Politecnico di Torino

Numerical optimization for large scale problems

Assignment on Unconstrained Optimization

Politecnico di Torino, A.Y. 2024/2025

Bono Giorgio s343572
Camolese Claudio s344788
Di Felice Andrea s337517

Contents

1 Overview

The purpose of the project is to become familiar with descent methods for finding the minimum of a function. The chosen methods were applied to four test functions to analyse the results in different dimensions.
For each function, there is a predefined starting point. Additionally, the functions are also analysed for 10 new starting points in the vicinity of the predefined starting point.

2 Methods

2.1 Nelder-Mead method

The first method presented is the *Nelder-Mead* method. This method does not take into account the derivatives of a function but instead considers a simplex. k-simplex is a k-dimensional polytope that is the convex hull of its k + 1 vertices.
The minimum is reached through four operations that are performed under certain conditions: reflection, expansion, contraction and shrinkage.

$$\begin{cases} x_r^k = \bar{x}_k + \rho(\bar{x}_k - x_{n+1}^k) \\ x_e^k = \bar{x}_k + \chi(x_r^k - \bar{x}_k) \\ x_c^k = \bar{x}_k - \gamma(\bar{x}_k - x_{n+1}^k) \\ x_i^{k+1} = x_1^k + \sigma(x_i^k - x_1^k) \end{cases} \tag{1}$$

where $\rho > 0$, $\chi > 1$, $0 < \gamma < 1$, $0 < \sigma < 1$ and $\bar{x}_k$ is the centroid of the best n points.
Let's see what these operations do in detail.

- **Reflection**: The worst point x_{n+1}^k of the simplex is reflected across the centroid of the remaining points to explore a potentially better solution.
- **Expansion**: If the reflection improves the objective significantly, the simplex is expanded further in that direction to accelerate convergence.
- **Contraction**: If reflection does not yield improvement, the simplex contracts towards the centroid to explore a smaller region.
- **Shrinkage**: If contraction fails, all points move closer to the best point, reducing the search space to refine the solution.

  The plot of this method can be seen in the following plots:

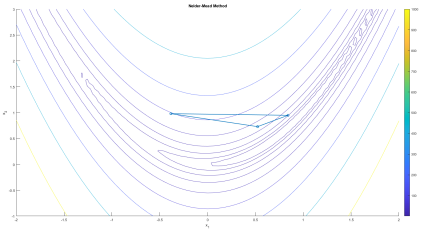

Figure 1: Initialization

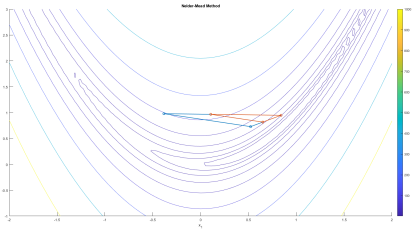

Figure 2: Shrinkage

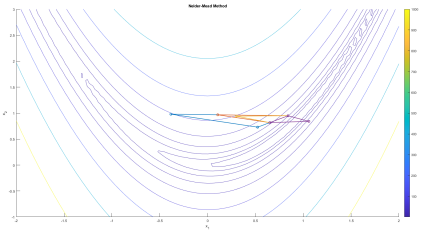

Figure 3: Reflection

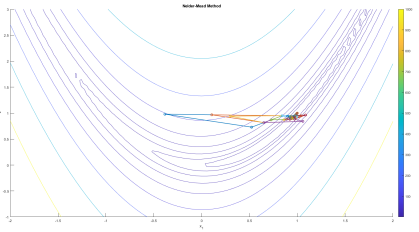

Figure 4: Convergence

The Nelder-Mead method is a zero-order method since it does not require information on the computation of the first or second derivatives of the function. This makes the method less computationally expensive but slower in convergence, as it does not exploit knowledge of the function's derivatives.

2.1.1 Fine-tuning

In the Nelder-Mead method, convergence strongly depends on the choice of parameters. For this reason, the parameters were carefully selected for each function by studying how the function decreases as the parameters change using random starting points different from x_0 given by the assignment. This process has been repeated N times and for each test, the mean of the bests parameter is memorized. This operation allows to decrease the dependence from the starting point of the method. This process has been done in the *Tuning.m* file (A.8).
The next plots are an example of how we have chosen the parameter ρ for the *extended Freudenstein and Roth function*.

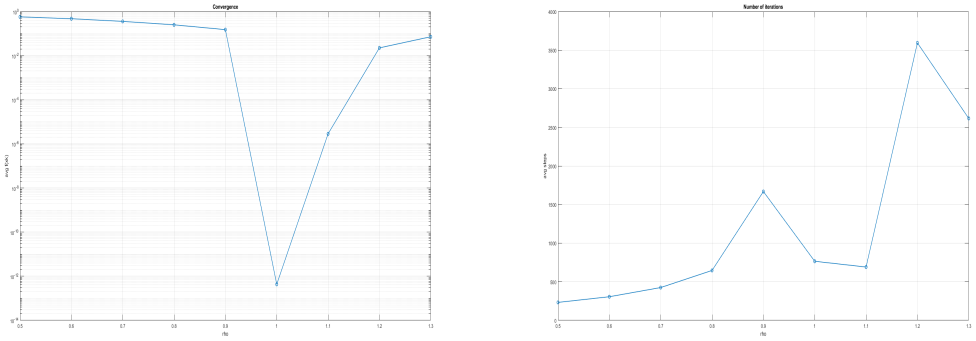

Figure 5: Convergence and iterations of ρ

As can be observed from the plots (5), the optimal choice of $\rho = 1$, as it ensures a better convergence and an acceptable number of iterations.

2.2 Modified Newton method

The *modified Newton* method is a variant of the standard Newton method. The Newton method builds a quadratic model for f around x^k.

$$m_k(p) = f(x^k) + \nabla f(x^k)^T p + \frac{1}{2} p^T \nabla^2 f(x^k) p \tag{2}$$

where p is defined as

$$p^k = \texttt{argmin}(m_k(p)) \Longrightarrow \nabla^2 f(x^k) p = -\nabla f(x^k) \tag{3}$$

However, the Newton method does not allow for the computation of p^k if $\nabla^2 f(x^k)$ is not positive definite. The remedy for this issue is to use the modified Newton method. The idea behind the modified Newton method is to replace the Hessian of the function, if it is not positive definite, with a matrix B_k that takes into account the Hessian with a slight perturbation E_k.

$$B_k = \nabla^2 f(x^k) + E_k \tag{4}$$

The perturbation must be such that B_k becomes positive definite while remaining small enough to preserve the second-order information of the function.
The perturbation matrix E_k is defined as:

$$\begin{cases} E_k = \tau_k I \\ \tau_k = max(0, \delta - \lambda_{min}(\nabla^2 f(x^k))) \end{cases} \tag{5}$$

In this way, if the starting Hessian is sufficiently positive definite, no correction is applied. Otherwise, if the smallest eigenvalue is negative, a correction is applied.

The Newton method is a second-order method, meaning it uses both gradient and Hessian information. As a result, it is computationally more expensive, but its convergence is quadratic.

In addition, in equation (4), the results are presented using both the exact derivatives solution of the function and the finite differences method.

3 Functions

In this section, the results obtained for the four test functions used are presented. For each function, the exact derivatives form of the Hessian matrix was studied in order to subsequently obtain more efficient code for computing the derivatives.

3.1 Rosenbrock function

The Rosenbrock function is defined as follows:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{6}$$

The function is to be tested only for a dimension equal to 2. It is necessary to apply the function with two different starting vectors: $x^{(0)} = (1.2, 1.2)$ and $x^{(0)} = (-1.2, 1)$.

3.2 Freudenstein and Roth function

The Freudenstein and Roth function is defined as follows:

$$F(x) = \frac{1}{2}\sum_{k=1}^{2} f_k^2(x) \tag{7}$$

$$f_k = x_k + ((5 - x_{k+1})x_{k+1} - 2)x_{k+1} - 13 \qquad , mod(k, 2) = 1 \tag{8}$$

$$f_k = x_{k-1} + ((x_{k+1} + 1)x_k - 14)x_k - 29 \qquad , mod(k, 2) = 0 \tag{9}$$

$$\bar{x_i} = 90, \quad mod(i, 2) = 1 \qquad \bar{x_i} = 60, \quad mod(i, 2) = 0 \tag{10}$$

The two dimensional plot of the function is the following:

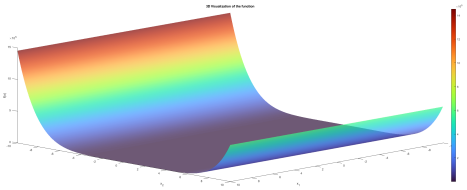

Figure 6: Freudenstein and Roth function 3D visualization

The Hessian matrix has this structure:

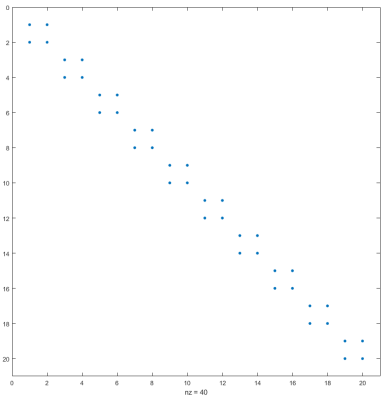

Figure 7: Hessian form of the *Freudenstein and Roth* function

As can be seen from (7), the Hessian has a form of a 2×2 block-diagonal matrix. Therefore, it can take advantage of its sparsity.

3.3 Problem 75

The problem 75 is defined as follows:

$$F(x) = \frac{1}{2}\sum_{k=1}^{2} f_k^2(x) \tag{11}$$

$$f_k = x_k - 1 \qquad , k = 1 \tag{12}$$

$$f_k = 10(k-1)(x_k - x_{k-1})^2 \qquad , 1 < k \leq n \tag{13}$$

$$\bar{x_i} = -1.2, \quad 1 < k < n \qquad x_l = -1, \quad l = n \tag{14}$$

The two dimensional plot of the function is the following:

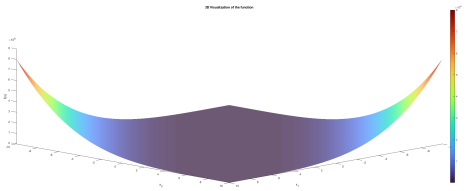

Figure 8: Problem 75 function

The Hessian matrix has this structure:

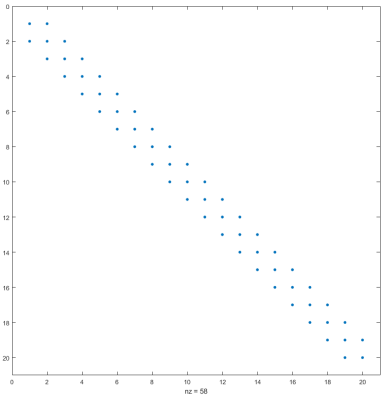

Figure 9: Hessian form of the *Problem 75* function

Has can be seen from (9), the Hessian has a form of a tridiagonal matrix. Therefore, it can take advantage of its sparsity.

3.4 Problem 76

The problem 76 is defined as follows:

$$F(x) = \frac{1}{2}\sum_{k=1}^{2} f_k^2(x) \tag{15}$$

$$f_k = x_k - \frac{x_{k+1}^2}{10} \qquad , 1 \leq k < n \tag{16}$$

$$f_k = x_k - \frac{x_1^2}{10} \qquad , k = n \tag{17}$$

$$\bar{x_l} = 2 \quad , l \geq 2 \tag{18}$$

The two dimensional plot of the function is the following:

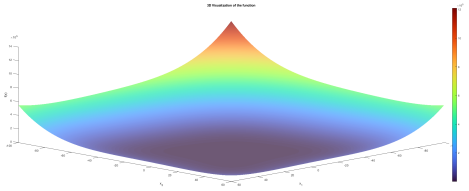

Figure 10: Problem 76 function

The Hessian matrix has the structure presented in (11).

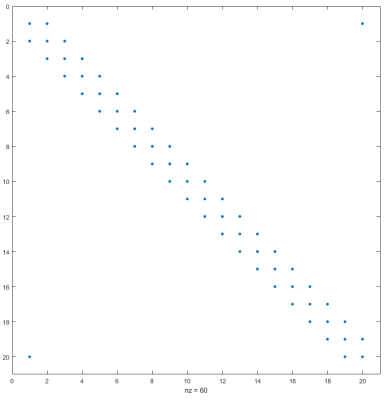

Figure 11: Hessian form of the *Problem 76* function

As can be seen from (11), the Hessian has a form of a tridiagonal matrix with a non zero element related to the $\frac{\partial}{\partial x_1}\frac{\partial}{\partial x_n}$ derivative. Therefore, it can take advantage of its sparsity.

4 Results

In this section, the results obtained for the functions mentioned in section (3) are presented. For each function, the results are analyzed both with and without preconditioning. Then, the behavior of the modified Newton method is examined using both exact derivatives and finite differences.

Preconditioning is a technique used to improve the convergence of iterative methods for solving linear systems. It involves transforming the original system into an equivalent one that has better numerical properties. A well-chosen preconditioner accelerates convergence and makes solving large systems more efficient. The chosen method is the *Preconditioned Conjugate Gradient* (pcg) since, given the nature of the Hessian matrix for the modified newton method, we are dealing with symmetric matrices. Preconditioner is used to improve convergence, and Incomplete Cholesky is a common choice for symmetric positive definite matrices. It produces a lower triangular matrix that serves as an effective preconditioner.

4.1 Nelder-Mead method

In this section, for each function, the behaviour of the *Nelder-Mead* is presented in different dimensions. For the Rosenbrock function the method is used just for two dimensions, while for the other functions the method is applied for 10, 25, 50 dimensions.
Then, for each function, the decreasing of the objective function and the increasing of the execution time with respect to the dimensions is also plotted.

