Assignment on unconstrained optimization

Andrea Cucchietti
s327134
Collegio di Ingegneria Informatica, del Cinema e Meccatronica
Politecnico di Torino
s327134@studenti.polito.it

Davide Elio Stefano Demicheli
s331531
Collegio di Ingegneria Informatica, del Cinema e Meccatronica
Politecnico di Torino
s331531@studenti.polito.it

Shakti Singh Rathore
s328222
Collegio di Ingegneria Informatica, del Cinema e Meccatronica
Politecnico di Torino
s328222@studenti.polito.it

***Abstract*—In this report we introduce the application of two optimization techniques, specifically the Nelder-Mead and Truncated Newton methods, to solve three distinct unconstrained minimization problems. The aim is to test and thoroughly compare the methods and their performance with respect to the expected theoretical behavior, using different values for the parameters and starting points. Regarding the case of the Truncated Newton method, results obtained using its preconditioned version and matrix-free implementation are also discussed.**

I. Methods

A. Truncated Newton method

The Truncated Newton method is an optimization method that computes at each step p_k, the approximate minimizer of the local quadratic model. The exact minimizer is solution of the equation (1).

$$H_f(x^{(k)}) \cdot p_k = -\nabla f(x^{(k)}) \tag{1}$$

The algorithm consists of two loops. The outer one uses p_k to update the current estimation. The inner loop contains a variant of the conjugate gradient method computing p_k. Furthermore, the CG method is interrupted if a direction gives a nonpositive value for $d^T \cdot H_f(x^{(k)}) \cdot d$. If this condition is verified, the inner loop returns the last result obtained, or $-\nabla f(x^{(k)})$ if this happens at the first iteration.
As a consequence, p_k is always a descent direction, even if $H_f(x^{(k)})$ is not positive definite.
The system is not solved exactly, the conjugate gradient method stops when a certain tolerance is reached. This threshold is dynamic, and it decreases while we are moving towards a smaller gradient. In this way, we do not "oversolve" the system when the x_k is distant from the solution.
Once the new p_k is defined, a backtracking strategy checking the Armijo conditions is performed.
The Matlab code is available in the Appendix in the listing 1.

1) Preconditioning: The conjugate gradient method in the inner loop is sensitive to the condition number of $H_f(x^{(k)})$. For this reason, especially when the condition number is large, solving the linear system using a preconditioner is a beneficial choice. The matrix M is obtained using the Incomplete LU factorization. Since the Hessian is not always positive definite, the Incomplete Cholesky factorization is not an option. The preconditioner was integrated into the algorithm as described in [1].
The Matlab code is available in the Appendix in the listing 2.

2) Matrix-free: The Hessian-free implementation does not use the explicit matrix $H_f(x^{(k)})$.
It exploits the fact that the Hessian is never present alone in the algorithm, but it is always multiplied by a vector. In particular, the product $H_f(x^{(k)}) \cdot p_k$ can be approximated as shown in (2).

$$H_f(x^{(k)}) \cdot p_k \approx \frac{\nabla f(x^{(k)} + h \cdot p_k) - \nabla f(x^{(k)})}{h} \tag{2}$$

The Matlab code is available in the Appendix in the listing 3.

B. Nelder-Mead method

The Nelder-Mead method is an optimization method, and, in this case, it is applied to minimization problems. It is a direct method and it is based on the concept of simplex.
At each step, the simplex is defined by $n+1$ points $x_i^{(k)}$ that are sorted by increasing value of $f(x_i^{(k)})$, where n is the dimensionality of the problem. $x_i^{(k)}$ refers to the i-th point at step k according to the sorting.
The points are then modified by the following operations:

- Reflection (4), the first operation of every new iteration.

$$\overline{x}^{(k)} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i^{(k)} \tag{3}$$

$$x_R^{(k)} = \overline{x}^{(k)} + \rho \cdot (\overline{x}^{(k)} - x_{n+1}^{(k)}) \tag{4}$$

if

$$f(x_1^{(k)}) \leq f(x_R^{(k)}) < f(x_n^{(k)})$$

then

$$x_{n+1}^{(k+1)} = x_R^{(k)}$$

- Expansion (5), performed if $f(x_R^{(k)}) < f(x_1^{(k)})$-

$$x_E^{(k)} = \overline{x}^{(k)} + \chi \cdot (x_R^{(k)} - \overline{x}^{(k)}) \tag{5}$$

$$x_{n+1}^{(k+1)} = argmax(f(x_E^{(k)}), f(x_R^{(k)}))$$

- Contraction, performed if instead $f(x_R^{(k)}) >= f(x_n^{(k)})$. When $f(x_R^{(k)}) <= f(x_{n+1}^{(k)})$, (6) is used, otherwise (7).

$$x_C^{(k)} = \overline{x}^{(k)} - \gamma \cdot (\overline{x}^{(k)} - x_R^{(k)}) \tag{6}$$

$$x_C^{(k)} = \overline{x}^{(k)} - \gamma \cdot (\overline{x}^{(k)} - x_{n+1}^{(k)}) \tag{7}$$

if

$$f(x_C^{(k)}) < f(x_{n+1}^{(k)})$$

then

$$x_{n+1}^{(k+1)} = x_C^{(k)}$$

- Shrinking (8), performed if $f(x_C^{(k)}) >= f(x_{n+1}^{(k)})$.

$$\begin{aligned} &x_i^{(k+1)} = x_1^{(k)} + \sigma \cdot (x_i^{(k)} - x_1^{(k)}), \ \forall i \in 2, ..., n+1 \\ &x_1^{(k+1)} = x_1^{(k)} \end{aligned} \tag{8}$$

The iterations are stopped when the stopping condition (9) is reached. It is the stopping criterion used by [4].
At the end of the algorithm, the point $x_1^{(end)}$ is the best approximation of the solution.

$$\sqrt{\frac{1}{n+1} \sum_{i=1}^{n+1} \left(f(\mathbf{x}_i) - \overline{f(\mathbf{x}_i)} \right)^2} \leq \varepsilon \tag{9}$$

From an implementation point of view, some considerations can be made.
First of all, storing the sequence of simplexes is not possible when the dimension of the problem increases. The three-dimensional matrix with the simplex sequence would have dimension $n \cdot (n+1) \cdot k_{max}$. A configuration with $kmax = 10^5$ and $n = 1000$ is not processable.
Instead of extracting only the portion of the sequence we used at the end, an alternative is to add a new layer with the new points at each iteration. In this way, when $kmax$ is not reached, the matrix can be stored.
This strategy, however, has the disadvantage of consistently increasing the execution time, and it is not much useful since the number of iterations is often still significant. For these reasons, we stored the sequences for the Rosenbrock tests, but not in the case of the three problems with a large n.

At each iteration, the points x_i of the simplex have to be sorted according to their function value. A naive solution would be to calculate, each time, every $f(x_i)$ and completely repeat the sorting.
But this is truly necessary only after the shrinking phase (8). In the other cases, only a single point changes, and it can be added in the correct position with a single iteration of the Insertion Sort.
The new position could be found with a binary search, but creating a new matrix using this information is more expensive than shifting the points.
This improvement decreased the execution time by a factor of 5.

The Matlab code is available in the Appendix in the listings 4 and 5.

II. Tests Rosenbrock

To test the two methods, we applied them to the Rosenbrock function, a well-known problem often used to test unconstrained optimization algorithms. The function is defined by

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1^2) \tag{10}$$

and has a global minimum at

$$(x1, x2) = (1, 1) \tag{11}$$

inside a long, narrow, parabolic-shaped flat valley.

We tested both methods using two distinct starting points:

$$(x1, x2) = (1.2, 1.2) \ ; \ (-1.2, 1) \tag{12}$$

For the Truncated Newton Method, we used the parameters:

$$\begin{aligned} btmax =& 50 \\ kmax =& 1000 \\ jmax =& 100 \\ c1 =& 1e-4 \\ rho =& 0.5 \\ fterm =& fterms\_suplin \\ tolgrad =& 1e-8 \\ h =& sqrt(eps) \end{aligned}$$

For the Nelder-Mead:

$$\begin{aligned} kmax =& 1000 \\ tol =& 1e-8 \\ rho =& 1 \\ chi =& 2 \\ gamma =& 0.5 \\ sigma =& 0.5 \end{aligned}$$

The two methods successfully converge to the global minimum for both starting points.
The first point is close to the solution. The second one is located on the opposite side of the function, in a flat valley (see Fig. 1). For this reason for the second starting point the convergence is more difficult. This fact is reflected in a higher number of iterations and time spent finding the minimizer.

