Politecnico di Torino

Corso di Laurea in
Ingegneria Matematica

Numerical Optimization for large scale problems - Assignment on Unconstrained Optimization

Ghezzi Lucia s325951
Girlando Emanuela s324699
Paradiso Giuliana s319688

5 Settembre 2024

Contents

1 Introductory analysis of the problem

This report aims at discussing and comparing the behaviour of two numerical methods for unconstrained optimization. The methods are used to solve the generic problem

$$\min_{x \in \mathbb{R}^n} f(x) \qquad f : \mathbb{R}^n \to \mathbb{R}$$

The two implemented methods are Modified Newton method and Truncated Newton method, which have both been tested on the Rosenbrock function first, and then applied to the following test problems:

- Rosenbrock function in R^2
- Generalized Broyden tridiagonal function - Problem 5
- Penalty function - Problem 27
- Variabily dimensioned function - Problem 28

The objective of the discussion is to compare the behaviour of the iterative methods, starting from different initial points; the comparison is based on:

- whether convergence to a minimum point has been reached
- the number of iterations necessary for convergence
- the norm of the gradient at the approximate solution
- the execution time
- the experimental rate of convergence

Truncated Newton method has been implemented both with and without preconditioning. All methods have been implemented using MATLAB software. The code is provided in the appendix.

2 Description of the methods

The chosen methods are variations of the Newton method, which relies on the approximation of the function f, at each iteration k, with a second-order Taylor expansion around the point $x^{(k+1)} = x^{(k)} + p \qquad \forall k \geq 0$:

$$f(x^{(k)} + p) \simeq f(x^{(k)}) + p^T \nabla f(x^{(k)}) + \tfrac{1}{2} p^T \nabla^2 f(x^{(k)}) p$$

The descent direction at every iteration is computed by solving the linear system:

$$\nabla^2 f(x^{(k)}) p = -\nabla f(x^{(k)}).$$

It is important to notice that p is truly a descent direction at iteration k only if $\nabla^2 f(x^{(k)})$ is positive definite, which is not always guaranteed.
The rate of convergence for Truncated Newton and Modified Newton method can vary depending on the problem and the specific implementation. Since under proper assumptions the pure Newton Method converges quadratically to the solution, in the best case, the two aforementioned methods achieve quadratic convergence, but, considering the use of an inexact line search, convergence might often result in only linear progress. Furthermore, since the minima of the functions are not known except for the first one, only an approximation of the true error at step k could be computed as:

$$e_k \approx ||x_k - x_{k-1}||$$

To compute the convergence rate, the following relationship was used:

$$p = \frac{log(\frac{||e_{k+1}||}{||e_k||})}{log(\frac{||e_k||}{||e_{k-1}||})}$$

2.1 Modified Newton method

Modified Newton method is used to overcome the issue of a non positive definite Hessian matrix $\nabla^2 f(x^{(k)})$, which can arise at some iterations. This method is based on the idea of applying a correction to the non positive definite Hessian by summing it with the correction matrix E_k, in such a way that the new matrix

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

is sufficiently positive definite, meaning that:

$$\lambda_{min}(B_k) \geq \delta > 0$$

where δ is a tolerance parameter and $\lambda_{min}(B_k)$ is the smallest eigenvalue of B_k.

E_k should be well conditioned, easy to factorize with direct methods and as small as possible in order to have B_k as close as possible to $\nabla^2(f(x_k))$. For these purposes the best choice is

$$E_k = \tau_k I$$

where I is the identity matrix and $\tau_k \in \mathbb{R}$.

Theoretically, $\tau_k = max(0, \delta - \lambda_{min}(\nabla^2 f(x^{(k)}))$, but, since computing the eigenvalues of a matrix can be computationally expensive, the following relation can be used to avoid this calculation:

$$\lambda_i(B_k) \leq ||B_k||_F$$

where $||B_k||_F := \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}}$ Frobenius norm.