4.1.1 Rosenbrock function

The results are the following:

x_0	$f(x_0)$	x_k	$f(x_k)$	Iterations	Time [s]
(1.2000, 1.2000)	5.8000	(1.000000, 1.000001)	2.89386×10^{-13}	70	0.14212
(-1.2000, 1.0000)	24.2000	(1.000000, 1.000000)	2.75443×10^{-13}	87	0.11985

Table 1: Results of the Rosenbrock function with different starting points in two dimensions

This can be also be shown in the following plots:

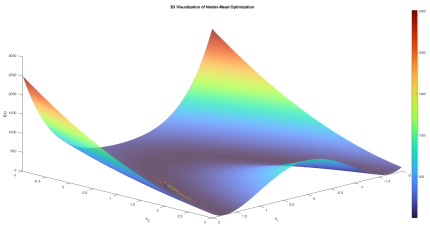

Figure 12: 3D plots with $x_0 = (1.2, 1.2)$

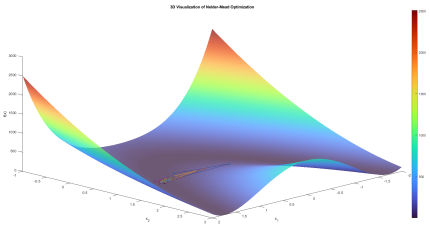

Figure 13: 3D plots with $x_0 = (-1.2, 1)$

It is also useful to analyse how we approach the minimum with respect to the current iteration.

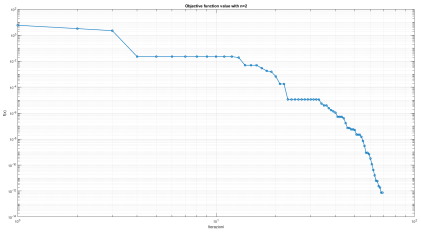

Figure 14: Objective function with $x_0 = (1.2, 1.2)$

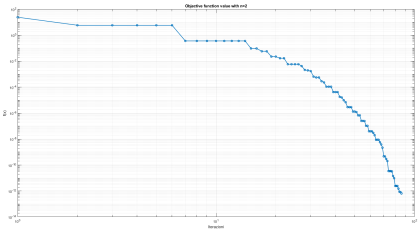

Figure 15: Objective function with $x_0 = (-1.2, 1)$

As can be observed from the graphs (14) and (15), the method allows finding the minimum of the function in a comparable time and with a slightly different number of iterations, despite the simplex starting from two different points.

4.1.2 Freudenstein and Roth function

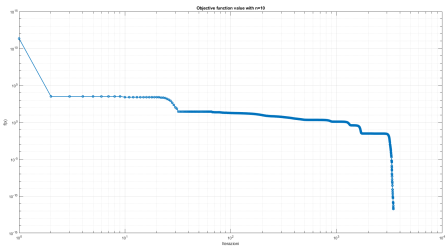

Figure 16: Objective function of Freudenstein and Roth function

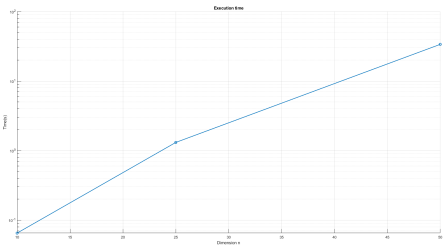

Figure 17: Execution time of Freudenstein and Roth function

In (16), the objective function strongly decreases when the minimum neighbourhood is reached from the method.

As can be seen from (17), the execution time is exponential with the respect to the increase of the dimension n, as we expected.

Point	$f(x_k)$	Iterations	Time (s)
Dimension $n = 10$			
x0	1.62122e-12	3457	0.0805
1	6.35417e-13	3504	0.0728
2	4.48282e-13	3004	0.0643
3	3.88043e-12	3576	0.0715
4	2.44965e-12	3333	0.0551
5	5.96587e-13	3521	0.0594
6	1.12249e-12	3870	0.0665
7	1.18532e-12	3230	0.0606
8	7.38954e-13	3505	0.0562
9	7.66771e-13	3713	0.0782
10	5.55962e-13	2871	0.0466
Dimension $n = 25$			
x0	1.16892e-12	28418	0.7733
1	3.48851e-12	26529	1.0061
2	1.38214e-12	24621	1.1512
3	2.70328e-12	27793	0.7088
4	1.94602e-12	27657	0.7553
5	5.52237e-12	28885	1.5216
6	1.4607e-12	27863	0.7069
7	3.37032e-12	28850	0.9065
8	1.73334e-12	26013	1.4108
9	1.98424e-12	25945	0.6576
10	1.39741e-12	29606	0.8314
Dimension $n = 50$			
x0	4.11277e-12	276623	28.8374
1	8.26614e-12	341052	33.6635
2	8.87558e-12	274627	27.1559
3	3.31132e-12	262504	24.9761
4	1.23132e-11	263824	26.5569
5	5.29272e-12	291961	28.7268
6	9.91449e-12	290961	27.9861
7	7.90259e-12	290888	29.6380
8	1.91851e-11	247105	24.4341
9	1.1468e-11	256670	25.9092
10	8.9032e-12	258476	25.6650

Table 2: Results of the Nelder-Mead method on the Extended Freudenstein and Roth function

From Table 2, it is clear that our *Nelder-Mead* method is performing extremely well even in a reasonable execution time for all the three dimensions. As we expected, increasing the dimensions of the problem would require more iterations to reach the minimum and consequently in a higher execution time.

4.1.3 Problem 75

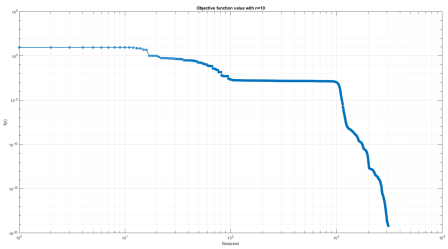

Figure 18: Objective function of problem 75

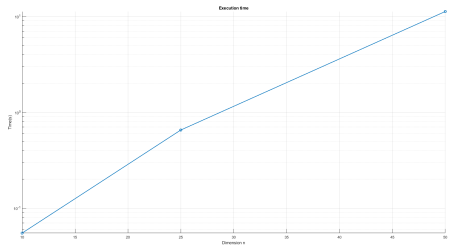

Figure 19: Execution time of problem 75

In (18), the objective function strongly decreases when the minimum neighbourhood is reached from the method.

As can be seen from (19), the execution time is exponential with the respect to the increase of the dimension n, as we expected.

Point	$f(x_k)$	Iterations	Time (s)
Dimension $n = 10$			
x0	6.34628e-20	3096	0.1225
1	1.19774e-21	3411	0.0662
2	2.57586e-19	3378	0.0573
3	1.79694e-20	2992	0.0743
4	1.13427e-19	2721	0.0526
5	5.20374e-19	2874	0.0487
6	6.69932e-21	3074	0.0661
7	1.92281e-18	2871	0.0451
8	3.79423e-20	3083	0.0473
9	2.61039e-20	3081	0.0484
10	8.13472e-20	2776	0.0430
Dimension $n = 25$			
x0	1.23196e-20	22287	0.4736
1	4.35327e-20	25054	0.5236
2	1.51271e-20	23631	0.5074
3	2.32258e-21	23550	0.9036
4	5.87179e-21	25162	0.9737
5	3.88304e-21	22426	0.4733
6	1.06292e-21	25293	0.5322
7	7.79341e-21	26410	0.5707
8	4.83113e-21	23080	0.4868
9	5.53550e-21	23788	0.4987
10	4.09670e-21	26231	1.0537
Dimension $n = 50$			
x0	2.14357e-20	217348	13.0694
1	8.36995e-21	206363	11.2357
2	1.39228e-20	177981	9.7818
3	5.52854e-21	205611	11.3906
4	1.33661e-20	180713	10.3491
5	3.34961e-20	197543	11.5534
6	2.02888e-19	198645	10.7251
7	1.38497e-21	248427	14.3024
8	3.34196e-21	185814	10.6848
9	3.34185e-19	162674	9.2878
10	4.13242e-20	233569	13.0641

Table 3: Results of the Nelder-Mead method on the problem 75.

From Table 3, it is clear that our Nelder-Mead method is performing extremely well even in a reasonable time execution for all the three dimensions. As we expected, increasing the dimensions of the problem would require more iterations to reach the minimum and consequently in a higher execution time.

4.1.4 Problem 76

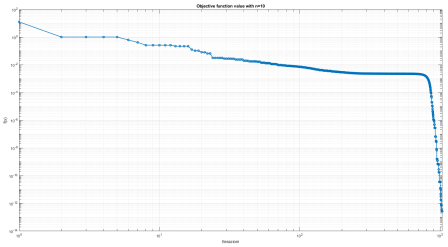

Figure 20: Objective function of problem 76

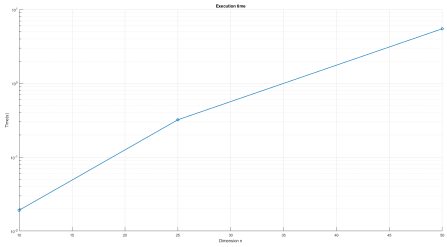

Figure 21: Execution time of problem 76

In (20), the objective function strongly decreases when the minimum neighbourhood is reached from the method.

As can be seen from (21), the execution time is exponential with the respect to the increase of the dimension n, as we expected.

Point	$f(x_k)$	Iterations	Time (s)
Dimension $n = 10$			
x0	2.68314e-13	1037	0.0265
1	1.66758e-13	1236	0.0379
2	1.63321e-13	956	0.0215
3	4.81229e-13	846	0.0179
4	3.23942e-13	905	0.0177
5	4.42672e-13	897	0.0171
6	1.96577e-13	935	0.0199
7	5.91331e-13	931	0.0155
8	2.37423e-13	700	0.0122
9	3.13318e-13	823	0.0143
10	1.00357e-12	724	0.0173
Dimension $n = 25$			
x0	1.86531e-12	10440	0.2570
1	1.0297e-12	10390	0.2207
2	1.20406e-12	8127	0.1823
3	1.18073e-12	9275	0.3628
4	9.67578e-13	9379	0.4893
5	7.28801e-13	8906	0.4653
6	1.08451e-12	10821	0.3746
7	5.17537e-13	12670	0.2907
8	4.526e-13	8298	0.1938
9	8.13156e-13	15307	0.3486
10	6.86093e-13	11800	0.2828

Table 4: Results of the Nelder-Mead method on the problem 76 for dimensions $n = 10$ and $n = 25$.

From Table 4 and 5, it is clear that our Nelder-Mead method is performing extremely well even in a reasonable time execution for all the three dimensions. As we expected, increasing the dimensions of the problem would require more iterations to reach the minimum and consequently in a higher execution time.

A more general consideration is that the the function evaluation in the minimum point is comparable for all the three dimensions, however, as we already said, the number of necessary

Point	$f(x_k)$	Iterations	Time (s)
Dimension $n = 50$			
x0	1.18603e-11	106850	6.9751
1	3.04513e-12	102868	5.5581
2	8.94853e-12	114085	6.2515
3	1.6374e-11	122067	7.0296
4	8.93533e-12	89761	4.8637
5	3.96011e-12	93869	5.0090
6	7.38762e-12	103819	5.4580
7	9.62418e-12	101713	5.3564
8	3.70351e-12	88994	4.7550
9	9.68541e-12	94121	5.5188
10	9.21121e-12	92156	5.0763

Table 5: Results of the Nelder-Mead method on the problem 76 for dimension $n = 50$.

iterations and time required increase.

4.2 Modified Newton method

In this section, for each function, the behaviour of the *Modified newton method* is presented in different dimensions. For the Rosenbrock function the method is used just for two dimensions, while for the other functions the method is applied for $n = 10^d$ with $d = 3, 4, 5$ dimensions.
For each function is also studied the case with exact derivatives and the case with finite differences and its h variations. Moreover, for the function that requires the use of the backtracking strategy, the bar plot of the number of backtrack operations is presented as well as the function evaluation with the respect of x_k.
Really important is to notice that we used this notation: if a function with the default set of parameters do not converge, the problem why it happens is explained. Instead, if a function would have converged if we had waited longer, its table is not presented. The sufficient waiting time for a function to converge is estimated to be 5 minutes. Beyond this time frame, we assume the function converges, but in a manner that takes too long and is outside the scope of the project.

4.3 Analytical derivatives

In this section, the results of the modified Newton method using analytical derivatives are analyzed. For each function, the gradient and the Hessian have been calculated analytically. By studying the exact derivative formulation of the function, the plots (7), (9), (11) already discussed in the previous section, are obtained.

4.3.1 Freudenstein and Roth function

Applying the *Modified Newton* method to the Freudenstein and Roth function in $n = 2$ dimensions gave the plots presented in (22) and (23).

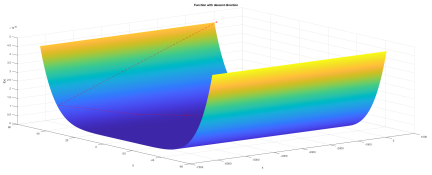

Figure 22: Freudenestein and Roth plot 3D with modified newton method

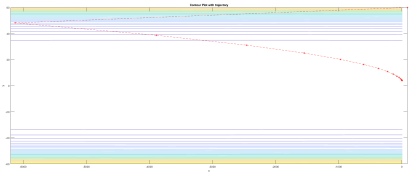

Figure 23: Freudenestein and Roth plot 2D with modified newton method

Plots (22) and (23) represents the variation of the $xseq$ sequence reached by the method in two and three dimensions.

Moreover, results obtained different dimensions and with or without preconditioning are presented.