Although the three implemented versions of the Truncated Newton method have similar performances, the one using preconditioning is the fastest to converge and requires fewer

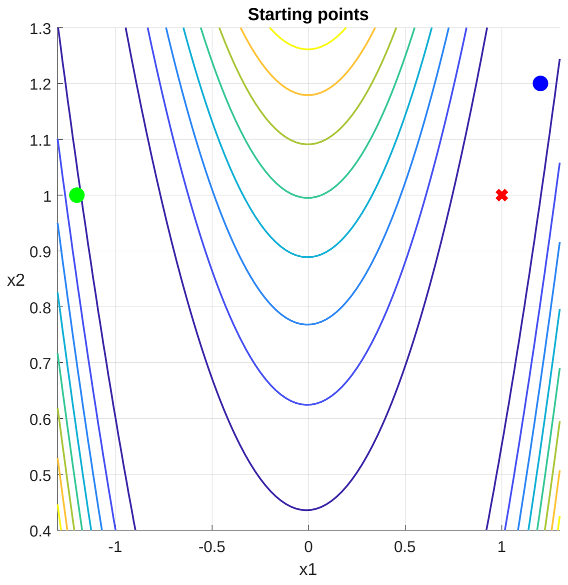

Fig. 1. Starting points. The blue dot is the $x_0 = (1.2, 1.2)$ and the green dot is $x_0 = (-1.2, 1)$. The red X is the minimizer.

inner iterations; the matrix-free version, instead, results slower than the other two and, for one outer iteration, it even reaches the maximum number of inner iterations. For a visual comparison of the three variants see Fig. 2 and 3.

To complete the discussion, we also computed for the three versions the experimental convergence rate, which is included between 1 and 2, as expected from the theory for a superlinear forcing term, and follows the logic of the previous observations.

The Nelder-Mead method, on the other hand, is slower, less accurate and requires more iterations than the Truncated Newton method for both the starting points. Some steps of the algorithm are shown in Fig. 4.

III. Introduction Problems

The three minimization problems we tackled are:

- Problem 76 in [2]
- Chained Wood function (Problem 2 in [2])
- Banded trigonometric problem (Problem 16 in [2])

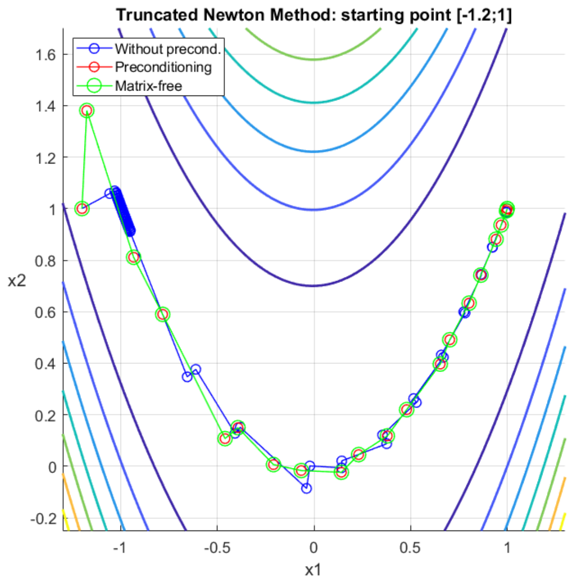

Fig. 2. Application of the three Truncated Newton method alternatives to the Rosenbrock function

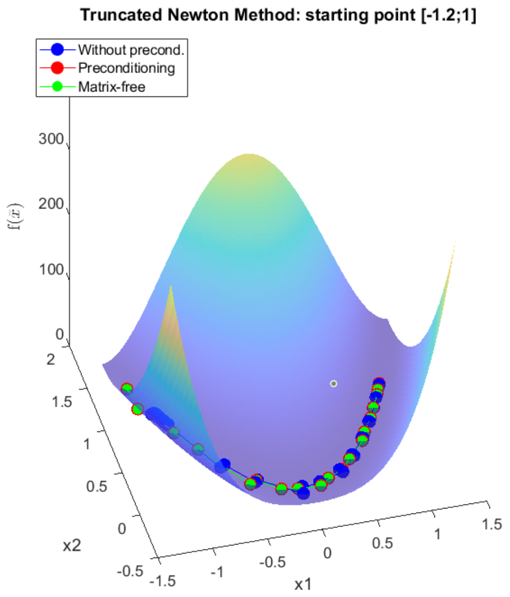

Fig. 3. 3D representation of Fig. 2

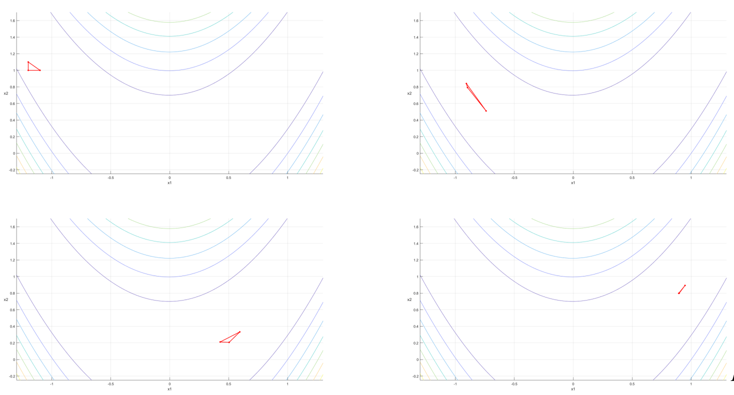

Fig. 4. Some steps of the Nelder-Mead (ordered by row) applied to the Rosenbrock function

Throughout this report, we are going to refer to these functions respectively as problem 1, problem 2, and problem 3.

$F(x)$ is the function to be minimized and the suggested starting point is indicated by $\bar{x}$.

A. Problem 1

The description, the gradient, and the Hessian of problem 1 are respectively (13), (14), and (15).

$$\begin{aligned} F(x) &= \frac{1}{2}\sum_{k=1}^{n} f_k^2(x), \\ f_k(x) &= x_k - \frac{x_{k+1}^2}{10}, \quad 1 \le k < n, \\ f_k(x) &= x_k - \frac{x_1^2}{10}, \quad k = n, \\ \bar{x}_l &= 2, \quad l \ge 1. \end{aligned} \tag{13}$$

There are two global minimizers for the problem (13). They are the $x \in \mathbb{R}^n$ such that every component is zero and the $x \in \mathbb{R}^n$ such that every component is 10. The global minimum is 0.

$$\begin{aligned} \frac{\partial F(x)}{\partial x_1} &= x_1 - \frac{1}{10}x_2^2 + \frac{1}{50}x_1^3 - \frac{1}{5}x_1 x_n, \\ \frac{\partial F(x)}{\partial x_k} &= x_k - \frac{1}{10}x_{k+1}^2 + \frac{1}{50}x_k^3 - \frac{1}{5}x_k x_{k-1}, \\ &\quad 1 < k < n, \\ \frac{\partial F(x)}{\partial x_n} &= x_n - \frac{1}{10}x_1^2 + \frac{1}{50}x_n^3 - \frac{1}{5}x_n x_{n-1}, \end{aligned} \tag{14}$$

$$\begin{aligned} \frac{\partial^2 F(x)}{\partial x_1^2} &= 1 + \frac{3}{50}x_1^2 - \frac{1}{5}x_n \\ \frac{\partial^2 F(x)}{\partial x_k^2} &= 1 + \frac{3}{50}x_k^2 - \frac{1}{5}x_{k-1} \\ &\quad 1 < k \le n, \\ \frac{\partial^2 F(x)}{\partial x_{k+1}\partial x_k} &= \frac{\partial^2 F(x)}{\partial x_k \partial x_{k+1}} = -\frac{1}{5}x_{k+1}, \\ &\quad 1 \le k < n, \\ \frac{\partial^2 F(x)}{\partial x_1 \partial x_n} &= \frac{\partial^2 F(x)}{\partial x_n \partial x_1} = -\frac{1}{5}x_1 \end{aligned} \tag{15}$$

