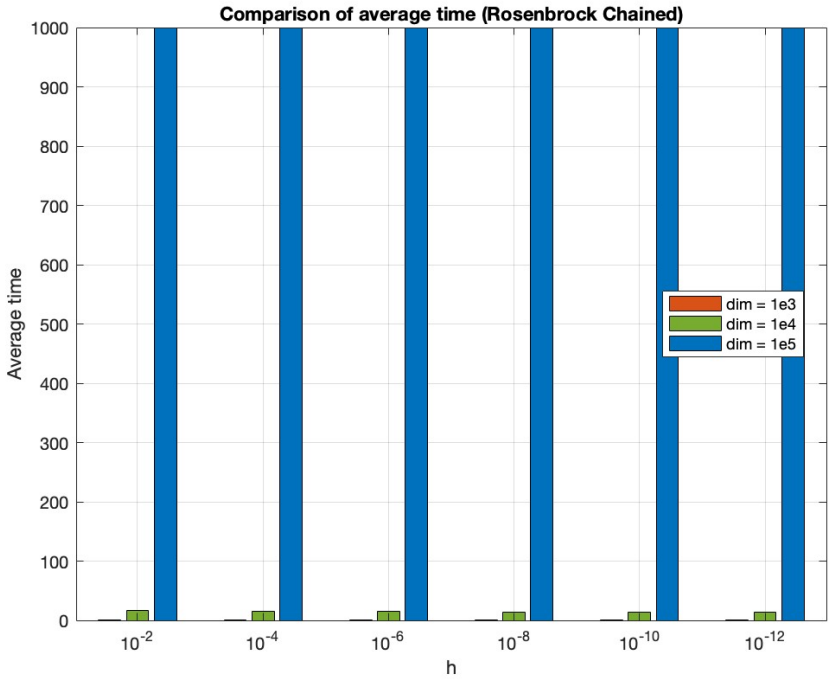

Figure 3.39: Average time need to converge compared the three different dimensions.

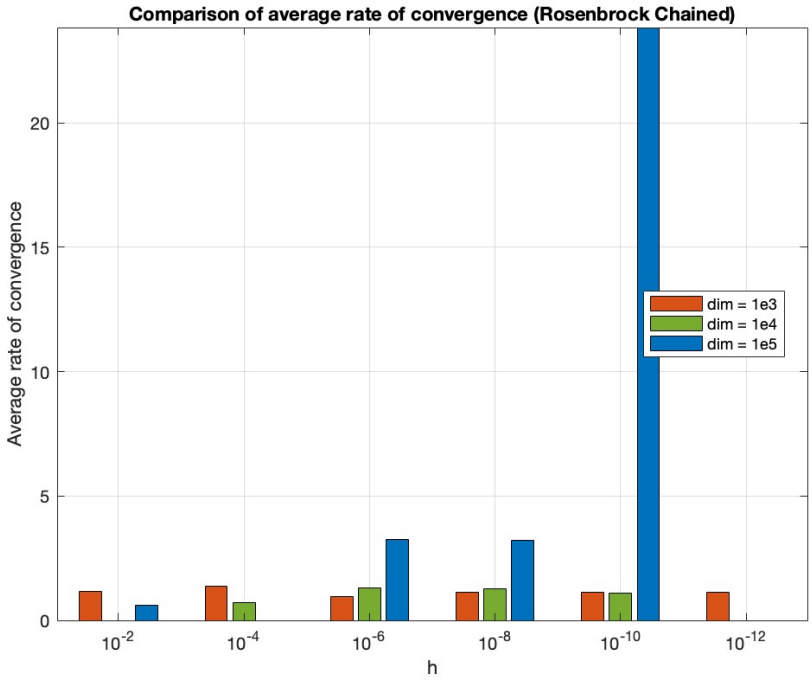

Figure 3.40: Average rate of convergence needed to converge compared the three different dimensions.

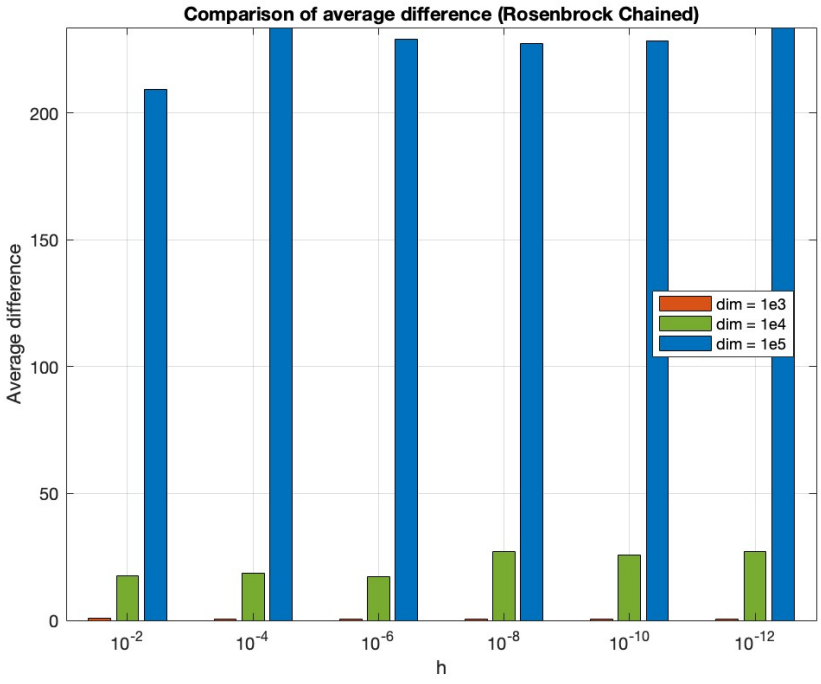

Figure 3.41: Average difference between last point found by the method and real minimizer compared the three different dimensions.

A few considerations can be deduced. First, exact derivatives are more precise then finite

differences due to the fact that the hessian matrix is highly ill conditioned. Second, the imposition of a time limit impact the precision of the method in terms of final point obtained and experimental rate of convergence.

3.7 Comparison Nelder-Mead - Modified Newton method (exact)

The Modified Newton Method is clearly superior in terms of accuracy, finding solutions relatively close to the real optimum even in extremely high dimensions (up to 10^5) with average error below 1.5. Nelder-Mead, on the other hand, loses significant accuracy already at much smaller dimensions (Dim 25 and 50) with average error of about 6.
Regarding efficiency in iterations and time both methods show a significant costs especially the modified Newton method for the highest tested dimensions. However, the Modified Newton Method still manages to maintain accuracy, whereas Nelder-Mead does not.

3.8 Chained Wood function

3.8.1 Nelder-mead

Results for the Nelder Mead method with chained Wood function.

Default tolerance for stopping criteria

1. first stopping criteria $\longrightarrow$ tol_simplex = 1e-07;
2. second stopping criteria $\longrightarrow$ tol_varf = 1e-07;
3. kmax = 10000

Tuned parameters
We found the best configuration of parameters for the suggested initial point. In the algorithm we form all possible configuration from values:

```
rho_vec = [0.25, 0.5, 1, 1.35, 1.75];
sigma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
gamma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
chi_vec = [1.1, 1.5, 2, 2.5, 3];
```

The best configuration of parameters we found is:

- for dimension = 10:

$$\rho = 1, \quad \sigma^2 = 0.5, \quad \chi = 2.5, \quad \gamma = 0.75$$

- for dimension = 25:

$$\rho = 1.35, \quad \sigma^2 = 0.5, \quad \chi = 1.5, \quad \gamma = 0.9$$

- for dimension = 50:

$$\rho = 1, \quad \sigma^2 = 0.1, \quad \chi = 2, \quad \gamma = 0.9$$

Before presenting the results, a few comments.
Tables 3.14, 3.15, 3.16 summarize the main outcomes of the algorithm when applied to the method using the 11 initial points (the first being the suggested one, and the others being random points). The first column shows the distance of the solution of the last iteration from the optimum. The third column indicates the reason why the method stopped, corresponding to the stopping criterion that was satisfied:

- flag = 1: The method reached the first stopping criterion (related to the size of the simplex)
- flag = 2: The method reached the second stopping criterion (related to the stationary region)
- flag = 3: The method reached the maximum number of iterations without satisfying one the first two stopping criteria.

The fourth column gives information about the order of magnitude of computational costs.

Results for dimension = 10:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	0.0001	876	2	0.1
$\mathbf{p}_2$	0.0002	4071	2	0.1
$\mathbf{p}_3$	0.0002	1759	2	0.1
$\mathbf{p}_4$	2.7587	10000	3	0.1
$\mathbf{p}_5$	1.9228	2295	2	0.1
$\mathbf{p}_6$	0.0002	3041	2	0.1
$\mathbf{p}_7$	0.0003	3040	2	0.1
$\mathbf{p}_8$	1.9222	10000	3	0.1
$\mathbf{p}_9$	1.9221	1611	2	0.1
$\mathbf{p}_{10}$	0.0001	3048	2	0.1
$\mathbf{p}_{11}$	0.0001	1209	2	0.1
sample average	0.7763	3613	/	0.1

Table 3.14: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random points)

Last 5 exponential rate of convergence for each point:

$$
\begin{aligned}
\text{Point 1:} &\quad \text{NaN} \quad 0 \quad 0.034 \quad \infty \quad \text{NaN} \\
\text{Point 2:} &\quad 0 \quad -0.59 \quad \infty \quad \text{NaN} \quad 0 \\
\text{Point 3:} &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad 0 \\
\text{Point 4:} &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
\text{Point 5:} &\quad 0 \quad \infty \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
\text{Point 6:} &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
\text{Point 7:} &\quad -\infty \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad 0 \\
\text{Point 8:} &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
\text{Point 9:} &\quad 0 \quad -\infty \quad \text{NaN} \quad \text{NaN} \quad 0 \\
\text{Point 10:} &\quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \quad \text{NaN} \\
\text{Point 11:} &\quad 0.70 \quad -2.77 \quad 0.061 \quad \infty \quad \text{NaN}
\end{aligned}
$$

Comments on results: There is sensitivity to the initial point, which affects the quality of the solution. Starting the method from points such as p_1 or p_{10}, the method converges very close to the optimum (0.001), while for points like p_4 or p_9, the distance between the convergence point and the optimum is on average about 2. The type of convergence also varies depending on the initial point: we can have no convergence or second-type convergence. For point p_4, for example, the method reaches the maximum number of iterations.
An atypical behavior is observed for point 4 (Figure 3.42 and Figure 3.43), whose simplex, as seen in Figure 3.42, does not appear to explore the space.
In fact, upon observing the simplex, it remains constant through the iterations, thus keeping both its size and the best point constant.

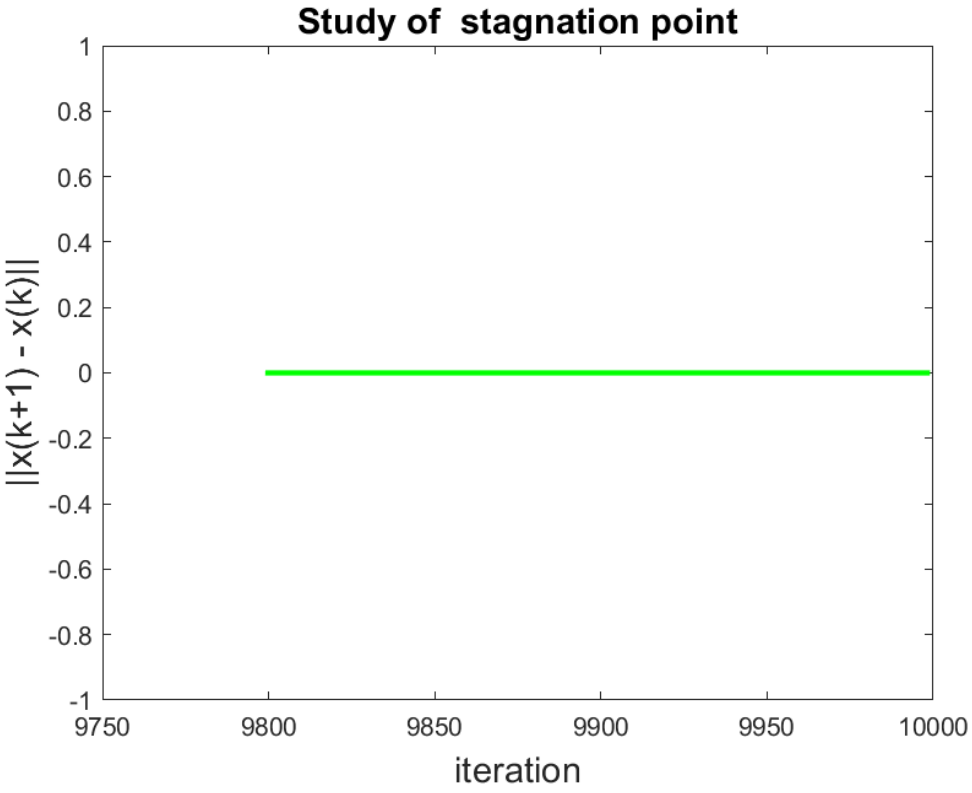

Figure 3.42: Variation of solution in the last iterations, initial point = p_4

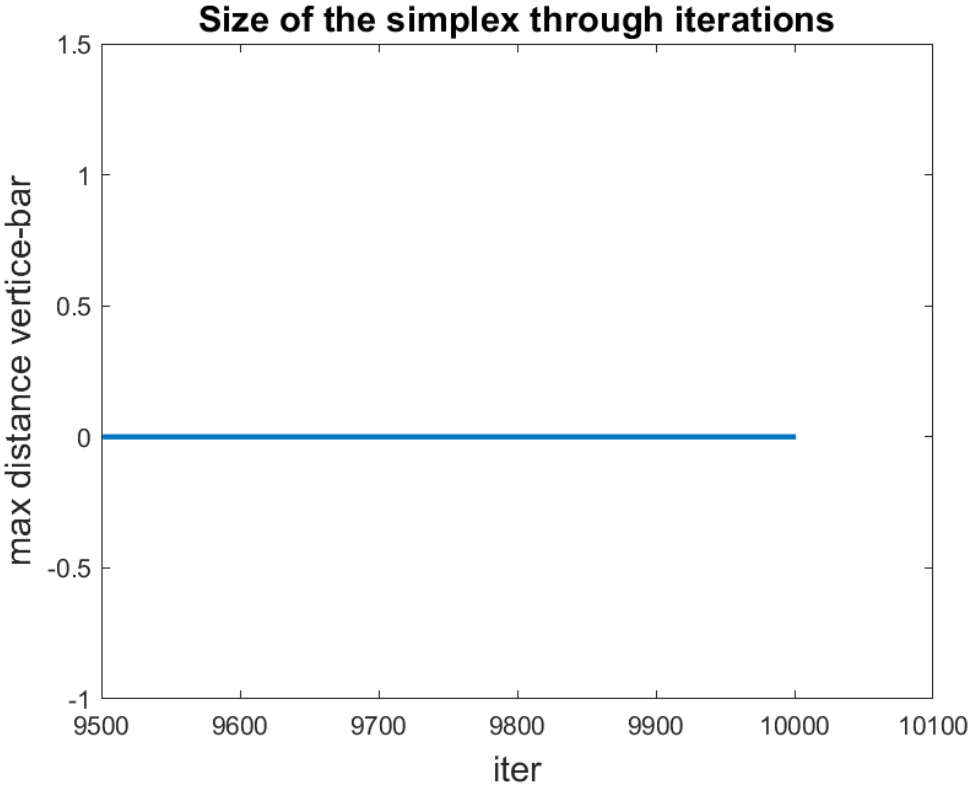

Figure 3.43: Size of the simplex through last iterations, initial point = p_4

It is noted that simplex's coordinates (with p_4) remains constantly equal to:

$$\begin{pmatrix}
0.9852 & 0.9705 & 1.0153 & 1.0308 & -0.9769 & 0.9650 & -0.9084 & 0.8362 & 1.0755 & 1.1571 \\
0.9851 & 0.9703 & 1.0154 & 1.0311 & -0.9768 & 0.9648 & -0.9084 & 0.8363 & 1.0755 & 1.1570 \\
0.9852 & 0.9705 & 1.0153 & 1.0309 & -0.9769 & 0.9650 & -0.9083 & 0.8361 & 1.0755 & 1.1571 \\
0.9852 & 0.9705 & 1.0153 & 1.0309 & -0.9768 & 0.9649 & -0.9084 & 0.8362 & 1.0755 & 1.1570 \\
0.9851 & 0.9703 & 1.0154 & 1.0310 & -0.9768 & 0.9648 & -0.9085 & 0.8363 & 1.0754 & 1.1570 \\
0.9850 & 0.9702 & 1.0154 & 1.0312 & -0.9767 & 0.9647 & -0.9085 & 0.8364 & 1.0754 & 1.1568 \\
0.9852 & 0.9705 & 1.0153 & 1.0308 & -0.9769 & 0.9651 & -0.9083 & 0.8360 & 1.0756 & 1.1573 \\
0.9850 & 0.9702 & 1.0154 & 1.0312 & -0.9767 & 0.9647 & -0.9085 & 0.8365 & 1.0754 & 1.1569 \\
0.9852 & 0.9704 & 1.0153 & 1.0309 & -0.9769 & 0.9649 & -0.9084 & 0.8363 & 1.0755 & 1.1571 \\
0.9850 & 0.9701 & 1.0154 & 1.0312 & -0.9768 & 0.9647 & -0.9085 & 0.8364 & 1.0754 & 1.1569 \\
0.9851 & 0.9703 & 1.0154 & 1.0310 & -0.9768 & 0.9648 & -0.9084 & 0.8364 & 1.0754 & 1.1569
\end{pmatrix}$$

The rows of the simplex matrix appear to be nearly linearly dependent and highly similar, which likely compromises its ability to effectively explore the search space.

Results for dimension = 25:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	0.4404	10000	3	1
$\mathbf{p}_2$	7.2122	10000	3	1
$\mathbf{p}_3$	3.9077	10000	3	1
$\mathbf{p}_4$	7.4077	10000	3	1
$\mathbf{p}_5$	3.4051	10000	3	1
$\mathbf{p}_6$	7.2124	10000	3	1
$\mathbf{p}_7$	3.1748	10000	3	1
$\mathbf{p}_8$	7.2670	10000	3	1
$\mathbf{p}_9$	5.1190	10000	3	1
$\mathbf{p}_{10}$	7.3992	10000	3	1
$\mathbf{p}_{11}$	2.9617	10000	3	1
sample average	5.5005	10000	/	1

Table 3.15: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	NaN	NaN	NaN	NaN	NaN
Point 5:	NaN	NaN	NaN	NaN	NaN
Point 6:	$-\infty$	NaN	NaN	NaN	NaN
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	NaN	NaN	NaN	NaN	0
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results: The method with this function and this dimension is rather poor:
- The average error of the solution from the optimum is 5.
- All iterations are exhausted before reaching any type of convergence, which leads to a high computational cost.

The average error consistently increased compared to the previous dimension. This rise could be attributed to both the higher dimensionality and the fact that the Wood function is not defined for odd dimensions. For instance, at dimension 24, the average error was 2.4434, while at dimension 26, it raise to 4.5378.

Results for dimension = 50:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	1.2504	10000	3	10
$\mathbf{p}_2$	9.6513	10000	3	10
$\mathbf{p}_3$	5.5529	10000	3	10
$\mathbf{p}_4$	9.8437	10000	3	10
$\mathbf{p}_5$	4.1307	10000	3	10
$\mathbf{p}_6$	5.1948	10000	3	10
$\mathbf{p}_7$	5.0834	10000	3	10
$\mathbf{p}_8$	2.7808	10000	3	10
$\mathbf{p}_9$	5.2271	10000	3	10
$\mathbf{p}_{10}$	3.4560	10000	3	10
$\mathbf{p}_{11}$	9.1621	10000	3	10
sample average	5.57	10000	/	10

Table 3.16: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point)

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	NaN	NaN	NaN	NaN	NaN
Point 5:	NaN	NaN	NaN	NaN	NaN
Point 6:	NaN	NaN	NaN	NaN	NaN
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	NaN	NaN	NaN	NaN	NaN
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results: The same behavior observed in dimension 25 is repeated, with worse results in terms of computational time: the order of magnitude is 10 seconds.

An overview:

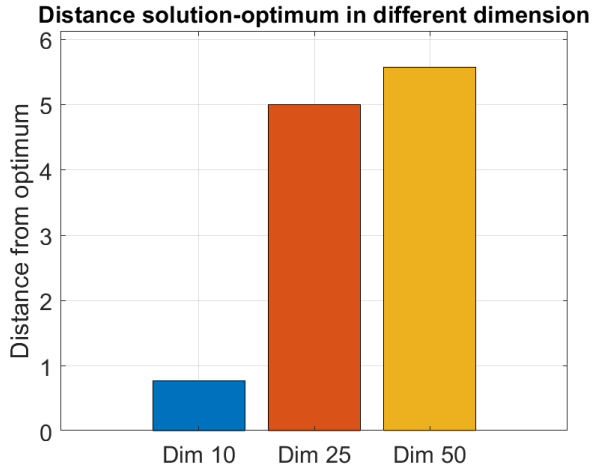

Figure 3.44: Chained Wood: Distance from optimum

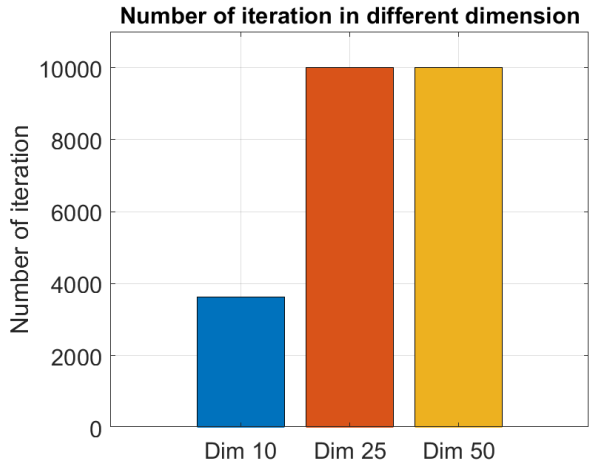

Figure 3.45: Chained Wood: Number of iteration

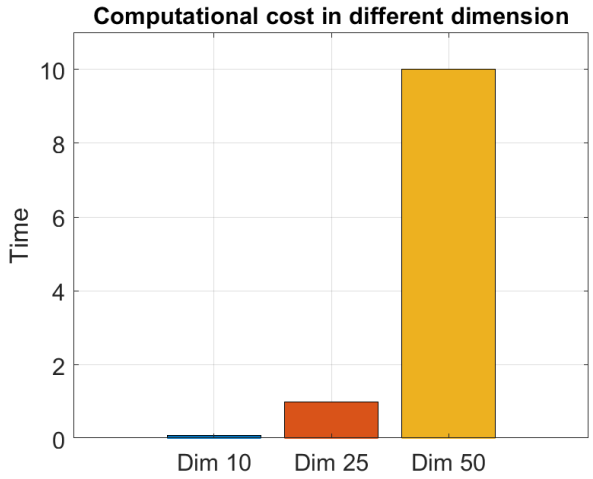

Figure 3.46: Chained Wood: Computational cost

3.8.2 Modified Newton method

Gradient and hessian matrix of the chained Wood function f.

$$\nabla f = \begin{cases} 400(x_1^3 - x_1 x_2) + 2(x_1 - 1) \\ -200x_1^2 + \frac{1101}{5}x_2 + \frac{99}{5}x_4 - 40 \\ 760x_i(x_i^2 - x_{i+1}) + 4(x_i - 1) \quad \forall i = 3, 5, 7, \ldots, n-3 \\ \frac{99}{5}(x_{i-2} + x_{i+2}) - 380x_{i-1}^2 + \frac{2102}{5}x_i - 80 \quad \forall i = 4, 6, 8, \ldots, n-2 \\ 360x_{n-1}(x_{n-1}^2 - x_n) + 2x_n - 2 \\ \frac{99}{5}x_{n-2} - 18x_{n-1}^2 + \frac{1101}{5}x_n - 40 \end{cases}$$

$$\nabla^2 f = \begin{bmatrix} 400(3x_1^2 - x_2) + 2 & -400x_1 & 0 & 0 & \cdots & \cdots & \cdots \\ -400x_1 & \frac{1101}{5} & 0 & \frac{99}{5} & \cdots & \cdots & \cdots \\ 0 & 0 & 2280x_3^2 + 4 - 760x_4 & -760x_3 & \cdots & \cdots & \cdots \\ 0 & \frac{99}{5} & -760x_3 & \frac{2102}{5} & 0 & \frac{99}{5} & \cdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \ddots & 0 & 1080x_{n-1}^2 - 360x_n + 2 & -360x_{n-1} \\ 0 & 0 & \cdots & \cdots & \frac{99}{5} & -360x_{n-1} & \frac{1001}{5} \end{bmatrix}$$

where elements in positions (i, i) and $(i, i+1)$ for $i = 3, 5, 7, \ldots, n-3$ have always the same structure

$$\begin{cases} H_{i,i} = 2280x_i^2 + 4 - 760x_{i+1} \\ H_{i,i+1} = -760x_i \\ \forall i = 3, 5, 7, \ldots, n-3 \end{cases} \qquad \begin{cases} H_{i,i-2} = \dfrac{99}{5} \\ H_{i,i-1} = -760x_{i-1} \\ H_{i,i} = \dfrac{2101}{5} \\ H_{i,i+1} = 0 \\ H_{i,i+2} = \dfrac{99}{5} \\ \forall i = 2, 4, 6, \ldots, n-3 \end{cases}$$

Finite differences

$$\nabla f \approx \begin{cases} 400(x_1^3 - x_1 x_2) + 2(x_1 - 1) + 400h_1^2 x_1 \\ -200x_1^2 + \frac{1101}{5}x_2 + \frac{99}{5}x_4 - 40 \\ 760x_i(x_i^2 - x_{i+1}) + 4(x_i - 1) + 760h_i x_i \quad \forall i = 3, 5, 7, \ldots, n-3 \\ \frac{99}{5}(x_{i-2} + x_{i+2}) - 380x_{i-1}^2 + \frac{2102}{5}x_i - 80 \quad \forall i = 4, 6, 8, \ldots, n-2 \\ 360x_{n-1}(x_{n-1}^2 - x_n) + 2x_n - 2 + 180h_{n-1}x_{n-1}^2 \\ \frac{99}{5}x_{n-2} - 18x_{n-1}^2 + \frac{1101}{5}x_n - 40 \end{cases}$$

while the hessian matrix computed with finite differences is equal to the hessian matrix with

exact derivatives except for the following components

$$
\begin{aligned}
H_{1,1} &\approx 400(3x_1^2 - x_2) + 2 + 200h_1^2 \\
H_{1,2} &\approx -200(2x_1 + h_1) \\
H_{2,1} &\approx -200(2x_1 + h_1) \\
H_{i,i} &\approx 2280x_i^2 + 4 - 380(2x_{i+1} + h_i^2) \quad \forall i = 3, 5, 7, \ldots, n-3 \\
H_{i,i+1} &\approx -100(76x_i + 38h_i) \quad \forall i = 3, 5, 7, \ldots, n-3 \\
H_{i,i-1} &\approx -100(76x_i + 38h_i) \quad \forall i = 2, 4, 6, \ldots, n-2 \\
H_{n-1,n-1} &\approx 180(6x_{n-1}^2 - 2x_n + h_{n-1}^2) + 2 \\
H_{n-1,n} &\approx -180(2x_{n-1} + h_n) \\
H_{n,n-1} &\approx -180(2x_{n-1} + h_n)
\end{aligned}
$$

At the beginning we implemented Hessian matrix with a sparse structure filling it with for loops for components in rows $3 : 2 : n-3$ and $4 : 2 : n-2$, then we realized that in dimension $n = 10^5$ the computation of the matrix required nearly $8s$ and this was too long, so we deleted for loops and created the matrix using *spdiags* because it is a matrix with five nonzero diagonals. This allowed us to compute the Hessian matrix in a time reduced by four orders of magnitude.