In the following plots is presented the decreasing function evaluation for the three different dimensions:

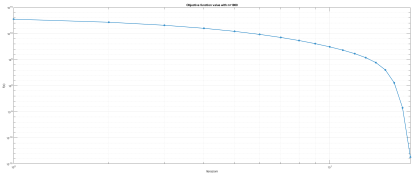

Figure 24: Freudenstein and Roth function evaluation with $n = 1000$

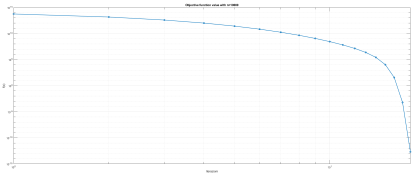

Figure 25: Freudenstein and Roth function evaluation with $n = 10000$

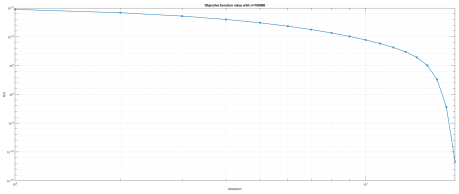

Figure 26: Freudenstein and Roth function evaluation with $n = 100000$

From Table 6, Table 7, Table 8, Table 9, Table 10, Table 11, it can be shown that *Freudenstein and Roth* function converged the same with and without preconditioning, obtaining the same results. Even the orders of magnitude of the execution times are the same in tables with and without preconditioning. This means that the Hessian matrix of the function is already well-conditioned and so preconditioning might not be necessary because the descent of the Newton method is already fast. This is reflected by the fact that the iteration numbers of tables both with and without preconditioning is the same.

4.3.2 Problem 75

Applying the *Modified Newton* method to the *problem 75* in dimensions three and two gives the plots presented in (27) and (28).

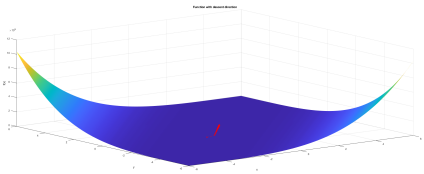

Figure 27: Problem 75 plot 3D with modified newton method

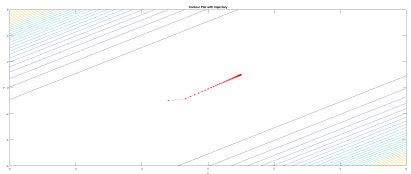

Figure 28: Problem 75 plot 2D with modified newton method

In the following plots is presented the decreasing function evaluation for different dimensions:

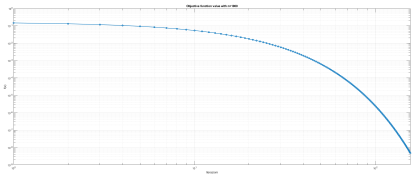

Figure 29: Problem 75 function evaluation with $n = 1000$

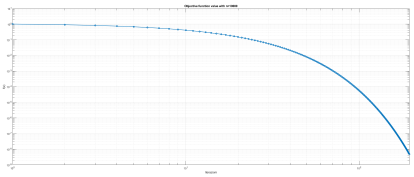

Figure 30: Problem 75 function evaluation with $n = 10000$

With *Problem 75* function, as can be seen from plots (29),(30), (31) and even from the Table 12, Table 13, Table 14, Table 15, Table 16, Table 17 the method requires an higher number of iterations.

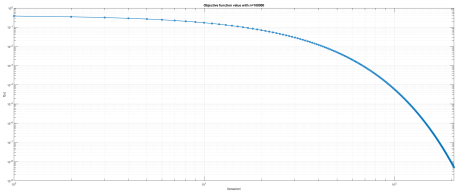

Figure 31: Problem 75 function evaluation with $n = 100000$

Nonetheless, the method converges in a few seconds also in 100000 dimensions thanks to the parameter "smallestabs" given in input to the function eigs that compute the smallest eigenvalue in absolute value of a sparce matrix. We can use this parameter only with this function because we know for sure that all the eigenvalues of the Hessian matrix are always positive since they are all constant, thanks to the analytical calculation. The "smallestabs" option is way faster than the "smallestreal" option which is used in all scenarios with the other functions. This enhance the convergence in few seconds.

4.3.3 Problem 76

Applying the *Modified Newton* method to the *problem 76* in three and two dimensions gave the plots presented in (32) and (33)

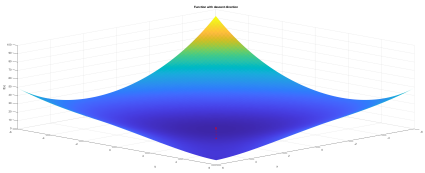

Figure 32: Problem 76 plot 3D with modified newton method

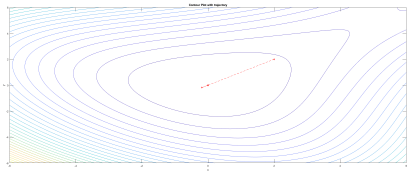

Figure 33: Problem 76 plot 2D with modified newton method

In the following plots is presented the decreasing function evaluation for different dimensions:

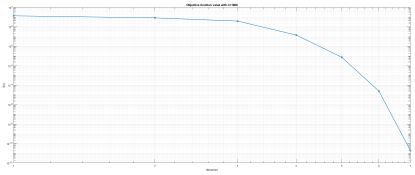

Figure 34: Problem 76 function evaluation with $n = 1e^3$

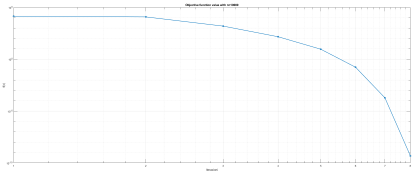

Figure 35: Problem 76 function evaluation with $n = 1e^4$

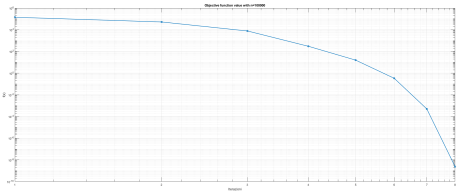

Figure 36: Problem 76 function evaluation with $n = 1e^5$

As can be seen from (34),(35),(36) and Table 20, Table 21, Table 22, Table 23, Table 24, Table 25, converges is reached with a small number of iterations and with a short time in the same way with and without preconditioning, obtaining the same results. Even the orders of magnitude of the execution times are the same in tables with and without preconditioning. This means that the Hessian matrix of the function is already well- conditioned and so preconditioning might not be necessary because the descent of the Newton method is already fast. This is reflected by the fact that the iteration numbers of tables both with and without preconditioning is the same.

Moreover, since the function requires the use of the backtracking strategy, the plot that shows the number of iterations in the backtracking for each iterations of the general method is presented in (37) and (38)

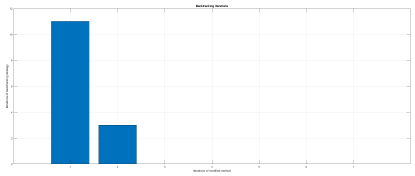

Figure 37: Problem 76 backtracking iterations with precondition

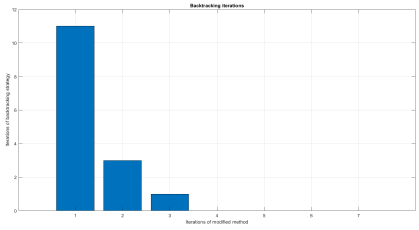

Figure 38: Problem 76 backtracking iterations with precondition

4.4 Finite differences

The Finite Difference method is a numerical technique used to approximate derivatives of functions. It involves replacing the continuous derivatives with discrete approximations by evaluating the function at a set of points. For example for the gradient:

$$f_{x_i} = \frac{\partial}{\partial x_i} f(\hat{x}) \approx \frac{f(\hat{x} + he_i) - f(\hat{x})}{h} \quad ,\forall i = 1, ..., n, \quad \texttt{forward} \tag{19}$$

$$f_{x_i} = \frac{\partial}{\partial x_i} f(\hat{x}) \approx \frac{f(\hat{x} + he_i) - f(\hat{x} - he_i)}{2h} \quad ,\forall i = 1, ..., n \quad \texttt{centered} \tag{20}$$

For the Hessian:

$$(Hf)_{ij} \approx \frac{f(\hat{x} + he_i) - 2f(\hat{x}) + f(\hat{x} - he_i)}{h} \quad ,\forall i = 1, ..., n, \quad \texttt{diagonal element} \tag{21}$$

$$(Hf)_{ij} \approx \frac{f(\hat{x} + he_i + he_j) - f(\hat{x} + he_i) - f(\hat{x} + he_i) + f(\hat{x})}{h} \quad ,\forall i = 1, ..., n, \quad \texttt{non diagonal element} \tag{22}$$

Since we are dealing with high-dimensional spaces, we must take advantage of the sparsity of the matrix. In this case, the preliminary analysis conducted in section (3) is extremely useful, since knowing the hessian from it can be implemented a more efficient code.
This allows us to compute the derivatives only for the points that we know are non-zero, optimizing the computational effort and reducing the complexity.

The finite differences method must be tested with $h = 10^{-k}$ with $k = 2, 4, 6, 8, 10, 12$, and then with $h_i = 10^{-k} |x_i|$.

Freudenstein and Roth function For the *Freudenstein and Roth* function no one of the standard h enhances converges. However, as can be seen from Table 26 and Table 27, if we use specific values of $h_i = 10^{-k} |x_i|$, convergence can be obtained for some starting points. This is probably to the fact that the composition of x_0 allows to explore sub-intervals in between $k = 2, 4, 6, 8, 10, 12$, enabling more probabilities of convergence.

Freudenstein with preconditioning:

- h=1e-2, h=1e-4, h=1e-8, h=1e-10, h=1e-12: max backtrack iterations reached
- h=1e-6: first two attempts fail, then high computation time

Freudenstein without preconditioning:

- h=1e-2: no convergence due to slow descent
- h=1e-4, h=1e-6, h=1e-12: error during ichol computation for 1e+3 and 1e+4
- h=1e-8: max backtrack iterations reached for 1e+3 and 1e+4

Freudenstein without preconditioning ($h\,|x_i|$):

- h=1e-2, h=1e-4, h=1e-6, h=1e-8, h=1e-10, h=1e-12: max backtrack iterations reached

Problem 75 For the *Problem 75* we obtain convergence for different combinations of h. They can be checked in Table 18, Table 30, Table 32, Table 35,Table 31, Table 33, Table 35

Problem 75 without preconditioning:

- h=1e-2, h=1e-4: no convergence (max backtrack iterations reached)
- h=1e-6, h=1e-8: converges only for x0, eigs error for others
- h=1e-10, h=1e-12: max backtrack with some convergences

Problem 75 without preconditioning ($h\,|x_i|$):

- h=1e-2, h=1e-4, h=1e-10, h=1e-12: max backtrack iterations reached
- h=1e-6, h=1e-8: max backtrack for x0, eigs error for others

Problem 76 For the *Problem 76* we obtain convergence almost everywhere, with reasoble execution times and iterations.

Problem 76 without preconditioning:

- h=1e-2: no convergence
- h=1e-4, h=1e-6, h=1e-8: converges for 1e+3 and 1e+4
- h=1e-10: converges for 1e+3 and 1e+4 but takes almost a minute
- h=1e-12: converges for 1e+3, fails for 1e+4

Problem 76 without preconditioning ($h\,|x_i|$):

- h=1e-12: max backtrack iterations reached
- h=1e-10, h=1e-8, h=1e-6, h=1e-4, h=1e-2: converge but to relatively high values

Since with *Problem 76* there are an high number of combinations of values able to reach convergence, using this function some key considerations can be extracted. In general, problems occurs when we have to deal with a really high/low number of h. This is because we are being to much demanding from the finite derivates or we are approximating too much the derivates.
In this case preconditioning enhance faster convergences for values of small h such as $h = 1e^{-12}$ and $h = 1e^{-10}$ where we can also compute the modified newton method for 10^5 dimensions. Comparing execution time for 10^4 dimensions it can be noticed several orders of magnitude of difference.

5 Conclusions

The project allowed us to evaluate the characteristics of the Nelder-Mead method and the Modified Newton method.
The *Nelder-Mead* method performs well for most functions, provided that proper fine-tuning is carried out. However, this method is not suitable for high-dimensional problems, where other methods, such as the Modified Newton method, prove to be more efficient.
The *Modified Newton* method, as mentioned, remains efficient even in very high dimensions. However, the main challenge lies in computing the smallest eigenvalue, which represents the bottleneck of the entire process. The method performs well with both the analytical derivative approach and the finite difference method, although the convergence time strongly depends on both the problem's dimensionality and the starting point x_0.

5.1 Further improvements

Since some values were not reported due to the time required to run the codes, having more powerful computers or selecting functions better suited to these two methods would have allowed for a presentation with more results. Moreover, since different computers were used, perfect reproducibility of results is not garanteed since different computers have different performances.

6 Tables results

Empty tables are used just as a reminder that the for that combination, the function is not converging.