B. Problem 2

The description, the gradient, and the Hessian of problem 2 are respectively (16), (17), and (18).

$$\begin{aligned} F(x) &= \sum_{j=1}^{k} 100(x_{i-1}^2 - x_i)^2 + \\ &+ (x_{i-1} - 1)^2 + 90(x_{i+1}^2 - x_{i+2})^2 + \\ &+ (x_{i+1} - 1)^2 + 10(x_i + x_{i+2} - 2)^2 + \\ &+ (x_i - x_{i+2})^2 \frac{1}{10}, \\ i &= 2j, k = \frac{n-2}{2} \end{aligned} \tag{16}$$

$$\begin{aligned} \frac{\partial F(x)}{\partial x_1} &= 400x_1(x_1^2 - x_2) + 2(x_1 - 1), \\ \frac{\partial F(x)}{\partial x_2} &= -200(x_1^2 - x_2) + 20(x_2 + x_4 - 2) + (x_2 - x_4)\frac{1}{5}, \\ \frac{\partial F(x)}{\partial x_{n-1}} &= 360x_{n-1}(x_{n-1}^2 - x_n) + 2(x_{n-1} - 1), \\ \frac{\partial F(x)}{\partial x_n} &= -180(x_{n-1}^2 - x_n) + \\ &+ 20(x_{n-2} + x_n - 2) - (x_{n-2} - x_n)\frac{1}{5}, \\ \frac{\partial F(x)}{\partial x_{2k+1}} &= 760x_{2k+1}(x_{2k+1}^2 - x_{2k+2}) + 4(x_{2k+1} - 1), \\ &\quad 1 < k < \frac{n-2}{2}, \\ \frac{\partial F(x)}{\partial x_{2k}} &= -380(x_{2k-1}^2 - x_{2k}) + 20(x_{2k-2} + x_{2k} - 2) + \\ &- (x_{2k-2} - x_{2k})\frac{1}{5} + 20(x_{2k} + x_{2k+2} - 2) + \\ &+ (x_{2k} - x_{2k+2})\frac{1}{5}, \quad 1 < k < \frac{n-2}{2}, \end{aligned} \tag{17}$$

$$\begin{aligned} \frac{\partial^2 F(x)}{\partial x_1^2} &= 1200x_1^2 - 400x_2 + 2, \\ \frac{\partial^2 F(x)}{\partial x_1 \partial x_2} &= -400x_1, \\ \frac{\partial^2 F(x)}{\partial x_2^2} &= 220.2, \end{aligned}$$

$$\frac{\partial^2 F(x)}{\partial x_2 \partial x_4} = 19.8,$$

$$\frac{\partial^2 F(x)}{\partial x_{n-1}^2} = 1080x_{n-1} - 360x_n + 2,$$

$$\frac{\partial^2 F(x)}{\partial x_{n-1} \partial x_n} = -360x_{n-1},$$

$$\frac{\partial^2 F(x)}{\partial x_n^2} = 200.2,$$

$$\frac{\partial^2 F(x)}{\partial x_n \partial x_{n-2}} = 19.8,$$

$$\frac{\partial^2 F(x)}{\partial x_{2k+1}^2} = 2280x_{2k+1}^2 - 760x_{2k+2} + 4, \quad 1 < k < \frac{n-2}{2}$$

$$\frac{\partial^2 F(x)}{\partial x_{2k+1} \partial x_{2k+2}} = -760x_{2k+1}, \quad 1 < k < \frac{n-2}{2}$$

$$\frac{\partial^2 F(x)}{\partial x_{2k}^2} = 420,4, \quad 1 < k < \frac{n-2}{2}$$

$$\frac{\partial^2 F(x)}{\partial x_{2k} \partial x_{2k-1}} = -760x_{2k-1}, \quad 1 < k < \frac{n-2}{2}$$

$$\frac{\partial^2 F(x)}{\partial x_{2k} \partial x_{2k-2}} = 19.8, \quad 1 < k < \frac{n-2}{2}$$

$$\frac{\partial^2 F(x)}{\partial x_{2k} \partial x_{2k+2}} = 19.8, \quad 1 < k < \frac{n-2}{2} \tag{18}$$

C. Problem 3

The function, its gradient, and the Hessian are respectively (19), (20), and (21). Note that the Hessian is a diagonal matrix and only the diagonal values are reported.

$$\begin{gathered} F(x) = \sum_{i=1}^{n} i[(1-cosx_i) + sinx_{i-1} - sinx_{i+1}], \\ x_0 = x_{n+1} = 0 \\ \bar{x}_i = 1, \quad i \geq 1. \end{gathered} \tag{19}$$

$$\begin{aligned} \frac{\partial F(x)}{\partial x_i} &= i \cdot sinx_i + 2 \cdot cosx_i, \qquad 1 \leq i < n \\ \frac{\partial F(x)}{\partial x_n} &= n \cdot sinx_n - (n-1) \cdot cosx_n \end{aligned} \tag{20}$$

$$\begin{aligned} \frac{\partial^2 F(x)}{\partial x_i^2} &= i \cdot cosx_i - 2 \cdot sinx_i \qquad 1 \leq i < n \\ \frac{\partial^2 F(x)}{\partial x_n^2} &= n \cdot cosx_n + (n-1) \cdot sinx_n \end{aligned} \tag{21}$$

The function (19) is periodic and has an infinite number of global minimizers. The value of the global minima changes according to the dimension of the problem n.

IV. Methodology

Using small values for the dimension of the problem n, as the tests on the Rosenbrock function showed, both the methods give good results without requiring any tuning. In order to have a more complete picture of the behaviour of the algorithms, the methodology we used to solve our three optimization problems aims to test the methods in situations in which the convergence is more difficult and a tuning of the parameters is required.

A. Truncated Newton method

We followed these steps for the Truncated Newton method, the matrix-free version, and the preconditioned version.

The starting configuration of parameters is given by:

- Backtracking:
  - $\rho = 0.8$
  - Maximum number of backtracking iterations $btmax = \lfloor log(10^{-10})/log(\rho) \rfloor$
- Maximum number of outer iterations $kmax = 100$
- Tolerance on the gradient for the stopping condition $tolgrad = 10^{-8}$
- Maximum number of inner iterations $jmax = 100$
- Forcing term (22)
- Low value of n, e.g. 10.

$$\eta_k = min(0.5, ||\nabla F(x_k)||_2) \tag{22}$$

We used this particular formula for $btmax$ in order to get $\rho^{btmax} \geq 10^{-10}$. This limit was selected to ensure progress in the value of x_k at each iteration.

a) First phase: tuning.
We used the starting points given by the problems and we:

- Increased n as long as the algorithm was still converging.
- At the first value of n for which the method was no longer converging, we increased $kmax$ until we obtained again convergence, or the execution time was too high.
  In the first case, we moved back to the previous step with the $kmax$ we obtained. Otherwise, we kept the last values of n and $kmax$ for which we had convergence and moved forward.
- Increased the value of $jmax$. In case of significant improvements, we moved again to the first point.
- Fixed the best $kmax$ and $jmax$ for a difficult n.
- Chose the best forcing term among (22), (23) and (24).
- Chose the best configuration of ρ and $c1$.

$$\eta_k = 0.5 \tag{23}$$

$$\eta_k = min(0.5, \sqrt{||\nabla F(x_k)||_2}) \tag{24}$$

b) Second phase: 5 starting points.
We then applied the three versions of the Truncated Newton method to other four starting points. We used both the initial and the tuned configurations.
The starting points were chosen case by case according to the problem. If the algorithm had not converged, then we considered the possibility of decreasing the value of $tolgrad$.

c) Third phase: comparison with Nelder-Mead.
We applied the methods also for the value of n obtained in IV-B and compared the results with the Nelder-Mead solutions.