The Modified Newton method algorithm is the following:

Starting from the initial guess $x^{(0)}$
for $k = 0, 1, 2...$

- build $B_k = \nabla^2 f(x^{(k)}) + E_k$, where
  - $E_k = 0$ if $\nabla^2 f(x^{(k)})$ is sufficiently definite positive,
  - otherwise $E_k = \tau_k I$ such that B_k is sufficiently definite positive
- solve the linear system $B_k p^{(k)} = -\nabla f(x^{(k)})$
- set $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$ with $\alpha^{(k)}$ such that it satisfies the Armijo backtracking condition

The algorithm for the computation of the matrix B_k is the following:

- set $\beta = ||\nabla^2 f(x^{(k)})||_F$
- if $[\nabla^2 f(x^{(k)})]_{ii} > 0 \quad \forall i = 1, ..., n$ set $\tau_0 = 0$;
  otherwise $\tau_0 = \frac{\beta}{2}$
- for $j = 0, 1, ...$
  - compute the Choleski factorization of $\nabla^2 f(x^{(k)}) + \tau_j I$
    if we succeed $B_k = \nabla^2 f(x^{(k)}) + \tau_j I$
    otherwise $B_k = \nabla^2 f(x^{(k)}) + \tau_j I$ is not definite positive, meaning that τ_j is not large enough, so we increase it: $\tau_{j+1} = max(2\tau_j, \frac{\beta}{2})$

2.2 Truncated Newton method

Truncated Newton method is used to overcome the issue of non positive definite Hessian matrices. It consists in a modification of Inexact Newton Method combined with a linesearch strategy in which we guarantee that $p^(k)$ is a descent direction even if the Hessian is not positive definite; while Newton method exploits direct methods for solving the linear system that provides the descent direction, Truncated Newton method uses Conjugate Gradient method (CG) to do so. Since CG method is designed for linear systems with positive definite coefficient matrix, in the case of Truncated Newton method the CG iterations are terminated as soon as a negative curvature direction arises. The CG iteration works as follows:

- the starting point for the CG iteration is $x^{(0)} = 0$
- if a search direction $p^{(i)}$ satisfies the condition: $p^{(i)T} A p^{(i)}$, meaning $p^{(i)}$ is a direction of negative curvature, the process is stopped. If this occurs at the first CG iteration, a new iterate $p^{(i)}$ is computed and the process stops; otherwise if the negative curvature condition is satisfied for iteration i, then $p^{(i-1)}$ is returned and the process stops.

The algorithm of Truncated Newton method is the following:

- given the initial guess $x^{(0)}$
- for $k = 0, 1, ...$
  - compute $p^{(k)}$ applying CG to the system $Az = b$, starting from $z^{(0)} = 0$ and ending CG iterations when $||Az^{(i)} - b|| < \eta_k ||\nabla f(x^{(k)})||$ or when $z^{(i)T} Az \leq 0$
  - set $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$ with $\alpha^{(k)}$ such that it satisfies the Armijo backtracking condition

It's important to acknowledge that CG is highly affected by the condition number of the matrix A: $K(A) = ||A||\ ||A^{-1}||$, meaning that the larger K(A), the slower the convergence. To lessen this problem it's possible to apply preconditioning to CG.

2.3 Line Search and Parameters for Newton Method and Truncated Newton Method

To determine an appropriate step size α_j during optimization, a line search strategy has been implemented. The backtracking line search employed incorporates the Armijo Condition, also known as the sufficient decrease condition which is:

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

where:

- x_k is the current point.
- p_k is the search direction.
- α is the step size.
- c_1 is a small parameter, typically in the range $(0, 1)$.

Newton Method requires the computation of the Hessian matrix, which can be computationally expensive. The key parameters involved are:

- **Step Size** α: determined through line search as described above.
- **Tolerance tolgrad**: convergence is considered achieved when the gradient norm $\|\nabla f(x)\|$ is below this threshold.
- **Maximum Iterations** $k_{\mathbf{max}}$: defines the maximum number of iterations allowed.

Truncated Newton Method is used to handle large-scale optimization problems more efficiently. Key parameters include the same of the previous one and furthermore:

- **Preconditioning**:
  - **Diagonal Matrix**: A preconditioner derived from the diagonal elements of the linear system matrix.
  - **Incomplete Cholesky Factorization**: A preconditioner based on the incomplete Cholesky factorization of the matrix.
- **Maximum Backtracking Iterations btmax**: The maximum number of iterations for the backtracking line search.