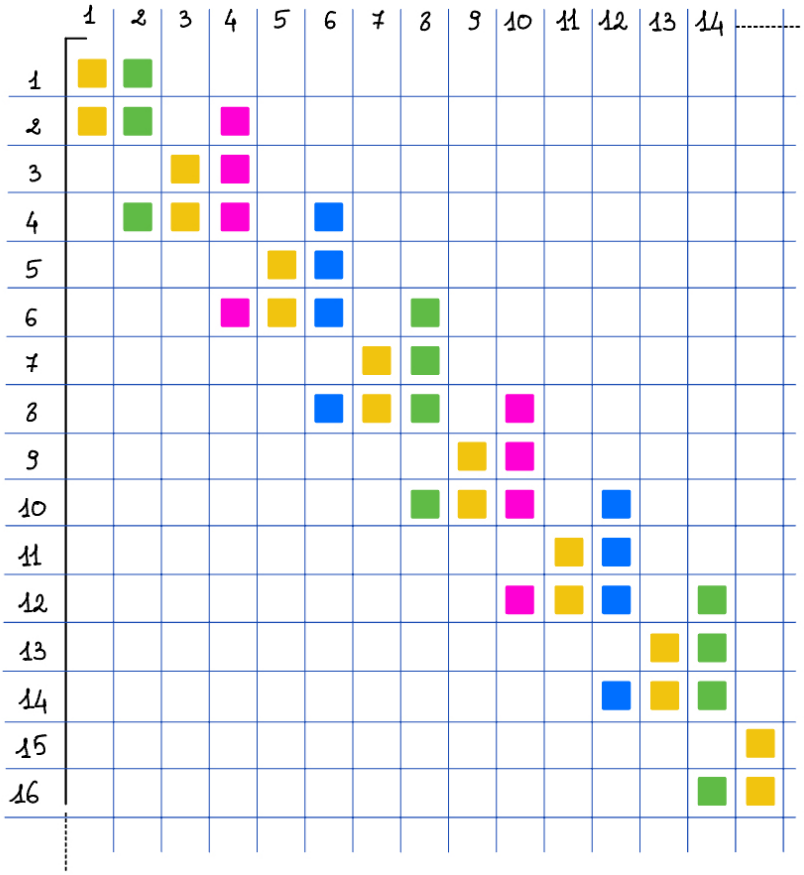

Figure 3.47: Hessian matrix of Wood function underling elements in the same column.

We also compute Hessian matrix with the step-length h in the form $h_i = 10^{-k}|x_i| \quad \forall k = 2, 4, 6, 8, 10, 12$ as the Jacobian of the exact gradient using the sparse structure of the Hessian

matrix. In particular we compute at the same time the gradient in the point x plus h_{i_s} corresponding to the i_s column with the same color in Fig 3.47 in order to make the minimum number of function evaluations.
By analyzing the derivatives of the function, we can deduce that all gradient's components are nullified by $x = \mathbf{1}$. This leads to conclusion that wood function assumes its global minimum in that point.
To better understand the behavior of the function $f(x)$ near the minimizer we report a plot of the function in 2D for $n = 2$.

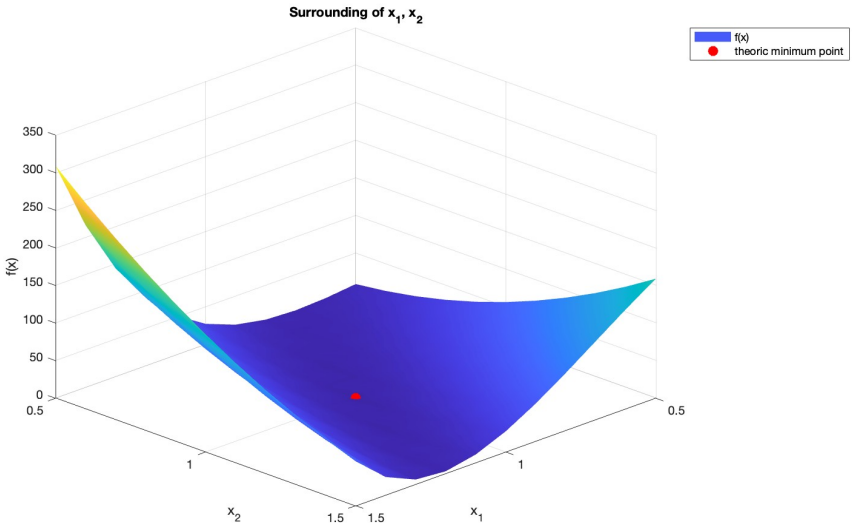

Figure 3.48: 2D plot of the function $f(x_1, x_2)$

From the plot, we can notice that the function is almost flat near the minimum; this may affect the performance of modified newton method because flat region may cause slow convergence or even stagnation for optimization solvers.

Results for modified Newton method with chained Wood function.
In this section are reported solutions from the application of Modified Newton Method to chained wood function considering exact derivatives. We store the Hessian matrix in a sparse and diagonal form due to the high dimension of the input vector $\{10^3, 10^4, 10^5\}$ memorizing only the upper diagonals of the matrix and defining the lower diagonals using the symmetry.
For each dimension we execute the method fixing the following parameters:

$$tol\_grad = 10^{-2} \quad kmax = 900 \quad \rho = 0.9 \quad c = 10^{-4} \quad btmax = 90$$

Specifically, we set a big tolerance tol_grad in order to compare the results of the method in the three different dimension, a smaller tolerance required a significant increment in time with dimension $n = 10^5$.
Next are reported some bar plots that compare results obtained by modified newton method applied to the problem in dimension $n = 10^3, 10^4, 10^5$ in terms of average time, average number of iterations, average difference between real minimizer and last point found and average rate of convergence.

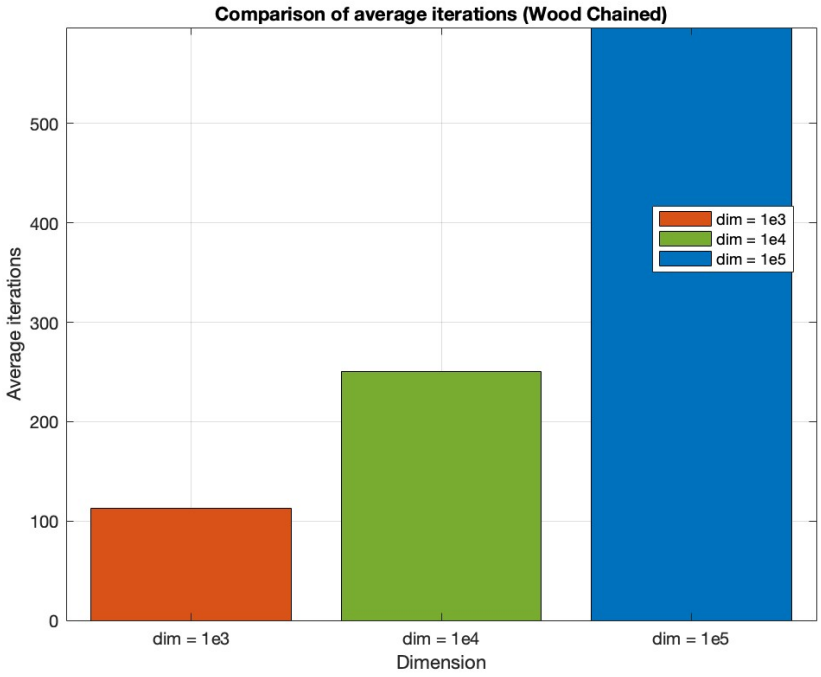

Figure 3.49: Average iterations needed to converge compared the three different dimensions.

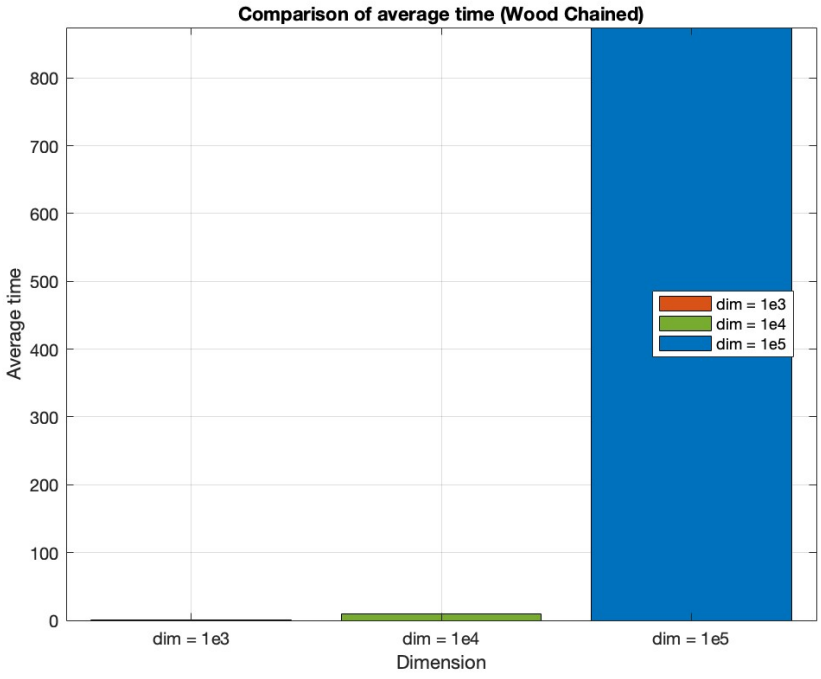

Figure 3.50: Average time needed to converge compared the three different dimensions.

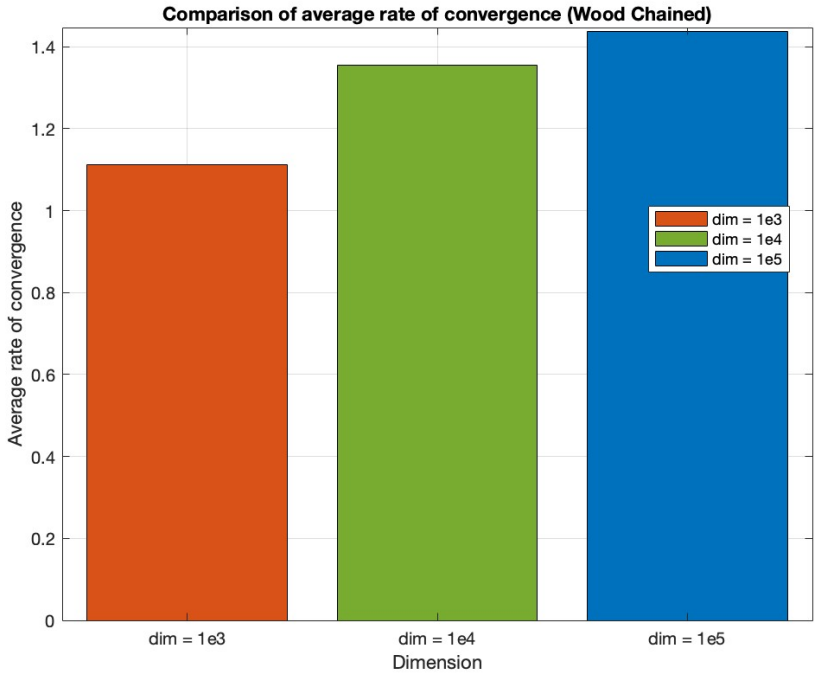

Figure 3.51: Average rate of convergence needed to converge compared the three different dimensions.

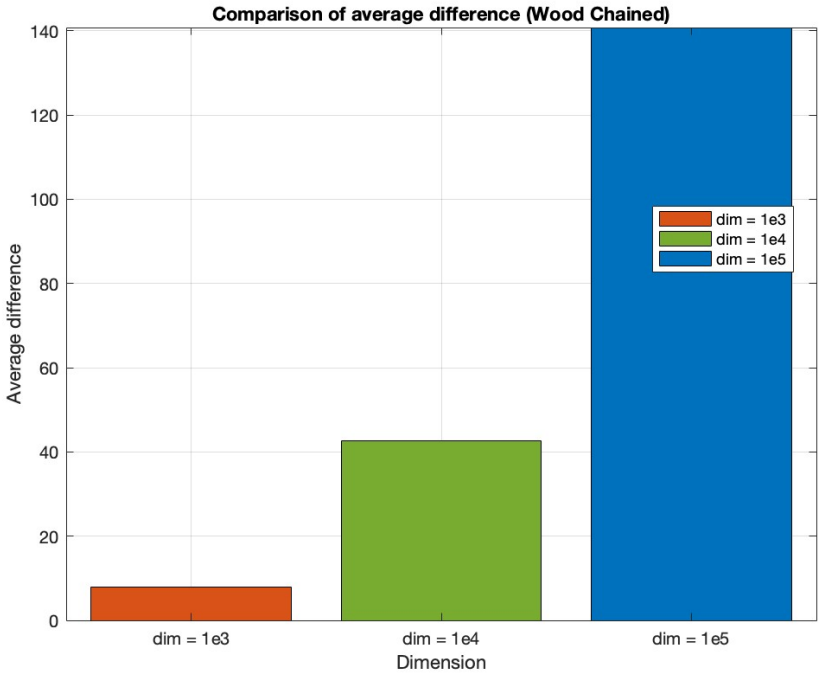

Figure 3.52: Average difference between last point found by the method and real minimizer compared the three different dimensions.

We executed the method and obtained only one failure, specifically in dimension $n = 10^4$ from

the 10^{th} starting point where backtracking failed to find a good step.
In Fig.(3.49) can be observed the increase of number of iterations as the dimension rises. In dimension $n = 10^3$ the average number of iterations is $iter = 112$, in dimension $n = 10^4$ it is $iter = 250$ and it becomes $iter = 596$ when $n = 10^5$. The same behavior is found in the average time required, it increases with dimension, as shown in Fig.(3.49).
Even though the method stops without failures, except in one case, the last point found is quite far from the optimum and the distance rises with dimension, as illustrated in Fig.(3.52), this is explained by Fig.(3.48) in which is underlined the flatness of the function near the minimizer.
Finally, the average rate of convergence is above 1 for each dimension and registered the same trend of iterations and time when input dimension becomes bigger.
The following figures report results obtained applying the method to the first point suggested in dimension $n = 10^3$ and setting parameters as illustrated at the beginning of this section.

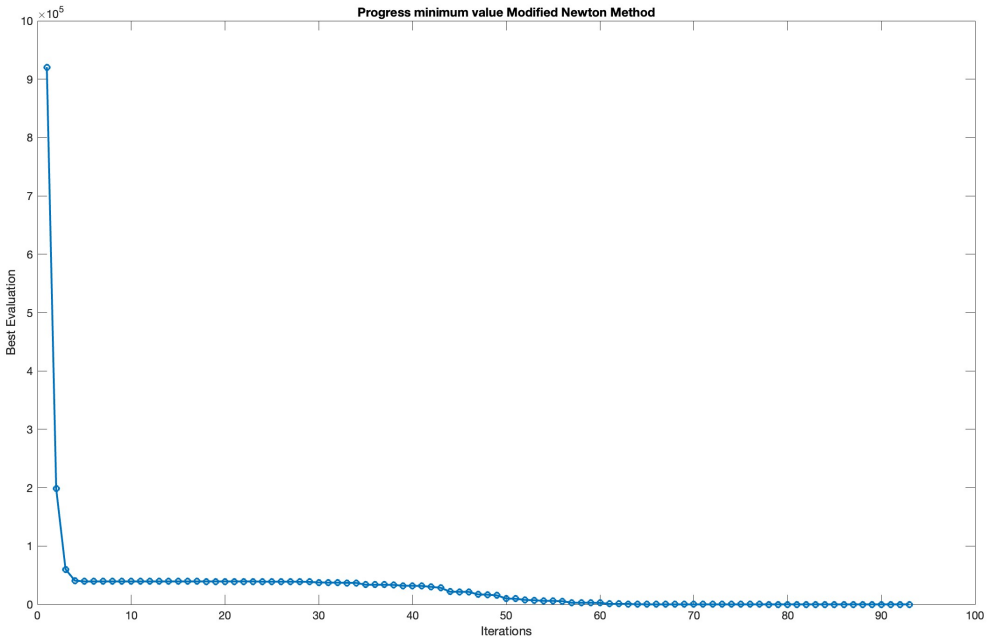

Figure 3.53: Function evaluation in the sequence of point found by the method until convergence with $\rho = 0.9$ starting from the first point suggested.

In Fig.(3.53) is illustrated the function evaluation in the sequence of point obtained by the method, it can be deduced an initial fast decrease in the first 4 iterations, then the function continue to get closer to the minimum but slowly. In Fig.(3.54) is reported the behavior of the gradient norm, surprisingly there is an increase during the first 60 iterations, then it oscillates around a value a little greater than 0 and from the iteration 72 it starts a slow decrease until it reaches the tolerance tol_grad.

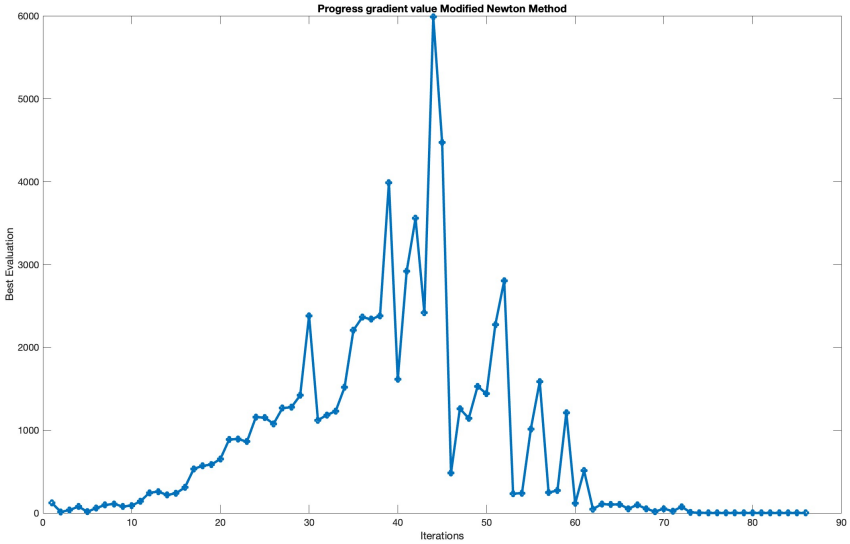

Figure 3.54: Gradient evaluation in the sequence of point found by the method until convergence with $\rho = 0.9$ starting from the first point suggested.

We tried to tune the parameters to see if the results could improve but this did not happen. Decreasing the value of $btmax$ caused an increase in the number of failures. Setting ρ to a value lower than 0.9 caused an increment in both number of iterations and time required without improving the distance between the real minimizer and the one obtained by the method.

Finite differences

In this section are reported results from the method applied to the function with derivatives computed with finite differences.
We use as parameters the ones resulting the best value configuration:

$$tol\_grad = 10^{-2} \quad kmax = 700 \quad \rho = 0.9 \quad c = 10^{-3} \quad btmax = 90$$

We chose $btmax = 90$ because a lower value significantly increased the number of failures; conversely, a higher value required a longer average time to reach convergence.
We imposed a limit time equal to $1000s$ after looking at the time required with exact derivatives, Fig.(3.50). Specifically, with exact derivatives the maximum time to converge was nearly $990s$ when $n = 10^5$.

Constant h

The number of failures vary with dimension n and the step-length value h. Specifically we ran the method for 11 different starting point for each $h = 10^{-k}$ with $k \in \{2, 4, 6, 8, 10, 12\}$:

- $n = 10^3 \quad \Rightarrow \quad \begin{cases} 11 \text{ failures with } h = 10^{-2} \\ 9 \text{ failures with } h = 10^{-4} \\ 0 \text{ failures with } h \in \{10^{-6}, 10^{-8}\} \\ 1 \text{ failure with } h = 10^{-10} \\ 2 \text{ failures with } h = 10^{-12} \end{cases}$

  Due to the backtracking's inability to find a good step.

- $n = 10^4 \quad \Rightarrow \quad \begin{cases} 11 \text{ failures with } h = 10^{-2} \\ 7 \text{ failures with } h = 10^{-4} \\ 0 \text{ failures with } h \in \{10^{-6}, 10^{-10}, 10^{-12}\} \\ 1 \text{ failure with } h = 10^{-8} \end{cases}$

  Due to the backtracking failed to find a good step.

- $n = 10^5 \quad \Rightarrow \quad \begin{cases} 11 \text{ failures with } h \in \{10^{-2}, 10^{-4}\} \\ 10 \text{ failures with } h \in \{10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\} \end{cases}$

  The method reached $time\_limit = 1000s$.

Looking at Fig.(3.55), the number of iterations increase with dimension and decrease as the step-length h tends to zero. Only with $h = 10^{-2}$ the average number of iteration is much greater for each dimension, this is because with that step-length value the derivatives are less accurate. As illustrated in Fig.(3.56), for dimension $n = 10^5$, it is quite less than $1000s$, because in most cases the method reached the limit time imposed.
The average rate of convergence is greater as h decreases. For $h \in \{10^{-4}, 10^{-10}\}$ and $n = 10^5$ the results differ from the others; specifically, in the first case it is negative, this is due to the ill-conditioned structure of the hessian matrix.
As a result, the difference in norm between the true minimum and the one found by the method is particularly significant, around 300, in dimension 10^5, Fig.(3.58).

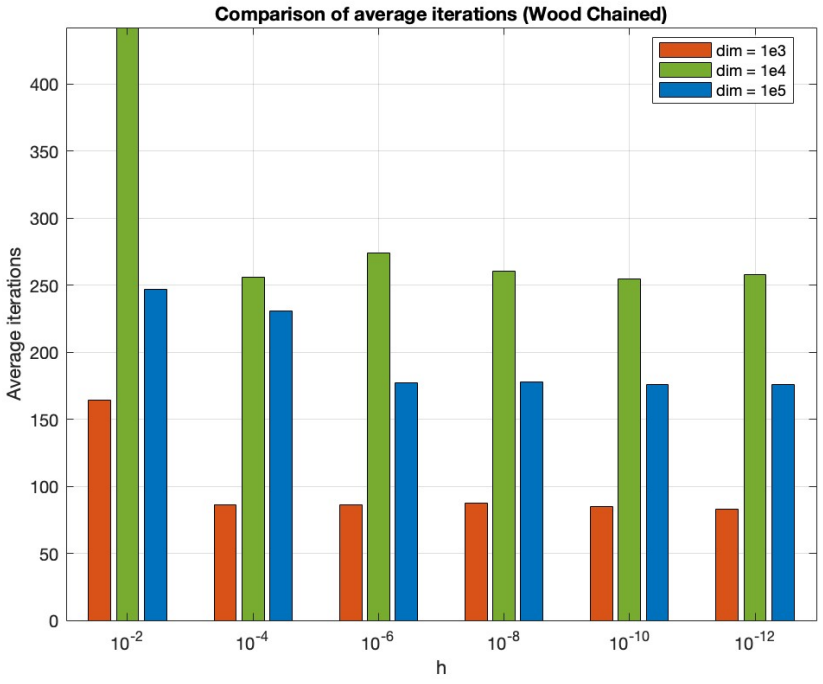

Figure 3.55: Average iterations needed to converge compared the three different dimensions.

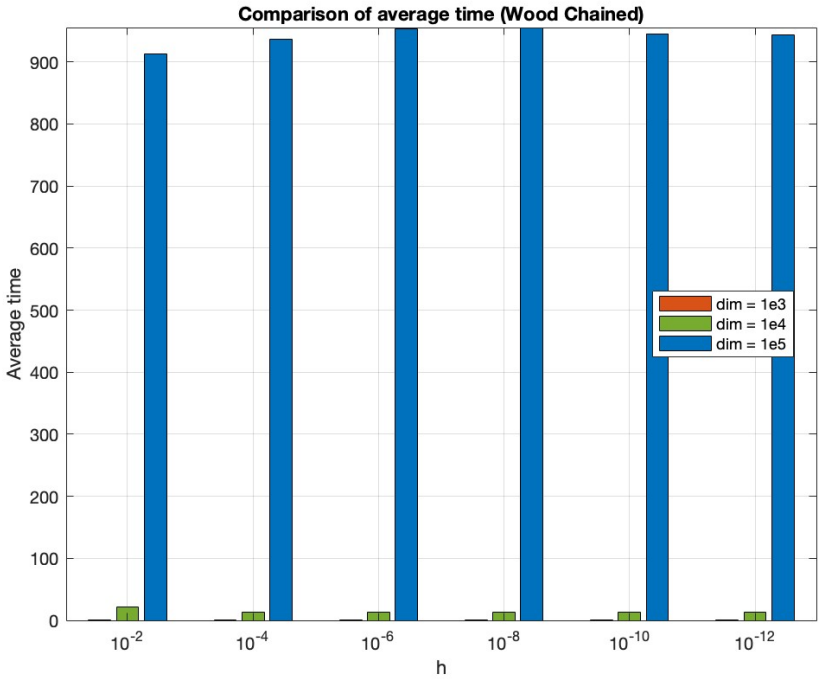

Figure 3.56: Average time needed to converge compared the three different dimensions.