B. Nelder-Mead method

Applying the Nelder-Mead method to solve the optimization problems, we followed the methodology reported here.
The first step was the construction of the initial simplex, defined by $n + 1$ points. To accomplish this, we took as reference the starting point indicated by the problem and defined the other n points modifying, for each of them, one dimension, following this procedure:

$$x_i = x_1, \quad i = 2, ..., n+1$$
$$x_i(i-1) = x_i(i-1) \cdot \frac{11}{10} + \Delta \quad i = 2, ..., n+1 \qquad (25)$$

The starting value of Δ is 0. During the tuning phase, we tried to modify the dimension of our simplex to achieve a better solution, increasing Δ. In the particular case in which the starting point was the origin, we only summed Δ to each dimension for enlarging the simplex.

In this way, the initial simplex is n-dimensional and is able to move freely in the whole solution space. We then stored these points in a $n \times (n+1)$ matrix where $x(:, 1)$ is the starting point of the problem and $x(:, i)$ for $i = 2, ..., n+1$ are the other n points we obtained. We report here the initial configuration of parameters we tested:

- $\rho = 1$ (reflection phase)
- $\chi = 2$ (expansion phase)
- $\gamma = 0.5$ (contraction phase)
- $\sigma = 0.5$ (shrinking phase)
- $kmax = 100$ (max number of iterations)
- $tol = 10^{-8}$ (tolerance for the stopping condition)

a) First phase: tuning.
We used the starting points given by the problems and we:

- Increased n as long as the algorithm was still converging.
- At the first n for which the algorithm was no longer converging, we increased the value of $kmax$ until we reached again convergence, or the execution time was too high. In the first case, we moved back to the first step with the $kmax$ we obtained. Otherwise, we kept the last values of n and $kmax$ for which we had convergence and moved forward. We also tested different dimensions for the starting simplex, following the procedure (25), but adding more than just the 10%. Then, we used the best Δ.
- Then, starting from ρ we tuned one parameter at a time.

b) Second phase: 5 starting points.
We then applied the Nelder and Mead method to other four starting points, chosen case by case according to the problem, using both the initial and the tuned configurations.

V. Numerical results

A. Problem 1

a) Choice of starting points: The points we chose are:

- the point given by the problem
- a point relatively far away from the two global minimizers
- a point between the two global minimizers
- a point close to the global minimizer $x = zeros(n, 1)$ (generated with the random seed equal to 123456)
- a difficult point for the Truncated Method, identified after many attempts with different points

They are summarised in Table I.

The Truncated Newton method and its variants performed well on the given starting point for this problem. They have always converged and in less than ten iterations for every n. The maximum number of internal j iterations and the number of external iterations k were nearly identical for every n among the variants, but the execution time was very different. The matrix-free version of the Truncated Newton method is orders of magnitude quicker than the other two alternatives for high n. For this reason, we used a logarithmic scale for both axes in Fig. 5 to compare the execution time of the three versions.

The reason is that the other two methods have to compute, store, and manage at each iteration a big Hessian matrix. This is computationally expensive, hence the methods result to be slower.

With the matrix-free implementation, even with $n = 1e7$ the problem was solved in 27 seconds. On the other hand, the other two alternatives can process problems up to $n = 40000$. Over that threshold, the Hessian cannot be stored, and, as a result, the error "MATLAB:array:SizeLimitExceeded" is shown.

The disadvantage of the matrix-free is that at each iteration we have only an approximation of the Hessian-vector product. This limitation is evident in the results presented in the following paragraphs.

b) Tuning Truncated Newton method: The methodologies presented in IV-A and IV-B were used to obtain the best configurations of parameters. The only change to the steps reported in the preceding sections is the starting point on which the tuning was done. As stated before, all the Truncated Newton method alternatives obtained great results on the given starting point and no tuning was needed. For this reason, we used the starting point $x_0^{(5)}$ presented in Table I. After the tuning on this point, we checked on $x_0^{(1)}$

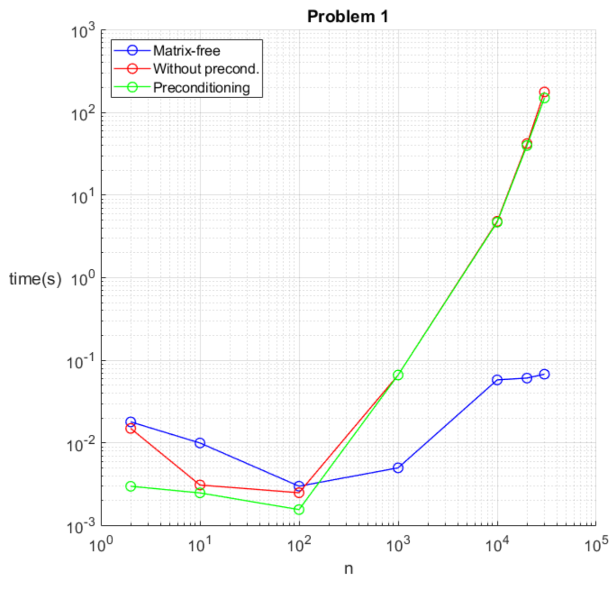

Fig. 5. Execution time of Truncated Newton method versions

TABLE I
STARTING POINTS PROBLEM 1

Name	**Point**
$x_0^{(1)}$	$2 * ones(n,1)$
$x_0^{(2)}$	$20 * ones(n,1)$
$x_0^{(3)}$	$4.75 * ones(n,1)$
$x_0^{(4)}$	$-0.1 + 0.2 * rand(n,1)$
$x_0^{(5)}$	$\sqrt{n:-1:1}'$

TABLE II
PARAMETERS CONFIGURATION
NELDER-MEAD PROBLEM 1

	n	**kmax**	Δ	ρ	χ	γ	σ
Values	200	1e5	10	1	2	0.5	0.5

that the results were still as good as the initial configuration.

Using as starting point $x_0^{(5)}$, the method was already not converging for $n = 55$, before the tuning. Moreover, the Armijo condition in the backtracking was not always satisfied.

The optimal parameters, obtained after the tuning, were the initial ones, except for $kmax$ which we increased up to $1e4$. Furthermore, the performance with the forcing terms (24) and (22) were similar. With (23) the algorithm did not converge.

c) Tuning Nelder-Mead: The steps presented in IV-B gave the configuration of parameters present in Table II. Only $kmax$ and Δ were changed to obtain better results.

d) Convergence rate: The experimental rate of convergence of the Truncated Newton method can be observed in Table III. The function in the listing 6, reported in the Appendix, was used to compute the values. Only converging test results are reported.
The experimental convergence rate does not match the theoretical results for different reasons. The first is that in many cases the algorithm converges in a few iterations ($k \approx 10$). There are not enough iterations to obtain a stable convergence rate and there are not enough points for the regression. The main reason is that the assumptions of the theorem on the convergence of the Truncated Newton method [3] are not guaranteed to be satisfied by this problem. $\nabla^2 F(x_0^{(i)})$ is not even positive definite for all the starting points. Hence the theorem cannot be applied.

Regarding the computation of the experimental convergence rate of the Nelder-Mead, we used the function in the listing 6 passing as arguments respectively the barycenter of our simplex at each step of the algorithm and the final barycenter.

TABLE III
EXPERIMENTAL RATE OF CONVERGENCE
TRUNCATED NEWTON METHOD
PROBLEM 1

Superlinear forcing term			
n	Starting point	k	Conv. rate
55	$x_0^{(1)}$	6	2.3
	$x_0^{(2)}$	7	1.2
	$x_0^{(3)}$	12	1.2
	$x_0^{(4)}$	3	2.6
	$x_0^{(5)}$	2.2e3	1.0
10000	$x_0^{(1)}$	6	1.4
	$x_0^{(2)}$	7	0.8
	$x_0^{(3)}$	12	0.9
	$x_0^{(4)}$	3	4.3
	$x_0^{(5)}$	$*$	$*$

Quadratic forcing term			
n	Starting point	k	Conv. rate
55	$x_0^{(1)}$	6	2.3
	$x_0^{(2)}$	7	1.2
	$x_0^{(3)}$	12	1.2
	$x_0^{(4)}$	3	2.6
	$x_0^{(5)}$	1.6e3	1.0
10000	$x_0^{(1)}$	6	1.4
	$x_0^{(2)}$	7	0.8
	$x_0^{(3)}$	12	0.9
	$x_0^{(4)}$	3	4.3
	$x_0^{(5)}$	$*$	$*$

$*$ The method converges, but not to a minimum

e) Application of the methods: Table IV shows the results of the two methods (with their variants) applied to the problem (13). In the table, the acronym "TPI" means "Time per iteration". The forcing term used was (24).
The comparison between the two methods is thoroughly discussed in section VI. The considerations apply also to this problem. One exception is that, in this particular case, the preconditioning is not beneficial.