Parameter Values

After an initial tuning phase, the values of the parameters providing the best results were selected:

- $c_1 = 1 \times 10^{-4}$
- $\rho = 0.8/0.5$
- btmax $= 10$
- tolgrad $= 1 \times 10^{-8}$
- $k_{\max} = 5 \times 10^3$

The line search strategy ensures an appropriate step size for both Newton Method and Truncated Newton Method. The parameters and preconditioning techniques play a crucial role in enhancing the efficiency and effectiveness of these optimization methods.

3 Results and comparison among the methods

3.1 Rosenbrock test function in $\mathbb{R}^2$

The Rosenbrock function is defined as:

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

The selected starting points are:

(a) $x_1 = (1.2, 1.2)$

(b) $x_2 = (-1.2, 1)$

(c) $x_3 = (0, 0)$

(d) $x_4 = (0.5; 1.5)$

(e) $x_5 = (-1; 1)$

The global minimum of the Rosenbrock function is known and is $x^* = (1, 1)$. This information was used to compute the experimental rate of convergence of the methods and to check whether convergence was reached or not.

The results obtained applying the Modified Newton Method with $\rho = 0.8$ are summarized in the following Table. The results are satisfying and show that for all initial points the method was able to converge to the minimum point with a few iterations and a low computational time.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	8	5.6227e-10	linear	0.0088 sec	yes
x_2	22	9.2699e-09	almost linear	0.018731 sec	yes
x_3	14	7.3225e-09	linear	0.0174 sec	yes
x_4	10	2.2821e-10	almost linear	0.0028 sec	yes
x_5	20	4.8122e-10	almost linear	0.0045 sec	yes

The results obtained applying the Truncated Newton Method with $\rho = 0.5$ without preconditioning are summarized in the following Table.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	10	0	variable	0.0182 sec	yes
x_2	64	9.1038e-15	variable	0.0160 sec	yes
x_3	18	1.7085e-12	variable	0.0185 sec	yes
x_4	8	8.0905e-13	variable	0.0175 sec	yes
x_5	63	5.3862e-14	almost linear	0.0186 sec	yes

The results obtained applying the Truncated Newton Method with $\rho = 0.5$ with preconditioning are summarized in the following Table.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	241	5.8741e-09	linear	0.0530 sec	yes
x_2	367	4.3353e-09	variable	0.7993	yes
x_3	431	6.4746e-09	almost linear	0.9506 sec	yes
x_4	246	1.6784e-09	linear	0.0390 sec	yes
x_5	445	9.1229e-09	linear	0.9406 sec	yes

Convergence is reached for Truncated Newton method for all initial points. The method without preconditioning attains the minimum within fewer iterations and in less time for almost all points with respect to when preconditioning is applied. This result reveals that applying preconditioning was not necessary in this case and only led to a longer computational time.

3.2 Generalized Broyden tridiagonal function - Problem 5

$$F(x) = \sum_{i=1}^{n} \left|(3 - 2x_i)x_i - x_{i-1} - x_{i+1} + 1\right|^p,$$

$$p = \frac{7}{3}, \quad x_0 = x_{n+1} = 0,$$

$$\bar{x}_i = -1, \quad i \geq 1.$$

The methods were tested with $n = 10^3$ and the selected starting points are:

(a) $x_1 = (0, 0, \ldots, 0)$;

(d) $x_2 = (-1, -1, \ldots, -1)$;

(c) $x_3 = (-1, 1, -1, 1, \ldots, 1)$;

(d) $x_4 = (0, \ldots, -1, 0, \ldots)$, which is a sparse vector with -1 in positions $10, 20, 30 \ldots$;

In this table we examine the results of the application of Modified Newton Method to the above mentioned function. The method attains convergences for all starting points in a small execution time and few iterations.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	24	3.3807e-09	linear	0.55739 sec	yes
x_2	24	4.234e-09	linear	0.55639 sec	yes
x_3	69	2.3631e-09	almost linear	1.2155 sec	yes
x_4	30	8.7525e-09	linear	0.57396 sec	yes