6.1 Analytical differences

6.1.1 Freudenstein and Roth function

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-14	5.494e-06	18	0.039
2.118e-14	7.987e-06	18	0.038
2.261e-14	8.252e-06	18	0.039
2.135e-14	8.018e-06	18	0.038
2.160e-14	8.064e-06	18	0.040
2.010e-14	7.780e-06	18	0.050
2.100e-14	7.951e-06	18	0.049
2.137e-14	8.021e-06	18	0.048
2.212e-14	8.162e-06	18	0.038
1.878e-14	7.520e-06	18	0.037
2.066e-14	7.887e-06	18	0.032

Table 6: Freudenstein and Roth function with $n = 10^3$ with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-14	5.494e-06	18	0.019
2.118e-14	7.987e-06	18	0.017
2.261e-14	8.252e-06	18	0.017
2.135e-14	8.018e-06	18	0.017
2.160e-14	8.064e-06	18	0.019
2.010e-14	7.780e-06	18	0.015
2.100e-14	7.951e-06	18	0.016
2.137e-14	8.021e-06	18	0.018
2.212e-14	8.162e-06	18	0.021
1.878e-14	7.520e-06	18	0.019
2.066e-14	7.887e-06	18	0.017

Table 7: Freudenstein and Roth function $n = 10^3$ without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-13	1.737e-05	18	0.175
2.087e-13	2.507e-05	18	0.148
2.116e-13	2.524e-05	18	0.154
2.113e-13	2.522e-05	18	0.158
2.071e-13	2.497e-05	18	0.155
2.136e-13	2.536e-05	18	0.161
2.167e-13	2.555e-05	18	0.148
2.093e-13	2.510e-05	18	0.156
2.156e-13	2.548e-05	18	0.157
2.050e-13	2.484e-05	18	0.147
2.148e-13	2.543e-05	18	0.155

Table 8: Freudenstein and Roth function with $n = 10^4$ with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-13	1.737e-05	18	0.152
2.087e-13	2.507e-05	18	0.158
2.116e-13	2.524e-05	18	0.151
2.113e-13	2.522e-05	18	0.138
2.071e-13	2.497e-05	18	0.133
2.136e-13	2.536e-05	18	0.126
2.167e-13	2.555e-05	18	0.124
2.093e-13	2.510e-05	18	0.120
2.156e-13	2.548e-05	18	0.114
2.050e-13	2.484e-05	18	0.132
2.148e-13	2.543e-05	18	0.281

Table 9: Freudenstein and Roth function $n = 10^4$ without preconditioning

6.2 Finite Differences

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-12	5.494e-05	18	2.221
2.131e-12	8.010e-05	18	2.044
2.116e-12	7.983e-05	18	1.749
2.120e-12	7.991e-05	18	2.228
2.115e-12	7.981e-05	18	1.895
2.116e-12	7.983e-05	18	1.836
2.111e-12	7.973e-05	18	2.326
2.098e-12	7.948e-05	18	2.781
2.117e-12	7.985e-05	18	1.755
2.130e-12	8.009e-05	18	2.079
2.124e-12	7.997e-05	18	1.584

Table 10: Freudenstein and Roth function with $n = 10^5$ with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.002e-12	5.494e-05	18	2.057
2.131e-12	8.010e-05	18	2.072
2.116e-12	7.983e-05	18	1.372
2.120e-12	7.991e-05	18	2.106
2.115e-12	7.981e-05	18	1.391
2.116e-12	7.983e-05	18	2.093
2.111e-12	7.973e-05	18	1.955
2.098e-12	7.948e-05	18	1.437
2.117e-12	7.985e-05	18	2.131
2.130e-12	8.009e-05	18	1.363
2.124e-12	7.997e-05	18	2.214

Table 11: Freudenstein and Roth function $n = 10^5$ without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.482e-09	9.468e-05	169	0.192
4.852e-09	9.851e-05	157	0.071
4.580e-09	9.571e-05	157	0.076
4.940e-09	9.940e-05	167	0.077
4.850e-09	9.848e-05	164	0.075
4.754e-09	9.751e-05	172	0.092
4.806e-09	9.805e-05	170	0.083
4.621e-09	9.614e-05	159	0.090
4.887e-09	9.886e-05	167	0.102
4.769e-09	9.766e-05	156	0.076
4.505e-09	9.492e-05	157	0.071

Table 12: Problem 75 n=10^3 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.482e-09	9.468e-05	169	0.154
4.852e-09	9.851e-05	157	0.029
4.580e-09	9.571e-05	157	0.034
4.940e-09	9.940e-05	167	0.031
4.850e-09	9.848e-05	164	0.033
4.754e-09	9.751e-05	172	0.031
4.806e-09	9.805e-05	170	0.034
4.621e-09	9.614e-05	159	0.031
4.887e-09	9.886e-05	167	0.031
4.769e-09	9.766e-05	156	0.026
4.505e-09	9.492e-05	157	0.027

Table 13: Problem 75 n=10^3 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.546e-09	9.535e-05	188	0.539
4.844e-09	9.842e-05	192	0.703
4.748e-09	9.745e-05	184	0.519
4.559e-09	9.548e-05	188	0.815
4.634e-09	9.627e-05	187	0.648
4.939e-09	9.939e-05	179	0.535
4.594e-09	9.586e-05	185	0.548
4.701e-09	9.697e-05	195	0.567
4.704e-09	9.699e-05	190	0.549
4.644e-09	9.638e-05	182	0.623
4.827e-09	9.825e-05	195	0.577

Table 14: Problem 75 n=10^4 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.546e-09	9.535e-05	188	1.184
4.844e-09	9.842e-05	192	0.512
4.748e-09	9.745e-05	184	0.656
4.559e-09	9.548e-05	188	0.795
4.634e-09	9.627e-05	187	0.663
4.939e-09	9.939e-05	179	0.432
4.594e-09	9.586e-05	185	0.420
4.701e-09	9.697e-05	195	0.727
4.704e-09	9.699e-05	190	0.815
4.644e-09	9.638e-05	182	0.593
4.827e-09	9.825e-05	195	0.810

Table 15: Problem 75 n=10^4 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.905e-09	9.905e-05	206	5.812
4.803e-09	9.801e-05	213	6.359
4.968e-09	9.968e-05	208	6.210
4.846e-09	9.845e-05	210	6.186
4.578e-09	9.569e-05	210	7.313
4.976e-09	9.976e-05	205	7.353
4.854e-09	9.853e-05	206	6.158
4.882e-09	9.881e-05	196	5.607
4.799e-09	9.797e-05	197	5.726
4.745e-09	9.742e-05	197	5.766
4.921e-09	9.920e-05	191	6.309

Table 16: Problem 75 n=10^5 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.905e-09	9.905e-05	206	8.071
4.803e-09	9.801e-05	213	7.642
4.968e-09	9.968e-05	208	7.274
4.846e-09	9.845e-05	210	6.709
4.578e-09	9.569e-05	210	6.930
4.976e-09	9.976e-05	205	8.034
4.854e-09	9.853e-05	206	6.878
4.882e-09	9.881e-05	196	7.181
4.799e-09	9.797e-05	197	6.860
4.745e-09	9.742e-05	197	6.599
4.921e-09	9.920e-05	191	5.888

Table 17: Problem 75 n=10^5 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
11.45	8.022e-05	9.98e-05	287

Table 18: Problem 75 n=10^4 with preconditioning with $h = 1e^{-8}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]

Table 19: Problem 75 n=10^4 without preconditioning $h = 1e^{-8}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.0825e-16	1.471e-08	5	0.006
8.748e-11	1.323e-05	7	0.012
1.279e-14	1.599e-07	8	0.016
4.007e-15	8.953e-08	7	0.018
7.756e-15	1.246e-07	7	0.019
6.545e-18	3.618e-09	8	0.022
5.157e-10	3.212e-05	7	0.026
1.969e-11	6.275e-06	7	0.017
9.389e-11	1.370e-05	7	0.012
1.917e-11	6.191e-06	7	0.013
3.886e-11	8.815e-06	7	0.015

Table 20: Problem 76 n=10^3 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.082e-16	1.471e-08	5	0.057
8.748e-11	1.323e-05	7	0.092
1.279e-14	1.599e-07	8	0.023
4.009e-15	8.954e-08	7	0.018
7.758e-15	1.246e-07	7	0.017
6.545e-18	3.618e-09	8	0.015
5.157e-10	3.212e-05	7	0.014
1.969e-11	6.275e-06	7	0.013
9.389e-11	1.370e-05	7	0.013
1.917e-11	6.191e-06	7	0.014
3.890e-11	8.821e-06	7	0.013

Table 21: Problem 76 n=10^3 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.0825e-15	4.653e-08	5	0.039
6.481e-11	1.139e-05	8	0.089
1.264e-10	1.590e-05	8	0.081
9.743e-13	1.396e-06	8	0.088
4.036e-09	8.984e-05	7	0.106
3.301e-12	2.570e-06	8	0.077
3.213e-12	2.535e-06	7	0.096
5.840e-10	3.418e-05	8	0.076
2.853e-12	2.389e-06	7	0.073
4.802e-10	3.099e-05	8	0.081
1.013e-11	4.502e-06	8	0.062

Table 22: Problem 76 n=10^4 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.082e-15	4.653e-08	5	0.044
6.482e-11	1.139e-05	8	0.093
1.264e-10	1.590e-05	8	0.098
9.743e-13	1.396e-06	8	0.091
4.036e-09	8.984e-05	7	0.111
3.301e-12	2.570e-06	8	0.089
3.213e-12	2.535e-06	7	0.137
5.840e-10	3.418e-05	8	0.092
2.853e-12	2.389e-06	7	0.095
4.802e-10	3.099e-05	8	0.098
1.014e-11	4.502e-06	8	0.079

Table 23: Problem 76 n=10^4 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.083e-14	1.471e-07	5	0.242
2.385e-09	6.907e-05	8	0.832
7.897e-10	3.974e-05	8	1.038
9.209e-10	4.292e-05	8	0.740
2.106e-09	6.490e-05	8	0.794
1.826e-09	6.044e-05	8	0.834
2.583e-09	7.187e-05	8	0.990
4.818e-10	3.104e-05	8	0.794
2.568e-19	7.167e-10	9	1.238
8.413e-10	4.102e-05	8	0.793
1.202e-09	4.903e-05	8	0.745

Table 24: Problem 76 n=10^5 with preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.082e-14	1.471e-07	5	0.495
2.385e-09	6.907e-05	8	1.158
7.898e-10	3.974e-05	8	1.561
9.209e-10	4.292e-05	8	1.187
2.106e-09	6.490e-05	8	1.148
1.826e-09	6.044e-05	8	1.447
2.583e-09	7.187e-05	8	1.357
4.818e-10	3.104e-05	8	1.141
2.568e-19	7.167e-10	9	1.675
8.413e-10	4.102e-05	8	1.069
1.202e-09	4.903e-05	8	1.025

Table 25: Problem 76 n=10^5 without preconditioning

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
6.007693	2.57133e-10	5.10375e-06	19

Table 26: Freudenstein and Roth n=10^3 with preconditioning with h=$1e^{-8}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
6.104	2.473e-06	4.880e-05	18
6.972	2.473e-06	5.851e-05	18
6.392	2.473e-06	5.945e-05	18
5.923	2.473e-06	5.842e-05	18
5.793	2.473e-06	5.901e-05	18
6.128	2.473e-06	5.682e-05	19
5.786	2.473e-06	5.790e-05	18
6.266	2.473e-06	5.836e-05	18
5.909	2.473e-06	5.933e-05	18
5.770	2.473e-06	5.573e-05	18
5.822	2.473e-06	5.777e-05	18

Table 27: Freudenstein and Roth n=10^3 with preconditioning with h=$1e^{-6}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
2.434e-09	9.949e-05	196	8.697
5.308e-09	7.129e-05	194	10.467
3.422e-09	6.349e-05	251	12.429
1.139e-09	3.797e-05	230	10.410

Table 28: Problem 75 n=10^3 with preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.894e-09	3.641e-05	207	11.62
8.105e-09	7.364e-05	131	6.93
3.061e-09	4.065e-05	139	8.68

Table 29: Problem 75 n=10^3 without preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
2.990e-09	3.380e-05	82	3.252
1.695e-09	4.007e-05	31	14.394
4.064e-09	3.980e-05	31	3.056
2.074e-09	3.568e-05	36	2.550
7.033e-09	8.255e-05	30	2.648
1.244e-09	4.009e-05	30	3.517
6.612e-09	8.461e-05	29	3.261

Table 30: Problem 75 n=10^3 with preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
5.560e-08	7.632e-05	56	3.083
1.550e-09	4.765e-05	35	5.298
9.689e-09	9.134e-05	32	14.018
1.432e-09	4.812e-05	34	4.228
4.641e-09	6.207e-05	35	2.953
5.244e-09	7.452e-05	34	3.058
2.597e-09	5.398e-05	32	3.721
2.909e-09	6.402e-05	29	2.905

Table 31: Problem 75 n=10^3 without preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
6.203e-08	7.346e-05	24	1.033

Table 32: Problem 75 n=10^3 with preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
4.769e-08	5.347e-05	25	1.64

Table 33: Problem 75 n=10^3 without preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.0189	9.842e-05	88	3.565

Table 34: Problem 75 n=10^3 with preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.019	9.842e-05	88	6.04

Table 35: Problem 75 n=10^3 without preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.083e-16	1.471e-08	5	0.094
8.748e-11	1.323e-05	7	0.134
1.279e-14	1.599e-07	8	0.089
4.007e-15	8.953e-08	7	0.114
7.756e-15	1.246e-07	7	0.077
6.545e-18	3.618e-09	8	0.079
5.157e-10	3.212e-05	7	0.083
1.969e-11	6.275e-06	7	0.072
9.389e-11	1.370e-05	7	0.074
1.917e-11	6.191e-06	7	0.077
3.886e-11	8.815e-06	7	0.074

Table 36: Problem 76 n=10^3 with preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.923e-11	6.201e-06	4	3.324
1.687e-09	5.809e-05	6	1.233
1.274e-10	1.596e-05	8	0.779
1.181e-09	4.861e-05	8	0.982
1.185e-14	1.539e-07	7	0.857
1.224e-15	4.947e-08	8	0.915
2.736e-14	2.339e-07	8	0.886
2.338e-14	2.163e-07	8	0.903
7.180e-15	1.198e-07	7	0.671
1.270e-09	5.039e-05	7	0.779
2.059e-16	2.029e-08	8	0.874

Table 37: Problem 76 n=10^3 without preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.083e-14	1.471e-07	5	0.320
2.385e-09	6.907e-05	8	0.938
7.897e-10	3.974e-05	8	1.215
9.209e-10	4.292e-05	8	0.860
2.106e-09	6.490e-05	8	0.977
1.826e-09	6.044e-05	8	1.036
2.583e-09	7.187e-05	8	1.283
4.818e-10	3.104e-05	8	1.318
2.568e-19	7.167e-10	9	2.102
8.413e-10	4.102e-05	8	1.922
1.202e-09	4.903e-05	8	1.709