TABLE IV
RESULTS N=200
PROBLEM 1

Nelder-Mead					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	9.7e4	$* * *$	1.0	14.1	1.4e-4
$x_0^{(2)}$	*		*	*	*
$x_0^{(3)}$	*		*	*	*
$x_0^{(4)}$	2.0e4		1.0	3.3	1.6e-4
$x_0^{(5)}$	*		*	*	*

Truncated Newton method					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	6	1.0	2.2	2.9e-3	4.9e-4
$x_0^{(2)}$	7	1.0	1.0	3.1e-3	4.5e-4
$x_0^{(3)}$	12	0.4	1.1	3.5e-3	2.9e-4
$x_0^{(4)}$	3	1.0	2.8	1.2e-3	4.0e-4
$x_0^{(5)}$	**	**	**	9.7e-3	**

Preconditioned Truncated Newton method					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	6	1.0	2.2	1.1e-2	1.8e-3
$x_0^{(2)}$	9	1.3	1.5	4.4e-3	5.0e-4
$x_0^{(3)}$	12	0.4	1.1	4.5e-3	4.0e-4
$x_0^{(4)}$	3	1.0	2.8	1.6e-3	5.3e-4
$x_0^{(5)}$	**	**	**	3.9e-2	**

Matrix-free Truncated Newton method					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	6	1.0	2.2	3.3e-3	5.4e-4
$x_0^{(2)}$	8	0.9	1.5	3.8e-3	4.8e-4
$x_0^{(3)}$	12	0.4	1.1	3.0e-3	2.40e-4
$x_0^{(4)}$	3	1.0	2.8	1.0e-3	3.4e-4
$x_0^{(5)}$	*	*	*	*	*

$* * *$ There are no internal iterations in this method
$**$ The method converges, but not to a minimum
$*$ The method does not converge

B. Problem 2

a) Choice of starting points: Since for the suggested point the Nelder and Mead algorithm was not performing well, for the other starting points we decided to get closer to the solution and test how far we could go from it while still reaching convergence.
Once we reached a distance for which we could not converge anymore, a starting point was chosen randomly in the range between the solution and the maximum distance.
The starting points we chose are:

- three points, $x_0^{(2)}$,$x_0^{(3)}$ and $x_0^{(4)}$ with increasing distance from the solution
- one random point, $x_0^{(5)}$, chosen in the way previously described

They are summarised in Table VI.

The two versions of the Truncated Newton Method performed differently for an increasing value of n. The version with preconditioning did fewer internal and external iterations and performed better for higher values of n. The Matrix-free approach was not considered for this problem. We conducted the tests for the choice of n on the suggested point. The number of iterations was strongly related to the choice of the starting point.

b) Tuning Truncated Newton method: The Truncated Newton method without preconditioning was no longer converging for $n = 1000$, unlike the version with preconditioning, so we decided to stop at this value for n. We tuned the parameters on the version with preconditioning. The two methods performed well overall, except for the case in which the starting point was the suggested one. For this reason, we chose this point for the tuning.

The tuning did not improve sensitively the solution, but brought an important reduction in the execution time, from minutes before the tuning, to seconds. In the end, the best choice for the parameters was:

- $rho = 0.8$
- $c1 = 1e^{-3}$
- $kmax = 1e^4$

c) Tuning Nelder-Mead: Following the methodology in section IV-B, we obtained the configuration present in Table V.

TABLE V
PARAMETERS CONFIGURATION
NELDER-MEAD PROBLEM 2

	n	**kmax**	Δ	ρ	χ	γ	σ
Values	40	1e5	10	1	1.25	0.8	0.5

The tuning for the Nelder and Mead resulted to be useful, improving the number of iterations and/or the value of the solution.

TABLE VI
STARTING POINTS PROBLEM 2

Name	**Point**
$x_0^{(1)}$	x_s
$x_0^{(2)}$	$ones(n,1) + 0.1$
$x_0^{(3)}$	$ones(n,1) + 10$
$x_0^{(4)}$	$ones(n,1) + 100$
$x_0^{(5)}$	$ones(n,1) + rand(n,1) * 100$

$$\begin{aligned} x_s(i) = 0 \quad for \quad i = 2, 4, \\ x_s(i) = 1 \quad for \quad i = 1, 3, \\ x_s(2k) = 0, \\ x_s(2k+1) = -2, \\ 3 < k < \frac{n}{2} \end{aligned} \tag{26}$$

d) Convergence rate: The considerations made for the experimental convergence rate for Problem 1, can also be applied for Problem 2. Table VIII shows the experimental rate of convergence of the Truncated Newton method without preconditioning. For all the starting points we used to test the Nelder and Mead method, except for the suggested one, the Truncated Newton Method was converging considerably fast.

For this reason we tested this method also using three more challenging points, always with $n = 1000$. This helped us prove that, for n high enough, the choice of the starting point is crucial for the algorithm to converge in a reasonable time.

TABLE VII
ADDITIONAL STARTING POINTS PROBLEM 2

Name	**Point**
$x_0^{(6)}$	$x_0^{(1)} - 10$
$x_0^{(7)}$	$x_0^{(1)} - 50$
$x_0^{(8)}$	$x_0^{(1)} - 70$

e) Consideration on the new points - Truncated Newton method: For these starting points the algorithm did not perform as well, in particular for $x_0^{(8)}$ the execution time was $t \approx 21s$, using the version without preconditioning. The preconditioned version in this case resulted to be significantly slower, and for $x_0^{(8)}$ it did not converge in a reasonable time.

TABLE VIII
EXPERIMENTAL RATE OF CONVERGENCE
TRUNCATED NEWTON METHOD
PROBLEM 2

Superlinear forcing term			
n	Starting point	k	Conv. rate
10000	$x_0^{(3)}$	33	0.8
	$x_0^{(4)}$	45	0.8
	$x_0^{(6)}$	946	0.4
	$x_0^{(7)}$	974	0.4
	$x_0^{(8)}$	$*$	$*$
Quadratic forcing term			
n	Starting point	k	Conv. rate
10000	$x_0^{(3)}$	32	0.8
	$x_0^{(4)}$	43	0.8
	$x_0^{(6)}$	943	0.4
	$x_0^{(7)}$	972	0.4
	$x_0^{(8)}$	1.0e3	0.5

$*$ The method does not converge

C. Problem 3

Following the two methodologies presented in section IV, we found the best configurations of parameters for the two algorithms and we tested them on the problem for two different dimensions $n = 10^2$ and 10^4, and five distinct starting points.

a) Choice of starting points: Since the function is periodic with period 2π and characterized by an infinite number of global minima, all the starting points chosen for this problem were taken inside the period $[-\pi, \pi]$.

In particular, analyzing the function, its gradient and the Hessian we deduced that, inside this period, the only minimizer, for n high enough, gets closer and closer to the origin. The maximizers, instead, are close to the extremes of this interval. For this reason, in addition to the starting point already given by the problem $x_0^{(2)}$, we chose:

TABLE IX
RESULTS N=40
PROBLEM 2

Nelder-Mead					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	$*$	$* * *$	$*$	$*$	$*$
$x_0^{(2)}$	2.2e4		1.0	1.1	5.2e-5
$x_0^{(3)}$	7.7e4		1.0	3.7	4.7e-5
$x_0^{(4)}$	$*$		$*$	$*$	$*$
$x_0^{(5)}$	$*$		$*$	$*$	$*$
Truncated Newton method					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	83	6.5	0.7	2.6e-2	1.8e-4
$x_0^{(2)}$	11	11.5	1.1	2.8e-2	1.9e-3
$x_0^{(3)}$	30	6.87	0.9	5.7e-2	1.9e-3
$x_0^{(4)}$	42	5.9	0.8	7e-2	1.6e-e
$x_0^{(5)}$	75	7.8	0.7	3e-2	3.4e-4
Preconditioned Truncated Newton method					
Starting point	k	j mean	Conv. rate	Time[s]	TPI[s]
$x_0^{(1)}$	336	0.1	1.0	0.4	1.2e-3
$x_0^{(2)}$	5	1	2.0	0.4	8.3e-2
$x_0^{(3)}$	14	1	1.1	0.4	3.0e-2
$x_0^{(4)}$	21	1	0.9	0.4	2.0e-2
$x_0^{(5)}$	28	1	1.0	0.2	7.8e-3

$*$ The method does not converge

- a point close to the solution, $x_0^{(1)}$, for which the convergence should not be difficult;
- a point far from the solution (and close to a maximizer), $x_0^{(3)}$ for which we expected problems of convergence;
- other two points, $x_0^{(4)}$ and $x_0^{(5)}$, are situated in intermediate positions to have a more complete picture.