In the following table we examine the results of the application of Truncated Newton method without preconditioning to function 5. Convergence is reached for all the points.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	34	2.0293e-09	variable	0.39126 sec	yes
x_2	16	4.0079e-09	linear	0.23212 sec	yes
x_3	52	9.4457e-09	variable	0.75356 sec	yes
x_4	47	2.4913e-09	linear	0.28399 sec	yes

In the next table we examine the results of the application of Truncated Newton method with preconditioning. In most cases the latter produces better results with respect to Truncated Newton method without preconditioning, despite an increment in the execution time.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	22	9.3076e-09	almost linear	2.5549 sec	yes
x_2	17	1.8529e-09	linear	2.1495 sec	yes
x_3	24	2.5069e-09	linear	2.5264 sec	yes
x_4	17	4.6843e-09	linear	2.0601 sec	yes

3.3 Penalty function 1 - Problem 27

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

where

$$f_k(x) = \begin{cases} \frac{1}{\sqrt{100000}}(x_k - 1), & \text{if } 1 \leq k \leq n, \\ \sum_{i=1}^{n} x_i^2 - \frac{1}{4}, & \text{if } k = n + 1, \end{cases}$$

with $m = n + 1$. The methods were tested with n=10^3 and the selected starting points are:

(b) $x_1 = (0, 0, 0, \ldots, 0)$;

(d) $x_2 = (1, 1, 1, \ldots, 1)$;

(c) $x_3 = (-1, -1, \ldots, -1)$;

(d) $x_4 = (\frac{1}{2}, \frac{1}{2}, \ldots, \frac{1}{2})$;

(d) $x_5 = linspace(1, 1/n, n)'$;

In this table we examine the results of the application of Modified Newton Method; The method was able to reach convergence for all starting points in a small execution time with linear experimental order of convergence.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	124	9.1124e-09	linear	4.9707 sec	yes
x_2	78	4.6798e-13	linear	2.9124 sec	yes
x_3	222	9.232e-9	almost linear	6.2981 sec	yes
x_4	136	9.7724e-09	linear	4.3950 sec	yes
x_5	391	3.5357e-13	almost linear	10.6944 sec	yes

In this table we examine the results of the application of Truncated Newton method without preconditioning

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	1	1.6001e-11	-	0.0253 sec	yes
x_2	29	1.6877e-12	almost linear	0.3554 sec	yes
x_3	165	9.9512e-09	linear	0.44796 sec	yes
x_4	136	9.7724e-09	linear	1.3637 sec	yes
x_5	110	1.6526e-14	linear	1.1262 sec	yes

In this table we examine the results of the application of Truncated Newton method with preconditioning

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	1	1.6001e-11	-	0.0890 sec	yes
x_2	29	8.8299e-12	linear	1.4612 sec	yes
x_3	165	9.9512e-09	linear	7.903 sec	yes
x_4	146	8.9666e-09	almost linear	184.9569 sec	yes
x_5	392	9.4084e-09	almost linear	100.28 sec	yes

The results are satisfying in both cases, since the methods achieved convergence for all stating points; For points x_4, x_5 in the 1^{st} case (no preconditioning) a few iterations were necessary and the experimental rate of convergence was assessed to be linear. In the 2^{nd}

case (with preconditioning) a large number of iterations and a long computational time was needed for the convergence, and the experimental rate of convergence appeared to be more variable, but still almost linear. For starting points x_2, x_3 the number of iterations necessary for convergence is the same, but preconditioning widens the computational time.