Table 38: Problem 76 n=10^4 with preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.082e-15	4.653e-08	5	0.092
6.481e-11	1.139e-05	8	0.138
1.264e-10	1.590e-05	8	0.137
9.743e-13	1.396e-06	8	0.155
4.036e-09	8.984e-05	7	0.171
3.301e-12	2.570e-06	8	0.132
3.212e-12	2.535e-06	7	0.159
5.840e-10	3.418e-05	8	0.158
2.853e-12	2.389e-06	7	0.204
4.802e-10	3.099e-05	8	0.148
1.013e-11	4.502e-06	8	0.131

Table 39: Problem 76 n=10^5 with preconditioning with h=$1e-12$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.083e-16	1.471e-08	5	0.007
8.748e-11	1.323e-05	7	0.022
1.279e-14	1.599e-07	8	0.017
4.007e-15	8.953e-08	7	0.017
7.756e-15	1.246e-07	7	0.026
6.545e-18	3.618e-09	8	0.033
5.157e-10	3.212e-05	7	0.018
1.969e-11	6.275e-06	7	0.014
9.389e-11	1.370e-05	7	0.012
1.917e-11	6.191e-06	7	0.015
3.886e-11	8.815e-06	7	0.014

Table 40: Problem 76 n=10^3 with preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
2.191e-16	1.940e-08	5	0.213
6.231e-16	3.516e-08	7	0.309
1.239e-17	4.625e-09	7	0.298
1.022e-15	4.508e-08	7	0.313
6.106e-16	3.481e-08	7	0.311
5.562e-17	1.033e-08	7	0.345
7.695e-16	3.910e-08	7	0.396
5.016e-16	3.153e-08	7	0.599
1.020e-16	1.409e-08	7	0.618
4.371e-15	9.339e-08	7	0.393
4.717e-15	9.702e-08	7	0.310

Table 41: Problem 76 n=10^3 without preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.082e-15	4.653e-08	5	0.035
6.481e-11	1.139e-05	8	0.092
1.264e-10	1.590e-05	8	0.087
9.743e-13	1.396e-06	8	0.092
4.036e-09	8.984e-05	7	0.120
3.301e-12	2.570e-06	8	0.090
3.212e-12	2.535e-06	7	0.107
5.840e-10	3.418e-05	8	0.095
2.853e-12	2.389e-06	7	0.094
4.802e-10	3.099e-05	8	0.098
1.013e-11	4.502e-06	8	0.077

Table 42: Problem 76 n=10^4 with preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
2.348e-14	2.135e-07	4	24.77
1.605e-11	5.665e-06	8	45.24
5.044e-11	1.004e-05	8	45.15
1.024e-10	1.431e-05	8	44.01
6.505e-11	1.141e-05	8	46.80
3.210e-11	8.012e-06	8	48.74
1.912e-11	6.184e-06	8	47.89
1.765e-10	1.879e-05	8	48.78
9.532e-11	1.381e-05	8	45.01
1.553e-16	1.679e-08	8	44.73
1.657e-10	1.820e-05	8	44.97

Table 43: Problem 76 n=10^4 without preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.083e-14	1.471e-07	5	0.251
2.385e-09	6.907e-05	8	0.887
7.897e-10	3.974e-05	8	1.208
9.209e-10	4.292e-05	8	0.849
2.106e-09	6.490e-05	8	0.976
1.826e-09	6.044e-05	8	1.022
2.583e-09	7.187e-05	8	1.092
4.818e-10	3.104e-05	8	0.967
2.568e-19	7.167e-10	9	1.352
8.413e-10	4.102e-05	8	0.931
1.202e-09	4.903e-05	8	0.867

Table 44: Problem 76 n=10^5 with preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]

Table 45: Problem 76 n=10^5 without preconditioning with h=$1e-10$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.094e-14	1.018e-08	5	0.345
8.420e-11	1.297e-05	7	0.366
2.382e-14	1.412e-07	8	0.399
1.592e-14	6.409e-08	7	0.348
1.943e-14	9.544e-08	7	0.345
1.250e-14	1.218e-09	8	0.386
4.977e-10	3.154e-05	7	0.366
1.894e-11	6.141e-06	7	0.353
9.202e-11	1.354e-05	7	0.353
1.898e-11	6.149e-06	7	0.350
3.430e-11	8.270e-06	7	0.355

Table 46: Problem 76 n=10^3 with preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.094e-14	1.018e-08	5	0.257
8.409e-11	1.296e-05	7	0.337
2.394e-14	1.420e-07	8	0.360
1.593e-14	6.416e-08	7	0.309
1.941e-14	9.524e-08	7	0.309
1.250e-14	1.229e-09	8	0.370
5.094e-10	3.191e-05	7	0.470
1.895e-11	6.142e-06	7	0.631
9.005e-11	1.340e-05	7	0.592
1.873e-11	6.108e-06	7	0.283
3.608e-11	8.482e-06	7	0.297

Table 47: Problem 76 n=10^3 without preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.094e-13	3.234e-08	5	42.764
3.452e-11	8.282e-06	8	67.868
4.177e-11	9.116e-06	8	67.191
1.021e-12	1.328e-06	8	66.818
4.012e-09	8.956e-05	7	61.486
3.289e-12	2.502e-06	8	84.435
3.072e-12	2.320e-06	7	64.774
6.320e-10	3.554e-05	8	73.240
2.813e-12	2.213e-06	7	65.530
5.060e-10	3.179e-05	8	73.404
1.206e-11	4.876e-06	8	73.898

Table 48: Problem 76 n=10^4 with preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.094e-13	3.233e-08	5	28.851
3.422e-11	8.246e-06	8	44.087
6.317e-11	1.122e-05	8	43.461
9.818e-13	1.299e-06	8	44.603
3.628e-09	8.516e-05	7	38.344
3.378e-12	2.537e-06	8	47.004
3.122e-12	2.340e-06	7	39.483
6.256e-10	3.536e-05	8	44.369
2.797e-12	2.206e-06	7	39.859
4.324e-10	2.939e-05	8	44.916
1.007e-11	4.451e-06	8	48.879

Table 49: Problem 76 n=10^4 without preconditioning with h=$1e-8$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.212e-10	2.413e-07	5	0.249
1.992e-10	1.136e-05	7	0.317
1.249e-10	5.728e-08	8	0.376
1.227e-10	2.456e-07	7	0.311
1.228e-10	2.489e-07	7	0.340
1.250e-10	2.371e-08	8	0.378
5.815e-10	2.922e-05	7	0.308
1.427e-10	5.026e-06	7	0.319
1.249e-10	5.970e-08	8	0.339
1.400e-10	5.027e-06	7	0.296
1.573e-10	7.350e-06	7	0.310

Table 50: Problem 76 n=10^3 with preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.212e-10	2.413e-07	5	0.241
1.992e-10	1.136e-05	7	0.283
1.249e-10	5.728e-08	8	0.324
1.227e-10	2.456e-07	7	0.301
1.228e-10	2.489e-07	7	0.389
1.250e-10	2.370e-08	8	0.447
5.817e-10	2.923e-05	7	0.305
1.427e-10	5.027e-06	7	0.314
1.249e-10	5.970e-08	8	0.376
1.400e-10	5.028e-06	7	0.616
1.574e-10	7.354e-06	7	0.626

Table 51: Problem 76 n=10^3 without preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.212e-09	7.622e-07	5	40.351
1.307e-09	9.657e-06	8	84.148
1.360e-09	1.390e-05	8	81.451
1.251e-09	8.431e-07	8	83.012
4.846e-09	8.258e-05	7	70.062
1.253e-09	1.712e-06	8	70.687
1.207e-09	1.620e-06	7	69.373
1.765e-09	3.105e-05	8	82.160
1.207e-09	1.642e-06	7	70.357
1.670e-09	2.751e-05	8	77.597
1.259e-09	3.507e-06	8	61.088

Table 52: Problem 76 n=10^4 with preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.212e-09	7.622e-07	5	30.400
1.307e-09	9.650e-06	8	46.007
1.360e-09	1.391e-05	8	47.147
1.251e-09	8.449e-07	8	44.974
4.846e-09	8.258e-05	7	39.620
1.253e-09	1.711e-06	8	44.821
1.207e-09	1.620e-06	7	39.851
1.765e-09	3.105e-05	8	45.860
1.207e-09	1.642e-06	7	39.157
1.671e-09	2.753e-05	8	44.263
1.259e-09	3.507e-06	8	43.467

Table 53: Problem 76 n=10^4 without preconditioning with h=$1e-6$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.244e-06	3.678e-06	5	0.293
1.248e-06	6.579e-06	7	0.410
1.250e-06	1.431e-06	8	0.503
1.248e-06	2.192e-06	7	0.411
1.248e-06	2.511e-06	7	0.412
1.250e-06	1.589e-07	8	0.442
1.248e-06	1.098e-05	7	0.518
1.249e-06	7.042e-06	7	0.475
1.250e-06	1.253e-06	8	0.449
1.248e-06	5.957e-06	7	0.379
1.248e-06	7.070e-06	7	0.408

Table 54: Problem 76 n=10^3 with preconditioning with h=$1e-4$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.244e-06	3.678e-06	5	0.271
1.248e-06	6.579e-06	7	0.382
1.250e-06	1.431e-06	8	0.343
1.248e-06	2.192e-06	7	0.295
1.248e-06	2.511e-06	7	0.322
1.250e-06	1.589e-07	8	0.765
1.248e-06	1.098e-05	7	0.633
1.249e-06	7.042e-06	7	0.340
1.250e-06	1.253e-06	8	0.311
1.248e-06	5.957e-06	7	0.318
1.248e-06	7.065e-06	7	0.307

Table 55: Problem 76 n=10^3 without preconditioning with h=$1e-4$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.244e-05	1.163e-05	5	53.178
1.250e-05	7.253e-06	8	85.781
1.250e-05	5.642e-06	8	86.765
1.250e-05	4.075e-06	8	80.446
1.250e-05	2.946e-05	7	72.074
1.250e-05	5.993e-06	8	90.566
1.244e-05	2.190e-05	7	74.135
1.250e-05	1.006e-05	8	81.444
1.245e-05	2.122e-05	7	70.807
1.250e-05	1.196e-05	8	81.167
1.250e-05	5.465e-06	8	87.500

Table 56: Problem 76 n=10^4 with preconditioning with h=$1e-4$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
1.244e-05	1.163e-05	5	30.563
1.250e-05	7.253e-06	8	44.707
1.250e-05	5.642e-06	8	44.467
1.250e-05	4.075e-06	8	44.348
1.250e-05	2.946e-05	7	38.891
1.250e-05	5.993e-06	8	47.173
1.244e-05	2.190e-05	7	39.377
1.250e-05	1.006e-05	8	43.654
1.245e-05	2.122e-05	7	41.916
1.250e-05	1.196e-05	8	48.518
1.250e-05	5.465e-06	8	46.396

Table 57: Problem 76 n=10^4 without preconditioning with h=$1e-4$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
6.03341e-09	6.75605e-05	30	0.322
9.06592e-09	9.59074e-05	34	0.421

Table 58: Problem 76 n=10^3 with preconditioning with h=$1e^{-10}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.252	3.57e-4	1000	54.82
0.270	4.03e-4	1000	52.20
0.282	4.32e-4	1000	53.61
0.274	4.14e-4	1000	47.33
0.268	4.10e-4	1000	47.08
0.266	4.03e-4	1000	50.72
0.275	4.17e-4	1000	46.13
0.276	4.28e-4	1000	49.43
0.268	4.00e-4	1000	49.48
0.263	3.97e-4	1000	49.62

Table 59: Problem 76 n=10^3 without preconditioning with h=$1e^{-10}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
9.96532e-17	1.41176e-08	5	0.256364

Table 60: Problem 76 n=10^3 with preconditioning with h=$1e^{-8}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.251	3.55e-4	1000	47.20
0.242	3.42e-4	1000	48.84
0.243	3.44e-4	1000	48.58
0.245	3.49e-4	1000	46.65
0.267	3.99e-4	1000	47.81
0.242	3.46e-4	1000	43.67
0.247	3.53e-4	1000	44.88
0.267	4.04e-4	1000	49.28
0.240	3.42e-4	1000	48.32
0.260	3.87e-4	1000	48.94

Table 61: Problem 76 n=10^3 without preconditioning with h=$1e^{-8}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
2.4203e-15	6.95743e-08	5	40.303335

Table 62: Problem 76 n=10^3 and n=10^4 with preconditioning with h=$1e^{-6}\,|x_i|$, only x_0

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.249	3.52e-4	1000	50.07
0.241	3.41e-4	1000	49.56
0.246	3.51e-4	1000	47.60
0.245	3.49e-4	1000	50.32
0.244	3.48e-4	1000	49.31
0.248	3.53e-4	1000	46.18
0.243	3.49e-4	1000	46.89
0.244	3.47e-4	1000	49.38
0.237	3.36e-4	1000	45.29
0.240	3.41e-4	1000	44.94

Table 63: Problem 76 n=10^3 without preconditioning with h=$1e^{-6}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
3.69949e-13	8.6013e-07	5	0.227851
3.69993e-12	2.72013e-06	5	37.443378

Table 64: Problem 76 n=10^3 and n=10^4 with preconditioning with h=$1e^{-4}\,|x_i|$, only x_0

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.249	3.52e-4	1000	50.07
0.241	3.41e-4	1000	49.56
0.246	3.51e-4	1000	47.60
0.245	3.49e-4	1000	50.32
0.244	3.48e-4	1000	49.31
0.248	3.53e-4	1000	46.18
0.243	3.49e-4	1000	46.89
0.244	3.47e-4	1000	49.38
0.237	3.36e-4	1000	45.29
0.240	3.41e-4	1000	44.94

Table 65: Problem 76 n=10^3 without preconditioning with h=$1e^{-4}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
3.02714e-09	7.74203e-05	4	0.181307
4.92682e-10	3.12336e-05	9	0.395027
4.48293e-10	2.97933e-05	9	0.387915
3.80972e-09	8.68533e-05	8	0.348957
1.6797e-09	5.76706e-05	7	0.305337
3.79189e-13	8.66495e-07	9	0.401504