The points are summarised in Table X. We generated $x_0^{(5)}$ with random seed equal to 123456.

b) Truncated Newton method: The gap between the performances of the two versions of the Truncated Newton method (the Matrix-free approach was not considered for this problem) becomes more and more important as n increases. For $n = 100$, as shown in Table XIV, the preconditioning does not significantly improve the standard method's performance. It performs fewer outer and inner iterations but in a similar execution time. For $n = 10000$, instead, while the preconditioned version converges for four starting points out of five, the other variant never does ($kmax = 100$ is not enough to converge and for higher $kmax$ it requires an unreasonable execution time).
The explanation of this result can be derived from the Hessian of the function: analyzing it we noticed an approximately linear increase of its condition number with the dimension of the problem n. This confirms the different behavior of the two variants we discussed before.
On the other hand, we noticed that, although the version with preconditioning outperforms the standard algorithm, its results get more easily stuck in points where the gradient is below the tolerance but which are not minima, as shown in table

TABLE X
STARTING POINTS PROBLEM 3

Name	Point
$x_0^{(1)}$	$zeros(n,1)$
$x_0^{(2)}$	$ones(n,1)$
$x_0^{(3)}$	$3 * ones(n,1)$
$x_0^{(4)}$	$linspace(-0.5, 0.5, n)'$
$x_0^{(5)}$	$rand(n,1)$

(see Table XI and XIV). In order to verify the nature of the stationary points we obtained, we computed the Hessian for the solution given by the method and checked its eigenvalues.

At the end of the tuning, we kept the original values:

- ρ=0.8 which gives the best performance with a good margin
- $c1$=1e-4 (other values give similar results)
- $jmax$ = 100 (higher values are either not reached or don't improve the method)
- $tolgrad$ = 1e-8 to have a fairly accurate result without requiring too heavy computations

We modified the value of k_{max}, from 100 up to 1000, obtaining more cases of convergence. It was not possible to increase it even more, due to unreasonable execution times ($> 30min$).

We then tested two different forcing terms, superlinear and quadratic, obtaining slight changes in the balance between inner and outer iterations, but, overall, very similar results. Regarding the experimental convergence rate, we made the same considerations already discussed for problem 1 (see the subsection V-A).

TABLE XI
RESULTS PRECONDITIONED NEWTON METHOD WITH N = 10000
PROBLEM 3

Preconditioned Truncated Newton Method			
Starting point	k	j mean	Time[s]
$x_0^{(1)}$	5	1.0	2.3
$x_0^{(2)}$	$**$	$**$	86.0
$x_0^{(3)}$	$*$	$*$	$*$
$x_0^{(4)}$	3	1.0	1.5
$x_0^{(5)}$	$**$	$**$	2.3

$**$ The algorithm converges, but not to a minimum
$*$ The algorithm does not converge

c) Nelder-Mead: Following the steps presented in IV-B we found the best configuration of parameters, which is reported in Table XII. Due to the nature of the problem, we made one change to the "standard" methodology: since the function is periodic and we decided to consider the period $[-\pi, \pi]$, we only summed a small positive Δ to each dimension to create the initial simplex. Having to deal with small numbers inside a period, this choice resulted more straightforward. In particular, starting from 0.1, we took the first Δ that improved considerably the algorithm's performance and noticed that higher values gave similar results.

TABLE XII
PARAMETERS CONFIGURATION
NELDER-MEAD PROBLEM 3

	n	kmax	Δ	ρ	χ	γ	σ
Values	100	1e5	1	1	1.5	0.9	0.5

As shown in Table XIII, the tuning resulted in significant improvement. The version with the original values for the parameters converges for only two starting points with high numbers of iterations. The tuned one, instead, finds the solution for four initial points out of five with considerably fewer iterations and shorter execution times.

Table XIV shows the results of the two methods applied to the problem (19) for n = 100 (for higher values of the dimension the Nelder-Mead does not converge and the comparison is meaningless). We used (22) as forcing term for the Truncated Newton method.

TABLE XIII
BEFORE/AFTER TUNING PROBLEM 3
NELDER-MEAD

	Without tuning			**With tuning**		
$\boldsymbol{x_0}$	$\boldsymbol{F(x_{end})}$	**k**	**Time[s]**	$\boldsymbol{F(x_{end})}$	**k**	**Time [s]**
$x_0^{(1)}$	-5.0e1	4.9e4	4.4	-5.0e1	2.1e4	1.8
$x_0^{(2)}$	$*$	$*$	$*$	-5.0e1	2.9e4	2.0
$x_0^{(3)}$	$*$	$*$	$*$	$*$	$*$	$*$
$x_0^{(4)}$	-5.0e1	4.8e4	4.1	-5.0e1	2.3e4	1.8
$x_0^{(5)}$	$*$	$*$	$*$	-5.0e1	2.5e4	1.9

$*$ the algorithm does not converge

VI. DISCUSSION

We report here our final observations regarding the two methods. Note that, for the sake of brevity, we will refer to the Nelder Mead method as NM and to the Truncated Newton as TN.

a) Comparison: The first thing we noticed, comparing the results we obtained for the three problems, is that with the NM method we cannot reach convergence for values of n comparable to those used for the TN. Also, the highest value of n for which the method still converges in a reasonable execution time is strictly related to the function we are considering.

The NM algorithm is derivative-free, it does not require any information about the gradient or the Hessian of the

TABLE XIV
RESULTS N=100
PROBLEM 3

Nelder-Mead					
x_0	k	j mean	Conv. rate	Time[s]	Time per iteration[s]
$x_0^{(1)}$	2.1e4	$***$	1.0	1.8	8.6e-5
$x_0^{(2)}$	2.9e4		1.0	2.0	6.9e-5
$x_0^{(3)}$	$*$		$*$	$*$	$*$
$x_0^{(4)}$	2.3e4		1.0	1.8	7.8e-5
$x_0^{(5)}$	2.5e4		1.0	1.9	7.6e-5
Truncated Newton method					
x_0	k	j mean	Conv. rate	Time[s]	Time per iteration[s]
$x_0^{(1)}$	13	16.5	1.2	3e-3	2.3e-4
$x_0^{(2)}$	16	10.3	1.3	8e-3	6.3e-4
$x_0^{(3)}$	27	8.4	1.3	7e-3	2.6e-4
$x_0^{(4)}$	11	10.6	1.7	1e-3	9.1e-5
$x_0^{(5)}$	14	8.0	1.4	2e-3	1.4e-4
Preconditioned Truncated Newton method					
x_0	k	j mean	Conv. rate	Time[s]	Time per iteration[s]
$x_0^{(1)}$	5	1.0	3.3	2e-3	4.0e-4
$x_0^{(2)}$	$**$	$**$	$**$	7e-3	$**$
$x_0^{(3)}$	$**$	$**$	$**$	6e-3	$**$
$x_0^{(4)}$	3	1.0	4.6	5e-3	1.7e-3
$x_0^{(5)}$	$**$	$**$	$**$	5e-3	$**$

$***$ There are no internal iterations in this method
$**$ The method converges, but not to a minimum
$*$ The method does not converge

function to be applied. Nevertheless, this good property is also the reason why we obtained not as accurate solutions as the ones we got using the other optimization algorithms. Indeed, the three versions of the TN method, having at their disposal information about the gradient and the Hessian, provide better solutions in terms of accuracy.

Another important aspect to take into account is that the NM algorithm is almost always slower in reaching convergence than the TN Method once the value of n is fixed. This is well described by the values of the experimental convergence rate we computed.