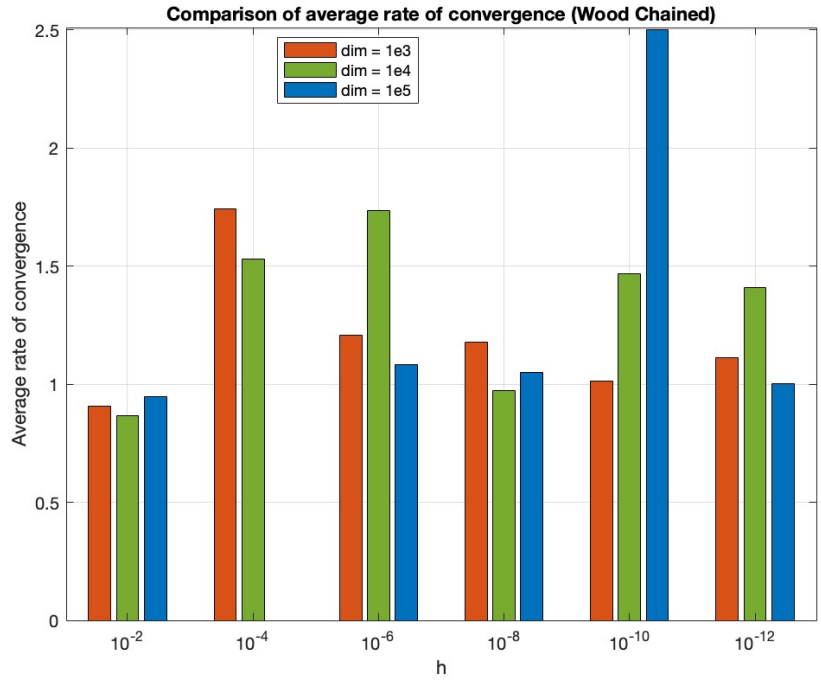

Figure 3.57: Average rate of convergence needed to converge compared the three different dimensions.

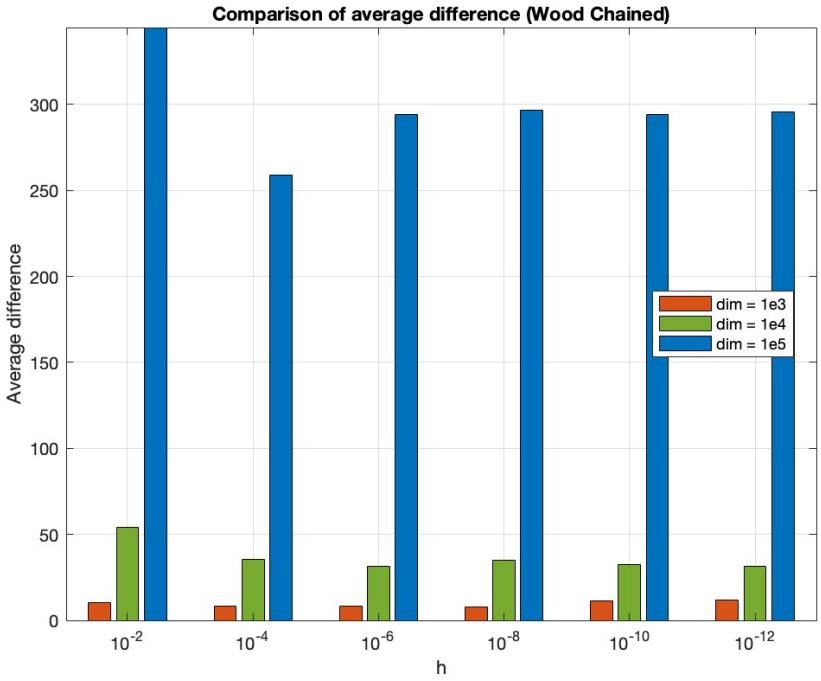

Figure 3.58: Average difference between last point found by the method and real minimizer compared the three different dimensions.

Variable h

The number of failures vary from $h_i = 10^{-k}|x_i|$, particularly:

- $n = 10^3 \quad \Rightarrow \quad \begin{cases} 0 \text{ failures with } k = 2, 4 \\ 1 \text{ failures with } k = 8, 10, 12 \\ 2 \text{ failures with } k = 6 \end{cases}$
  because the backtracking was not able to find a good step in the maximum number of iterations imposed, $btmax = 90$.

- $n = 10^4 \quad \Rightarrow \quad \begin{cases} 0 \text{ failures with } k = 4 \\ 2 \text{ failures with } k = 8, 10 \\ 3 \text{ failures with } k = 2, 6, 12 \end{cases}$
  due to the same reason for dimension $n = 10^3$.

- $n = 10^5 \quad \Rightarrow \quad \begin{cases} 10 \text{ failures with } k = 4, 6, 8, 10, 12 \\ 11 \text{ failures with } k = 2 \end{cases}$
  where the method stopped because it reached $time\_limit = 1000s$

In Fig.(3.59) can be observed the number of iterations needed, as dimension increases, it raises. On average, $h_i = 10^{-2}|x_i|$ requires more time and number of iterations in each dimension except for $n = 10^5$, because with that value we obtain the least accurate approximation.
The precision decreases with the increment of dimension, the average maximum difference between the real minimizer and the last point found is register for the biggest h_i, as shown in Fig.(3.62). Finally, the experimental rate of convergence differs with dimension and h value, Fig.(3.61).

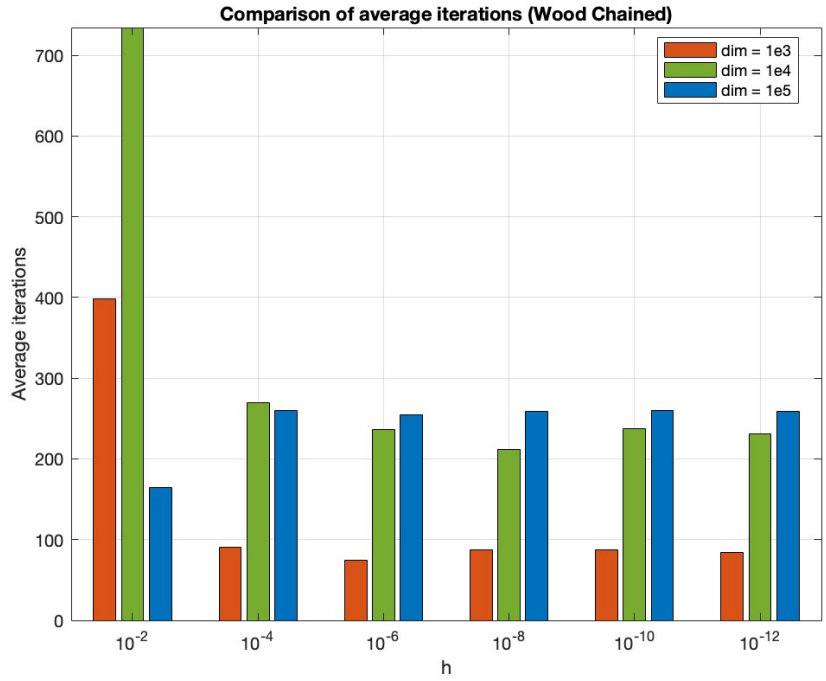

Figure 3.59: Average iterations needed to converge compared the three different dimensions.

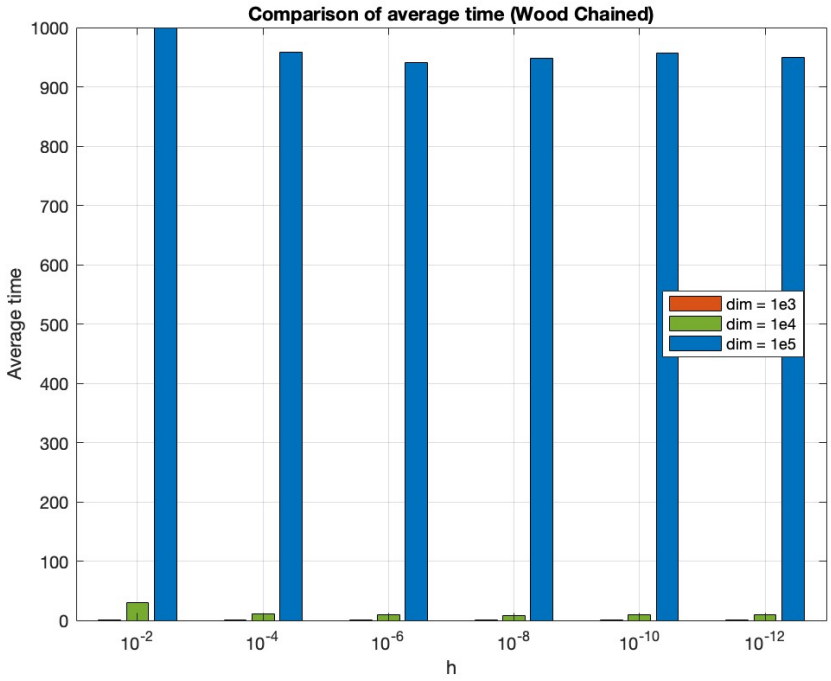

Figure 3.60: Average time needed to converge compared the three different dimensions.

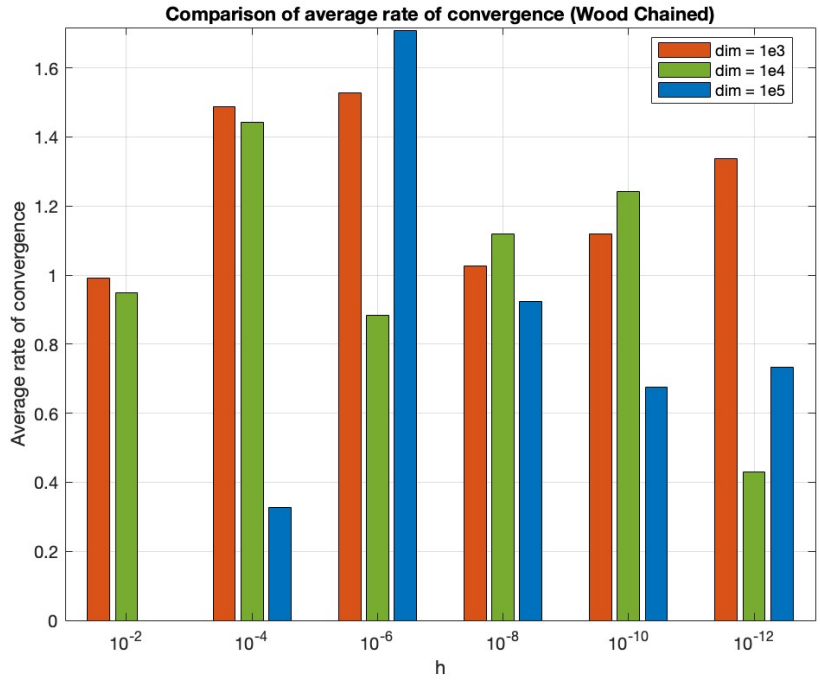

Figure 3.61: Average rate of convergence needed to converge compared the three different dimensions.

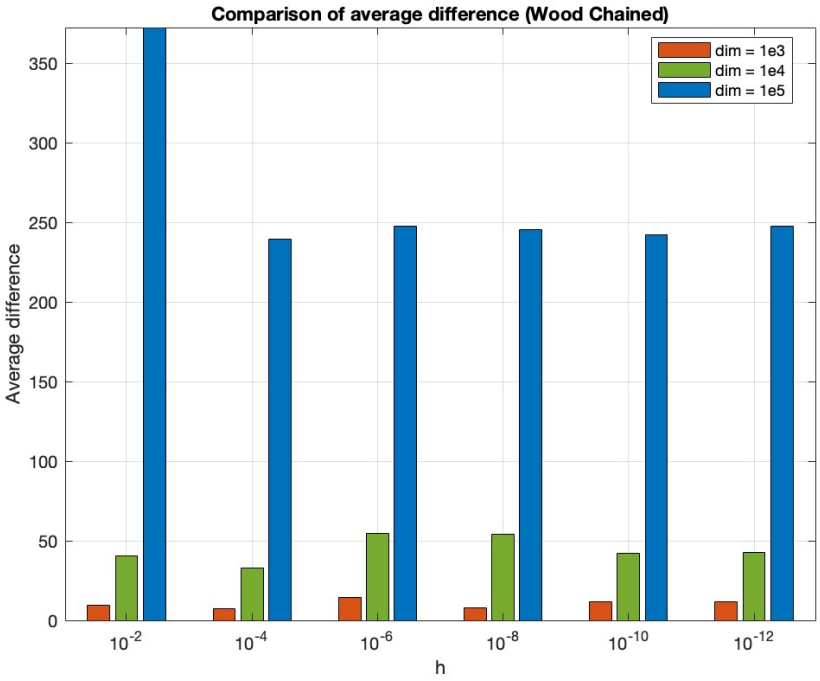

Figure 3.62: Average difference between last point found by the method and real minimizer compared the three different dimensions.

3.9 Comparison Nelder-Mead - Modified Newton method (exact)

Neither the Modified Newton Method nor the Nelder-Mead Method demonstrate effective performance when applied to the Chained Wood function. The Nelder-Mead Method frequently fails to converge within the specified maximum number of iterations. Conversely, while the Modified Newton Method converges, it does so at a significant distance from the optimum. Furthermore, the performance of both methods deteriorates with increasing dimensionality, yielding poor results at the highest dimensions tested.

3.10 Chained Powell function

3.10.1 Nelder-mead

Results for the Nelder Mead method with Chained Powell function.

Default tolerance for stopping criteria
Section ... for detail about stopping criteria.

1. first stopping criteria $\longrightarrow$ tol_simplex = 1e-07;
2. second stopping criteria $\longrightarrow$ tol_varf = 1e-07;
3. kmax = 10000

Tuned parameters
We found the best configuration of parameters for the suggested initial point. In the algorithm we form all possible configuration from values:

```
rho_vec = [0.25, 0.5, 1, 1.35, 1.75];
sigma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
gamma_vec = [0.1, 0.25, 0.5, 0.75, 0.9];
chi_vec = [1.1, 1.5, 2, 2.5, 3];
```

The best configuration of parameters we found is:

- for dimension = 10:

$$\rho = 1.30, \quad \sigma^2 = 0.1, \quad \chi = 2, \quad \gamma = 0.75$$

- for dimension = 25:

$$\rho = 1.35, \quad \sigma^2 = 0.1, \quad \chi = 1.1, \quad \gamma = 0.75$$

- for dimension = 50:

$$\rho = 1, \quad \sigma^2 = 0.1, \quad \chi = 1.1, \quad \gamma = 0.9$$

Before presenting the results, a few comments.
Tables 3.17, 3.18, 3.19 summarize the main outcomes of the algorithm when applied to the method using the 11 initial points (the first being the suggested one, and the others being random points). The first column shows the distance of the solution from the optimum. The third column indicates the reason why the method stopped, corresponding to the stopping criterion that was satisfied:

- flag = 2: The method reached the first stopping criterion.
- flag = 3: The method reached the second stopping criterion.
- flag = 1: The method reached the maximum number of iterations without satisfying either of the first two stopping criteria.

The last column presents the order of magnitude of computational time.

Results for dimension = 10:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	0.0261	584	2	0.1
$\mathbf{p}_2$	0.0115	665	2	0.1
$\mathbf{p}_3$	0.0172	701	2	0.01
$\mathbf{p}_4$	0.0136	598	2	0.01
$\mathbf{p}_5$	0.0286	549	2	0.01
$\mathbf{p}_6$	0.0123	551	2	0.01
$\mathbf{p}_7$	0.0105	712	2	0.01
$\mathbf{p}_8$	0.0299	574	2	0.01
$\mathbf{p}_9$	0.0098	667	2	0.01
$\mathbf{p}_{10}$	0.0111	555	2	0.01
$\mathbf{p}_{11}$	0.0084	676	2	0.01
sample average	0.0163	621.09	/	0.0304

Table 3.17: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point)

Last 5 exponential rate of convergence for each point:

Point 1: NaN NaN NaN NaN NaN
Point 2: NaN NaN NaN NaN 0
Point 3: 0.33 ∞ NaN NaN NaN
Point 4: 0 0.41 -1.78 ∞ NaN
Point 5: NaN NaN NaN NaN NaN
Point 6: NaN NaN NaN NaN NaN
Point 7: 0 0.34 1.02 -3.07 $-\infty$
Point 8: $-\infty$ NaN NaN NaN NaN
Point 9: 0 ∞ NaN NaN NaN
Point 10: NaN NaN NaN NaN 0
Point 11: NaN NaN NaN NaN NaN

Comments on results
All the errors are quite small, suggesting that the method successfully converged close to the optimal solution. The number of iterations is relatively low compared to previous tests, indicating that the method had a relatively fast convergence with a computational cost also relatively low compared to the 1 or 10 seconds observed for the Chained Wood function.

Results for dimension = 25:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	1	1307	2	0.1
$\mathbf{p}_2$	1	1335	2	0.1
$\mathbf{p}_3$	1	1362	2	0.1
$\mathbf{p}_4$	1	1315	2	0.1
$\mathbf{p}_5$	1	1327	2	0.1
$\mathbf{p}_6$	1	1342	2	0.1
$\mathbf{p}_7$	1	1348	2	0.1
$\mathbf{p}_8$	1	1325	2	0.1
$\mathbf{p}_9$	1	1325	2	0.1
$\mathbf{p}_{10}$	1	1347	2	0.1
$\mathbf{p}_{11}$	1	1354	2	0.1
sample average	1	1343	/	0.1

Table 3.18: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random point)

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	NaN	NaN	NaN	NaN	NaN
Point 5:	NaN	NaN	NaN	NaN	NaN
Point 6:	NaN	NaN	NaN	NaN	NaN
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	NaN	NaN	NaN	NaN	NaN
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results
The norm of the difference between the convergence point and the optimum is always equal to 1 for all 11 initial points. This suggests that the algorithm was unable to significantly reduce the error. This persistent distance from the minimizer might be influenced by the nature of the Chained Powell function, which is well-defined for even dimensions. Supporting this, when considering dimension 24, the average error was 0.01722, while at dimension 26, it increased to 0.0276: two values which are lower than average error in dimension 25. This behavior can be explained by the fact that, at odd dimensions, the optimization process fails to consider the last coordinate, resulting in a lack of information and a degraded performance.
The number of iterations varies between 1307 and 1362 with limited differences between the different initial points.

Results for dimension = 50:

Initial point	$\|\|\bar{x}_{conv} - x^*\|\|$	# iterations	flag	time (s)
$\mathbf{p}_1$	0.0331	10000	3	10
$\mathbf{p}_2$	0.0373	10000	3	10
$\mathbf{p}_3$	0.0388	10000	3	10
$\mathbf{p}_4$	0.0253	10000	3	10
$\mathbf{p}_5$	0.0280	10000	3	10
$\mathbf{p}_6$	0.0375	10000	3	10
$\mathbf{p}_7$	0.0537	10000	3	10
$\mathbf{p}_8$	0.0612	10000	3	10
$\mathbf{p}_9$	0.0495	10000	3	10
$\mathbf{p}_{10}$	0.0314	10000	3	10
$\mathbf{p}_{11}$	0.0341	10000	3	10
sample average	0.0391	10000	/	10

Table 3.19: Table summarizing the main results from the algorithm for the method applied with the 11 initial points (first one is the one suggested, the other one are the random points)

Last 5 exponential rate of convergence for each point:

Point 1:	NaN	NaN	NaN	NaN	NaN
Point 2:	NaN	NaN	NaN	NaN	NaN
Point 3:	NaN	NaN	NaN	NaN	NaN
Point 4:	NaN	NaN	NaN	NaN	NaN
Point 5:	NaN	NaN	NaN	NaN	NaN
Point 6:	NaN	NaN	NaN	NaN	NaN
Point 7:	NaN	NaN	NaN	NaN	NaN
Point 8:	NaN	NaN	NaN	NaN	NaN
Point 9:	NaN	NaN	NaN	NaN	NaN
Point 10:	NaN	NaN	NaN	NaN	NaN
Point 11:	NaN	NaN	NaN	NaN	NaN

Comments on results:
The norm of the difference between the convergence point and the optimum varies between 0.0253 and 0.0612, with an average of 0.0391 which is a good convergence considering the fact that the dimension is high.
The number of iterations is always 10000 for all 11 initial points, suggesting that the algorithm reached the maximum allowed number of iterations without meeting a convergence criterion, which leads to a high computational cost.

An overview:

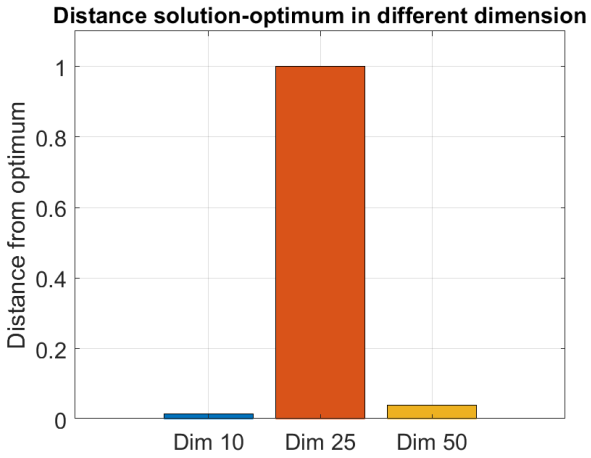

Figure 3.63: Powell: distance from optimum

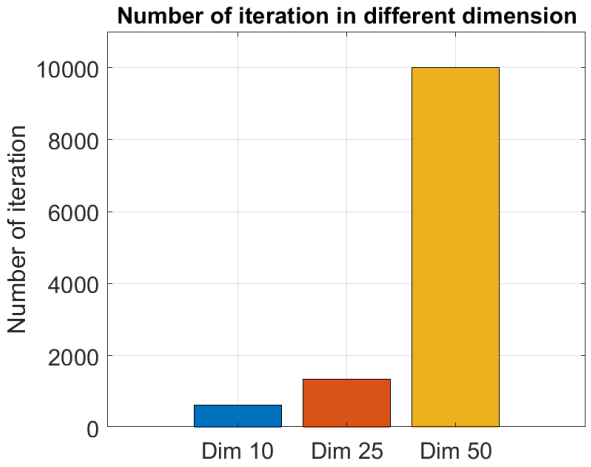

Figure 3.64: Powell: number of iterations

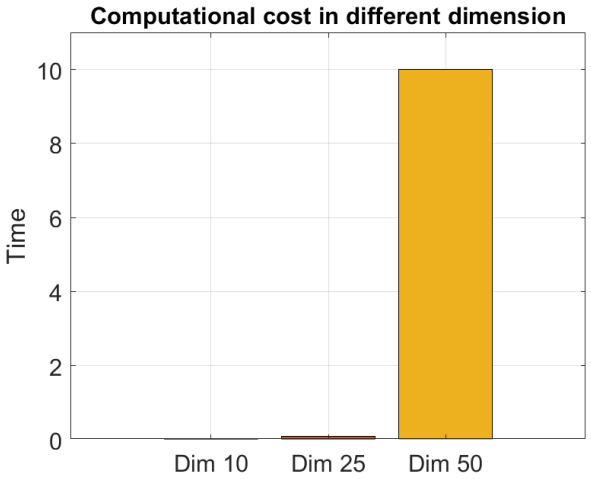

Figure 3.65: Powell: computational cost

3.10.2 Modified Newton method

Gradient and hessian matrix of the Powell function f.

$$\begin{aligned}
\frac{\partial f}{\partial x_1} &= 40(x_1^3 - 3x_1^2x_4 + 3x_1x_4^2 - x_4^3) + 2(x_1 + 10x_2) \\
\frac{\partial f}{\partial x_2} &= 20(x_1 + 10x_2) + 4(x_2^3 - 6x_2^2x_3 + 12x_2x_3^2 - 8x_3^3) \\
\frac{\partial f}{\partial x_i} &= 2(6x_i + 5x_{i-1}) + 8(x_{i-1}(6x_{i-1}x_i - x_{i-1}^2 - 12x_i^2) + \\
&\quad + 13x_i^3 - 5x_{i+3}(3x_i^2 + 3x_ix_{i+3} - x_{i+3}^2)) \quad \forall i = 3:2:n-3 \\
\frac{\partial f}{\partial x_i} &= 10(x_{i-1} + 21x_i - 4x_{i-3}^3 + 12x_{i-3}^2x_i - 12x_{i-3}x_i^2) + \\
&\quad + 4(11x_i^3 - 6x_i^2x_{i+1} + 12x_ix_{i+1}^2 - 8x_{i+1}^3) \quad \forall i = 4:2:n-2 \\
\frac{\partial f}{\partial x_{n-1}} &= 10(x_{n-1} - x_n) - 8(x_{n-2}^3 + 6x_{n-2^2}x_{n-1} - 12x_{n-2}x_{n-1}^2 + 8x_{n-1}^3) \\
\frac{\partial f}{\partial x_n} &= -10(x_{n-1} + x_n - 4x_{n-3}^3 + 12x_{n-3}^2x_n - 12x_{n-3}x_n^2 + 4x_n^3)
\end{aligned}$$

The hessian matrix has a particular diagonal structure.

$$H = \begin{bmatrix}
\boxed{D_1} & \boxed{U} & 0 & \boxed{S} & 0 & \cdots & 0 \\
\boxed{L} & \boxed{D_2} & \boxed{U} & 0 & \cdots & \cdots & 0 \\
0 & \boxed{L} & \boxed{D_3} & \boxed{U} & 0 & \cdots & \boxed{S} \\
\boxed{T} & 0 & \boxed{L} & \boxed{D_4} & \boxed{U} & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & \boxed{L} & \boxed{D_{n-1}} & \boxed{U} & 0 \\
0 & \cdots & \boxed{T} & 0 & \boxed{L} & \boxed{D_n} &
\end{bmatrix}$$

Main diagonal elements:

$$\begin{aligned}
D_1 &= 120(x_1 - x_4)^2 + 2 \\
D_2 &= 200 + 12(x_2 - 2x_3)^2 \\
D_i &= 12(1 + 4x_{i-1}^2 - 16x_{i-1}x_i + 26x_i^2 - 20x_ix_{i+3} + 10x_{i+3}^2) \quad \forall i = 3:2:n-3 \\
D_i &= 210 + 12(x_{i-3}^3 - 2x_{i-3}x_i + 11x_i^2 - 4x_ix_{i+1} + 4x_{i+1}^2) \quad \forall i = 4:2:n-2 \\
D_{n-1} &= 10 + 48(x_{n-2} - 2x_{n-1})^2 \\
D_n &= 10 + 120(x_{n-3} - x_n)^2
\end{aligned}$$

Diagonal in position +1

$$\begin{aligned}
U_2 &= 20 \\
U_3 &= -42(x_2 - x_3)^2 \\
U_{i+1} &= 10 \quad \forall i = 3:2:n-3 \\
U_{i+1} &= -24(x_i - x_{i+1})^2 \quad \forall i = 4:2:n-2 \\
U_n &= -10
\end{aligned}$$

Diagonal in position +3

$$\begin{aligned}
S_4 &= -120(x_1 - x_4)^2 \\
S_{i+3} &= -120(x_i - x_{i+3})^2 \quad \forall i = 3:2:n-3
\end{aligned}$$

L blocks have the same elements of U blocks and T block the same of S blocks.