Table 66: Problem 76 n=10^3 with preconditioning with h=$1e^{-2}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
0.254	3.59e-4	1000	44.76
0.237	3.37e-4	1000	49.13
0.247	3.53e-4	1000	48.85
0.242	3.45e-4	1000	48.51
0.244	3.45e-4	1000	45.71
0.235	3.33e-4	1000	49.01
0.243	3.46e-4	1000	47.16
0.243	3.47e-4	1000	47.71
0.237	3.37e-4	1000	44.50
0.237	3.38e-4	1000	45.13

Table 67: Problem 76 n=10^3 without preconditioning with h=$1e^{-2}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]
7.56e-13	1.22e-06	5	32.96

Table 68: Problem 76 n=10^4 with preconditioning with h=$1e^{-2}\,|x_i|$

$f(x_k)$	$\|\|\nabla f(x_k)\|\|$	Iterations	Time [s]

Table 69: Problem 76 n=10^4 without preconditioning with h=$1e^{-2}\,|x_i|$

Parameter	Value
btmax	15
c1	1.000×10^{-4}
delta	1.000×10^{-4}
kmax	1000
mode	"smallestreal"
random_seed	337517
rho	0.500
tolgrad	1.000×10^{-4}

Table 70: Global parameters

A Nelder mead method

A.1 main.m

```
clc
close all
clear

% Set random seed to enhance reproducibility
rng(337517)

% Open menu configuration: choice of function
[f, choice]= menu();

dimensions=[10, 25, 50];
title_plot = ["Extended Freudenstein and Roth function", "Problem
    75", "Problem 76"];

% Parameters of Nelder-Mead method
rho = 1;
chi = 1.1;
gamma = 0.7;
sigma = 0.6;

k = 1;

% Store execution times for time plot
execution_times = zeros(10, length(dimensions));

% Compute Nelder-Mead method for rosenbrock function with different
% starting point
rosenbrock([1.2;1.2], rho, chi, gamma, sigma)
rosenbrock([-1.2;1], rho, chi, gamma, sigma)

for n=dimensions
    % Starting point with respect to the choosen function
    if choice==1
        x0 = 60*ones(n, 1);
        x0(1:2:end) = 90;
    elseif choice==2
        x0 = -1.2*ones(n, 1);
        x0(end) = -1;
    else
        x0 = 2* ones(n, 1);
    end

    % Creation of the hypercube
```

```
    hypercube = x0 + (rand(n, 10) * 2 - 1);

    disp("------------------")
    fprintf("Dimension n=%g\n",n)
    disp("------------------")

    % Nelder-Mead method computed with starting point x0
    [xk, fk, execution_time, iter] = nelder(f,x0, n, rho, chi,
    gamma, sigma);
    print_results(xk, fk, execution_time, iter, title_plot(choice))
    ;

    % Nelder-Mead method computed for each column vector of the
    hypercube
    for j= 1:size(hypercube,2)
        point=hypercube(:,j);
        [xk, fk, execution_time, iter] = nelder(f, point, n, rho,
    chi, gamma, sigma);
        execution_times(j, k) = execution_time;
        print_results(xk, fk, execution_time, iter, title_plot(
    choice));
    end

    k=k+1;
end

% Plot execution time with respect to the dimensions
figure;
hold on;
grid on;
plot(dimensions, mean(execution_times, 1), '-o', 'LineWidth', 1.5);
title('Execution time');
xlabel('Dimension n');
ylabel('Time(s)');
set(gca, 'YScale', 'log');
grid on;
hold off;
```

A.2 menu.m

```
function [f,choice]=menu()

datasets = {'Extended Freudenstein and Roth function', 'Problem 75'
    , 'Problem 76'};

% Pop up the menu with the choice of the function
[choice, ok] = listdlg('PromptString', 'Select a functions:', ...
                       'SelectionMode', 'single', ...
                       'ListString', datasets, ...
                       'ListSize', [250, 250], ...
                       'Name', 'Function selection', ...
                       'OKString', 'Select', ...
                       'CancelString', 'Cancel');

% Check if the user correctly select a function
if ok
    % Choice of the function
    switch choice
        case 1
            f = @(x) extendedFreudensteinRoth(x);
```

```
        case 2
          f = @(x) problem75(x);
        case 3
          f=@(x) problem76(x);
    end
    disp('Function loaded!');
else
    error('Abort operation');
end
```

A.3 nelder.m

```
function [xk, fk, execution_time, k] = nelder(f, x, n, rho, chi,
    gamma, sigma)
    % Parameters
    max_iter = 500000;
    tol = 1e-6;
    k = 1;

    % Simplex generator
    simplex = generate_simplex(x, n);

    % check of the dimensions of input array x and value n
    if length(x) ~= n
        error("The dimension of the array and the number of
    dimensions n must coincide")
    end

    % Store the initial point as one of the points of the simplex (
    used in tuning)
    % if ~isempty(x)
    %     simplex(:, 1) = x;
    % end

    % Only for rosenbrock function: plot 2D
    if n==2
        figure;
        hold on;
        fcontour(@(x, y) (1 - x).^2 + 100 * (y - x.^2).^2, [-2, 2,
    -1, 3], 'LineWidth', 0.6, 'LevelList', logspace(0, 3, 9));
        colormap parula;
        colorbar;
        title('Nelder-Mead Method');
        xlabel('x_1');
        ylabel('x_2');
    end

    % Store all the f values here
    f_values_history = zeros(1,max_iter);

    tic

    % Stopping criteria
    while k <= max_iter && max(vecnorm(simplex - mean(simplex, 2),
    2, 1)) >= tol
        f_values = f(simplex);
        [~, idx] = sort(f_values);
        % Sort simplex points based on f_values
        simplex = simplex(:, idx);

```

```
44          % Store best f_value of the simplex (with lowest f_value)
45          f_values_history(k)=f_values(1);
46
47          % Compute the centroid of all the simplex points minus the
    worst (with highest f_value)
48          centroid = mean(simplex(:, 1:end-1), 2);
49
50          % Reflection
51          x_r = centroid + rho * (centroid - simplex(:, end));
52          f_r = f(x_r);
53
54          % Update worst point with reflected point
55          if f_r < f_values(end-1) && f_r >= f_values(1)
56              simplex(:, end) = x_r;
57
58          elseif f_r < f_values(1)
59              % Expansion
60              x_e = centroid + chi * (x_r - centroid);
61
62              % Update worst point with expanded point
63              if f(x_e) < f_r
64                  simplex(:, end) = x_e;
65              else
66                  simplex(:, end) = x_r;
67              end
68
69          else
70              if f_r < f_values(end)
71                  % Contraction
72                  x_c = centroid + gamma * (x_r - centroid);
73              else
74                  x_c = centroid + gamma * (simplex(:, end) -
    centroid);
75              end
76
77              % Update worst point with contracted point
78              if f(x_c) < f_values(end)
79                  simplex(:, end) = x_c;
80              else
81                  % Shrinkage
82                  simplex(:, 2:end) = simplex(:, 1) + sigma * (
    simplex(:, 2:end) - simplex(:, 1));
83              end
84          end
85
86          % Only for rosenbrock function: plot simplex points
87          if n == 2
88              plot([simplex(1, :), simplex(1, 1)], [simplex(2, :),
    simplex(2, 1)], '-o', 'LineWidth', 1.5);
89          end
90
91          k = k + 1;
92      end
93
94      % Final values
95      xk = simplex(:, 1);
96      fk = f(xk);
97      execution_time=toc;
98      f_values_history= f_values_history(:,1:k);
99
100     % Objective function with respect to the k-th iteration
```

```
    % figure;
    % semilogy(1:k, f_values_history, '-o', 'LineWidth', 1.5, '
    MarkerSize', 5);
    % title(sprintf("Objective function value with n=%g", n));
    % xlabel('Iterations');
    % ylabel('f(x)');
    % set(gca, 'XScale', 'log')
    % set(gca, 'YScale', 'log')
    % grid on;
end
```

A.4 nelder3d.m

```
function [xk, fk, execution_time, k] = nelder3D(f, x, n, rho, chi,
    gamma, sigma)
    % Parameters
    max_iter = 500000;
    tol = 1e-6;
    k = 1;

    % Simplex generator
    simplex = generate_simplex(x, n);

    % check of the dimensions of input array x and value n
    if length(x) ~= n
        error("The dimension of the array and the number of
    dimensions n must coincide")
    end

    % Store the initial point as one of the points of the simplex (
    used in tuning)
    % if ~isempty(x)
    %     simplex(:, 1) = x;
    % end

    % Only for Rosenbrock function 3D plot
    if n == 2
        plot_nelder_mead_3d(f, [-2, 2], [-1, 3], [0, 2500]);
        hold on;
    end

    % Store all the f values here
    f_values_history = zeros(1,max_iter);

    tic

    % Stopping criteria
    while k <= max_iter && max(vecnorm(simplex - mean(simplex, 2),
    2, 1)) >= tol
        f_values = f(simplex);
        [~, idx] = sort(f_values);
        % Sort simplex points based on f_values
        simplex = simplex(:, idx);

        % Store best f_value of the simplex (with lowest f_value) (
    with highest f_value)
        f_values_history(k)=f_values(1);

        % Compute the centroid of all the simplex points minus the
    worst
```

```
        centroid = mean(simplex(:, 1:end-1), 2);

        % Reflection
        x_r = centroid + rho * (centroid - simplex(:, end));
        f_r = f(x_r);

        % Update worst point with reflected point
        if f_r < f_values(end-1) && f_r >= f_values(1)
            simplex(:, end) = x_r;

        elseif f_r < f_values(1)
            % Expansion
            x_e = centroid + chi * (x_r - centroid);

            % Update worst point with expanded point
            if f(x_e) < f_r
                simplex(:, end) = x_e;
            else
                simplex(:, end) = x_r;
            end

        else
            if f_r < f_values(end)
                % Contraction
                x_c = centroid + gamma * (x_r - centroid);
            else
                x_c = centroid + gamma * (simplex(:, end) -
    centroid);
            end

            % Update worst point with contracted point
            if f(x_c) < f_values(end)
                simplex(:, end) = x_c;
            else
                % Shrinkage
                simplex(:, 2:end) = simplex(:, 1) + sigma * (
    simplex(:, 2:end) - simplex(:, 1));
            end
        end

        % Only for rosenbrock function: plot simplex points
        if n == 2
            simplex_z = arrayfun(@(i) f(simplex(:, i)), 1:size(
    simplex, 2));
            plot3([simplex(1, :), simplex(1, 1)], ...
                  [simplex(2, :), simplex(2, 1)], ...
                  [simplex_z, simplex_z(1)], '-o', 'MarkerSize', 6)
    ;
        end

        k = k + 1;
     end

     % Final values
     xk = simplex(:, 1);
     fk = f(xk);
     execution_time=toc;
     f_values_history= f_values_history(:,1:k);

     % Objective function with respect to the k-th iteration
     figure;
```

```
    semilogy(1:k, f_values_history, '-o', 'LineWidth', 1.5, '
    MarkerSize', 5);
    title(sprintf("Objective function value with n=%g", n));
    xlabel('Iterazioni');
    ylabel('f(x)');
    set(gca, 'XScale', 'log')
    set(gca, 'YScale', 'log')
    grid on;
end
```

A.5 plot_nelder_mead_3d.m

```
function plot_nelder_mead_3d(f, xlim, ylim, zlim)

    % xlim and ylim are the limits of the plot for x and y axis
    [X, Y] = meshgrid(linspace(xlim(1), xlim(2), 100), linspace(
    ylim(1), ylim(2), 100));
    Z = arrayfun(@(x, y) f([x; y]), X, Y);

    figure;
    hold on;
    surf(X, Y, Z, 'FaceAlpha', 0.7, 'EdgeColor', 'none');
    colormap turbo;
    colorbar;
    title('3D Visualization of Nelder-Mead Optimization');
    xlabel('x_1');
    ylabel('x_2');
    zlabel('f(x)');
    view(135, 30);
    shading interp;
end
```

A.6 generate_simplex.m

```
function simplex = generate_simplex(x, n)
    simplex = rand(n, n+1);
    simplex(:, 1) = x;

    % Check if points are affinely independent, if the rank of the
    % differential matrix is not n, it regenerate the simplex
    while rank(simplex(:, 2:end) - simplex(:, 1)) < n
        simplex = rand(n, n+1);
        simplex(:, 1) = x;
    end
end
```

A.7 print_results.m

```
function print_results(xk, fk, execution_time, k, title)
    fprintf('**** NELDER MEAD METHOD: FINISHED POINT: %s\n', title)
    ;
    fprintf('**** NELDER MEAD METHOD: RESULTS *****\n');
    % fprintf('xk: %f;\n', xk);
    fprintf('f(xk): %g;\n', fk);
    fprintf('N. of Iterations: %d;\n', k);
    fprintf('Elapsed time is %g seconds.\n\n', execution_time);
end
```

A.8 tuning.m

```
clear
close all
clc

% Dimensions
n = 10;
% Iterations
N = 500;

rho_range = 0.5:0.1:1.3;
chi_range = 1.1:0.1:1.9;
gamma_range = 0.1:0.1:0.9;
sigma_range = 0.1:0.1:0.9;

steps = zeros(1, N);
res = zeros(1, N);
avg_steps = zeros(1, length(rho_range));
avg_res = zeros(1, length(rho_range));

% Select function
[f,choice]= menu();

% Parameters to check
rho = 1;
chi = 1.1;
gamma = 0.7;
sigma = 0.6;

i = 1;

for rho = rho_range
    for j = 1:N
        [xk, fk, execution_time, k] = ...
            nelder(f, [], n, rho, chi, gamma, sigma);
        steps(j) = k;
        res(j) = fk;
    end
    avg_steps(i) = mean(steps);
    avg_res(i) = mean(res);
    i = i+1;
    disp(i);
end

% Display the average number of iteration with respect to parameter
figure;
plot(rho_range, avg_steps, '-o', 'LineWidth', 1.2);
title("Number of iterations");
xlabel('rho');
ylabel('avg steps');
grid on;

% Display the average value of f with respect to parameter
figure;
plot(rho_range, avg_res, '-o', 'LineWidth', 1.2);
title("Convergence");
xlabel('rho');
ylabel('avg f(xk)');
set(gca, 'YScale', 'log')
grid on;
```