3.4 Variably dimensioned function - Problem 28

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

where

$$f_k(x) = \begin{cases} x_k - 1, & \text{if } 1 \le k \le n, \\ \sum_{i=1}^{n} i(x_i - 1), & \text{if } k = n+1, \\ \left(\sum_{i=1}^{n} i(x_i - 1)\right)^2, & \text{if } k = n+2, \end{cases}$$

with $m = n + 2$.
The methods were tested with n=10^3 and the selected starting points are:

(b) $x_1 = (0, 0, 0, \ldots, 0)$;

(d) $x_2 = (1, 1, 1, \ldots, 1)$;

(c) $x_3 = (\frac{1}{2}, \frac{1}{2}, \ldots, \frac{1}{2})$;

(d) $x_4 = (-1, -1, \ldots, -1)$;

(d) $x_5 = linspace(1, 1/n, n)'$;

In the next table we examine the results of the application of Modified Newton Method to the selected function; The method attains convergences for all starting points, with a quite large number of iterations and a fairly long computational time. The rate of convergence resulted to be linear.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	399	9.4975e-09	almost linear	11.5500 sec	yes
x_2	298	1.2116e-09	linear	7.0449 sec	yes
x_3	578	2.2341e-09	linear	16.7861 sec	yes
x_4	272	3.2116e-09	linear	6.7983 sec	yes
x_5	681	1.9538e-09	linear	13.5058 sec	yes

In this table we examine the results of the application of Truncated Newton method without preconditioning. In this case the method converges starting from x_2, x_3, x_4 in fewer iterations and less computational time with respect to Modified Newton method at a linear rate; nonetheless, for x_1 and x_5 convergence is not achieved with the considered tolerance on the gradient norm (still it is of the order 10^{-7}, so fairly small).

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	5000	4.1529e-07	-	-	no
x_2	190	9.7744e-09	linear	1.4994 sec	yes
x_3	468	2.3468e-10	linear	2.9196 sec	yes
x_4	539	5.3430e-09	linear	3.3049 sec	yes
x_5	5000	8.9231e-07	-	-	no

In this table we examine the results of the application of Truncated Newton method with preconditioning. As for the previous test functions, preconditioning widens the computational time necessary for convergence. Moreover, it is possible to notice that starting from x_1 preconditioning is crucial for the convergence, while when using x_5 as a starting point convergence is not achieved neither with preconditioning.

$x^{(0)}$	num.iterations	$\|\nabla f(x^*)\|$	exp rate of conv.	execution time	conv. reached
x_1	1439	1.1890e-09	variable	544.0973 sec	yes
x_2	267	1.9272e-09	variable	312.8285 sec	yes
x_3	567	4.7626e-09	linear	547.9713	yes
x_4	2217	9.4975e-09	linear	1.6300e+03 sec	yes
x_5	5000	1.0536e-08	-	-	no

4 Final considerations

A total of 57 experiments were carried out, resulting in 54 successful outcomes, often with relatively few iterations and 3 unsuccessful results. Even when the algorithm did not succeed, it got very close to a minimum point, as hinted by the small gradient norm. The observed behaviour suggests that the methods are efficient.

Generally, Modified Newton Method tends to outperform Truncated Newton Method, although there are specific cases (such as in some starting points of Problem 28) where Truncated Newton method without preconditioning performs better in terms of execution time and number of iterations; nonetheless, convergence is always reached in both cases. Additionally, Truncated Newton method with preconditioning takes more time with respect to the other methods to reach convergence, and it sometimes fails (see Problem 28).

In some instances, the observed convergence rate aligns with the theoretical predictions, while in others it differs, likely due to potential violations of theoretical assumptions. Execution is generally not excessively time demanding; nonetheless, a notable increase in running time is observed when preconditioning is applied, likely due to the iterative computation of the incomplete Choleski factor and of a matrix inverse.