Finite differences:

$$
\begin{aligned}
\frac{\partial f}{\partial x_1} &= 40(x_1^3 - 3x_1^2x_4 + 3x_1x_4^2 - x_4^3 + h_1^2x_1) + 2(x_1 + 10x_2) \\
\frac{\partial f}{\partial x_2} &= 20(x_1 + 10x_2) + 4(x_2^3 - 6x_2^2x_3 + 12x_2x_3^2 - 8x_3^3 - 2h_2^2x_3) \\
\frac{\partial f}{\partial x_i} &= 2(6x_i + 5x_{i-1}) + 8(x_{i-1}(6x_{i-1}x_i - x_{i-1}^2 - 12x_i^2) + \\
&\quad + 13x_i^3 - 5x_{i+3}(3x_i^2 + 3x_ix_{i+3} - x_{i+3}^2) + h_i^2(16x_i - 5h_i^2x_{i+2})) \quad \forall i = 3:2:n-3 \\
\frac{\partial f}{\partial x_i} &= 10(x_{i-1} + 21x_i - 4x_{i-3}^3 + 12x_{i-3}^2x_i - 12x_{i-3}x_i^2) + \\
&\quad + 4(11x_i^3 - 6x_i^2x_{i+1} + 12x_ix_{i+1}^2 - 8x_{i+1}^3 - 2h_i^2(5x_{i-3} + 5x_i - 2x_{i+1})) \quad \forall i = 4:2:n-2 \\
\frac{\partial f}{\partial x_{n-1}} &= 10(x_{n-1} - x_n) - 8(x_{n-2}^3 + 6x_{n-2^2}x_{n-1} - 12x_{n-2}x_{n-1}^2 + 8x_{n-1}^3 + 8h_{n-1}^2x_{n-1}) \\
\frac{\partial f}{\partial x_n} &= -10(x_{n-1} + x_n - 4x_{n-3}^3 + 12x_{n-3}^2x_n - 12x_{n-3}x_n^2 + 4x_n^3 + 4h_n^2(x_{n-3} + x_n))
\end{aligned}
$$

As for exact derivatives we exploit the symmetry of the matrix computing only upper diagonals. Main diagonal elements:

$$
\begin{aligned}
D_1 &= 120(x_1 - x_4)^2 + 2 + 20h_1^2 \\
D_2 &= 200 + 12(x_2 - 2x_3)^2 + 2h_2^2 \\
D_i &= 12(1 + 4x_{i-1}^2 - 16x_{i-1}x_i + 26x_i^2 - 20x_ix_{i+3} + 10x_{i+3}^2) + 22h_i^2 \quad \forall i = 3:2:n-3 \\
D_i &= 210 + 12(x_{i-3}^3 - 2x_{i-3}x_i + 11x_i^2 - 4x_ix_{i+1} + 4x_{i+1}^2) + 22h_i^2 \quad \forall i = 4:2:n-2 \\
D_{n-1} &= 10 + 48(x_{n-2} - 2x_{n-1})^2 + 2h_{n-1}^2 \\
D_n &= 10 + 120(x_{n-3} - x_n)^2 + 20h_n^2
\end{aligned}
$$

Diagonal in position +1

$$
\begin{aligned}
U_2 &= 20 \\
U_3 &= -24(x_1 - 2x_4)^2 - 16h_3^2 - 48h_3x_3 + 24h_2x_2 \\
U_{i+1} &= 10 \quad \forall i = 3:2:n-3 \\
U_{i+1} &= -24(x_i - x_{i+1})^2 - 16h_{i+1}^2 - 48h_{i+1}x_{i+1} + 24h_ix_i \quad \forall i = 4:2:n-2 \\
U_{n-1} &= -24(x_{n-2} - 2x_{n-1})^2 - 8h_{n-2}^2 - 32h_{n-1}^2 + 24h_{n-2}h_{n-1} + 24(2h_{n-1} - h_{n-2})(x_{n-2} - 2x_{n-1}) \\
U_n &= -10
\end{aligned}
$$

Diagonal in position +3

$$
\begin{aligned}
S_4 &= -120(x_1 - x_4)^2 - 40h_1^2 - 40h_4^2 + 60h_1h_4 + 120(x_1 - x_4)(h_4 - h_1) \\
S_{i+3} &= -120(x_i - x_{i+3})^2 - 40(h_i^2 - h_{i+3}^2) + 60h_ih_{i+3} + 120(x_i - x_{i+3})(h_{i+3} - h_i) \quad \forall i = 3:2:n-3
\end{aligned}
$$

We use the same finite differences shown before for constant and variable h, specifically when h is equal to a fixed value for each x component, many term can be sum and expressions become more compact.
We have also implemented hessian matrix as the Jacobian of the exact gradient when applying

the step-length h adapted to each component of the input vector x, computing it doing the least number of function evaluation as possible using the particular structure of the matrix shown in Fig.(3.66) in order to minimize the computational cost.

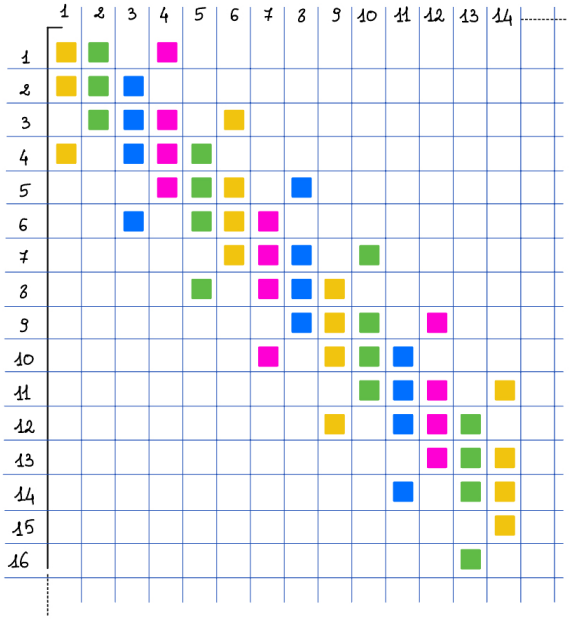

Figure 3.66: Hessian matrix of Powell function underling elements in the same column.

By analyzing the derivatives of the function, we can deduce that all gradient's components are nullified by $x = (0, \ldots, 0)$. This leads to conclusion that Powell function assumes its global minimum in that point.
To better understand the behavior of the function $f(x)$ near the minimizer we report a plot of the function in 2D for $n = 2$.

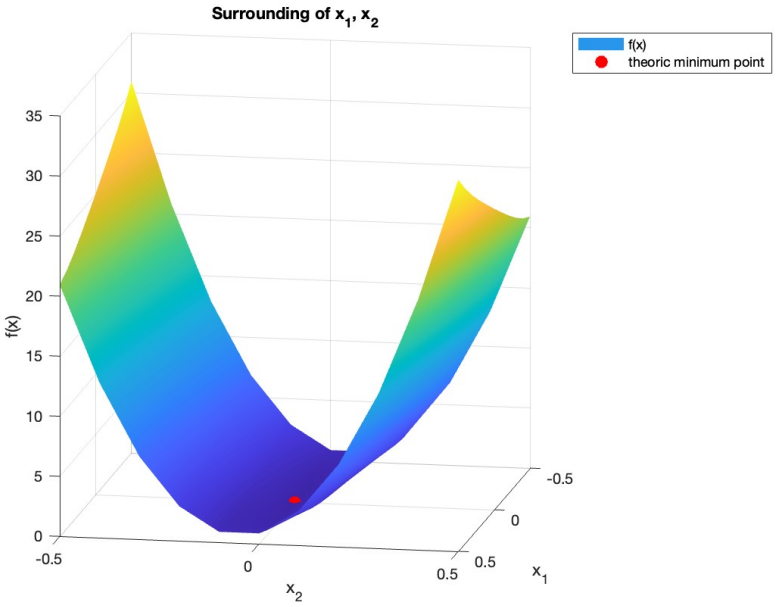

Figure 3.67: 2D plot of the function $f(x_1, x_2)$

From the plot we can deduce that the function is well-behaved near the minimizer, this can lead to fast convergence when applying an optimizer solver.

Results for modified Newton method with chained Powell function.

In this section are reported solutions from the application of Modified Newton Method to the problem considering exact derivatives. We store the Hessian matrix in a sparse and diagonal structure due to the high dimension of the input vector $\{10^3, 10^4, 10^5\}$ memorizing only the upper diagonals of the matrix and defining the lower diagonals using the symmetry of the matrix. This implementation significantly reduces computational time in dimension 10^5: the Hessian is now computed in approximately 50% of the time required by the original version. For each dimension we run the method starting from 11 different points, where the first is fixed and the other 10 are a perturbation of it.
For each dimension we have used the same set of parameters:

$$tol\_grad = 10^{-3} \quad kmax = 200 \quad \rho = 0.9 \quad c = 10^{-4} \quad btmax = 100$$

Next are reported some bar plots that compare the results obtained by modified newton method applied to Powell function in dimensions $n = 10^3, 10^4, 10^5$ in terms of average time, average number of iterations, average difference between minimum point and minimum found and average rate of convergence.

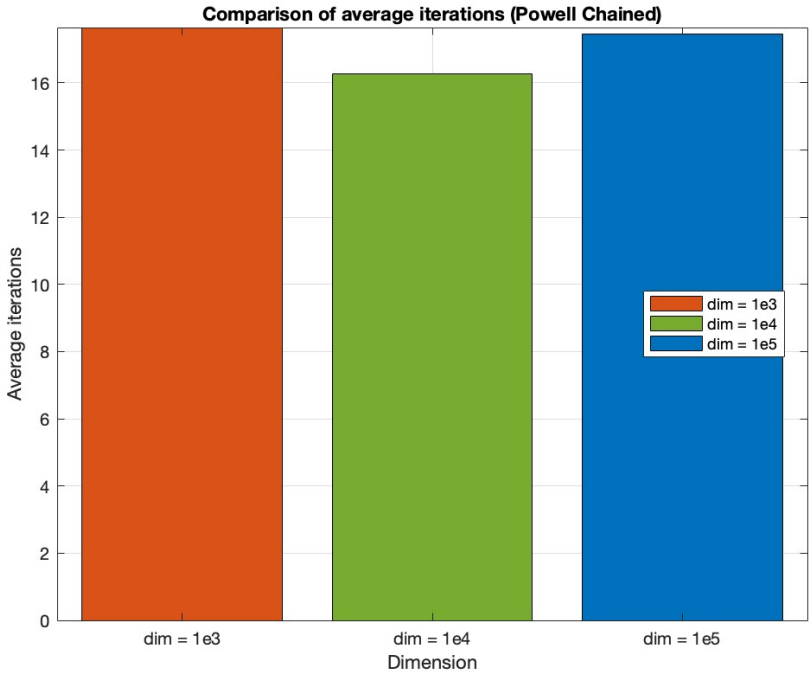

Figure 3.68: Average iterations needed to converge compared the three different dimensions.

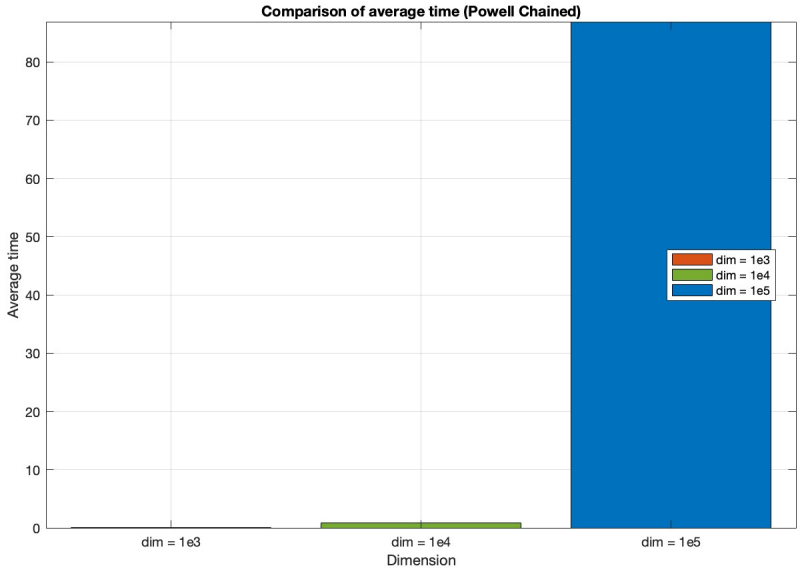

Figure 3.69: Average time needed to converge compared the three different dimensions.

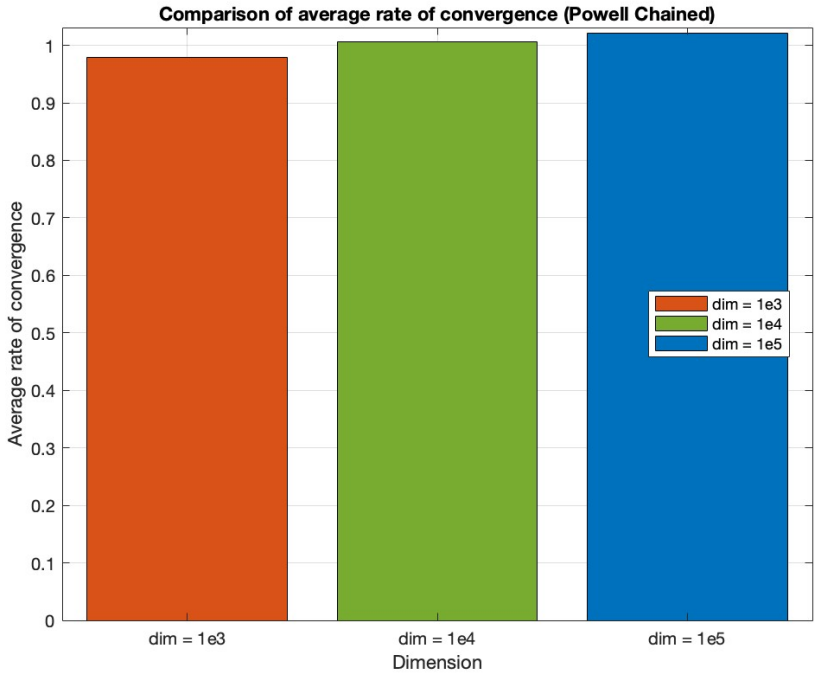

Figure 3.70: Average rate of convergence needed to converge compared the three different dimensions.

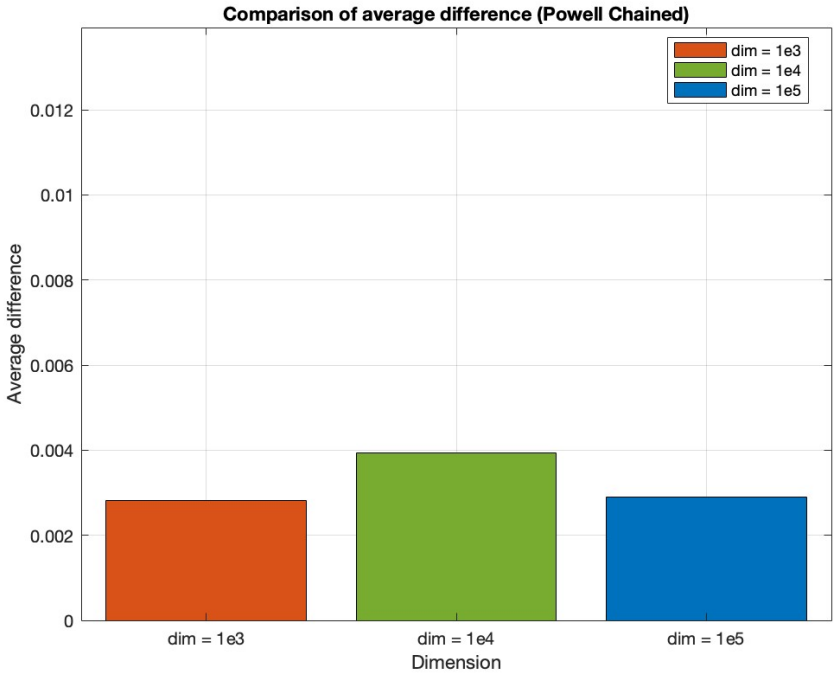

Figure 3.71: Average difference between last point found by the method and real minimizer compared the three different dimensions.

What can be deduced from the bar plots is that the method converges to a point whose norm differs from the minimizer starting from the third decimal digit. The number of iterations is quite similar for all dimension, approximately 17. The average time differs looking at dimension $n = 10^5$, where are needed on average $85s$. The experimental rate of convergence is nearly 1 for all dimensions, this is due to the small number of iterations required to converge; specifically, the experimental rate of convergence approximates the order of convergence when the number of iterations are quite large. In addition, the number of failures registered is 0.

For example, looking at the behavior of the function evaluations in intermediate points found by the method, we have observed that, due to backtracking, the method registers a fast decrease in the first iterations and then less progress in the last iterations until the optimum. In Fig.(3.72) is shown the behavior of the function evaluations setting parameters as before.

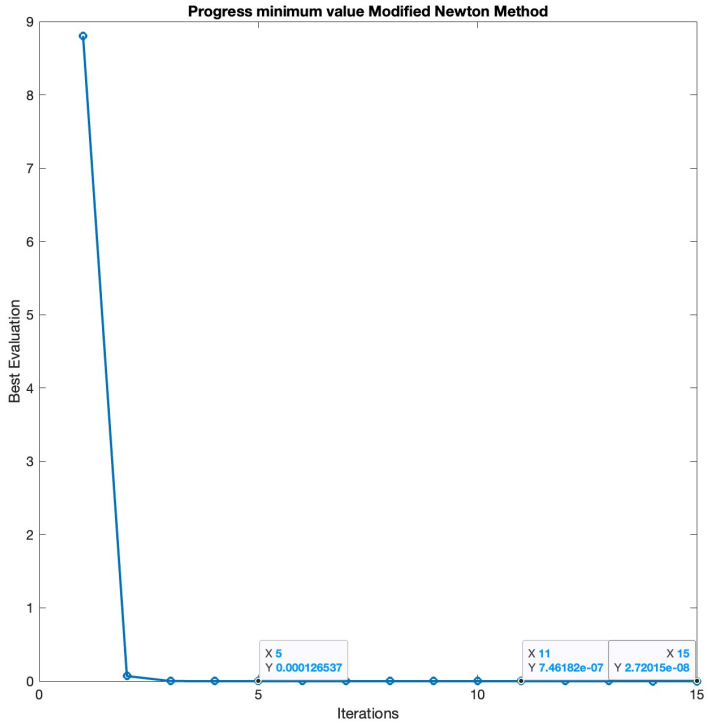

Figure 3.72: Function evaluation in the sequence of point found by the method until convergence with $\rho = 0.9$ starting from the first point suggested.

Tuning parameters

We have not observed differences changing backtracking parameters neither in number of iterations nor in speed of convergence, this is due to the peculiar structure of Powell function, it is significant not flat near the minimum and this fast the convergence.

Finite differences

In this section are reported results from the method applied to the problem with derivatives computed with finite differences.
We set the following parameters

$$tol\_grad = 10^{-2} \quad kmax = 200 \quad \rho = 0.9 \quad c = 10^{-4} \quad btmax = 50$$

Constant h

The method converges without failures in a few iterations, on average 12 for dimension 10^3 and 10^5 and one less for dimension 10^4 for all h values, as shown in Fig.(3.68). Looking at Fig.(3.74), as registered with exact derivatives, the method needs much more time when applied in dimension 10^5, nearly $60s$.
The average rate of convergence is above 1 for each h value and dimension $10^3, 10^5$, a little bit slow for dimension 10^4, this is due to the small number of iterations.
Finally the last point found differs in norm from the real minimizer from the second decimal digit.

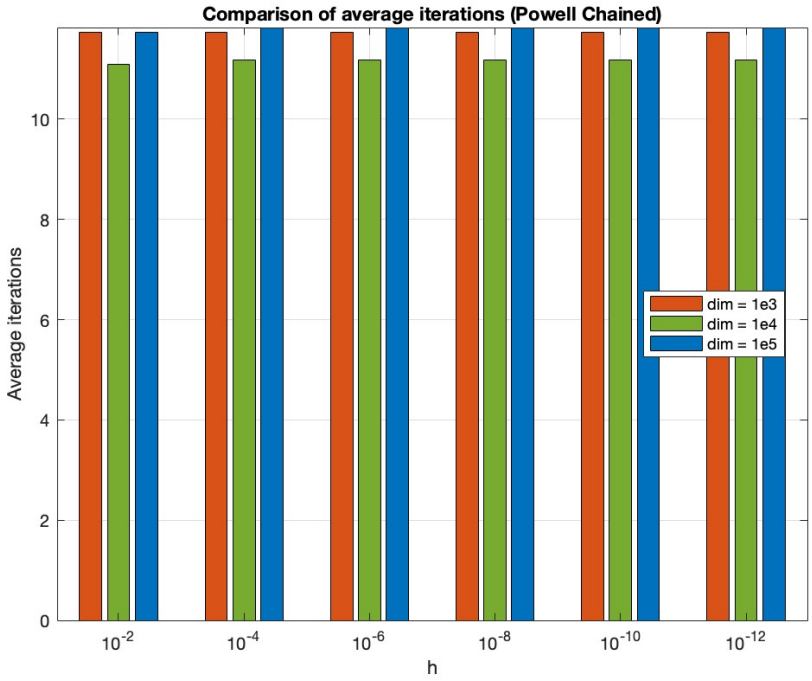

Figure 3.73: Average iterations needed to converge compared the three different dimensions varying the step-length h.

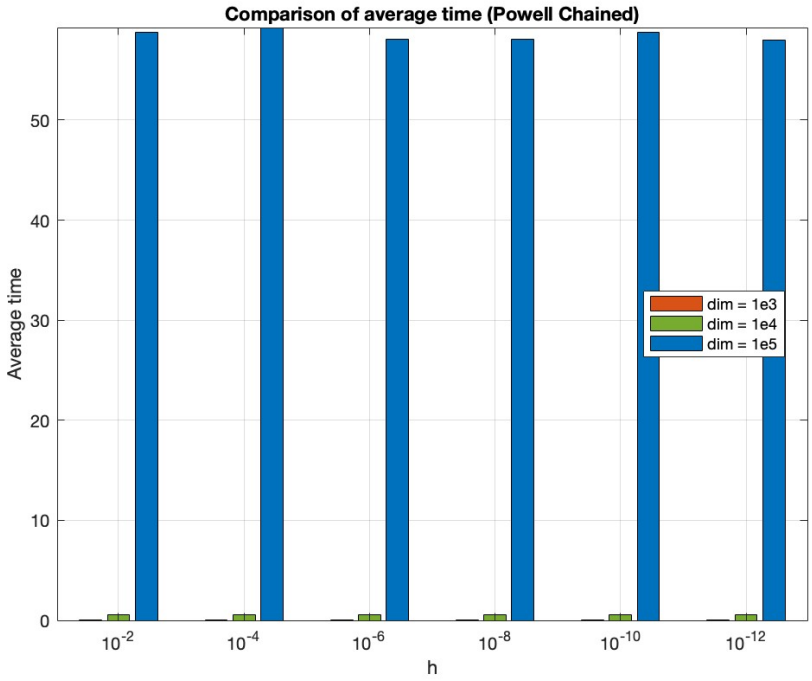

Figure 3.74: Average time needed to converge compared the three different dimensions varying the step-length h.

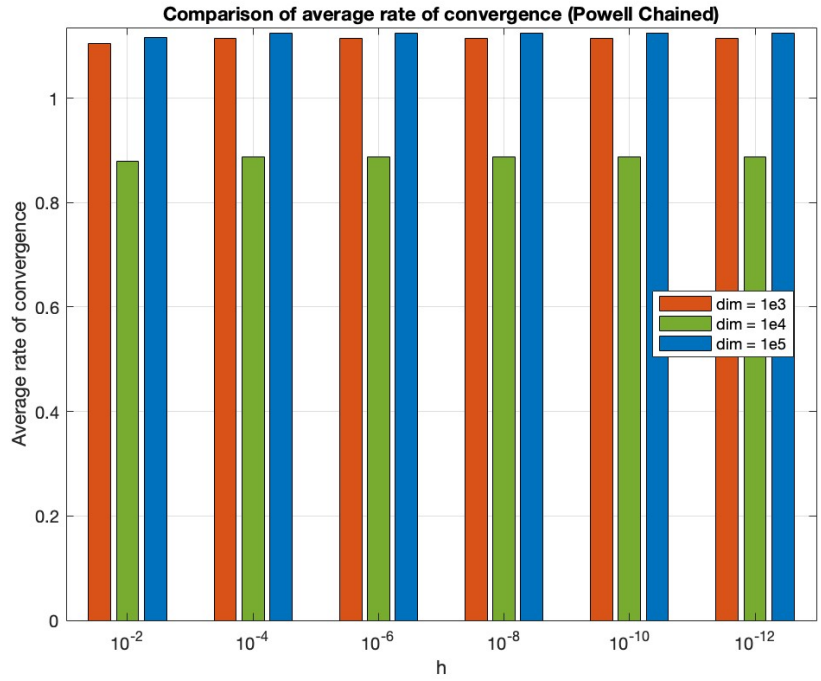

Figure 3.75: Average rate of convergence needed to converge compared the three different dimensions varying the step-length h.