A.9 rosenbrock.m

```
function rosenbrock(x0, rho, chi, gamma, sigma)
    % Starting point and relative f value
    f = @(x) rosenbrock_function(x);
    fprintf('x0: %f;\n', x0);
    fprintf('f(x0): %f;\n', f(x0));

    [xk, fk, execution_time, k] = nelder(f, x0, 2, rho, chi, gamma,
    sigma);
    print_results(xk, fk, execution_time, k, "Rosenbrock function 2
   D");
end
```

A.10 rosenbrock_function.m

```
function f=rosenbrock_function(x)
    if size(x,2)==1
        % If x is a column vector
        f= 100*(x(2)-x(1).^2).^2 + (1-x(1)).^2;
    else
        % If x is a simplex
        f=100*(x(2,:)-x(1,:).^2).^2+(1-x(1,:)).^2;
        f=f';
    end
end
```

A.11 extendedFreudensteinRoth.m

```
function F = extendedFreudensteinRoth(simplex)
    [n, m] = size(simplex);

    % F is a column vector whose values are f evaluations in each
   column of
    % the simplex
    F = zeros(m, 1);

    % For each point of the simplex
    for i = 1:m
        x = simplex(:, i);
        Fi = 0;

        % Compute F for i-th point of the simplex
        for k = 1:n
            if mod(k, 2) == 1
                if k == n
                    fk = x(k) - 13;
                else
                    fk = x(k) + ((5 - x(k+1)) * x(k+1) - 2) * x(k
   +1) - 13;
                end
            else
                fk = x(k-1) + ((x(k) + 1) * x(k) - 14) * x(k) - 29;
            end
            Fi = Fi + 0.5 * fk^2;
        end

        F(i) = Fi;
    end
end
```

A.12 problem75.m

```
function F = problem75(simplex)
    [n, m] = size(simplex);

    % F is a column vector whose values are f evaluations in each
    column of
    % the simplex
    F = zeros(m, 1);

    % For each point of the simplex
    for k = 1:m
        x = simplex(:, k);
        f_k = zeros(1, n);
        f_k(1) = x(1) - 1;

        % Compute F for i-th point of the simplex
        for i = 2:n
            f_k(i) = 10 * (i - 1) * (x(i) - x(i - 1))^2;
        end
        F(k) = 0.5 * sum(f_k.^2);
    end
end
```

A.13 problem76.m

```
function F = problem76(X)
    [n, m] = size(X);

    % F is a column vector whose values are f evaluations in each
    column of
    % the simplex
    F = zeros(m, 1);

    % For each point of the simplex
    for i = 1:m
        x = X(:, i);
        fk = zeros(n, 1);

        % Compute F for i-th point of the simplex
        for k = 1:n-1
            fk(k) = x(k) - (x(k+1)^2) / 10;
        end

        fk(n) = x(n) - (x(1)^2) / 10;
        F(i) = 0.5 * sum(fk.^2);
    end
end
```

B Modified Newton method

B.1 main_modified_newton.m

```
clear
close all
clc

load("global_parameters.mat")
warning('off', 'all')
```

```
7
8  % Parameters
9  h=1e-2;
10 type = "fw";
11 rng(random_seed)
12
13 % Menu configuration
14 [f, x0, grad_f, Hess_f, c1, mode, opts, title_function, choice,
       prec_choice]= menu_modified(h, type);
15
16 % Modified newton method on rosenbrock function in two different
       starting
17 % point
18 rosenbrock([1.2; 1.2], 100, tolgrad, 1e-4, rho, btmax, delta, "
       smallestreal", struct(), prec_choice);
19 rosenbrock([-1.2; 1], 100, tolgrad, 1e-4, rho, btmax, delta, "
       smallestreal", struct(), prec_choice);
20
21 % n = 1000, n = 10000, n = 100000
22 for n = logspace(3, 5, 3)
23     disp('------------------')
24     disp(['DIMENSION:',mat2str(n)])
25     disp('------------------')
26
27     % Starting point with respect to dimension n
28     x0_val = x0(n);
29
30     % Execution starting time
31     tic
32     % Modified newton method in starting point x0
33     [xk, fk, gradfk_norm, k, failure] = ...
34         modified_newton_bcktrck(n, x0_val, f, grad_f, Hess_f, kmax,
       tolgrad, c1, rho, btmax, delta, mode, opts,prec_choice);
35     % Execution ending time
36     toc
37     print_results(xk, fk, gradfk_norm, k, kmax, title_function,
       failure);
38
39     % Generate hypercube
40     random_points = x0_val + (rand(n, 10) * 2 - 1);
41
42     % Modified newton method for each column vector of the
       hypercube
43     for i = 1:10
44         tic
45         [xk, fk, gradfk_norm, k, failure] = ...
46             modified_newton_bcktrck(n, random_points(:,i), f,
       grad_f, Hess_f, kmax, tolgrad, c1, rho, btmax, delta, mode, opts
       ,prec_choice);
47         toc
48         print_results(xk, fk, gradfk_norm, k, kmax, title_function,
       failure);
49     end
50 end
51
52 if n == 2
53     if choice == 1
54         [X, Y] = meshgrid(linspace(-6200, 10, 1000), linspace(-60,
       60, 1000));
55     else
56         [X, Y] = meshgrid(linspace(-6, 6, 100), linspace(-6, 6,
```

```
    100));
57      end
58      Z = arrayfun(@(x, y) f([x; y]), X, Y);
59
60      % Contour Plot
61      figure;
62      contour(X, Y, Z, 30); hold on;
63      plot(xseq(1, :), xseq(2, :), '--*r');
64      title('Contour Plot with trajectory');
65      xlabel('x'); ylabel('y');
66      hold off;
67
68      % Surface Plot
69      figure;
70      surf(X, Y, Z, 'EdgeColor', 'none'); hold on;
71      plot3(xseq(1, :), xseq(2, :), arrayfun(@(i) f(xseq(:, i)), 1:
    size(xseq, 2)), '--*r');
72      title('Function with descent direction');
73      xlabel('x'); ylabel('y'); zlabel('f(x)');
74      hold off;
75  end
```

B.2 menu_modified.m

```
1   function [f, x0, grad, H, c1, mode, opts, title_function, choice,
    prec_choice] = menu_modified(h, type)
2
3       % Menu choice of the function
4       datasets = {'Extended Freudenstein and Roth function', 'Problem
        75', 'Problem 76'};
5       [choice, ok] = listdlg('PromptString', 'Select a function:', ...
6                              'SelectionMode', 'single', ...
7                              'ListString', datasets, ...
8                              'ListSize', [250, 250], ...
9                              'Name', 'Function selection', ...
10                             'OKString', 'Select', ...
11                             'CancelString', 'Cancel');
12
13      if ~ok
14          error('Abort operation');
15      end
16
17      % Menu choice of derivative method
18      deriv_methods = {'Analytical derivatives', 'Finite differences'
    };
19      [deriv_choice, flag] = listdlg('PromptString', 'Select a
    differentiation method:', ...
20                                     'SelectionMode', 'single', ...
21                                     'ListString', deriv_methods, ...
22                                     'ListSize', [250, 250], ...
23                                     'Name', 'Differentiation method'
    , ...
24                                     'OKString', 'Select', ...
25                                     'CancelString', 'Cancel');
26
27      if ~flag
28          error('Abort operation');
29      end
30
31      % Menu choice of preconditioning method
```

```
32      prec_methods = {'Use preconditioning', 'Do not use
    preconditioning'};
33      [prec_choice, prec] = listdlg('PromptString', 'Select a
    preconditioning method:', ...
34                                    'SelectionMode', 'single', ...
35                                    'ListString', prec_methods, ...
36                                    'ListSize', [250, 250], ...
37                                    'Name', 'Preconditioning method'
    , ...
38                                    'OKString', 'Select', ...
39                                    'CancelString', 'Cancel');
40
41      if ~prec
42          error('Abort operation');
43      end
44
45      % Choice of the function and its parameters
46      switch choice
47          case 1
48              f = @(x) extended_freudenstein_newton(x);
49              x0 = @(n) (60 * ones(n, 1)) .* (mod((1:n)', 2) == 0) +
    (90 * ones(n, 1)) .* (mod((1:n)', 2) ~= 0);
50              c1=1e-6;
51              mode='smallestreal';
52              opts.maxit=1000;
53              opts.tol=1e-6;
54              title_function = 'Extended Freudenstein and Roth
    function';
55
56              if deriv_choice == 1
57                  grad= @(x) extended_freudenstein_grad(x);
58                  H = @(x) extended_freudenstein_hessian(x);
59
60              else
61                  grad=@(x) findiff_gradf(f, x, h, type);
62                  H = @(x) extended_freudenstein_findiff_hessian(f, x
    , sqrt(h));
63              end
64
65          case 2
66              f = @(x) problem75_newton(x);
67              x0 = @(n) [-1.2 * ones(n-1, 1); -1];
68              c1=1e-4;
69              mode="smallestabs";
70              opts=struct();
71              title_function = 'Problem 75';
72
73              if deriv_choice == 1
74                  grad= @(x)problem75_grad(x);
75                  H = @(x) problem75_hessian(x);
76              else
77                  mode = "smallestreal";
78                  opts.maxit=1000;
79                  opts.tol=1e-6;
80                  grad=@(x) findiff_gradf(f, x, h, type);
81                  H = @(x) problem75_findiff_hessian(f, x, sqrt(h));
82              end
83
84          case 3
85              f = @(x) problem76_newton(x);
86              x0 =@(n) 2* ones(n, 1);
```

```
            c1=1e-4;
            mode='smallestreal';
            opts.maxit=1000;
            opts.tol=1e-6;
            title_function = 'Problem 76';

            if deriv_choice == 1
                grad=@(x) problem76_grad(x);
                H = @(x) problem76_hessian(x);
            else
                grad=@(x)findiff_gradf(f, x, h, type);
                H = @(x) problem76_findiff_hessian(f, x, sqrt(h));
            end
    end

    disp('Function and differentiation method loaded!');
end
```

B.3 modified_newton_bcktrck.m

```
function [xk, fk, gradfk_norm, k, failure] =
    modified_newton_bcktrck(n, x0, f, gradf, Hessf, kmax, tolgrad,
    c1, rho, btmax, delta, mode, opts,prec_choice)

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

% Store all the values of backtracking iterations
btseq = zeros(1, kmax);
% Store all the values for f history plot
f_values_history = zeros(1,kmax);
% Sequence of xk (used for plots, to try it, add xseq to output
    parameters)
% xseq = zeros(n, kmax);

% Flags
failure = false;
bcktrck = false;
eigFound = false;

% Starting values
xk = x0;
fk = f(xk);
f_values_history(1) = fk;
gradfk = gradf(xk);
k = 0;
gradfk_norm = norm(gradfk);

% Stopping criteria
while k < kmax && gradfk_norm >= tolgrad
    % Compute hessian matrix
    Hk = Hessf(xk);
    Bk = Hk;

    % Try to compute the incomplete cholesky factorization
    try
        L = ichol(Bk, struct('type', 'nofill'));
    % If it fails because the matrix is not positive definite, make
    it
    % positive definite
    catch ME
```

```
37          if contains(ME.message, 'nonpositive pivot') || contains(ME
    .message, 'nonsingular')
38              % If the eigs function do not converge to a minimum try
     to
39              % increase the number of max iteration until an upper
    bound
40              while opts.maxit <= 1e+5 && eigFound == false
41                  eigFound = true;
42                  % Compute smallest eigenvalue
43                  lambda_min = eigs(Hk,1,mode,opts);
44                  % If eigs fails, it returns NaN so try it again
    increasing
45                  % maxIterations
46                  if isnan(lambda_min)
47                      opts.maxit = opts.maxit * 10;
48                      eigFound = false;
49                  end
50              end
51              % If eigs completely fails exit from the method
52              if eigFound == false
53                  failure = true;
54                  disp("Error during eigs computation.")
55                  break;
56              end
57
58              if lambda_min <= 0
59                  % Correction of the Hessian matrix
60                  tau_k = max(0, delta - lambda_min);
61                  Bk = Hk + tau_k * speye(n);
62              end
63              % Recompute incomplete cholesky factorization
64              if prec_choice==1
65                  try
66                      L = ichol(Bk, struct('type', 'nofill'));
67                  catch ME
68                      failure = true;
69                      fprintf("Error during ichol computation: %s\n",
     ME.message);
70                      break;
71                  end
72              end
73          else
74              % If the error of ichol is not "the input matrix is not
75              % positive definite" throw an error
76              failure = true;
77              fprintf("Error during ichol computation: %s\n", ME.
    message);
78              break;
79          end
80      end
81
82      % If the user selects the preconditioning option
83      if prec_choice==1
84          % compute precodnitioning
85          tol = 1e-4;
86          maxIter = 100;
87          [pk,~] = pcg(Bk, -gradfk, tol, maxIter, L, L');
88      else
89          % compute pk without preconditioning
90          pk = -Bk \ gradfk;
91      end
```

```
92
93      % Compute parameters for Armijo condition for backtracking
        strategy
94      alpha = 1;
95
96      xnew = xk + alpha * pk;
97      fnew = f(xnew);
98
99      c1_gradfk_pk = c1 * gradfk' * pk;
100     bt = 0;
101
102     % Backtracking strategy
103     while bt < btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
104         alpha = rho * alpha;
105         xnew = xk + alpha * pk;
106         fnew = f(xnew);
107         bt = bt + 1;
108         % flag used to know if plot backtrack iterations
109         bcktrck = true;
110     end
111
112     % If Armijo condition is not satisfied and bt index reaches the
        maximum
113     % number of iterations exit from the method
114     if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
115         failure = true;
116         disp("Max backtrack iterations reached")
117         break;
118     end
119
120     % Update all the parameters for the next iteration
121     k = k + 1;
122     xk = xnew;
123     fk = fnew;
124     f_values_history(k) = fk;
125     gradfk = gradf(xk);
126     gradfk_norm = norm(gradfk);
127     btseq(k) = bt;
128     % xseq(:, k) = xk;
129 end
130
131 % Reshape of the backtrack vector
132 btseq = btseq(1:k);
133 f_values_history = f_values_history(1:k);
134 % xseq = xseq(:, 1:k);
135
136 % If there are not errors and the backtracking strategy has been
        done al
137 % least one time plot backtrack values
138 if failure == false && bcktrck == true
139     figure;
140     bar(btseq);
141     title('Backtracking iterations');
142     xlabel('Iterations of modified method');
143     ylabel('Iterations of backtracking strategy');
144     grid on;
145 end
146
147 % Objective function with respect to the k-th iteration
148 figure;
149 semilogy(1:k, f_values_history, '-o', 'LineWidth', 1.5, 'MarkerSize
```

```
    ', 5);
title(sprintf("Objective function value with n=%g", n));
xlabel('Iterazioni');
ylabel('f(x)');
set(gca, 'XScale', 'log')
set(gca, 'YScale', 'log')
grid on;

end
```