5 Appendix

```
clear
close all
clc
%% ROSENBROCK FUNCTION IN 2D
% parameters
c1 = 1e-4; % used for the Armijo condition
rho = 0.8; %it works better with 0.8 for Modified Newton and Truncated Newton without preconditioning
btmax = 10; %max numer of iterations in the backtracking cycle
gradftol = 1e-8; % tolerance on gradient norm (stopping criterion)
kmax = 5e3; %max number of outer iterations

% starting points
x0=[1.2;1.2];
%x0=[-1.2;1];
%x0=[0;0];
%x0=[-1;-1];
%x0=[0.4;-0.4];

% function handles for f(x), its gradient and its Hessian matrix
fR = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
gradfR = @(x) [-400*x(1)*(x(2)-x(1)^2)-2*(1-x(1)); 200*(x(2)-x(1)^2)];
HfR= @(x) [-400*x(2)+1200*x(1)^2+2, -400*x(1) ; -400*x(1), 200];

%% MODIFIED NEWTON METHOD
disp ('***** START OF MODIFIED NEWTON METHOD FOR ROSENBROCK FUNCTION *****')
tic ;
[xk, fk, k, gradfk_norm, xseq, btseq] =...
   modifiednewton(x0, fR, gradfR, HfR, gradftol, kmax, c1, rho, btmax);
elapsed_time = toc ;

disp ('***** FINISHED *****')
disp ('***** MODIFIED NEWTON METHOD RESULTS *****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence given the minimum point
x_min=[1;1];
[pmedian, pseq] = exp_conv_rate_min(x_min, k, xseq);

disp ('***************')

%% TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
disp ('***** START OF TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING FOR ROSENBROCK FUNCTION *****')
tic ;
[xk , fk , gradfk_norm , k, xseq , btseq ] = ...
     truncated_newton(x0 , fR, gradfR , HfR , kmax , gradftol , c1 , rho , btmax);
elapsed_time = toc ;

disp ('***** FINISHED *****')
```

```
disp ('***** TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING RESULTS *****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence given the minimum point
x_min=[1;1];
[pmedian, pseq] = exp_conv_rate_min(x_min, k, xseq);

disp ('***************')

%% TRUNCATED NEWTON METHOD WITH PRECONDITIONING
rho = 0.5; % preconditioned Truncated Newton method works better with rho=0.5
disp ('***** START OF TRUNCATED NEWTON METHOD WITH PRECONDITIONING FOR ROSENBROCK FUNCTION *****')
tic ;
[xk , fk , gradfk_norm , k, xseq , btseq ] = ...
    truncated_newton_pre(x0 , fR, gradfR , HfR , kmax , gradftol , c1 , rho , btmax );
elapsed_time = toc ;

disp ***** FINISHED *****
disp ('***** TRUNCATED NEWTON METHOD WITH PRECONDITIONING RESULTS*****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence given the minimum point
x_min=[1;1];
[pmedian, pseq] = exp_conv_rate_min(x_min, k, xseq);

disp ('***************')

clear
close all
clc

%% PROBLEM 27: PENALTY FUNCTION
% parameters
n=1e3; %dimension of the R^n space
c1 = 1e-4; % used for the Armijo condition
rho = 0.8; %works better with 0.8 rather than 0.5
btmax = 10; %max numer of iterations in the backtracking cycle
gradftol = 1e-8; % tolerance on gradient norm (stopping criterion)
kmax = 5e3; %max number of outer iterations

% starting points
x0=zeros(n,1);
%x0=ones(n,1);
%x0=-*ones(n,1);
%x0=0.5*ones(n,1);
%x0=linspace(1,n,n)';

% function handles for f(x), its gradient and its Hessian matrix
f27 = @(x) 1/2*(sum( (1/sqrt(100000)*(x(n:1)-1)).^2 )) ...
```

```
    + 1/2*sum( (x(1:n).^2 -1/4) ).^2;
gradf27 = @ (x) (1/ sqrt(100000)*( x(1: end) - 1) + ...
    2*x(1: end).*sum( (x(1: end ).^2 - 1/4) ) ) ;
Hf27_diag = @ (x) diag ((1/ sqrt (100000) + ...
    2* sum( x (1: end) .^2 - 1/4) ) + 4*( x(1: end ) .^2) ) ;
Hf27_nondiag = @ (x) 4 * ( x * x') - diag (4 * x .^2) ;
Hf27 = @ (x) Hf27_diag ( x ) + Hf27_nondiag ( x ) ;

%% MODIFIED NEWTON METHOD
disp ('***** START OF MODIFIED NEWTON METHOD FOR PROBLEM 27 FUNCTION
   *****')
tic ;
[xk, fk, k, gradfk_norm, xseq, btseq] =...
    modifiednewton(x0, f27, gradf27, Hf27, gradftol, kmax, c1, rho,
       btmax);