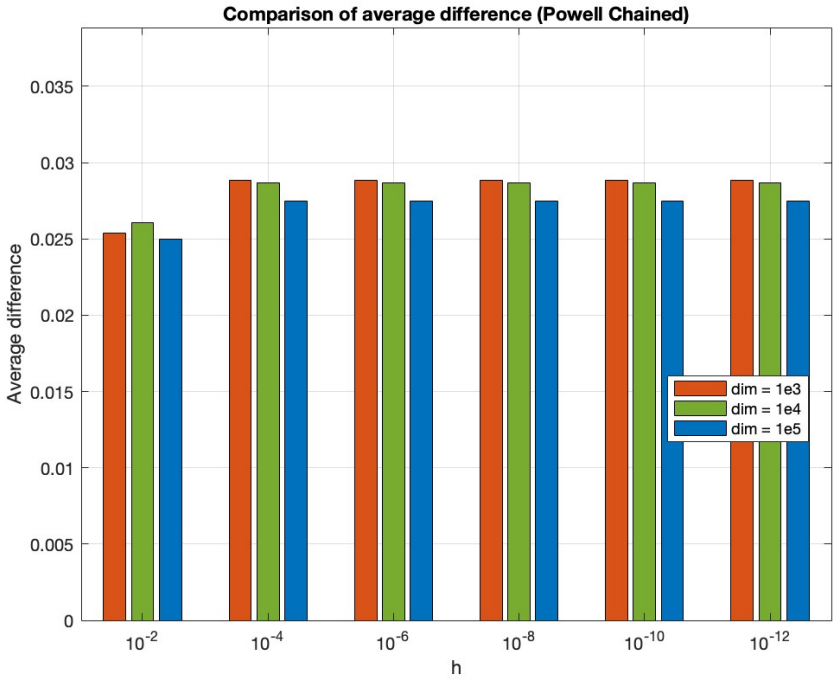

Figure 3.76: Average difference between last point found by the method and real minimizer compared the three different dimensions varying the step-length h.

As in the case of exact derivatives we have not observed improvement changing backtracking

parameters. In Fig.(3.77) and Fig.(3.78) are presented the best function evaluations in the sequence of points found by the method fixing all backtracking parameters changing only ρ starting from the first suggested point in dimension 10^5 and with $h = 10^{-2}$. There are no significant differences neither in points nor in number of iterations, which are still 11.

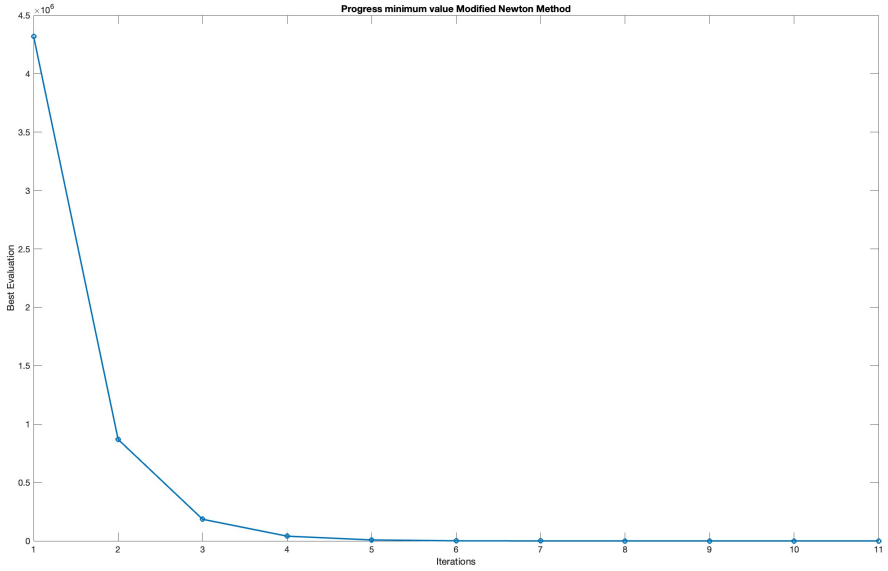

Figure 3.77: Best function evaluations in the sequence obtained by the method with $\rho = 0.9$ and other parameters as before.

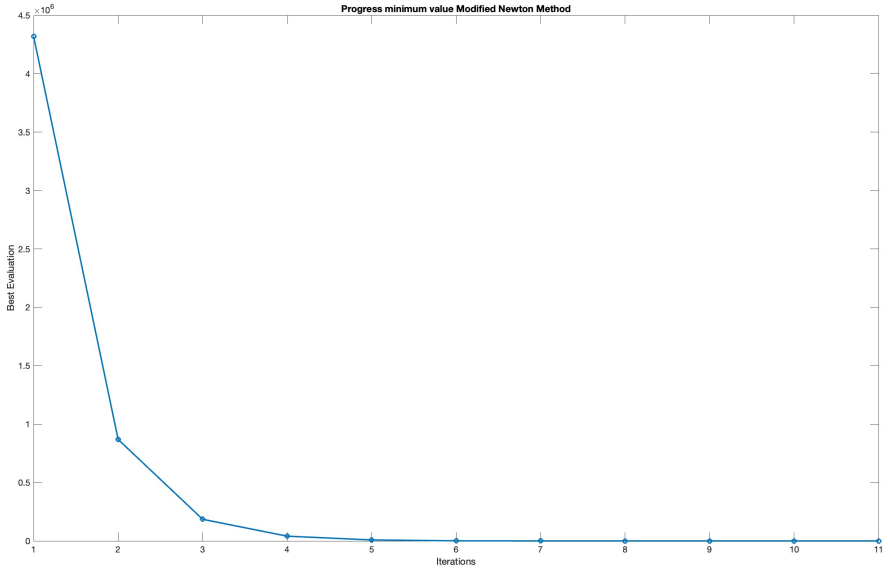

Figure 3.78: Best function evaluations in the sequence obtained by the method with $\rho = 0.3$ and other parameters as before.

Variable h

In this section are reported results obtained with finite differences and a step-length h adapted to each component of the input vector x.

During the test we have observed that hessian matrix computed as the Jacobian of the exact gradient needed too much time to be computed (nearly $9s$ in dimension 10^5) so we decided to use centered finite differences implemented exploiting function evaluation in each point in order to reduce numerical cancellation as much as possible, in dimension 10^5 this required nearly $3s$ to be evaluated.

The average number of iterations required to converge is the same as in the previous test with finite differences computed with constant h, Fig.(3.79).

Looking at Fig.(3.80) for each h value the dimension 10^5 is the slowest, approximately $60s$, and it can be seen that $h = 10^{-6}$ it is the fast above all.

The experimental rate of convergence shown in Fig.(3.81) differs between h value, specifically in dimension 10^4, it is approximately 1 for $h_i = 10^{-2}|x_i|$ and nearly 0.9 for all the others; it is quite bigger for dimension 10^3 and 10^5, this is due to the greater average number of iterations.

Finally, the average difference between real minimizer and the last point found by the method is in norm less than 0.03, specifically it decreases as dimension increases for each h_i value except for $h_i = 10^{-2}|x_i|$, where in dimension 10^3 the last point is more accurate, Fig.(3.82).

Additionally, we have not registered any failures.

What can be noted comparing results obtained with finite differences and constant and variable h is that results from the second test are more accurate for $h_i = 10^{-2}|x_i|$ in terms of average distance from real minimizer and average rate of convergence in dimension $n = 10^4$.

Comparing results obtained with exact and approximated derivatives can be seen that halving the maximum number of backtracking iterations $btmax$ has a significant influence on both number of iterations and time required to converge. Looking at dimension 10^5, while in the first case the method needed on average 16 iterations and $90s$, when using finite differences with $btmax = 50$ it required nearly 11 iterations and $60s$, approximately two-thirds of the time required with exact derivatives and $btamx = 100$; on the other hand, this decrease has not effected hugely the other results.

During the phase of tuning we have not observed specific benefits except for the decrease of $btmax$.

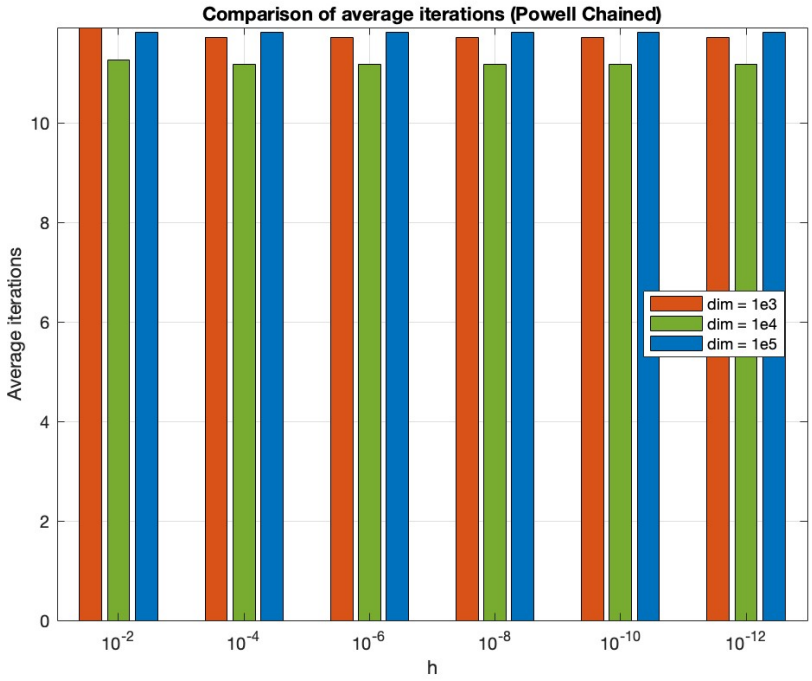

Figure 3.79: Average iterations needed to converge compared the three different dimensions.

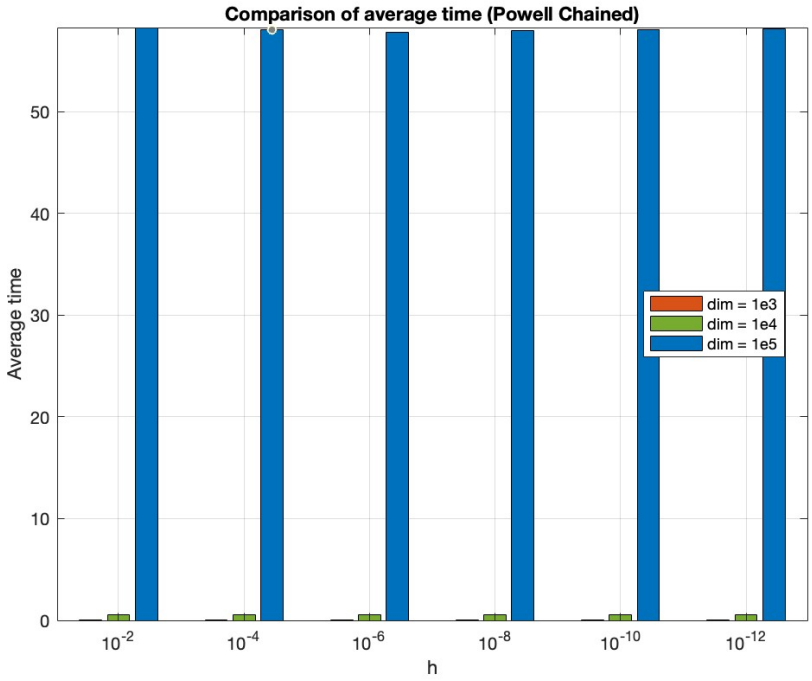

Figure 3.80: Average time needed to converge compared the three different dimensions.

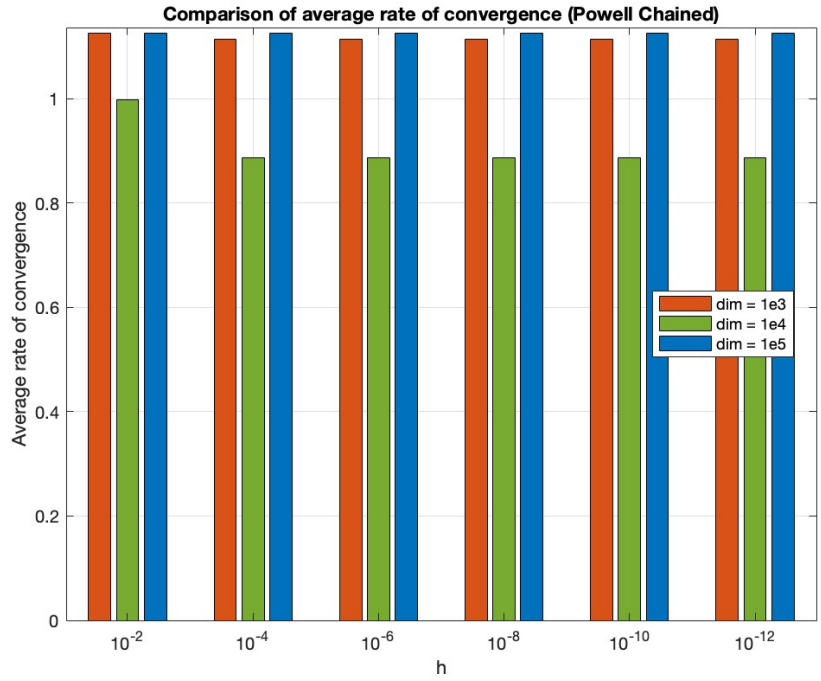

Figure 3.81: Average rate of convergence needed to converge compared the three different dimensions.

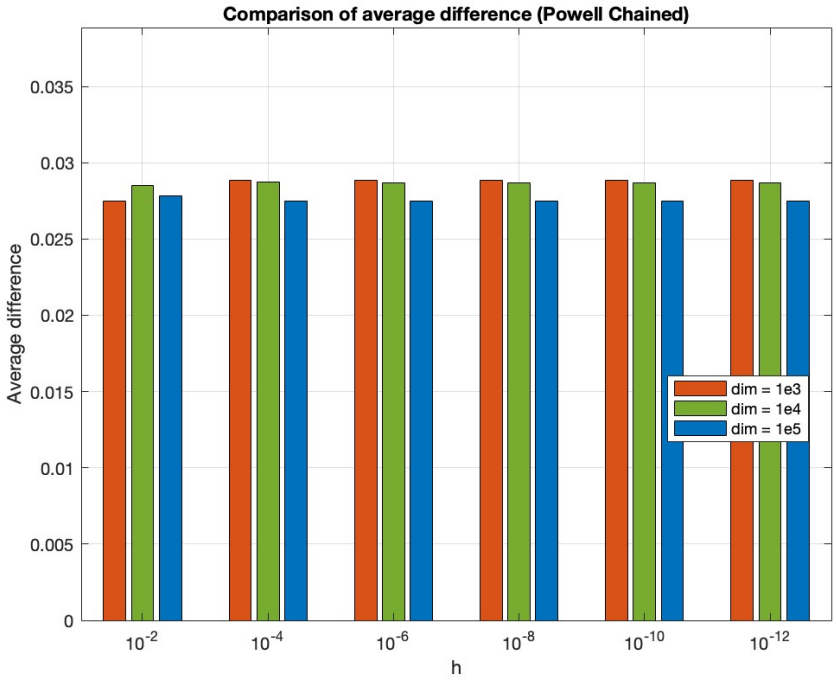

Figure 3.82: Average difference between last point found by the method and real minimizer compared the three different dimensions.

3.11 Comparison Nelder-Mead - Modified Newton method (exact)

The final point of iteration of the Nelder-Mead method provides a relatively good approximation of the minimizer in dimensions 10 and 50, but not in dimension 25. In contrast, the convergence points obtained through the modified Newton's method are significantly better, also considering the higher dimension, exhibiting a mean distance of approximately 0.003.
In terms of the number of iterations required, the modified Newton's method outperforms the Nelder-Mead method, using only on average about 0.005 percent of the iterations needed by the latter. However, the Newton's method tends to take longer to complete. This is an expected outcome, as each iteration of the modified Newton's method is considerably more computationally expensive due to the larger dimensionality of the simulation.

Code for Nelder-Mead Method

Below are the main scripts written in MATLAB for the Nelder-Mead method.
The full code can be found on GitHub:
https://github.com/cristina210/Assignement-numerical-optimization

A.1 Main scripts for nelder-mead method

A.1.1 *nelder_mead*

This function implements the Nelder-Mead method to minimize a given function.

Inputs:

- **f**: Function to minimize.
- **simplex**: Initial simplex (rows = points, columns = coordinates).
- **kmax**: Maximum number of iterations.
- **rho, chi, gamma, sigma**: Coefficients for reflection, expansion, contraction, and shrinkage.
- **n**: Dimension.
- **tol1**: Tolerance for the stopping criterion based on the size of the simplex, indicating convergence to a single point.
- **tol2**: Tolerance for the stopping criterion based on the function value, triggered when the simplex reaches a stationary region for f.

Outputs:

- **k**: Number of iterations performed.
- **simplex**: Final simplex.
- **x_vec**: Best point of the simplex during iterations.
- **flag**: Status flag indicating the output of the algorithm:
  - 1: Maximum number of iterations reached.
  - 2: Stopping criterion based on **tol1** is met.
  - 3: Stopping criterion based on **tol2** is met.

```
function [k, simplex, x_vec, flag, size_vec] = nelder_mead(f, simplex,
   kmax, rho, chi, gamma, n, sigma, tol1, tol2)
flag = 0;
tol3 = 10^(-8);

% Function handle for updating certain quantities
upDate_quantities = @(k, simplex, x_bar, f1, fend) deal( ...
    k+1, ...
    max(vecnorm(simplex - x_bar, Inf, 2)), ...
    abs(fend - f1));

% initialization
history = [];

f_val = zeros(n+1, n+1);
for i=1:n+1
    f_val(i,:) = [f(simplex(i,:)), simplex(i,:)];
end
% f_val is a matrix of size (n+1, n+1) that contains the function values
% and the coordinates of the points in the simplex during the execution
% of the Nelder-Mead algorithm. The first column of f_val contains the
   function
% values f computed at the points of the simplex.
% Each row represents a point in the simplex,
% so f_val(i,1) is the function value evaluated at the i-th point.

x_bar = mean(simplex(1:n, :));

x_vec = simplex(1,:);
% x_vec contains best point of the simplex through iterations

distance_bar = max(vecnorm(simplex - x_bar, Inf, 2));
% distance_bar represents the maximum distance between the barycenter (
   x_bar)
% and the points of the simplex.
% distance_bar is used as a convergence criterion: when it becomes
   smaller than
% a predefined tolerance (tol1) the method stops.

size_vec = [distance_bar];
% size_vec contains the maximum distance between a vertice of the
   symplex
% anche the barycenter rapresenting the dimension of the simplex

delta_f = 1;
% delta_f represents the absolute difference between the function values
% at the first and last points of the simplex.
% delta_f is used as a convergence criterion: when it becomes smaller
   than
% a predefined tolerance (tol2) the method stops.

k = 0;

while k < kmax && distance_bar > tol1 && delta_f > tol2
    % disp("simplex:")
    % disp(simplex)
```

```
% disp("best point:")
% disp(simplex(1,:))
% A = simplex(2:end, :) - simplex(1, :);
% det_A = det(A'*A);
% disp("Determinant of matrix of difference: " + det_A);
if k ~= 1
    x_vec = [x_vec; simplex(1,:)];
end
f_val = sortrows(f_val);
% f_val is sorted based on the function values to ensure
% that the best point is always at the top of the simplex.
simplex = f_val(:,2:end);
x_bar = mean(simplex(1:n, :));

% reflection
x_r = x_bar + rho*(x_bar - simplex(n+1,:));
f1 = f_val(1,1);
fr = f(x_r);
fn = f_val(n, 1);
fend = f_val(n+1, 1);
if (f1 < fr || abs(f1-fr) <= tol3) && ( fr < fn || abs(fr-fn) < tol3
   )
    simplex(end,:) = x_r;
    f_val(end,:) = [fr, x_r];
    [k, distance_bar, delta_f] = upDate_quantities(k, simplex, x_bar
       , f1, fend);
    size_vec = [size_vec; distance_bar];
    continue
elseif fr < f1
    % expansion
    x_e = x_bar + chi*(x_r - x_bar);
    fe = f(x_e);
    history = [history; "exp"];
    if fe < fr
        simplex(end, :) = x_e;
        f_val(end,:) = [fe, x_e];
        [k, distance_bar, delta_f] = upDate_quantities(k, simplex,
           x_bar, f1, fend);
        size_vec = [size_vec; distance_bar];
        continue
    else
        simplex(end,:) = x_r;
        f_val(end,:) = [fr, x_r];
        [k, distance_bar, delta_f] = upDate_quantities(k, simplex,
           x_bar, f1, fend);
        size_vec = [size_vec; distance_bar];
        continue
    end
elseif (fr > fn || abs(fn-fr) <= tol3)
    % contraction
    if fend < fr
        x_c = x_bar - gamma*(x_bar - simplex(end,:));
    else
        x_c = x_bar - gamma*(x_bar - x_r);
    end
    fc = f(x_c);
```

```
        history = [history; "cont"];
        if fc < fend
            simplex(end, :) = x_c;
            f_val(end,:) = [fc, x_c];
            [k, distance_bar, delta_f] = upDate_quantities(k, simplex,
               x_bar, f1, fend);
            size_vec = [size_vec; distance_bar];
            continue
        else
            % shrinking
            for i=2:n+1
                simplex(i,:) = simplex(1,:) + sigma*(simplex(i,:) -
                   simplex(1,:));
                f_val(i,:) = [f(simplex(i,:)), simplex(i,:)];
            end
            history = [history; "shrink"];
            [k, distance_bar, delta_f] = upDate_quantities(k, simplex,
               x_bar, f1, fend);
            size_vec = [size_vec; distance_bar];
            continue
        end
    end
end
count_shrinks = sum(history == "shrink");
count_contr = sum(history == "cont");
count_exp = sum(history == "exp");
disp("How many shrink:")
disp(count_shrinks)
disp("How many expansion:")
disp(count_exp)
disp("How many contraction:")
disp(count_contr)
if k == 1
   warning('The␣initial␣simplex␣provided␣as␣input␣is␣not␣adequate␣to␣
      explore␣the␣surrounding␣space:␣the␣function␣stops␣at␣the␣first␣
      iteration')
end
if k == kmax
   warning('Maximum␣iteration␣limit␣reached')
   flag = 3;  % no convergence
end
if distance_bar <= tol1
   %disp('The simplex is converging to a point')
   flag = 1; % convergence of type 1
end
if delta_f <= tol2
   %disp('The simplex has reached a stationary region for f')
   flag = 2; % convergence of type 2
end

end
```

A.1.2 *NelderMead_simplex*

This function constructs an initial simplex for the Nelder-Mead method in a specified dimension. It verifies that the simplex is non-degenerate by checking the linear independence of its vertices.

Inputs:

- **dim**: Dimension.
- **initial_point**: Coordinates of a point for the simplex.

Outputs:

- **simplex_initial**: Matrix containing the vertices of the initial simplex.
- **flag**: Status flag indicating whether the simplex is valid.

```
function [simplex_initial, flag] = NelderMead_simplex(dim, initial_point
    )
flag = 0;
simplex_initial = zeros(dim+1, dim);
for i = 1:(dim+1)
    for j = 1:(dim)
        if i >= 2 && i == j + 1
            simplex_initial(i,j) = 1;
        else
            simplex_initial(1,:) = initial_point;
        end
    end
end
% Check if symplex is invalid
vettore_differenze = zeros(dim,dim);
for i = 2:dim+1
    vettore_differenze(i-1,:) = simplex_initial(1,:) - simplex_initial(i
        ,:);
end
if rank(vettore_differenze) ~= dim
    disp("Initial symplex is invalid")
    flag = 1;
end
end
```

A.1.3 *compute_errorRatio*

This function computes the convergence rate of an iterative method given the convergence point. It outputs a vector of convergence rates and plots the convergence rates over iterations.
Formula for convergence rates: $\frac{||e_{k+1}||}{||e_k||}$ with $e_k = x^k - x^*$

Inputs:

- **x_bar**: Matrix of solution estimates per iteration.
- **n_iter**: Number of total iterations.
- **x_opt**: Convergence point.

Outputs:

- **vec_rate**: Vector of convergence rates.

```
function [vec_rate] = compute_exp_rate_conv2(x_bar, n_iter, x_opt)
vec_rate = zeros(1,n_iter);
for i=1:(length(x_bar)-1)
    e_succ = norm(x_bar(i+1,:)-x_opt);
    e_k = norm(x_bar(i,:)-x_opt);
    if e_k ~= 0
        ratio = e_succ/e_k;
        vec_rate(i) = ratio;
    end
end
figure;
vec_rate = vec_rate(1:(length(x_bar)-1));
plot(1:length(vec_rate), vec_rate, 'LineWidth', 2)
title('Convergence␣rate','FontSize', 16)
xlabel('iteration', 'FontSize', 14)
ylabel('Error␣ratios', 'FontSize', 14)
```

A.1.4 *compute_exp_rate*

This function computes the exponential convergence rates of an iterative method. It returns a vector of convergence rates, calculated using the logarithms of the ratios of consecutive errors at each iteration:

$$\frac{\log\left(\frac{||e_{k+1}||}{||e_k||}\right)}{\log\left(\frac{||e_k||}{||e_{k-1}||}\right)}$$

Inputs:

- **x**: Matrix of solution estimates per iteration.
- **n_iter**: Total number of iterations.

Outputs:

- **vec_rate**: Vector of exponential convergence rates.

```
function [vec_rate] = compute_exp_rate(x, n_iter)
vec_rate = zeros(1,n_iter);
for i=3:(length(x)-1)
    e_succ = norm(x(i+1,:)-x(i,:));
    e_k = norm(x(i,:)-x(i-1,:));
    e_prev = norm(x(i-1,:)-x(i-2,:));
    num = log(e_succ/e_k);
    den = log(e_k/e_prev);
    vec_rate(i) = num/den;
end
vec_rate = vec_rate(3:(length(x)-1));
```

A.1.5 *stagnation_func*

This function calculates and visualizes the norm of the differences between consecutive simplex barycenters. It is intended to study the presence of a *stagnation point* by plotting the changes in successive iterations.

Inputs:

- **x**: A matrix where each row represents an iteration point.