In the table IV, we can appreciate how the execution time of the two algorithms, starting from the same point, is noticeably different. After all, we are comparing a direct method and a Newton method. This gap in the performances of the two algorithms is also reflected in the number of outer iterations performed by them, in the order of $10^4, 10^5$ for the NM, three orders of magnitude less for the TN.

In order to have a more complete picture of the differences, since the TN presents two levels of iterations (outer on k, inner on j), we also computed the time spent by the two methods for each iteration. On average the iterations of NM always require less time, as we expected from the theory.

Regarding the tuning, for both algorithms it resulted to be useful to obtain better results in terms of execution time and number of steps, even when it did not improve the accuracy of our solution.

One last difference between the two methods is the sensitivity to the choice of the starting point. The NM is not significantly affected by how we choose the initial condition, especially for small values of n. This good property, however, became weaker when we tried values of n close to non-convergence, as shown in the comparison tables of the three problems. Opposite considerations can be made for the TN: for this method the choice of the starting point is crucial for the convergence of the algorithm.

b) Last observations about Truncated Newton Method:

- The choice of the forcing terms does not have a big impact on the solution, just barely noticeable changes in the number of inner and outer iterations.
- The number of backtracking steps at each iteration and the number of inner iterations of the algorithm are related to the problem we are solving and the starting point we are using.
- The use of preconditioning most of the time helps to find the solution faster, performing much fewer inner iterations, especially when the Hessian is ill-conditioned for large values of n. In some cases, however, it converges not to minima, but to saddle points.

c) Conclusion: In conclusion, given the results we obtained, we can state that, if we have to solve an unconstrained optimization problem:

- of dimension n low
- with gradient and Hessian difficult to compute
- for which we are not interested in having an extremely accurate solution
- within a reasonable execution time

the best choice is the Nelder Mead method.

If instead our problem:

- is of dimension n high
- has a reasonable complexity of the gradient and Hessian (which is not always positive definite)
- requires a very accurate solution and fast convergence
- has a solution the position of which we can approximate a priori

then the best choice is the Truncated Newton method.

The following are some aspects that might be worth considering to further improve the obtained results for the Truncated Newton method:

- use a sparse representation for the Hessian, which is diagonal or quasi-diagonal for the three problems we solved, saving a considerable amount of storage
- try different preconditioners which may provide better results

REFERENCES

[1] J. Nocedal and S. J. Wright, Numerical Optimization, 2nd ed., Springer, 2006.

[2] Luksan, Ladislav, and Vlček, Test Problems for Unconstrained Optimization, 2003.
[3] S. G. Nash, A survey of truncated-Newton methods, 2000.
[4] J. A. Nelder, R. Mead, A Simplex Method for Function Minimization, The Computer Journal, Volume 7, Issue 4, January 1965, Pages 308–313

APPENDIX

TRUNCATED NEWTON METHOD

Listing 1. Truncated Newton method

```
function [xk, fk, gradfk_norm, k, xseq,
   btseq, internal_jseq, flag_converged,
   ...
    flag_armijo_always_satisfied,
        flag_enough_internal_steps] =
        truncated_newt(x0, f, gradf, Hessf
        , fterm, kmax, internal_jmax, ...
    tolgrad, c1, rho, btmax)

    % initialize the variables
    flag_converged = false;
    flag_armijo_always_satisfied = true;
    flag_enough_internal_steps = true;

    % compute the value for the Armijo
        condition
    farmijo = @(fk, alpha, gradfk, pk)
        ...
         fk + c1 * alpha * gradfk' * pk;

    xseq = zeros(length(x0), kmax);
    btseq = zeros(1, kmax);
    internal_jseq = zeros(1, kmax);
    k=0;
    xk = x0;

    fk = f(xk);
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk, 2);

    while k<kmax && gradfk_norm >=
        tolgrad
        % initialization for the Conj.
            grad. method
        % The direction pk, computed by
            the CG, is the new direction
            for
        % the update of xk
        zj = 0;
        rj = -gradfk;
        % k is not useful for the forcing
             terms we have defined, but in
        % general it can be used
        tol = fterm(k, -rj) * gradfk_norm
            ;
        dj = rj;
        j = 0;
        Bk = Hessf(xk);

        % Conj. grad. method
        while j<internal_jmax
            w = Bk * dj;
            if dj'* w <= 0
                if j ==0
                    % at the first step
                        rj is -gradfk
                    pk = rj;
                    break;
                else
                    pk = zj;
                    break;
                end
            end
            alpha = rj'*dj/(dj'*w);
            zj = zj + alpha * dj;
            rnew = rj - alpha * w;
            j = j+1;
            if norm(rnew, 2)<tol
                pk = zj;
                break;
            end
            beta = rnew'*rnew/(rj'*rj);
            dj = rnew + beta *dj;
            rj = rnew;
        end
        if j>=internal_jmax && norm(rnew,
            2)>=tol
            pk = zj;
            flag_enough_internal_steps =
                false;
        end

        alpha = 1;

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

        % backtracking checking the
            Armojo condition
        bt = 0;
        while bt < btmax && fnew >=
            farmijo(fk, alpha, gradfk, pk)
            alpha = rho * alpha;
            xnew = xk + alpha * pk;
            fnew = f(xnew);

            bt = bt + 1;
        end

        if fnew >= farmijo(fk, alpha,
            gradfk, pk)
            flag_armijo_always_satisfied=
                false;
        end

        % update variables for the next
            iteration
        xk = xk + alpha * pk;
        fk = f(xk);
        gradfk = gradf(xk);
        gradfk_norm = norm(gradfk, 2);
        k=k+1;

        % save variables for the output
        xseq(:, k) = xk;
        btseq(1, k) = bt;
        internal_jseq(1, k) = j;
    end

    if gradfk_norm < tolgrad
        flag_converged=true;
    end

    % trim the variables keeping only
        used cells
    xseq = xseq(:, 1:k);
    btseq = btseq(:, 1:k);
    internal_jseq = internal_jseq(:, 1:k)
        ;
end
```

Matrix-free Truncated Newton method

Listing 2. Matrix-free Truncated Newton method

```
function [xk, fk, gradfk_norm, k, xseq,
    btseq, internal_jseq, flag_converged,
    ...
    flag_armijo_always_satisfied,
        flag_enough_internal_steps] =
        truncated_newt_matrix_free(x0, f,
        gradf, h, fterm, kmax,
        internal_jmax, ...
    tolgrad, c1, rho, btmax)

    % initialize the variables
    flag_converged = false;
    flag_armijo_always_satisfied = true;
    flag_enough_internal_steps = true;

    % compute the value for the Armijo
        condition
    farmijo = @(fk, alpha, gradfk, pk)
        ...
        fk + c1 * alpha * gradfk' * pk;

    xseq = zeros(length(x0), kmax);
    btseq = zeros(1, kmax);
    internal_jseq = zeros(1, kmax);
    k=0;
    xk = x0;

    fk = f(xk);
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk, 2);

    while k<kmax && gradfk_norm >=
        tolgrad
        % initialization for the Conj.
            grad. method
        % The direction pk, computed by
            the CG, is the new direction
            for
        % the update of xk
        zj = 0;
        rj = -gradfk;
        % k is not useful for the forcing
             terms we have defined, but in
        % general it can be used
        tol = fterm(k, -rj) * gradfk_norm
            ;
        dj = rj;
        j = 0;
        % perform the product of the
            Hessian with a vector without
            computing
        % explicitly the Hessian
        Hessf_vet_prod = @(xk, dir) (
            gradf(xk+h*dir)-gradf(xk))/h;

        while j<internal_jmax
            w = Hessf_vet_prod(xk, dj);
            if dj'* w <=0
                if j ==0
                    % at the first step
                        rj is -gradfk
                    pk = rj;
                    break;
                else
                    pk = zj;
                    break;
                end
            end
            alpha = rj'*dj/(dj'*w);
            zj = zj + alpha * dj;
            rnew = rj - alpha * w;
            j = j+1;
            if norm(rnew, 2)<tol
                pk = zj;
                break;
            end
            beta = rnew'*rnew/(rj'*rj);
            dj = rnew + beta *dj;
            rj = rnew;
        end
        if j>=internal_jmax && norm(rnew,
          2)>=tol
          pk = zj;
          flag_enough_internal_steps =
              false;
      end

      alpha = 1;

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

      % backtracking checking the
          Armojo condition
      bt = 0;
      while bt < btmax && fnew >=
          farmijo(fk, alpha, gradfk, pk)
          alpha = rho * alpha;
          xnew = xk + alpha * pk;
          fnew = f(xnew);

          bt = bt + 1;
      end

      if fnew >= farmijo(fk, alpha,
          gradfk, pk)
          flag_armijo_always_satisfied=
              false;
      end

      xk = xk + alpha * pk;

      % update variables for the next
          iteration
      fk = f(xk);
      gradfk = gradf(xk);
      gradfk_norm = norm(gradfk, 2);
      k=k+1;

      % save variables for the output
      xseq(:, k) = xk;
      btseq(1, k) = bt;
      internal_jseq(1, k) = j;
  end

  if gradfk_norm < tolgrad
      flag_converged=true;
  end

  % trim the variables keeping only
      used cells
  xseq = xseq(:, 1:k);
  btseq = btseq(:, 1:k);
  internal_jseq = internal_jseq(:, 1:k)
      ;

end
```