B.4 print_results.m

```
function print_results(xk, fk, gradfk_norm, k, kmax, title, failure)
    fprintf('**** MODIFIED NEWTON METHOD: FINISHED POINT: %s\n', title);
    fprintf('**** MODIFIED NEWTON METHOD: RESULTS *****\n');

    % Print results only if there are no errors in modified newton method
    if failure == false
        % fprintf('xk: %s;\n', mat2str(xk));
        fprintf('f(xk): %g;\n', fk);
        fprintf('Gradient norm: %g;\n', gradfk_norm);
        fprintf('N. of Iterations: %d/%d;\n\n', k, kmax);
    end
end
```

B.5 rosenbrock.m

```
function rosenbrock(x0, kmax, tolgrad, c1, rho, btmax, delta, mode, opts,prec_choice)
    f = @(x) 100*(x(2)-x(1).^2).^2 + (1-x(1)).^2;
    grad_f = @(x) [-400*x(1)*(x(2) - x(1)^2) + 2*(x(1) - 1);
                   200*(x(2) - x(1)^2)];
    Hess_f = @(x) sparse([1200*x(1)^2 - 400*x(2) + 2, -400*x(1);
                   -400*x(1), 200]);

    [xk, fk, gradfk_norm, k, failure] = ...
    modified_newton_bcktrck(2, x0, f, grad_f, Hess_f, kmax, tolgrad, c1, rho, btmax, delta, mode, opts,prec_choice);
    print_results(xk, fk, gradfk_norm, k, kmax, "Rosenbrock function 2D", failure);
end
```

B.6 extended_freudenstein_newton.m

```
function F = extended_freudenstein_newton(x)
    [m, ~] = size(x);

    F = 0;
    for k = 1:m
        if mod(k, 2) == 1
            if k == m
                fk = x(k) - 13;
            else
                fk = x(k) + ((5 - x(k+1)) * x(k+1) - 2) * x(k+1) - 13;
```

```
            end
        else
            fk = x(k-1) + ((x(k) + 1) * x(k) - 14) * x(k) - 29;
        end
         F =F+ fk^2;
    end
    F=0.5*F;
end
```

B.7 extended_freudenstein_hessian.m

```
function H = extended_freudenstein_hessian(x)
    n = size(x, 1);

    % vector of principal diagonal and first upper diagonal
    vectDiag0 = zeros(1, n);
    vectDiag1 = zeros(1, n-1);

    for k = 1:n
        if mod(k, 2) == 1
                vectDiag0(k) = 2;
            if k < n
                vectDiag1(k) = 12*x(k+1)-16;
            end
        else
            vectDiag0(k) = 30*x(k)^4 -80*x(k)^3 +12*x(k)^2 -240*x(k) +12*x(k-1) +12;
        end
    end
    H = sparse(1:n,1:n,vectDiag0,n,n) + sparse(2:n,1:n-1,vectDiag1,n,n) + sparse(1:n-1,2:n,vectDiag1,n,n);

end
```

B.8 extended_freudenstein_grad.m

```
function grad = extended_freudenstein_grad(x)
    n = size(x, 1);
    grad = zeros(n, 1);

    for k = 1:n
        if mod(k, 2) == 1

            if k < n
                grad(k)= x(k)+((5-x(k+1))*x(k+1)-2)*x(k+1)-13 + x(k)+((x(k+1)+1)*x(k+1)-14)*x(k+1)-29;
            else
                grad(k)= x(k)-13 + x(k)-29;
            end
        else
            grad(k)=6*x(k)^5 -20*x(k)^4 +  4 *x(k)^3 -120*x(k)^2 +12*x(k) +12*x(k)*x(k-1) -16*x(k-1) +432;
        end
    end
end
```

B.9 extended_freudenstein_findiff_hessian

```
function [H] = extended_freudenstein_findiff_hessian(f, x, h)
n = length(x);

% vector of principal diagonal and first upper diagonal
Hessf0 = zeros(1,n);
Hessf1= zeros(1,n-1);

for j=1:n
    % Elements on the principal diagonal
    xh_plus = x;
    xh_minus = x;
    xh_plus(j) = xh_plus(j) + h;
    xh_minus(j) = xh_minus(j) - h;
    Hessf0(j) = (f(xh_plus) - 2*f(x) + f(xh_minus))/(h^2);
    if mod(j,2)~=0
        % Elements on the first upper diagonal
        i=j+1;
        xh_plus_ij = x;
        xh_plus_ij([i, j]) = xh_plus_ij([i, j]) + h;
        xh_plus_i = x;
        xh_plus_i(i) = xh_plus_i(i) + h;
        xh_plus_j = x;
        xh_plus_j(j) = xh_plus_j(j) + h;
        Hessf1(j) = (f(xh_plus_ij) - ...
            f(xh_plus_i) - f(xh_plus_j) + f(x))/(h^2);
    end
end
    % Creation of the sparse hessian
    H = sparse(1:n,1:n,Hessf0,n,n) + sparse(2:n,1:n-1,Hessf1,n,n) +
    sparse(1:n-1,2:n,Hessf1,n,n);
end
```

B.10 problem75_newton.m

```
function F = problem75_newton(x)
    n = length(x);

    f_k = zeros(n, 1);
    f_k(1) = x(1) - 1;

    for i = 2:n
        f_k(i) = 10 * (i - 1) * (x(i) - x(i - 1))^2; % Caso 1 < k
       n
    end

    F = 0.5 * sum(f_k.^2);
end
```

B.11 problem75_grad.m

```
function grad = problem75_grad(x)
    n = size(x, 1);
    grad = zeros(n, 1);

    for k = 1:n
        if k == 1
            grad(k) = x(k)-1 - 10 * (x(k+1)-x(k));
        elseif k == n
            grad(k) = 10 * (k-1) * (x(k)-x(k-1));
```

```
        else
            grad(k) = 10 * (k-1) * (x(k)-x(k-1)) - 10 * k *(x(k+1)-
    x(k));
        end
    end
end
```

B.12 problem75_findiff_hessian.m

```
function [H] = problem75_findiff_hessian(f, x, h)
    n = length(x);

    % vector of principal diagonal and first upper diagonal
    Hessf0 = zeros(1, n);
    Hessf1 = zeros(1, n-1);

    for j = 1:n
        % Elements on the principal diagonal
        x_plus = x;
        x_plus(j) = x_plus(j) + h;
        x_minus = x;
        x_minus(j) = x_minus(j) - h;
        Hessf0(j) = (f(x_plus) - 2*f(x) + f(x_minus)) / h^2;

        if j < n
        % Elements on the first upper diagonal
        i=j+1;
        xh_plus_ij = x;
        xh_plus_ij([i, j]) = xh_plus_ij([i, j]) + h;
        xh_plus_i = x;
        xh_plus_i(i) = xh_plus_i(i) + h;
        xh_plus_j = x;
        xh_plus_j(j) = xh_plus_j(j) + h;
        Hessf1(j) = (f(xh_plus_ij) - ...
            f(xh_plus_i) - f(xh_plus_j) + f(x))/(h^2);
        end
    end

    % Creation of the sparse hessian
    H = sparse(1:n, 1:n, Hessf0, n, n) + ...
        sparse(2:n, 1:n-1, Hessf1, n, n) + ...
        sparse(1:n-1, 2:n, Hessf1, n, n);
end
```

B.13 problem75_hessian.m

```
function H = problem75_hessian(x)
    n = size(x, 1);

    % vector of principal diagonal and first upper diagonal
    vectDiag0 = zeros(1,n);
    vectDiag1 = zeros(1,n-1);

    for k = 1:n
        if k == 1
            vectDiag0(k) = (10*k)+1;
            vectDiag1(k) = -10*k;
        elseif k == n
            vectDiag0(k) = 10*k;
```

```
        else
            vectDiag0(k) = 10*(k-1) + 10*k;
            vectDiag1(k) = -10*k;
        end
    end
    H = sparse(1:n,1:n,vectDiag0,n,n) + sparse(2:n,1:n-1,vectDiag1,
   n,n) + sparse(1:n-1,2:n,vectDiag1,n,n);
end
```

B.14 problem76_newton.m

```
function F_val = problem76_newton(x)
    n = length(x);
    fk = zeros(n, 1);

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

    fk(n) = x(n) - (x(1)^2) / 10;
    F_val = 0.5 * sum(fk.^2);
end
```

B.15 problem76_grad.m

```
function grad = problem76_grad(x)
    n = size(x, 1);
    grad = zeros(n, 1);

    % Transform x in a circular vector
    x=[x(n); x; x(1)];

    for k = 2:n+1
        grad(k-1)= (x(k)^3)/50 - (x(k)*x(k-1))/5 + x(k) -(x(k+1)^2)
   /10;
    end
end
```

B.16 problem76_hessian.m

```
function H = problem76_hessian(x)
    n = size(x, 1);

    % vector of principal diagonal and first upper diagonal
    vectDiag0 = zeros(1, n);
    vectDiag1 = zeros(1, n-1);

    for k = 1:n
        if k > 1
            vectDiag0(k)=3/50*x(k)^2 -x(k-1)/5+1;
        else
            vectDiag0(k)=3/50*x(k)^2 -x(n)/5+1;
        end
        if k < n
            vectDiag1(k) = -x(k+1)/5;
        end
    end

```

```
    % Creation of the sparse hessian
    H = sparse(1:n,1:n,vectDiag0,n,n) + sparse(2:n,1:n-1,vectDiag1,
    n,n) + sparse(1:n-1,2:n,vectDiag1,n,n);
    % Add elements in position (1,n) and (n,1) based on the
    structure of the hessian
    H(1,n) = - x(1)/5;
    H(n,1) = -x(1)/5;
end
```

B.17 problem76_findiff_hessian.m

```
function [H] = problem76_findiff_hessian(f, x, h)
n = length(x);

% vector of principal diagonal and first upper diagonal
Hessf0 = zeros(1,n);
Hessf1= zeros(1,n-1);

for j=1:n
    % Elements on the principal diagonal
    xh_plus = x;
    xh_minus = x;
    xh_plus(j) = xh_plus(j) + h;
    xh_minus(j) = xh_minus(j) - h;
    Hessf0(j) = (f(xh_plus) - 2*f(x) + f(xh_minus))/(h^2);

    if j < n
        % Elements on the first upper diagonal
        i=j+1;
        xh_plus_ij = x;
        xh_plus_ij([i, j]) = xh_plus_ij([i, j]) + h;
        xh_plus_i = x;
        xh_plus_i(i) = xh_plus_i(i) + h;
        xh_plus_j = x;
        xh_plus_j(j) = xh_plus_j(j) + h;
        Hessf1(j) = (f(xh_plus_ij) - ...
            f(xh_plus_i) - f(xh_plus_j) + f(x))/(h^2);
    end

end

% Creation of the sparse hessian
H = sparse(1:n,1:n,Hessf0,n,n) + sparse(2:n,1:n-1,Hessf1,n,n) +
    sparse(1:n-1,2:n,Hessf1,n,n);

% Add elements in position (1,n) and (n,1) based on the structure
    of the hessian
xh_plus_ij = x;
xh_plus_ij([1, n]) = xh_plus_ij([1, n]) + h;
xh_plus_i = x;
xh_plus_i(1) = xh_plus_i(1) + h;
xh_plus_j = x;
xh_plus_j(n) = xh_plus_j(n) + h;
H(1,n) = (f(xh_plus_ij) - ...
    f(xh_plus_i) - f(xh_plus_j) + f(x))/(h^2);

H(n,1) = H(1,n);

end
```

B.18 findiff_gradf.m

```
function [gradfx] = findiff_gradf(f, x, h, type)

gradfx = zeros(size(x));

switch type
    % forward finite differences
    case 'fw'
        for i=1:length(x)
            xh = x;
            xh(i) = xh(i) + h;
            gradfx(i) = (f(xh) - f(x))/ h;

        end
    % centered finite differences
    case 'c'
        for i=1:length(x)
            xh_plus = x;
            xh_minus = x;
            xh_plus(i) = xh_plus(i) + h;
            xh_minus(i) = xh_minus(i) - h;
            gradfx(i) = (f(xh_plus) - f(xh_minus))/(2 * h);
        end
    % backward finite differences
    otherwise
        for i=1:length(x)
            xh = x;
            xh(i) = xh(i) + h;
            gradfx(i) = (f(xh) - f(x))/h;
        end
end
end
```