Outputs:

- **vec_e_k**: A row vector containing the norms of the differences between consecutive simplex barycenters.

```
function [vec_e_k] = stagnation_func(x_bar)
vec_e_k = zeros(1,length(x_bar)-1);
for i=2:length(x_bar)
    e_k = norm(x_bar(i,:) - x_bar(i-1,:));
    vec_e_k(1,i-1) = e_k;
end
figure;
plot(1:length(vec_e_k), vec_e_k, 'g', 'LineWidth', 2)
title('Study␣of␣␣stagnation␣point','FontSize', 16)
xlabel('iteration', 'FontSize', 14)
ylabel('||x(k+1)␣-␣x(k)||', 'FontSize', 14)
```

A.2 General script for tuning parameters in Nelder Mead

This function, *comparePar2*, is designed to evaluate the performance of the Nelder-Mead optimization algorithm when applied to the Rosenbrock function. It systematically compares different configurations of four key parameters: ρ, σ, γ, and χ.
For each parameter, l distinct values are tested, leading to a total of l^4 unique parameter combinations. For every one of these configurations, the function calculates:

- The **number of iterations** required for the algorithm to converge.
- The **distance from the optimal solution**.

The primary objective is to find the parameter configuration that minimizes a weighted sum of the iteration count and the convergence error.

Inputs:

- **dim**: The dimensionality of the problem.
- **f**: The objective function to be minimized.
- **initial_point**: The starting point for the Nelder-Mead algorithm.
- **x_opt**: The known optimal solution of the objective function.
- **l**: The number of different values to test for each of the four parameters (ρ, σ, γ, χ).

- `rho_vec`, `sigma_vec`, `gamma_vec`, `chi_vec`: Vectors containing the possible values to be tested for ρ, σ, γ, and χ, respectively.

Outputs:

- `pos1`, `pos2`, `pos3`, `pos4`: These are the indices of the best parameter configuration that minimizes the weighted sum of iterations and convergence error.
  - **pos1** corresponds to the index of the optimal ρ value.
  - **pos2** corresponds to the index of the optimal σ value.
  - **pos3** corresponds to the index of the optimal γ value.
  - **pos4** corresponds to the index of the optimal χ value.

```
function [pos1, pos2, pos3, pos4] = comparePar2(dim, f, initial_point,
    x_opt, l,rho_vec,sigma_vec, gamma_vec,chi_vec)

% for Nelder Mead
kmax = 5000;
tol_simplex = 1e-07;
tol_varf = 1e-07;

% for choosing the best configuration
weight_k = 1*10^(-5);
weight_opt = 1 - weight_k;

% initial simplex
[simplex_initial, flag2] = NelderMead_simplex(dim, initial_point);

configuration_k = zeros(l,l,l,l); % store the number of iteration for
    each configuration of par
configuration_err_conv = zeros(l,l,l,l); % store the distance from
    optimal point for each configuration of par
for i_rho = 1:l
    for i_sig = 1:l
        for i_gam = 1:l
            for i_chi = 1:l
                [k, simplex,x, flag]  = nelder_mead(f, simplex_initial,
                    kmax, rho_vec(i_rho), chi_vec(i_chi), gamma_vec(i_gam
                    ), dim, sigma_vec(i_sig), tol_simplex, tol_varf);
                configuration_k(i_rho, i_sig, i_gam, i_chi) = k;
                configuration_err_conv(i_rho, i_sig, i_gam, i_chi) =
                    norm(x(end,:) - x_opt);
            end
        end
    end
end

% calculate the sum of number of iteration and distance from optimal
    point
% weighed.
configuration_qnt = (weight_k*configuration_k) + (weight_opt*
    configuration_err_conv);

% find the configuration which minimize the quantity
[min_value, lin_index] = min(configuration_qnt(:));
[pos1, pos2, pos3, pos4] = ind2sub(size(configuration_qnt), lin_index);
```

A.3 Exercise 2

In this script the Nelder Mead method is applied to minimize the Rosenbrock function in two dimensions with two different starting point.

```
dim = 2;
initial_point1 = [1.2,1.2];
initial_point2 = [-1,1.2];
f = @(x) 100*(x(2)-x(1)^2)^2+(1-x(1))^2 ; % Rosenbrock function
x_opt = [1,1];

% Tuned parameters for point 1
rho1 = 0.7;
sigma1 = 0.1;
gamma1 = 0.2;
chi1 = 1.5;

% Tuned parameters for point 2
rho2 = 0.7;
sigma2 = 0.1;
gamma2 = 0.6;
chi2 = 2;

% tol for first stopping criterion
tol_simplex = 1e-07;
% tol for second stopping criterion
tol_varf = 1e-15;
% maximum number of iteration
kmax = 10000;

% Create initial simplex
[simplex_initial1, flag1] = NelderMead_simplex(dim, initial_point1);
[simplex_initial2, flag2] = NelderMead_simplex(dim, initial_point2);

% Nelder mead output:
% [# iteration, last simplex, flag about convergence, type of move
    through
% iteration (expansion, contraction, shrinking), max distance between
% barycenter and vertices through iterations]
tic;
[k1, simplex1, x_1, flag1, type_of_move1, size_vec_1]  = nelder_mead(f,
   simplex_initial1, kmax, rho1, chi1, gamma1, dim, sigma1, tol_simplex,
    tol_varf);
time1 = toc;
tic;
[k2, simplex2, x_2, flag2, type_of_move2, size_vec_2]  = nelder_mead(f,
   simplex_initial2, kmax, rho2, chi2, gamma2, dim, sigma2, tol_simplex,
    tol_varf);
time2 = toc;

% Output
disp("Regarding point of convergence...")
disp("convergence for [1.2,1.2] ?")
disp(flag1)
disp("convergence for [-1,1.2] ?")
disp(flag2)
```

```
disp("[1.2,1.2] initial point, convergence point")
disp(x_1(end,:))
disp("distance from the optimum:")
disp(norm(x_1(end,:) - x_opt))
disp("[-1,1.2] initial point, convergence point")
disp(x_2(end,:))
disp("distance from the optimum:")
disp(norm(x_2(end,:) - x_opt))
disp("  ")
disp("Regarding speed of convergence...")
disp("[1.2,1.2] initial point, number iteration before convergence")
disp(k1)
disp("[-1,1.2] initial point, number iteration before convergence")
disp(k2)
disp("[1.2,1.2] initial point, computational costs")
disp(time1)
disp("[-1,1.2] initial point, computational costs")
disp(time2)
disp("size of last simplex for point [1.2,1.2] ")
disp(size_vec_1(end))
disp("size of last simplex for point [-1,1.2]")
disp(size_vec_2(end))

% Theorical rates:
vec_rate1 = compute_errorRatio(x_1, k1, x_opt);
%vec_rate2 = compute_errorRatio(x_2, k2, x_opt);

% Experimental rates:
disp("Last 5 exponential rate of initial point [1.2,1.2]:")
vec_rate3 = compute_exp_rate(x_1, k1);
fprintf('%g␣', vec_rate3(end-5:end));
fprintf('\n');
disp("Last 5 exponential rate of initial rates [-1,1.2]")
vec_rate4 = compute_exp_rate(x_2, k2);
fprintf('%g␣', vec_rate4(end-5:end));
fprintf('\n');

% Stagnation:
vec_increments1 = stagnation_func(x_1);
vec_increments2 = stagnation_func(x_2);

% Picture
f = @(x, y) 100*(y - x.^2).^2 + (1 - x).^2;
x_interval = linspace(-2, 2, 500);
y_interval = linspace(-1, 3, 500);
nelderMead_picture2D(f, x_interval, y_interval, x_1, initial_point1,
   type_of_move1)
nelderMead_picture2D(f, x_interval, y_interval, x_2, initial_point2,
   type_of_move2)

figure;
plot(1:length(size_vec_1), size_vec_1)
xlabel('iter');
ylabel('max␣distance␣vertice-x_bar');
title('Size␣of␣the␣simplex␣through␣iterations');
```

```
figure;
plot(1:length(size_vec_2), size_vec_2)
xlabel('iter');
ylabel('max distance vertice-x_bar');
title('Size of the simplex through iterations');

figure;
plot(1:length(x_1), vecnorm(x_1 - x_opt, 2, 2),'LineWidth', 1.5,  'Color
    ', 'b')
xlabel('iter','FontSize', 14);
ylabel('distance x-x_opt', 'FontSize', 14);
title('Distance from optimum through iterations','FontSize', 14);

figure;
plot(1:length(x_2), vecnorm(x_2 - x_opt, 2, 2),'LineWidth', 1.5,  'Color
    ', 'b')
xlabel('iter','FontSize', 14);
ylabel('distance x-x_opt', 'FontSize', 14);
title('Distance from optimum through iterations','FontSize', 14);
```

A.4 Exercise 3

This function runs the Nelder-Mead method for a given objective function, utilizing a fixed initial point and an additional 10 random points generated within a hypercube constructed from the fixed initial point. Its primary purpose is to assess the algorithm's performance, specifically measuring convergence rates and execution times.
Inputs:

- **dim**: The dimension of the problem.
- **f**: The objective function to be minimized.
- **x_initial**: The fixed initial point for the Nelder-Mead algorithm.
- **x_opt**: The known optimal solution for the objective function, used for performance evaluation.

Outputs:

- **vec_time**: A 1×11 vector storing the times for the Nelder-Mead algorithm:
  - **vec_time(1)**: Time taken for the single, fixed initial point.
  - **vec_time(2:end)**: Times taken for each of the 10 random initial points.
- **k1**: The number of iterations required for convergence when using the single, fixed initial point.
- **x1**: The final solution (specifically, the best point of the simplex) obtained when starting from the single, fixed initial point.
- **k_10_points**: A 1×10 vector, where each element represents the number of iterations required for convergence for each of the 10 random initial points.
- **x_10_points**: A $10 \times (\text{dim} + 1)$ matrix. Each row of this matrix contains:

  - The final solution (the best point of the simplex) obtained for each respective random initial point.
  - A success/failure flag, included as the last element of the row.

- **lista_rates**: A cell array. Each cell contains a vector of convergence rates (calculated based on the distance to the known optimal solution) for each of the 11 applications of the method (one for the fixed initial point, and 10 for the random initial points).

- **lista_err**: A cell array. Each cell contains a vector of the norms of the differences between consecutive barycenters of the simplex, calculated for each of the 11 applications of the method.

```
function [vec_time, k1, x1, k_10_points,x_10_points, lista_rates,
   lista_err] = NelderMead_for_10Points_2(dim,f,x_initial,x_opt,rho,
   sigma,chi,gamma)

% Parameters for Nelder Mead
kmax = 10000;
tol_simplex = 1e-07;
tol_varf = 1e-07;
vec_time = zeros(1,11); % contains computational costs
lista_rates = {};  % contains rates of convergence
lista_err = {};    % contains increments of x_bar

% Nelder Mead method with suggested initial points
[simplex_initial, flag] = NelderMead_simplex(dim, x_initial);
tic
[k1, simplex,x1, flag,size_vec]  = nelder_mead(f, simplex_initial, kmax,
    rho, chi, gamma, dim, sigma, tol_simplex, tol_varf);
vec_time(1,1) = toc;

% Outputs
figure;
plot(length(size_vec)-500:length(size_vec), size_vec(length(size_vec)
   -500:length(size_vec)),'LineWidth', 2)
xlabel('iter','FontSize', 14);
ylabel('max␣distance␣vertice-bar', 'FontSize', 14);
title('Size␣of␣the␣simplex␣through␣iterations','FontSize', 14);

figure;
plot(length(x1)-500:length(x1), vecnorm(x1(length(x1)-500:length(x1),:)
   - x_opt, 2, 2),'LineWidth', 2)
xlabel('iter','FontSize', 14);
ylabel('distance␣x-x_opt', 'FontSize', 14);
title('Distance␣from␣optimum␣through␣iterations','FontSize', 14);


vec_error = compute_errorRatio(x1, k1, x_opt);
vec_err = stagnation_func(x1);
vec_rate = compute_exp_rate(x1,k1);
lista_rates{1} = vec_rate;
lista_err{1} = vec_err;

x1 = x1(end,:);
x1 = [x1, flag];
```

```
% Nelder Mead method with 10 random initial points from ipercube
% Generate points
l_bound = x_initial - 1;
u_bound = x_initial + 1;
M_ten_initial_points = zeros(dim, 10);
for i=1:dim
    coord_random = l_bound(i)*ones(1,10) + (u_bound(i) - l_bound(i)) *
       rand(1, 10);
    M_ten_initial_points(i,:) = coord_random;
end

% Nelder mead for each point
x_10_points = zeros(10,dim + 1);
k_10_points = zeros(1,10);

for i = 1:10
    initial_point = M_ten_initial_points(:,i)';
    disp("Point:")
    disp(i+1)
    [simplex_initial, flag] = NelderMead_simplex(dim, initial_point);
    tic
    [k, simplex, x, flag, size_vec]  = nelder_mead(f, simplex_initial,
       kmax, rho, chi, gamma, dim, sigma, tol_simplex, tol_varf);
    vec_time(1,i+1) = toc;
    row_10_points = [x(end,:), flag];
    x_10_points(i,:) = row_10_points ;
    k_10_points(1,i) = k;

    % Outputs

    vec_error_i = compute_errorRatio(x, k, x_opt);
    vec_rate_i = compute_exp_rate(x,k);
    vec_err_i = stagnation_func(x);
    figure;
    %plot(length(size_vec)-500:length(size_vec), size_vec(length(
       size_vec)-500:length(size_vec)),'LineWidth', 2)
    plot(size_vec, 'LineWidth', 2);
    xlabel('iter','FontSize', 14);
    ylabel('max distance vertice-bar', 'FontSize', 14);
    title('Size of the simplex through iterations','FontSize', 14);
    lista_rates{end+1} = vec_rate_i;
    lista_err{end+1} = vec_err_i;
end

% Outputs

vec_error_i = compute_errorRatio(x, k, x_opt);
vec_rate_i = compute_exp_rate(x,k);
vec_err_i = stagnation_func(x);
figure;
%plot(length(size_vec)-500:length(size_vec), size_vec(length(size_vec)
   -500:length(size_vec)),'LineWidth', 2)
plot(size_vec, 'LineWidth', 2);
xlabel('iter','FontSize', 14);
ylabel('max distance vertice-bar', 'FontSize', 14);
```

```
title('Size␣of␣the␣simplex␣through␣iterations','FontSize', 14);
lista_rates{end+1} = vec_rate_i;
lista_err{end+1} = vec_err_i;
%disp(size_vec)
end
```

Code for Modified Newton Method

Below are the scripts written for the Modified Newton method. The code can be found on GitHub: *https://github.com/cristina210/Assignement-numerical-optimization*

B.5 Main scripts for Modified Newton method

B.5.1 *modified_newton_method*

This function implements the Modified Newton method to minimize a given function.

```
function [xk, fk, gradfk_norm, k, ...
    xseq, btseq, failure] = modified_newton_method(x0, f, gradf, hessf,
       kmax, tolgrad, c1, btmax, n, rho, h0, flag_h)

% function [xk, fk, gradfk_norm, k, xseq, btseq, failure] =
   modified_newton_method(x0, f, gradf, hessf, kmax, tolgrad, c1, btmax,
    n, rho, h0, flag_h)
% Function that performs the Newton optimization method when the hessian
% matrix is not positive definite, using bactracking strategy for the
% step-length selection.
%
% INPUTS:
% x0 = n-dimensional column vector;
% f = function handle that describes a function R^n->R;
% gradf = function handle that describes the gradient of f;
% hessf = function handle that describes the Hessian of f;
% kmax = maximum number of iterations permitted;
% tolgrad = value used as stopping criterion w.r.t. the norm of the
% gradient;
% c1 =  the  factor of the Armijo condition that must be a scalar in
   (0,1);
% btmax =  maximum  number of steps for updating alpha during the
% backtracking strategy;
% n = dimension of x0;
% rho =  fixed  factor, lesser than 1, used for reducing alpha;
% h0 = step-length through which we compute the expansion x = x_bar + h
   used
% to compute finite difference;
% flag_h = flag used to indicate which h to use, if flag_h = 1, need to
% adapt h at each value, if flag_h = 0, h is costant, even in the case
   with
% exact derivatives where h = 0;
%
```

```
% OUTPUTS:
% xk = the last x computed by the function with preconditioning;
% fk = the value f(xk);
% gradfk_norm = value of the norm of gradf(xk);
% k = index of the last iteration performed;
% xseq = 4-by-k matrix where the columns are the elements xk of the
% sequence;
% btseq = 1-by-k vector where elements are the number of backtracking
% iterations at each optimization step with xk;
% failure = flag which indicates if there is stagnation during
% backtracking.


% Pre-allocate
xk = x0;
fk = f(xk);
xseq = zeros(n, 4);
btseq = zeros(1, kmax);
beta = 10^-3;
failure = false;
h = ones(n,1) * h0;

if flag_h == 1
    h = h0*abs(xk);
end

gradfk = gradf(xk,h);
gradfk_norm = norm(gradfk);

farmijo = @(fk, alpha, c1_gradfk_pk) fk + alpha * c1_gradfk_pk;
k = 0;

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

time_limit = 1000;
start_time = tic;
while k < kmax && gradfk_norm > tolgrad

    if toc(start_time) > time_limit
        failure = true;
        warning('The method stopped because it reached time_limit')
        break;
    end

    if flag_h == 1
        h = h0*abs(xk);
    end
    hessfk = hessf(xk,h);

    % solve with Cholesky correction
    L = choleski_added_multiple_identity(hessfk, beta, n);
    y = L \ (-gradfk);
    pk = L' \ y;
```

```
% Backtracking
alpha = 1;
xnew = xk + alpha * pk;
fnew = f(xnew);
c1_gradfk_pk = c1 * gradfk' * pk;
bt = 0;

while bt < btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
    alpha = rho * alpha;

    xnew = xk + alpha * pk;

    if norm(xnew-xk,2) < 10^-12
        failure = true;
        warning('Stagnation␣in␣backtracking')
        break
    end
    fnew = f(xnew);
    bt = bt + 1;
end

if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
    warning('Backtracking␣failed␣to␣find␣a␣good␣step.');
    failure = true;
    btseq(k+1, 1) = bt;
    xk = xnew;
    k = k+1;
    break;
end

if failure == true
    warning("Stagnation in bactracking: change btmax or rho
       parameters.")
    break
end

% Update
xk = xnew;
fk = fnew;
gradfk = gradf(xk,h);
gradfk_norm = norm(gradfk);

k = k + 1;
xseq(:, mod(k, 4) + 1) = xk;
btseq(k) = bt;


% best_values(k) = fk;
% best_gradf(k) = norm(gradfk);
% if mod(k, 10) == 0
%     figure(2);
%     plot(best_values(6:k), '-o', 'MarkerSize', 6, 'LineWidth', 2);
%     xlabel('Iterations', 'FontSize', 14);
%     ylabel('Best Evaluation', 'FontSize', 14);
```

```
    %     title('Progress minimum value Modified Newton Method', '
       FontSize', 16);
    %     set(gca, 'FontSize', 12);
    %     drawnow;
    %
    %     figure(3);
    %     plot(best_gradf(5:k), '-o', 'MarkerSize', 6, 'LineWidth', 3);
    %     xlabel('Iterations', 'FontSize', 14);
    %     ylabel('Best Evaluation', 'FontSize', 14);
    %     title('Progress gradient value Modified Newton Method', '
       FontSize', 16);
    %     set(gca, 'FontSize', 12);
    %     drawnow;
    % end

end

if (k == kmax || failure) && gradfk_norm > tolgrad
    failure = true;
end

btseq = btseq(1:k);

% Add final block
% if mod(k, 10) ~= 0 && k > 0
%     figure(1);
%     %hold off;
%     plot(best_values(1:k), '-o', 'MarkerSize', 6, 'LineWidth', 2);
%     xlabel('Iterations', 'FontSize', 14);
%     ylabel('Best Evaluation', 'FontSize', 14);
%     title('Progress minimum value Modified Newton Method', 'FontSize',
    16);
%     set(gca, 'FontSize', 12);
%     drawnow;
% end
```

B.5.2 *choleski_added_multiple_identity*

This function implements the addition of a multiple of the identity matrix to a Cholesky factorization.

```
function L = choleski_added_multiple_identity(hessfk, beta, n)

% function L = choleski_added_multiple_identity(hessfk, beta, n)
%
% Function that modifies a possibly indefinite Hessian matrix by adding
   a
% scaled identity matrix to ensure positive definiteness, and computes
   its
% Cholesky factorization.
%
% INPUTS:
% hessfk = symmetric Hessian matrix (possibly indefinite) of size n-by-n
   ;
```

```
% beta = small positive scalar used to ensure the perturbed matrix
    becomes
%         positive definite;
% n = dimension of the matrix hessfk;
%
% OUTPUT:
% L = lower triangular matrix such that L * L' = Bk, where Bk is a
%     positive definite modification of hessfk.
%
% DESCRIPTION:
% The function iteratively adds tau_k * I to hessfk, where tau_k is
% initialized based on the smallest diagonal entry of hessfk. If hessfk
    is
% already positive definite, tau_k starts at zero. Otherwise, a positive
% shift is applied.
% At each iteration, a Cholesky decomposition is attempted. If it fails
% (i.e., the matrix is not positive definite), tau_k is increased (up to
    a
% maximum of 1e5) and the process is repeated, up to kmax iterations.
% If a positive definite matrix is found, the function returns its
% Cholesky factor. Otherwise, it raises an error.

kmax = 500;

% Find minimun element of the diagonal
min_diag_hess = min(diag(hessfk));

% Initialization of tauk
if min_diag_hess > 0
    tauk = 0;
else
    tauk = -min_diag_hess + beta;
end

flag_pos_def = 1;
k = 0;
I = speye(n);

while k < kmax && flag_pos_def ~= 0
    % Bk is created as a sparse matrix
    Bk = hessfk + tauk * I;

    % Cholesky factoritazion to check if the matrix is positive definite
    [L, flag_pos_def] = chol(Bk, 'lower');

    if flag_pos_def == 0
        break;
    else
        % tauk = max(10 * tauk, beta);
        % tauk = max(2 * tauk, beta);
        tauk = min(max(2 * tauk, beta), 1e5);
    end

    k = k + 1;
end
```

```
if tauk > 1e6
    error('Tauk too big.')
elseif k == kmax && flag_pos_def == 1
    error('Bk NOT positive definite')
end

end
```

B.5.3 *compute_exp_rate_conv_multi*

This function implements the addition of a multiple of the identity matrix to a Cholesky factorization.

```
function rate = compute_exp_rate_conv_multi(xseq)

% function rate = compute_exp_rate_conv_multi(xseq)
%
% Function that estimates the experimental rate of convergence (ERC)
% of an iterative optimization method, based on the last three iterates.
%
% INPUT:
% xseq = n-by-m matrix where each column represents the iterate x_k of
    the
%        optimization method. The function assumes m >= 3 and uses only
%        the last three iterates (i.e., x_{k}, x_{k-1}, x_{k-2});
%
% OUTPUT:
% rate = estimate of the experimental rate of convergence;

num = xseq(:,end) - xseq(:,end-1);
denom = xseq(:,end-1) - xseq(:,end-2);
rate = log(sum(num.^2))/log(sum(denom.^2));

end
```

B.5.4 *create_bar_plot*

This function create a bar plot to compare three vectors of performance metrics.

```
function create_bar_plot(vec1, vec2, vec3, flag, flag_fun)

% function create_bar_plot(vec1, vec2, vec3, flag)
%
% This function creates a grouped bar plot to compare three vectors of
% performance metrics (e.g., time, error, iterations, or convergence
    rate)
% across different problem dimensions (e.g., 1e3, 1e4, 1e5).
% If the vectors contain only one element each, the x-axis will
    represent
% the dimensions directly. Otherwise, the x-axis will show values of the
% step size (h), and the bars will be grouped by step size, with colors
```

```
% indicating different dimensions.
%
% INPUTS:
%   vec1 - Vector of metrics for dimension 1e3
%   vec2 - Vector of metrics for dimension 1e4
%   vec3 - Vector of metrics for dimension 1e5
%   flag - A string indicating the type of metric being plotted
%          ('time', 'diff', 'iter', or other)

    n = length(vec1);
    rates = [vec1(:), vec2(:), vec3(:)];  % Each column is a dimension

    figure;

    if n == 1
        for i = 1:3
            bar(i, rates(1, i), 'FaceColor', get_color(i));
            hold on;
        end
        xlim([0.5, 3.5]);
        xticks(1:3);
        xticklabels({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'});
        xlabel('Dimension');
    else
        hb = bar(rates);
        colors = [0.8500 0.3250 0.0980;
                  0.4660 0.6740 0.1880;
                  0.0000 0.4470 0.7410];
        for i = 1:3
            hb(i).FaceColor = colors(i,:);
        end

        x_labels = arrayfun(@(k) sprintf('10^{-%d}', 2*k), 1:n, '
           UniformOutput', false);
        xticks(1:n);
        xticklabels(x_labels);
        xlabel('h');
    end

    ylim([0, max(rates(:)) + 0.01]);

    switch flag
        case 'time'
            ylabel('Average time');
            metric_title = 'Comparison of average time';
        case 'diff'
            ylabel('Average difference');
            metric_title = 'Comparison of average difference';
        case 'iter'
            ylabel('Average iterations');
            metric_title = 'Comparison of average iterations';
        otherwise
            ylabel('Average rate of convergence');
            metric_title = 'Comparison of average rate of convergence';
    end
```

```
    problem_name = '';
    switch flag_fun
        case 'ros'
            problem_name = ' (Rosenbrock Chained)';
        case 'wood'
            problem_name = ' (Wood Chained)';
        case 'pow'
            problem_name = ' (Powell Chained)';
    end

    title([metric_title, problem_name]);


    legend({'dim = 1e3', 'dim = 1e4', 'dim = 1e5'}, 'Location', 'best');
    grid on;
end

function c = get_color(i)

% function get_color(i)
% Helper function to return a predefined color for each dimension index.
%
% INPUTS:
%   i - Index of the vector (1, 2, or 3)
%
% OUTPUTS:
%   c - RGB color triplet
    color_map = [0.8500 0.3250 0.0980;
                 0.4660 0.6740 0.1880;
                 0.0000 0.4470 0.7410];
    c = color_map(i, :);
end
```

B.5.5 *create_picture*

```
function create_picture(matrix, rho_vec, c1_vec, point, val_y,
   value_studied)
%
% function create_picture(matrix, rho_vec, c1_vec, point)
% Function that create a bar plot showing how the number of iterations
% needed to converge vary base on the different combination of rho and
   c1
% values
%
% INPUTS:
% matrix = matrix with length(rho_vec) rows and length(c1_vec) columns
% whose entries are the iterations required for each values combination
% rho_vec = vector containing all the rho values
% c1_vec = vector containing all the c1 values
% point = term which specifies which is the starting point (1,2)

    figure;

    data = matrix';
    b = bar(data, 'grouped');
```

```
    colors = lines(length(rho_vec));
    for i = 1:length(rho_vec)
        b(i).FaceColor = colors(i, :);
    end

    xticks(1:length(c1_vec));
    xticklabels(arrayfun(@(x) sprintf('%.0e', x), c1_vec, 'UniformOutput
        ', false));
    xlabel(val_y, 'FontSize', 14);
    ylabel(value_studied, 'FontSize', 14);
    legend(arrayfun(@(x) sprintf('\\rho = %.1f', x), rho_vec, '
        UniformOutput', false), ...
            'Location', 'BestOutside');
    title(sprintf('%s vs %s from point %d', value_studied, val_y, point)
        , 'FontSize', 16);
    grid on;

end
```