Listing 3. Preconditioned Truncated Newton

```
function [xk, fk, gradfk_norm, k, xseq,
    btseq, internal_jseq, flag_converged,
    ...
    flag_armijo_always_satisfied,
        flag_enough_internal_steps] =
        truncated_newt_prec(x0, f, gradf,
        Hessf, fterm, kmax, internal_jmax,
         ...
    tolgrad, c1, rho, btmax)

    % initialize the variables
    flag_converged = false;
    flag_armijo_always_satisfied = true;
    flag_enough_internal_steps = true;

    % computes the value for the Armijo
        condition
    farmijo = @(fk, alpha, gradfk, pk)
        ...
         fk + c1 * alpha * gradfk' * pk;

    xseq = zeros(length(x0), kmax);
    btseq = zeros(1, kmax);
    internal_jseq = zeros(1, kmax);
    k=0;
    xk = x0;

    fk = f(xk);
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk, 2);

    while k<kmax && gradfk_norm >=
        tolgrad
        % initialization for the Conj.
            grad. method
        % The direction pk, computed by
            the CG, is the new direction
            for
        % the update of xk
        zj = 0;
        rj = -gradfk;
        Bk = Hessf(xk);

        % computation of the matrix for
            the preconditioning
        [L, U, P] = ilu(sparse(Bk));
        y0 = U\(L\(P*rj));
        yj = y0;

        % k is not useful for the forcing
             terms we have defined, but in
        % general it can be used
```

```
tol = fterm(k, -rj) * gradfk_norm
    ;
dj = yj;
j = 0;
while j<internal_jmax
    w = Bk * dj;
    if dj'* w<=0
        if j ==0
            % at the first step
                rj is -gradfk
            pk = rj;
            break;
        else
            pk = zj;
            break;
        end
    end
    alpha = rj'*yj/(dj'*w);
    zj = zj + alpha * dj;
    rnew = rj - alpha * w;
    j = j+1;
    if norm(rnew, 2)<tol
        pk = zj;
        break;
    end
    temp = L\(P*rnew);
    ynew = U\temp;
    beta = rnew'*ynew/(rj'*yj);
    dj = ynew + beta *dj;
    rj = rnew;
    yj = ynew;
end
if  j>=internal_jmax && norm(rnew
    , 2)>=tol
    pk = zj;
    flag_enough_internal_steps =
        false;
end

alpha = 1;

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

% backtracking checking the
    Armojo condition
bt = 0;
while bt < btmax && fnew >=
    farmijo(fk, alpha, gradfk, pk)
    alpha = rho * alpha;
    xnew = xk + alpha * pk;
    fnew = f(xnew);

    bt = bt + 1;
end
        if fnew >= farmijo(fk, alpha,
            gradfk, pk)
             flag_armijo_always_satisfied=
                false;
        end

        xk = xk + alpha * pk;

        % update variables for the next
            iteration
        fk = f(xk);
        gradfk = gradf(xk);
        gradfk_norm = norm(gradfk, 2);
        k=k+1;

        % save variables for the output
        xseq(:, k) = xk;
        btseq(1, k) = bt;
        internal_jseq(1, k) = j;
    end

    if gradfk_norm < tolgrad
        flag_converged=true;
    end

    % trim the variables keeping only
        used cells
    xseq = xseq(:, 1:k);
    btseq = btseq(:, 1:k);
    internal_jseq = internal_jseq(:, 1:k)
        ;

end
```

Nelder-Mead method

Listing 4. Nelder-Mead method

```
function [xmin,fmin,k,xseq] =
    NM_method_impr(f,x0,kmax,rho,chi,gamma
    ,sigma,tol)
    % Shrinking phase of a single point
    fshrinking = @(x1,xi) x1 + sigma*(xi-
        x1) ;

    % Variable initialization
    k = 1;
    xk = x0;
    % n is the number of points in a
        simplex (n-1 is the dimension of
        the problem)
    [m,n] = size(x0);
    flag_shr = true;
    xseq = zeros(m,kmax);
    while k<kmax
        %SORTING PHASE
        if flag_shr
        % a complete sorting is
            performed only at the
            start and after a
            shrinking phase
        fxks = f(xk);
        [fxks,sortedInd] = sort(fxks)
            ;
        xk = xk(:,sortedInd);
        flag_shr=false;
    else
        fnew = f(xk(:,end));
        [xk,fxks ] = insertion_sort(
            xk, fxks(1:end), fnew);
    end


    sumNbest = sum(xk(:,1:end-1),2);%
        saved in a separate variable
        because this computation is
        used twice
    xseq(:,k) = (sumNbest+xk(:,end))/
        n; %barycenter of best "
        num_points" used for the exp.
        conv. rate

    % TERMINATION
    change_rate = sqrt(sum((fxks-mean
        (fxks)).^2)/(n));
    if change_rate <= tol
        break
    end

    %REFLECTION PHASE
    xb = sumNbest/(n-1); %barycenter
        of best "num_points"-1 points
    xr = xb + rho*(xb-xk(:,end));
    if(f(xr)<fxks(end-1) && f(xr)>=
        fxks(1))
        xk(:,end) = xr;
    elseif(f(xr)<fxks(1))
        %EXPANSION PHASE
        xe = xb + chi*(xr-xb);
        if(f(xe)<f(xr))
            xk(:,end) = xe;
        else
            xk(:,end) = xr;
        end
    else
        %CONTRACTION PHASE
        if(f(xr)<fxks(end))
            xc = xb - gamma*(xb-xr);
        else
            xc = xb - gamma*(xb-xk(:,
                end));
        end
        if(f(xc)>= fxks(end))
            %SHRINKING PHASE
            flag_shr=true;
            fshrSpecific = @(x)
                fshrinking(xk(:,1),x);
            xk(:,2:end) =
                fshrSpecific(xk(:,2:
                end));
        else
            xk(:,end) = xc;
        end
    end
    k = k+1;
end
if(k == kmax)
    xseq = xseq(:,1:k-1);
else
    xseq = xseq(:,1:k);
end
xmin = xk(:,1);
fmin = f(xmin);
end
```

Insertion sort

Listing 5. Single iteration of the Insertion sort

```
function [X, fval] = insertion_sort(X,
    fval, fnew)
    xnew = X(:, end);
    i=length(fval);
    while i>=2
        if fval(i-1)>fnew
            X(:, i) = X(:, i-1);
            fval(i) = fval(i-1);
        else
            break;
        end
        i=i-1;
    end
    X(:, i) = xnew;
    fval(i) = fnew;
end
```

Convergence rate

Listing 6. Computation of the convergence rate with regression

```
function [p] = compute_conv_rate(xseq,
    xsol)
    [~, k] = size(xseq);
    % k-2 because the last point in xseq
        is the solution
    y = zeros(1, k-2);
    x = zeros(1, k-2);

    for i=1:k-2
        y(1,i) = log(norm(xseq(:, i+1)-
            xsol)); % log||e_k+1||
        x(1,i) = log(norm(xseq(:, i) -
            xsol)); % log||e_k||
    end

    % regression
    p = x(:)\y(:);
end
```