B.5.6 *modNewtonMethod_picture2D*

```
function modNewtonMethod_picture2D(f, x_interval, y_interval, xseq1,
    xseq2, initial_point1, initial_point2)
%
% function modNewtonMethod_picture2D(f, x_interval, y_interval, xseq1,
    xseq2, initial_point1, initial_point2)
%
% Function that take the picture whit color and countout map of the
% Rosenbrock function, showing the steps needed to the modified Newton
% method to converge to the minimum point startinf from the two
    different
% initial point given.
%
% INPUTS:
% f = function handle that describes a function R^n->R;
% x_interval = x coords in which the function is evaluated;
% y_interval = y coords in which the function is evaluated;
% xseq1 = sequences of points find each step until the method converges
% from initial_point1;
% xseq2 = sequences of points find each step until the method converges
% from initial_point2;
% initial_point1 = 1^st starting point
% initial_point2 = 2^nd starting point
%

[X, Y] = meshgrid(x_interval, y_interval);
Z = f(X, Y);
figure;
imagesc(x_interval, y_interval, Z);
set(gca, 'YDir', 'normal');
colorbar;
set(gca, 'ColorScale', 'log');
hold on;
contour(X, Y, Z, 50, 'LineColor', 'k');
```

```
plot(initial_point1(1), initial_point1(2), 'rp', 'MarkerSize', 12, '
   MarkerFaceColor', 'r');
plot(initial_point2(1), initial_point2(2), 'cp', 'MarkerSize', 12, '
   MarkerFaceColor', 'c');
plot(xseq1(1, :), xseq1(2, :), 'ro-', 'LineWidth', 1.3, 'MarkerSize',
   3.5)
plot(xseq2(1, :), xseq2(2, :), 'co-', 'LineWidth', 1.3, 'MarkerSize',
   3.5)
plot(1, 1,  'yo', 'MarkerFaceColor', 'y', 'MarkerSize', 5);
hold off;
xlabel('x');
ylabel('y');
title('Rosenbrock␣function␣-␣Color␣and␣Contour␣Map');

end
```

B.5.7 *print_function*

This function print the function f in 2D.

```
function print_function(x_opt, f)
    i = 1;  % index 1^st variable to vary
    j = 2;  % index 2^nd variable to vary

    % Meshgrid in the neighbourhood of x_opt
    delta = 0.5;
    step = 0.1;
    x1 = x_opt(i);
    x2 = x_opt(j);
    [xi, xj] = meshgrid(x1 - delta : step : x1 + delta, ...
                        x2 - delta : step : x2 + delta);

    Z = zeros(size(xi));

    % Evaluation of f varying xi, xj
    for row = 1:size(xi, 1)
        for col = 1:size(xi, 2)
            x = x_opt;
            x(i) = xi(row, col);
            x(j) = xj(row, col);
            Z(row, col) = f(x);
        end
    end

    % Plot of the neighbourhood of x_opt
    figure;
    surf(xi, xj, Z);
    xlabel(sprintf('x_%d', i));
    ylabel(sprintf('x_%d', j));
    zlabel('f(x)');
    title(sprintf('Surrounding␣of␣x_%d,␣x_%d', i, j));
    shading interp;
    view(135,30);
    hold on;
```

```
    f_min = f(x_opt);
    plot3(x_opt(1), x_opt(2), f_min, 'ro', 'MarkerSize', 8, '
       MarkerFaceColor', 'r');
    legend('f(x)', 'theoric minimum point');
    hold off;

end
```

B.5.8 *Exercise 2*

```
clc
clear all
close all

% set the seed to obtain the same results while running the program
seed_value = min(343341, 343428);
rng(seed_value);

dim = 2;
initial_point1 = [1.2; 1.2];
initial_point2 = [-1.2; 1];
initial_points = [initial_point1, initial_point2];
x_opt = ones(2,1);

% function definition
f = @(x) 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;

% gradient of the function
gradf = @(x,h) [400*x(1)^3 - 400*x(1)*x(2) + 2*x(1) - 2;
                    200*(x(2) - x(1)^2)];

% Hessian matrix of the function
hessf = @(x,h) [ 1200*x(1)^2 - 400*x(2) + 2, -400*x(1);
                    -400*x(1), 200];

% Implementation of centered finite differences approximation for gradf
   and hessf
gradf_finite_diff_rosenbrock = @(x, h)[
    -400*x(1)*x(2) + 400*x(1)^3 + 400*h(1).^2*x(1) - 2 + 2*x(1);
    200*x(2) - 200*x(1)^2
    ];

% in this way I obtain the hessian matrix for the case of h costante and
% variable, in the 1^st case I pass to the function h 2x1 with equal
% components
hess_finite_diff_rosenbrock = @(x, h)[
    1200*x(1)^2 - 400*x(2) + 2 + 200*h(1)^2,  -400*x(1) - 200*h(1);
    -400*x(1) - 200*h(1)                , 200
    ];

% parameters used by the function modified_netwon_method
rho = 0.5;
tolgrad = 10^(-7);
c1 = 10^(-4);
```

```
btmax = 40;
kmax = 500;

flag_h = 0; % h not used, compute the exact derivatives
h = 0;

time = zeros(2,1);
xseq_tot = cell(1,2);
iter = zeros(2,1);
failure_tot = zeros(2,1);

% compute the experimental rate of convergence
vec_rate = zeros(1,2);
vec_rate_pre = zeros(1,2);

for i=1:2
    tic;
    [xk, fk, gradfk_norm, k, ...
        xseq, btseq, failure] = modified_newton_method(initial_points(:,
           i), f, gradf, hessf, kmax, ...
        tolgrad, c1, btmax, dim, rho, h, flag_h);
    time(i) = toc;
    iter(i) = k;
    xseq_tot{i} = xseq;
    vec_rate(i) = compute_exp_rate_conv_multi(xseq(:,end-4:end));
    failure_tot(i) = failure;
end


% Picture
f_picture = @(x, y) 100*(y - x.^2).^2 + (1 - x).^2;
x_interval = linspace(-2, 2, 500);
y_interval = linspace(-1, 3, 500);
%modNewtonMethod_picture2D(f_picture, x_interval, y_interval, xseq_tot
   {1}, xseq_tot{2}, initial_point1, initial_point2)


% testing different parameters for bactracking strategy with rho     0.5
    and
% c1     10^-4 and starting from initial_point1
rho_vec = [0.1, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9];
c1_vec = [10^(-7), 10^(-5), 10^(-4), 10^(-3), 0.01, 0.1];

time = zeros(length(rho_vec), length(c1_vec));
test_diff_c1_rho = zeros(length(rho_vec), length(c1_vec));
failure_tot = zeros(length(rho_vec), length(c1_vec));

time1 = zeros(length(rho_vec), length(c1_vec));
test_diff_c1_rho1 = zeros(length(rho_vec), length(c1_vec));
failure_tot1 = zeros(length(rho_vec), length(c1_vec));
for i=1:length(rho_vec)
    for j=1:length(c1_vec)
        tic;
        [xk, fk, gradfk_norm, k, ...
        xseq, btseq, failure] = modified_newton_method(initial_point1, f
           , gradf, hessf, kmax,...
```

```
        tolgrad, c1, btmax, dim, rho_vec(i), h, flag_h);
        time(i,j) = toc;
        test_diff_c1_rho(i, j) = k;
        failure_tot(i,j) = failure;

        tic;
        [xk1, fk1, gradfk_norm1, k1, ...
        xseq1, btseq1, failure1] = modified_newton_method(initial_point2
            , f, gradf, hessf, kmax,...
        tolgrad, c1, btmax, dim, rho_vec(i), h, flag_h);
        time1(i,j) = toc;
        test_diff_c1_rho1(i, j) = k1;
        failure_tot1(i,j) = failure1;
    end
end

% Find which rho and c1 values lead to the fastest convergence
min_value = min(test_diff_c1_rho(:));
[index_rho, index_c1] = find(test_diff_c1_rho == min_value);
min_rho = rho_vec(index_rho(1));
min_c1 = c1_vec(index_c1(1));
create_picture(test_diff_c1_rho, rho_vec, c1_vec, 1, 'c1', 'iter')
create_picture(test_diff_c1_rho1, rho_vec, c1_vec, 2, 'c1', 'iter')

% Finite differences


% parameter used for finite differences
h_vec = 10.^[-2, -4, -6, -8, -10, -12];


% Testing different values of h solving pcg without preconditioning
% starting from both initial_point1 and initial_point2
flag_h = 0; % we do not use a specific increment for each value x_i

time_finite_diff = zeros(length(rho_vec),length(h_vec));
norm_err_conv = zeros(length(rho_vec), length(h_vec));
iter = zeros(length(rho_vec), length(h_vec));
failure_tot = zeros(length(rho_vec),length(h_vec));
vec_rate = zeros(length(rho_vec),length(h_vec));

time_finite_diff2 = zeros(length(rho_vec),length(h_vec));
norm_err_conv2 = zeros(length(rho_vec), length(h_vec));
iter2 = zeros(length(rho_vec), length(h_vec));
failure_tot2 = zeros(length(rho_vec),length(h_vec));
vec_rate2 = zeros(length(rho_vec),length(h_vec));

% Testing the finite differences applying a specific h_i when
% differentiating with respect to the variable x_i if flag_h == 1
flag_h2 = 1;

time_finite_diff_h_var = zeros(length(rho_vec), length(h_vec));
norm_err_conv_h_var = zeros(length(rho_vec), length(h_vec));
iter_h_var = zeros(length(rho_vec), length(h_vec));
failure_tot_h_var = zeros(length(rho_vec),length(h_vec));
vec_rate_h_var = zeros(length(rho_vec),length(h_vec));
```

```
time_finite_diff_h_var2 = zeros(length(rho_vec), length(h_vec));
norm_err_conv_h_var2 = zeros(length(rho_vec), length(h_vec));
iter_h_var2 = zeros(length(rho_vec), length(h_vec));
failure_tot_h_var2 = zeros(length(rho_vec),length(h_vec));
vec_rate_h_var2 = zeros(length(rho_vec),length(h_vec));

tolgrad = 10^(-7);
c1 = 10^(-4);
btmax = 40;
for i=1:length(rho_vec)
    for j=1:length(h_vec)
        tic;
        [xk, fk, gradfk_norm, k, ...
        xseq, btseq, failure] = modified_newton_method(initial_point1, f
           , gradf_finite_diff_rosenbrock, ...
        hess_finite_diff_rosenbrock, kmax,...
        tolgrad, c1, btmax, dim, rho, h_vec(j), flag_h);
        time_finite_diff(i,j) = toc;
        iter(i,j) = k;
        failure_tot(i,j) = failure;
        norm_err_conv(i,j) = norm(x_opt - xk,2);
        vec_rate(i,j) = compute_exp_rate_conv_multi(xseq);

        tic;
        [xk2, fk2, gradfk_norm2, k2, ...
        xseq2, btseq2, failure2] = modified_newton_method(initial_point2
           , f, gradf_finite_diff_rosenbrock, ...
        hess_finite_diff_rosenbrock, kmax,...
        tolgrad, c1, btmax, dim, rho, h_vec(j), flag_h);
        time_finite_diff2(i,j) = toc;
        iter2(i,j) = k2;
        failure_tot2(i,j) = failure2;
        norm_err_conv2(i,j) = norm(x_opt - xk2,2);
        vec_rate2(i,j) = compute_exp_rate_conv_multi(xseq2);

        tic;
        [xk_h_var, fk_h_var, gradfk_norm_h_var, k_h_var, ...
        xseq_h_var, btseq_h_var, failure_h_var] = modified_newton_method
           (initial_point1, f, gradf_finite_diff_rosenbrock, ...
        hess_finite_diff_rosenbrock, kmax,...
        tolgrad, c1, btmax, dim, rho, h_vec(j), flag_h2);
        time_finite_diff_h_var(i,j) = toc;
        iter_h_var(i,j) = k_h_var;
        failure_tot_h_var(i,j) = failure_h_var;
        norm_err_conv_h_var(i,j) = norm(x_opt - xk_h_var,2);
        vec_rate_h_var(i,j) = compute_exp_rate_conv_multi(xseq_h_var);

        tic;
        [xk_h_var2, fk_h_var2, gradfk_norm_h_var2, k_h_var2, ...
        xseq_h_var2, btseq_h_var2, failure_h_var2] =
           modified_newton_method(initial_point2, f,
           gradf_finite_diff_rosenbrock, ...
        hess_finite_diff_rosenbrock, kmax,...
        tolgrad, c1, btmax, dim, rho, h_vec(j), flag_h2);
        time_finite_diff_h_var2(i,j) = toc;
```

```
        iter_h_var2(i,j) = k_h_var;
        failure_tot_h_var2(i,j) = failure_h_var;
        norm_err_conv_h_var2(i,j) = norm(x_opt - xk_h_var,2);
        vec_rate_h_var2(i,j) = compute_exp_rate_conv_multi(xseq_h_var2);
    end
end

create_picture(iter, rho_vec, h_vec, 1, 'h', 'iter')
create_picture(iter_h_var, rho_vec, h_vec, 1, 'h\_var', 'iter')
```

B.5.9 *Exercise 3*

We create three different scripts, one for each function, and impose manually the dimension n of the input vector x. Each script contain function handle for the function, the gradient and hessian matrix computed with both exact derivatives and finite differences. Then we execute the test for exact derivatives, finite differences with constant and variable h and save the output information into a file named *output.mat*.

First part, common to each function

```
clc
clear all
close all

% set the seed to obtain the same results while running the program
seed_value = min(343341, 343428);
rng(seed_value);
dim = 10^4;
disp("dimension:")
disp(dim)
```

Rosenbrock chained

```
% function definition
f1_ros = @(x) sum(100*(x(1:end-1).^2 - x(2:end)).^2 + (x(1:end-1) - 1)
    .^2);

% gradient of the function
grad_f1_ros = @(x,h) sparse([
    2 * x(1) + 400 * x(1)^3 - 400 * x(1)*x(2) - 2;
    -200 * x(1:end-2).^2 + 202 * x(2:end-1) + 400 * x(2:end-1).^3 - 400
       * x(2:end-1).*x(3:end) - 2;
    -200 * x(end - 1)^2 + 200 * x(end)
]);

% Hessian matrix of the function
hess_f1_ros = @(x,h) spdiags([
    sparse([-400*x(1:end-1); 0]), ...
    sparse([1200 * x(1)^2 - 400 * x(2) + 2; ...
            202 + 1200 * x(2:end - 1).^2 - 400 * x(3:end); ...
            200]), ...
    sparse([0; -400*x(1:end-1)])
], [-1, 0, 1], length(x), length(x));
```

```
% Implementation of centered finite differences approximation for gradf
    and
% hessf:
gradf_finite_diff_ros = @(x,h) [400*x(1)^3 + 400*h(1)^2*x(1) - 400*x(1)*
    x(2) + 2*x(1) - 2; ...
     -200*x(1:end-2).^2 + 202*x(2:end-1) + 400*x(2:end-1).^3 + 400*h(2:
        end-1).^2.*x(2:end-1) - 400*x(2:end-1).*x(3:end) - 2; ...
     -200*x(end-1)^2 + 200*x(end)];


hess_finite_diff_ros = @(x, h) spdiags([
     [-400 * x(1:end-1) - 200 * h; 0],...
     [1200 * x(1)^2 + 200 * h^2 - 400 * x(2) + 2; ...
     1200 * x(2:end-1).^2 - 400 * x(3:end) + 200 * h^2 + 202; ...
     200],...
     [0; -400 * x(1:end-1) - 200 * h]
     ], [-1, 0, 1 ], length(x), length(x));

hess_finite_diff_ros_h_var = @(x, h) spdiags([
     [-400 * x(1:end-1) - 200 * h(1:end-1); 0],...
     [1200 * x(1)^2 + 200 * h(1)^2 - 400 * x(2) + 2; ...
     1200 * x(2:end-1).^2 - 400 * x(3:end) + 200 * h(2:end-1).^2 + 202;
         ...
     200],...
     [0; -400*x(1:end-1) - 200 * h(1:end-1)]
     ], [-1, 0, 1 ], length(x), length(x));

x1_rosenbrock = repmat([-1.2; 1.0], dim/2, 1); % because dim is even
x1_opt = ones(dim,1);
l_bound = x1_rosenbrock - 1;
u_bound = x1_rosenbrock + 1;
M_ten_initial_points = l_bound + (u_bound - l_bound) .* rand(dim, 10);
starting_points = [x1_rosenbrock, M_ten_initial_points];
```

Wood chained

```
% function definition
f2_wood = @(x) sum(100*(x(1:2:end-3).^2 - x(2:2:end-2)).^2 + (x(1:2:end
    -3) - 1).^2) + ...
     sum(90*(x(3:2:end-1).^2 - x(4:2:end)).^2 + (x(3:2:end-1) - 1).^2) +
         ...
     sum(10*(x(2:2:end-2) + x(4:2:end) - 2).^2 + (x(2:2:end-2) - x(4:2:
        end)).^2 / 10);

% gradient of the function:
function g = grad_wood_fun(x,h)
     n = length(x);
     g = sparse(n,1);

     g(1) = 400 * x(1)^3 - 400 * x(1) * x(2) + 2 * x(1) - 2;
     g(2) = -200 * x(1)^2 + (1101/5) * x(2) + 99/5 * x(4) - 40;

     g(n-1) = 360 * x(n-1)^3 - 360 * x(n-1) * x(n) + 2 * x(n-1) - 2;
```

```
    g(n) = (99/5) * x(n-2) - 180 * x(n-1)^2 + (1001/5) * x(n) - 40;

    % elementi 4:2:n-2
    g(4:2:n-2) = (99/5) * x(2:2:n-4) - 380 * x(3:2:n-3).^2 + (2102/5) *
       x(4:2:n-2) + (99/5) * x(6:2:n) - 80;

    % elementi 3:2:n-3
    g(3:2:n-3) = 760 * x(3:2:n-3).^3 + 4 * x(3:2:n-3) - 760 * x(3:2:n-3)
        .* x(4:2:n-2) - 4;
end
grad_wood = @(x,h) grad_wood_fun(x,h);

% Hessian matrix of the function:
function H = hess_wood_fun(x,h)
    n = length(x);

    % Pre-allocate vectors for diagonals
    main_diag = sparse(n,1);
    sup_diag1 = sparse(n,1); % sup-diagonal +1
    sup_diag2 = sparse(n,1); % sup-diagonal +2

    % --- Special elements --- %
    main_diag(1) = 1200 * x(1)^2 - 400 * x(2) + 2;
    sup_diag1(2) = -400 * x(1);
    main_diag(2) = 1101 / 5;
    sup_diag2(2) = 99 / 5;
    main_diag(n-1) = 1080 * x(n-1)^2 - 360 * x(n) + 2;
    sup_diag1(n) = -360 * x(n-1);
    main_diag(n) = 1001 / 5;

    % --- odd i = 3,5,...,n-3 --- %
    idx_odd = 3:2:n-3;
    main_diag(idx_odd) = 2280 * x(idx_odd).^2 + 4 - 760 * x(idx_odd+1);
    sup_diag1(idx_odd+1) = -760 * x(idx_odd);

    % --- even i = 4,6,...,n-2 --- %
    main_diag(idx_odd+1) = 2102 / 5;
    sup_diag2(idx_odd+3) = 99 / 5;

    sub_diag1 = [sup_diag1(2:n); 0];
    sub_diag2 = [sup_diag2(3:n); zeros(2,1)];

    % Final construction of the sparse hessian matrix
    H = spdiags([sub_diag2, sub_diag1, main_diag, sup_diag1, sup_diag2],
        ...
                [-2, -1, 0, 1, 2], n, n);
end
hess_wood = @(x,h) hess_wood_fun(x,h);

% Implementation of centered finite differences approximation for gradf
   and
% hessf:
function g = gradf_finite_diff_wood_fun(x,h)
    n = length(x);
    g = sparse(n,1);
```

```
    g(1) = 400 * x(1)^3 - 400 * x(1) * x(2) + 2 * x(1) - 2 + 400 * h(1)
       ^2 * x(1);
    g(2) = -200 * x(1)^2 + (1101/5) * x(2) + 99/5 * x(4) - 40;

    g(n-1) = 360 * x(n-1)^3 - 360 * x(n-1) * x(n) + 2 * x(n-1) - 2 + 180
        * h(n-1) * x(n-1)^2;
    g(n) = (99/5) * x(n-2) - 180 * x(n-1)^2 + (1001/5) * x(n) - 40;

    % elements 4:2:n-2
    g(4:2:n-2) = (99/5) * x(2:2:n-4) - 380 * x(3:2:n-3).^2 + (2102/5) *
       x(4:2:n-2) + (99/5) * x(6:2:n) - 80;

    % elements 3:2:n-3
    g(3:2:n-3) = 760 * x(3:2:n-3).^3 + 4 * x(3:2:n-3) - 760 * x(3:2:n-3)
        .* x(4:2:n-2) - 4 + 760 * h(3:2:n-3) .* x(3:2:n-3);
end
gradf_finite_diff_wood = @(x,h) gradf_finite_diff_wood_fun(x,h);

function H = hess_finite_diff_wood_fun(x,h)
    n = length(x);

    % Prealloca vettori per le diagonali
    main_diag = sparse(n,1);
    sup_diag1 = sparse(n,1); % sopra diagonale a +1
    sup_diag2 = sparse(n,1); % sopra diagonale a +2

    % --- Elementi speciali ---
    main_diag(1) = 1200 * x(1)^2 - 400 * x(2) + 2 + 200 * h(1)^2;
    sup_diag1(2) = -400 * x(1) - 200 * h(1);
    main_diag(2) = 1101 / 5;
    sup_diag2(4) = 99 / 5;
    main_diag(n-1) = 1080 * x(n-1)^2 - 360 * x(n) + 2 + 180 * h(n-1)^2;
    sup_diag1(n) = -360 * x(n-1) - 180 * h(n);
    main_diag(n) = 1001 / 5;

    % --- Loop sui dispari i = 3,5,...,n-3 ---
    idx_odd = 3:2:n-3;
    main_diag(idx_odd) = 2280 * x(idx_odd).^2 + 4 - 760 * x(idx_odd+1) +
        380 * h(idx_odd).^2;
    sup_diag1(idx_odd+1) = -760 * x(idx_odd) - 380 * h(idx_odd);

    % --- Loop sui pari i = 4,6,...,n-2 ---
    main_diag(idx_odd+1) = 2102 / 5;
    sup_diag2(idx_odd+3) = 99 / 5;

    sub_diag1 = [sup_diag1(2:n); 0];
    sub_diag2 = [sup_diag2(3:n); zeros(2,1)];

    % Costruzione finale della matrice Hessiana sparsa
    H = spdiags([sub_diag2, sub_diag1, main_diag, sup_diag1, sup_diag2],
        ...
                [-2, -1, 0, 1, 2], n, n);
end
hess_finite_diff_wood = @(x,h) hess_finite_diff_wood_fun(x,h);

n = 1:dim;
```

```
x2_wood = zeros(dim,1);
x2_wood(mod(n,2) == 1 & n <= 4) = -3;
x2_wood(mod(n,2) == 1 & n > 4) = -2;
x2_wood(mod(n,2) == 0 & n <= 4) = -1;
x2_wood(mod(n,2) == 0 & n > 4) = 0;
x2_opt = ones(dim,1);
l_bound = x2_wood - 1;
u_bound = x2_wood + 1;
M_ten_initial_points = l_bound + (u_bound - l_bound) .* rand(dim, 10);
starting_points = [x2_wood, M_ten_initial_points];
```

Powell chained

```
% function definition
f3_powell = @(x) sum(arrayfun(@(j) ...
        (x(2*j-1) + 10*x(2*j))^2 + ...
        5*(x(2*j+1) - x(2*j+2))^2 + ...
        (x(2*j) - 2*x(2*j+1))^4 + ...
        10*(x(2*j-1) - x(2*j+2))^4, ...
        1:(length(x)-2)/2));

% gradient of the function
function g = grad_powell_fun(x,h)
    n = length(x);
    g = sparse(n,1);

    g(1) = 40 * x(1)^3 + 2 * x(1) + 20 * x(2) - 120 * x(1)^2 * x(4) +
       120 * x(1) * x(4)^2 - 40 * x(4)^3;
    g(2) = 20 * x(1) + 200 * x(2) + 4 * x(2)^3 - 24 * x(2)^2 * x(3) + 48
        * x(2) * x(3)^2 - 32 * x(3)^3;

    idx_odd = 3:2:n-3;

    g(idx_odd) = 12 * x(idx_odd) + 10 * x(idx_odd-1) - 8 * x(idx_odd-1)
       .^3 + 48 * x(idx_odd-1).^2 .* x(idx_odd) ...
        - 96 * x(idx_odd-1) .* x(idx_odd).^2 + 104 * x(idx_odd).^3 - 120
            * x(idx_odd).^2 .* x(idx_odd+3) + 120 * x(idx_odd) .* x(
           idx_odd+3).^2 ...
        - 40 * x(idx_odd+3).^3;

    g(idx_odd+1) = 10 * x(idx_odd) + 210 * x(idx_odd+1) - 40 * x(idx_odd
       -2).^3 + ...
        120 * x(idx_odd-2).^2 .* x(idx_odd+1) - 120 * x(idx_odd-2) .* x(
           idx_odd+1).^2 + ...
        44 * x(idx_odd+1).^3 - 24 * x(idx_odd+1).^2 .* x(idx_odd+2) + 48
            * x(idx_odd+1) .* x(idx_odd+2).^2 - 32 * x(idx_odd+2).^3;


    g(n-1) = 10 * x(n-1) - 10 * x(n) - 8 * x(n-2)^3 + 48 * x(n-2)^2 * x(
       n-1) - 96 * x(n-2) * x(n-1)^2 + 64 * x(n-1)^3;
    g(n) = - 10 * x(n-1) + 10 * x(n) - 40 * x(n-3)^3 + 120 * x(n-3)^2 *
       x(n) - 120 * x(n-3) * x(n)^2 + 40 * x(n)^3;
end
grad_powell = @(x,h) grad_powell_fun(x,h);
```

```
% Hessian matrix of the function:
function H = hess_powell_fun(x,h)
    n = length(x);

    % Pre-allocate vectors for diagonals
    main_diag = sparse(n,1);
    sup_diag1 = sparse(n,1); % sup-diagonal +1
    sup_diag3 = sparse(n,1); % sup-diagonal +3

    % Diagonal elements
    main_diag(1) = 120 * x(1)^2 + 2 - 240 * x(1) * x(4) + 120 * x(4)^2;
    main_diag(2) = 200 - 48 * x(2) * x(3) + 48 * x(3)^2 + 12 * x(2)^2;
    main_diag(n-1) = 10 + 48 * x(n-2)^2 - 192 * x(n-2) * x(n-1) + 192 *
       x(n-1)^2;
    main_diag(n) = 10 + 120 * x(n-3)^2 - 240 * x(n-3) * x(n) + 120 * x(n
       )^2;

    % --- Special elements --- %
    sup_diag1(2) = 20;
    sup_diag3(4) = -120 * x(1)^2 + 240 * x(1) * x(4) - 120 * x(4)^2;
    sup_diag1(3) = -24 * x(2)^2 + 96 * x(2) * x(3) - 96 * x(3)^2;
    sup_diag1(n) = -10;

    % --- odd i = 3,5,...,n-3 --- %
    idx_odd = 3:2:n-3;
    main_diag(idx_odd) = 12 + 48 * x(idx_odd-1).^2 - 192 * x(idx_odd-1)
       .* x(idx_odd) + 312 * x(idx_odd).^2 - 240 * x(idx_odd) .* x(
       idx_odd+3) + 120 * x(idx_odd+3).^2;
    sup_diag1(idx_odd+1) = 10;
    sup_diag3(idx_odd+3) = -120 * x(idx_odd).^2 + 240 * x(idx_odd) .* x(
       idx_odd+3) - 120 * x(idx_odd+3).^2;

    % --- even i = 4,6,...,n-2 --- %
    main_diag(idx_odd+1) = 210 + 120 * x(idx_odd-2).^2 - 240 * x(idx_odd
       -2) .* x(idx_odd+1) + 132 * x(idx_odd+1).^2 - 48 * x(idx_odd+1)
       .* x(idx_odd+2) + 48 * x(idx_odd+2).^2;
    sup_diag1(idx_odd+2) = -24 * x(idx_odd+1).^2 + 96 * x(idx_odd+1) .*
       x(idx_odd+2) - 96 * x(idx_odd+2).^2;
    sup_diag1(n-1) = -24 * x(n-2)^2 + 96 * x(n-2) * x(n-1) - 96 * x(n-1)
       ^2;

    sub_diag1 = [sup_diag1(2:n); 0];
    sub_diag3 = [sup_diag3(4:n); zeros(3,1)];

    % Final construction of the sparse hessian matrix
    H = spdiags([sub_diag3, sub_diag1, main_diag, sup_diag1, sup_diag3],
        ...
                [-3, -1, 0, 1, 3], n, n);
end
hess_powell = @(x,h) hess_powell_fun(x,h);

% Implementation of centered finite differences approximation for gradf
   and
% hessf:
function g = gradf_finite_diff_powell_fun(x,h)
    n = length(x);
```

```
    g = sparse(n,1);
    g(1) = 40 * x(1)^3 + 2 * x(1) + 20 * x(2) - 120 * x(1)^2 * x(4) +
       120 * x(1) * x(4)^2 - 40 * x(4)^3 + 40 * h(1)^2 * x(1);
    g(2) = 20 * x(1) + 200 * x(2) + 4 * x(2)^3 - 24 * x(2)^2 * x(3) + 48
        * x(2) * x(3)^2 - 32 * x(3)^3 - 8 * h(2)^2 * x(3);

    idx_odd = 3:2:n-3;

    % elements 3:2:n-3
    g(idx_odd) = 12 * x(idx_odd) + 10 * x(idx_odd+1) - 8 * x(idx_odd-1)
       .^3 + 48 * x(idx_odd-1).^2 .* x(idx_odd) - ...
        96 * x(idx_odd-1) .* x(idx_odd).^2 + 104 * x(idx_odd).^3 - 120 *
            x(idx_odd).^2 .* x(idx_odd+3) + 120 * x(idx_odd) .* x(
           idx_odd+3).^2 - ...
        40 * x(idx_odd+3).^3 + 104 * h(idx_odd).^2 .* x(idx_odd) - 40 *
           h(idx_odd).^2 .* x(idx_odd+2);

    % elements 4:2:n-2
    g(idx_odd+1) = 10 * x(idx_odd) + 210 * x(idx_odd+1) - 40 * x(idx_odd
       -2).^3 + ...
        120 * x(idx_odd-2).^2 .* x(idx_odd+1) - 120 * x(idx_odd-2) .* x(
           idx_odd+1).^2 + ...
        44 * x(idx_odd+1).^3 - 24 * x(idx_odd+1).^2 .* x(idx_odd+2) + 48
            * x(idx_odd+1) .* x(idx_odd+2).^2 - 32 * x(idx_odd+2).^3 +
           ...
        - 40 * h(idx_odd+1).^2 .* x(idx_odd-2) + 40 * h(idx_odd+1).^2 .*
            x(idx_odd+1) - 16 * h(idx_odd+1).^2 .* x(idx_odd+2);

    g(n-1) =  10 * x(n-1) - 10 * x(n) - 8 * x(n-2)^3 + 48 * x(n-2)^2 * x
       (n-1) - 96 * x(n-2) * x(n-1)^2 + 64 * x(n-1)^3 + 64 * h(n-1)^2 *
       x(n-1);
    g(n) = - 10 * x(n-1) + 10 * x(n) - 40 * x(n-3)^3 + 120 * x(n-3)^2 *
       x(n) - 120 * x(n-3) * x(n)^2 + 40 * x(n)^3 - 40 * h(n)^2 * x(n-3)
        + 40 * h(n)^2 * x(n);

end
gradf_finite_diff_powell = @(x,h) gradf_finite_diff_powell_fun(x,h);

function H = hess_finite_diff_powell_fun(x,h)
    n = length(x);

    % Pre-allocate vectors for diagonals
    main_diag = sparse(n,1);
    sup_diag1 = sparse(n,1); % sup-diagonal +1
    sup_diag3 = sparse(n,1); % sup-diagonal +3

    % Diagonal elements
    main_diag(1) = 120 * x(1)^2 + 2 - 240 * x(1) * x(4) + 120 * x(4)^2 +
        20 * h(1)^2;
    main_diag(2) = 200 - 48 * x(2) * x(3) + 48 * x(3)^2 + 12 * x(2)^2 +
       2 * h(2)^2;
    main_diag(n-1) = 10 + 48 * x(n-2)^2 - 192 * x(n-2) * x(n-1) + 192 *
       x(n-1)^2 + 2 * h(n-1)^2;
    main_diag(n) = 10 + 120 * x(n-3)^2 - 240 * x(n-3) * x(n) + 120 * x(n
       )^2 + 20 * h(n)^2;
```

```
    % --- Special elements --- %
    sup_diag1(2) = 20;
    sup_diag3(4) = -120 * x(1)^2 + 240 * x(1) * x(4) - 120 * x(4)^2 - 20
        * h(4)^2;
    sup_diag1(3) = -24 * x(2)^2 + 96 * x(2) * x(3) - 96 * x(3)^2 -16 * h
       (3)^2 - 48 * h(3) * x(3) + 24 * h(2) * x(2);
    sup_diag1(n) = -10;

    % --- odd i = 3,5,...,n-3 --- %
    idx_odd = 3:2:n-3;
    main_diag(idx_odd) = 12 + 48 * x(idx_odd-1).^2 - 192 * x(idx_odd-1)
       .* x(idx_odd) + 312 * x(idx_odd).^2 - 240 * x(idx_odd) .* x(
       idx_odd+3) + 120 * x(idx_odd+3).^2 + 22 * h(idx_odd).^2;
    sup_diag1(idx_odd+1) = 10;
    sup_diag3(idx_odd+3) = -120 * x(idx_odd).^2 + 240 * x(idx_odd) .* x(
       idx_odd+3) - 120 * x(idx_odd+3).^2 - 20 * h(idx_odd+3).^2;

    % --- even i = 4,6,...,n-2 --- %
    main_diag(idx_odd+1) = 210 + 120 * x(idx_odd-2).^2 - 240 * x(idx_odd
       -2) .* x(idx_odd+1) + 132 * x(idx_odd+1).^2 - 48 * x(idx_odd+1)
       .* x(idx_odd+2) + 48 * x(idx_odd+2).^2 + 22 * h(idx_odd+1).^2;
    sup_diag1(idx_odd+2) = -24 * x(idx_odd+1).^2 + 96 * x(idx_odd+1) .*
       x(idx_odd+2) - 96 * x(idx_odd+2).^2 - 48 * h(idx_odd+2).*x(
       idx_odd+2) + 24 * h(idx_odd+1) .* x(idx_odd+1)- 16 * h(idx_odd+2)
       .^2;
    sup_diag1(n-1) = -24 * x(n-2)^2 + 96 * x(n-2) * x(n-1) - 96 * x(n-1)
       ^2 - 16 * h(n-1)^2 - 48 * h(n-1) * x(n-1) + 24 * h(n-2) * x(n-2);

    sub_diag1 = [sup_diag1(2:n); 0];
    sub_diag3 = [sup_diag3(4:n); zeros(3,1)];

    % Final construction of the sparse hessian matrix
    H = spdiags([sub_diag3, sub_diag1, main_diag, sup_diag1, sup_diag3],
        ...
                [-3, -1, 0, 1, 3], n, n);
end
hess_finite_diff_powell = @(x,h) hess_finite_diff_powell_fun(x,h);

function H = hess_finite_diff_powell_fun_h_var(x,h)
    n = length(x);

    % Pre-allocate vectors for diagonals
    main_diag = sparse(n,1);
    sup_diag1 = sparse(n,1); % sup-diagonal +1
    sup_diag3 = sparse(n,1); % sup-diagonal +3

    % Diagonal elements
    main_diag(1) = 120 * x(1)^2 + 2 - 240 * x(1) * x(4) + 120 * x(4)^2 +
        20 * h(1)^2;
    main_diag(2) = 200 - 48 * x(2) * x(3) + 48 * x(3)^2 + 12 * x(2)^2 +
       2 * h(2)^2;
    main_diag(n-1) = 10 + 48 * x(n-2)^2 - 192 * x(n-2) * x(n-1) + 192 *
       x(n-1)^2 + 2 * h(n-1)^2;
    main_diag(n) = 10 + 120 * x(n-3)^2 - 240 * x(n-3) * x(n) + 120 * x(n
       )^2 + 20 * h(n)^2;
```

```
    % --- Special elements --- %
    sup_diag1(2) = 20;
    sup_diag3(4) = -120 * x(1)^2 + 240 * x(1) * x(4) - 120 * x(4)^2 - 40
        * h(1)^2 - 40 * h(4)^2 + 60 * h(1) * h(4) + 120 * h(4) * (x(1) -
        x(4)) - 120 * h(1) * (x(1) - x(4));
    sup_diag1(3) = -24 * x(2)^2 + 96 * x(2) * x(3) - 96 * x(3)^2 -16 * h
       (3)^2 - 48 * h(3) * x(3) + 24 * h(2) * x(2);
    sup_diag1(n) = -10;

    % --- odd i = 3,5,...,n-3 --- %
    idx_odd = 3:2:n-3;
    main_diag(idx_odd) = 12 + 48 * x(idx_odd-1).^2 - 192 * x(idx_odd-1)
       .* x(idx_odd) + 312 * x(idx_odd).^2 - 240 * x(idx_odd) .* x(
       idx_odd+3) + 120 * x(idx_odd+3).^2 + 22 * h(idx_odd).^2;
    sup_diag1(idx_odd+1) = 10;
    sup_diag3(idx_odd+3) = -120 * x(idx_odd).^2 + 240 * x(idx_odd) .* x(
       idx_odd+3) - 120 * x(idx_odd+3).^2 - 40 * h(idx_odd).^2 - 40 * h(
       idx_odd+3).^2 ...
        + 60 * h(idx_odd) .* h(idx_odd+3) + 120 * h(idx_odd+3) .* (x(
           idx_odd) - x(idx_odd+3)) - 120 * h(idx_odd) .* (x(idx_odd) -
           x(idx_odd+3));

    % --- even i = 4,6,...,n-2 --- %
    main_diag(idx_odd+1) = 210 + 120 * x(idx_odd-2).^2 - 240 * x(idx_odd
       -2) .* x(idx_odd+1) + 132 * x(idx_odd+1).^2 - 48 * x(idx_odd+1)
       .* x(idx_odd+2) + 48 * x(idx_odd+2).^2 + 22 * h(idx_odd+1).^2;
    sup_diag1(idx_odd+2) = -24 * x(idx_odd+1).^2 + 96 * x(idx_odd+1) .*
       x(idx_odd+2) - 96 * x(idx_odd+2).^2 ...
        -16 * h(idx_odd+2).^2 - 48 * h(idx_odd+2) .* x(idx_odd+2) + 24 *
            h(idx_odd+1) .* x(idx_odd+1);

    sup_diag1(n-1) = -24 * x(n-2)^2 + 96 * x(n-2) * x(n-1) - 96 * x(n-1)
       ^2 - 8 * h(n-2)^2 - 32 * h(n-1)^2 + 24 * h(n-2) * h(n-1) + 24 * x
       (n-2) * (2*h(n-1) - h(n-2)) + 48 * x(n-1) * (h(n-2) - 2 * h(n-1))
       ;

    sub_diag1 = [sup_diag1(2:n); 0];
    sub_diag3 = [sup_diag3(4:n); zeros(3,1)];

    % Final construction of the sparse hessian matrix
    H = spdiags([sub_diag3, sub_diag1, main_diag, sup_diag1, sup_diag3],
        ...
                [-3, -1, 0, 1, 3], n, n);
end
hess_finite_diff_powell_h_var = @(x,h) hess_finite_diff_powell_fun_h_var
   (x,h);

n = 1:dim;
x3_powell = zeros(dim, 1);
x3_powell(mod(n,4) == 1) = 3;
x3_powell(mod(n,4) == 2) = -1;
x3_powell(mod(n,4) == 3) = 0;
x3_powell(mod(n,4) == 0) = 1;
x3_opt = zeros(dim,1);

l_bound = x3_powell - 1;
```

```
u_bound = x3_powell + 1;
M_ten_initial_points = l_bound + (u_bound - l_bound) .* rand(dim, 10);
starting_points = [x3_powell, M_ten_initial_points];
```

Test with exact derivatives (example with chained Rosenbrock)

```
rho = 0.9;
tolgrad = 10^(-2);
c1 = 10^(-4);
btmax = 75;
% kmax = 1500;  % for dim = 10^3
% kmax = 15000; % for dim = 10^4
kmax = 150000;  % for dim = 10^5

flag_h = 0;
h = 0;

diff_ros = zeros(11,1);
time_ros = zeros(11, 1);
iter_ros = zeros(11, 1);
rate_ros = zeros(1,11);
failure_ros = zeros(1,11);
for index=1:11
    tic;
    [xk, fk, gradfk_norm, k, ...
        xseqk, btseq, failure] = modified_newton_method(starting_points
           (:,index), f1_ros, grad_f1_ros, hess_f1_ros, kmax, ...
        tolgrad, c1, btmax, dim, rho, h, flag_h);
    time_ros(index) = toc;
    iter_ros(index) = k;
    diff_ros(index) = norm(xk - x1_opt,2);
    rate_ros(index) = compute_exp_rate_conv_multi(xseqk);
    failure_ros(index) = failure;
    disp(gradfk_norm)
end
final_rate_exact_deriv_ros = mean(rate_ros, 'omitnan');
```

Test with finite difference (example with chained Rosenbrock)

```
% Finite differences
% parameter used for finite differences
h_vec = 10.^[-2, -4, -6, -8, -10, -12];

% Testing different values of h, fixed and varying from point to point
flag_h2 = 1; % Hessian matrix computed with finite differences and h
   variable

num_point = 11;

time1_finite_diff_ros = zeros(length(h_vec),num_point);
rate_h_ros = zeros(1, num_point);
final_rate_ros = zeros(1, length(h_vec));
norm_err_conv_ros = zeros(num_point, length(h_vec));
iters_ros = zeros(num_point, length(h_vec));
```

```
failure1_ros = zeros(length(h_vec),num_point);

time2_finite_diff_ros = zeros(length(h_vec),num_point);
rate_h2_ros = zeros(1, num_point);
final_rate2_ros = zeros(1,length(h_vec));
norm_err_conv2_ros = zeros(num_point, length(h_vec));
iters2_ros = zeros(num_point, length(h_vec));
failure2_ros = zeros(length(h_vec),num_point);

final_point_temporary_ros = zeros(dim,11);
final_point_ros = cell(1,11);
final_point_temporary2_ros = zeros(dim,11);
final_point2_ros = cell(1,11);

for i=1:length(h_vec)
    disp(i)
    for index=1:num_point
        disp(index)
        tic;
        [xk, fk, gradfk_norm, k, ...
            xseqk, btseq, failure] = modified_newton_method(
               starting_points(:,index), f1_ros, ...

               gradf_finite_diff_ros, hess_finite_diff_ros_h_var, kmax
               ,...
               tolgrad, c1, btmax, dim, rho, h_vec(i), flag_h);
        time1_finite_diff_ros(i, index) = toc;
        iters_ros(index, i) = k;
        norm_err_conv_ros(index,i) = norm(x1_opt - xk,2);
        rate_h_ros(index) = compute_exp_rate_conv_multi(xseqk);
        final_point_temporary_ros(:,index) = xk;
        failure1_ros(i, index) = failure;

        disp(index)
        tic;
        [xk2, fk2, gradfk_norm2, k2, ...
            xseqk2, btseq2, failure2] = modified_newton_method(
               starting_points(:,index), f1_ros,...
            grad_f1_ros, hess_finite_diff_ros_h_var, kmax,...
            tolgrad, c1, btmax, dim, rho, h_vec(i), flag_h2);
        time2_finite_diff_ros(i, index) = toc;
        iters2_ros(index, i) = k2;
        norm_err_conv2_ros(index,i) = norm(x1_opt - xk2,2);
        rate_h2_ros(index) = compute_exp_rate_conv_multi(xseqk2);
        final_point_temporary2_ros(:,index) = xk2;
        failure2_ros(i, index) = failure2;
    end
    final_rate_ros(i) = mean(rate_h_ros, 'omitnan');
    final_rate2_ros(i) = mean(rate_h2_ros, 'omitnan');
    final_point_ros{i} = final_point_temporary_ros;
    final_point2_ros{i} = final_point_temporary2_ros;
end
```

Export in an output file *output_ mat*

```
filename = 'output.mat';
base_vars = {'time_ros', 'diff_ros', 'iter_ros', '
   final_rate_exact_deriv_ros', 'failure_ros'};
base_vars = {'time1_finite_diff_ros', 'norm_err_conv_ros', 'iters_ros',
   'final_rate_ros', 'failure1_ros'};
base_vars = {'time2_finite_diff_ros', 'norm_err_conv2_ros', 'iters2_ros'
   , 'final_rate2_ros', 'failure2_ros'};

for i = 1:length(base_vars)
    base_name = base_vars{i};
    new_name = sprintf('%s_%d', base_name, dim);

    val = eval(base_name);

    eval([new_name ' = val;']);

    if exist(filename, 'file')
        save(filename, new_name, '-append');
    else
        save(filename, new_name);
    end
end
```

B.5.10 Plot of results

```
% Exercise 3, analysis of Modified Newton method
% This script implements and tests the Modified Newton method
% simulations are performed to analyze the convergence,
% computational cost, stagnation and empirical rates of convergence of
   the method in
% various dimensions (10^3,10^4,10^5) and multiple initial points (one
   suggested from
% PDF and 10 points from the ipercube).
%% Compute graphics in order to compare results in different dimension
   %%
clc
clear all
load("output.mat")

% Rosenbrock chained %
create_bar_plot(mean(time_ros_1000), mean(time_ros_10000), mean(
   time_ros_100000), 'time', 'ros')
create_bar_plot(mean(diff_ros_1000), mean(diff_ros_10000), mean(
   diff_ros_100000), 'diff', 'ros')
create_bar_plot(mean(iter_ros_1000), mean(iter_ros_10000), mean(
   iter_ros_100000), 'iter', 'ros')
create_bar_plot(mean(final_rate_exact_deriv_ros_1000), mean(
   final_rate_exact_deriv_ros_10000), mean(
   final_rate_exact_deriv_ros_100000), 'rate', 'ros')

% Wood chained %
create_bar_plot(mean(time_wood_1000), mean(time_wood_10000), mean(
   time_wood_100000), 'time', 'wood')
create_bar_plot(mean(diff_wood_1000), mean(diff_wood_10000), mean(
   diff_wood_100000), 'diff', 'wood')
```

```
create_bar_plot(mean(iter_wood_1000), mean(iter_wood_10000), mean(
    iter_wood_100000), 'iter', 'wood')
create_bar_plot(mean(final_rate_exact_deriv_wood_1000), mean(
    final_rate_exact_deriv_wood_10000), mean(
    final_rate_exact_deriv_wood_100000), 'rate', 'wood')

% Powell chained %
create_bar_plot(mean(time_pow_1000), mean(time_pow_10000), mean(
    time_pow_100000), 'time', 'pow')
create_bar_plot(mean(diff_pow_1000), mean(diff_pow_10000), mean(
    diff_pow_100000), 'diff', 'pow')
create_bar_plot(mean(iter_pow_1000), mean(iter_pow_10000), mean(
    iter_pow_100000), 'iter', 'pow')
create_bar_plot(mean(final_rate_exact_deriv_pow_1000), mean(
    final_rate_exact_deriv_pow_10000), mean(
    final_rate_exact_deriv_pow_100000), 'rate', 'pow')

%% Finite differences %%
% (1) fixed h %
clc
clear all
load("output.mat")

% Rosenbrock chained %
create_bar_plot(mean(time1_finite_diff_ros_1000,2), mean(
    time1_finite_diff_ros_10000,2), mean(time1_finite_diff_ros_100000,2),
     'time', 'ros')
create_bar_plot(mean(norm_err_conv_ros_1000), mean(
    norm_err_conv_ros_10000), mean(norm_err_conv_ros_100000), 'diff', '
    ros')
create_bar_plot(mean(iters_ros_1000), mean(iters_ros_10000), mean(
    iters_ros_100000), 'iter', 'ros')
create_bar_plot(final_rate_ros_1000, final_rate_ros_10000,
    final_rate_ros_100000, 'rate', 'ros')

% Wood chained %
create_bar_plot(mean(time1_finite_diff_wood_1000,2), mean(
    time1_finite_diff_wood_10000,2), mean(time1_finite_diff_wood_100000
    ,2), 'time', 'wood')
create_bar_plot(mean(norm_err_conv_wood_1000), mean(
    norm_err_conv_wood_10000), mean(norm_err_conv_wood_100000), 'diff', '
    wood')
create_bar_plot(mean(iters_wood_1000), mean(iters_wood_10000), mean(
    iters_wood_100000,2), 'iter', 'wood')
create_bar_plot(final_rate_wood_1000, final_rate_wood_10000,
    final_rate_wood_100000, 'rate', 'wood')

% Powell chained %
create_bar_plot(mean(time1_finite_diff_pow_1000,2), mean(
    time1_finite_diff_pow_10000,2), mean(time1_finite_diff_pow_100000,2),
     'time', 'pow')
create_bar_plot(mean(norm_err_conv_pow_1000), mean(
    norm_err_conv_pow_10000), mean(norm_err_conv_pow_100000), 'diff', '
    pow')
create_bar_plot(mean(iters_pow_1000), mean(iters_pow_10000), mean(
    iters_pow_100000), 'iter', 'pow')
```

```
create_bar_plot(final_rate_pow_1000, final_rate_pow_10000,
   final_rate_pow_100000, 'rate', 'pow')

%% Finite differences %%
% (2) variable h %
clc
clear all
load("output.mat")

% Rosenbrock chained %
create_bar_plot(mean(time2_finite_diff_ros_1000,2), mean(
   time2_finite_diff_ros_10000,2), mean(time2_finite_diff_ros_100000,2),
    'time', 'ros')
create_bar_plot(mean(norm_err_conv2_ros_1000), mean(
   norm_err_conv2_ros_10000), mean(norm_err_conv2_ros_100000), 'diff', '
   ros')
create_bar_plot(mean(iters2_ros_1000), mean(iters2_ros_10000), mean(
   iters2_ros_100000), 'iter', 'ros')
create_bar_plot(final_rate2_ros_1000, final_rate2_ros_10000,
   final_rate2_ros_100000, 'rate', 'ros')


% Wood chained %
create_bar_plot(mean(time2_finite_diff_wood_1000,2), mean(
   time2_finite_diff_wood_10000,2), mean(time2_finite_diff_wood_100000
   ,2), 'time', 'wood')
create_bar_plot(mean(norm_err_conv2_wood_1000,2), mean(
   norm_err_conv2_wood_10000,2), mean(norm_err_conv2_wood_100000,2), '
   diff', 'wood')
create_bar_plot(mean(iters2_wood_1000,2), mean(iters2_wood_10000,2),
   mean(iters2_wood_100000,2), 'iter', 'wood')
create_bar_plot(final_rate2_wood_1000, final_rate2_wood_10000,
   final_rate2_wood_100000, 'rate', 'wood')

% Powell chained %
create_bar_plot(mean(time2_finite_diff_pow_1000'), mean(
   time2_finite_diff_pow_10000'), mean(time2_finite_diff_pow_100000'), '
   time', 'pow')
create_bar_plot(mean(norm_err_conv2_pow_1000), mean(
   norm_err_conv2_pow_10000), mean(norm_err_conv2_pow_100000), 'diff', '
   pow')
create_bar_plot(mean(iters2_pow_1000), mean(iters2_pow_10000), mean(
   iters2_pow_100000), 'iter', 'pow')
create_bar_plot(final_rate2_pow_1000, final_rate2_pow_10000,
   final_rate2_pow_100000, 'rate', 'pow')
```

Bibliography

[1] Michael Baudin, *Nelder-Mead User's Manual*, https://www.scilab.org/sites/default/files/neldermead.pdf

[2] Jorge Nocedal and Sthephen J. Wright, *Numerical optimization*