elapsed_time = toc ;
disp ('***** FINISHED *****')
disp ('***** MODIFIED NEWTON METHOD RESULTS *****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence without knowing the minimum point
[pmedian, pseq] = exp_conv_rate(k, xseq);

disp ('***************')

%% TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
disp ('***** START OF TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
   FOR PROBLEM 27 FUNCTION *****')
tic ;
[xk , fk , gradfk_norm , k, xseq , btseq ] = ...
     truncated_newton(x0 , f27, gradf27 , Hf27 , kmax , gradftol , c1 ,
         rho , btmax);
elapsed_time = toc ;
disp ('***** FINISHED *****')
disp ('***** TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING RESULTS
   *****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence without knowing the minimum point
[pmedian, pseq] = exp_conv_rate(k, xseq);

disp ('***************')

%% TRUNCATED NEWTON METHOD WITH PRECONDITIONING
disp ('*****   START OF TRUNCATED NEWTON METHOD WITH PRECONDITIONING
   FOR PROBLEM 27 FUNCTION *****')
tic ;
[xk , fk , gradfk_norm , k, xseq , btseq ] = ...
     truncated_newton_pre(x0 , f27, gradf27 , Hf27 , kmax , gradftol ,
        c1 , rho , btmax );
elapsed_time = toc ;
disp ('***** FINISHED *****')
disp ('***** TRUNCATED NEWTON METHOD WITH PRECONDITIONING RESULTS *****
   ')
```

```
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence without knowing the minimum point
[pmedian, pseq] = exp_conv_rate(k, xseq);

disp ('***************')

clear
close all
clc

%% PROBLEM 28: VARIABLY DIMENSIONED FUNCTION
% parameters
n=1e3; %dimension of the R^n space
c1 = 1e-4; % used for the Armijo condition
rho = 0.8; %works better with 0.8 rather than 0.5
btmax = 10; %max numer of iterations in the backtracking cycle
gradftol = 1e-8; % tolerance on gradient norm (stopping criterion)
kmax = 5e3; %max number of outer iterations

% starting points
x0=zeros(n,1);
%x0=ones(n,1);
%x0=-ones(n,1);
%x0=0.5*ones(n,1);
% x0=linspace(1,n,n)';

% function handles for f(x), its gradient and its Hessian matrix
f28 = @(x) 1/2*sum((x(1:n)-1).^2) ...
    + 1/2*(sum( (1:n)*(x(1:n)-1) ).^2) ...
    + 1/2*(sum( (1:n)*(x(1:n)-1) ).^4);
gradf28 = @ (x) ( x(1: end ) + (1: n )'.* sum ((1: n )'.*( x (1: end )
    -1) ) + ...
    2*(1: n )'.* sum ((1: n )'.*( x (1: end) -1) ) .^3) ;
Hf28 = @ (x) ((1: n )'.*(1: n ) + ...
    6*(1: n )'.*(1: n ) .*( sum ((1: n )'.*( x (1:end ) -1) ) ) .^2 +
       diag ( ones (n ,1) ) ) ;

%% MODIFIED NEWTON METHOD
disp ('***** START OF MODIFIED NEWTON METHOD FOR PROBLEM 28 FUNCTION
    *****')
tic ;
[xk, fk, k, gradfk_norm, xseq, btseq] =...
     modifiednewton(x0, f28, gradf28, Hf28, gradftol, kmax, c1, rho,
        btmax);
elapsed_time = toc ;
disp ('***** FINISHED *****')
disp ('***** MODIFIED NEWTON METHOD RESULTS *****')
disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])

% experimental rate of convergence without knowing the minimum point
[pmedian, pseq] = exp_conv_rate(k, xseq);

disp ('***************')
```

```
208
209 %% TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
210 disp ('*****   START OF TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
        FOR PROBLEM 28 FUNCTION *****')
211 tic ;
212 [xk , fk , gradfk_norm , k, xseq , btseq ] = ...
213      truncated_newton(x0 , f28, gradf28 , Hf28 , kmax , gradftol , c1 ,
           rho , btmax);
214 elapsed_time = toc ;
215 disp ('***** FINISHED *****')
216 disp ('***** TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING RESULTS
       *****')
217 disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
218 disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
219 disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])
220
221 % experimental rate of convergence without knowing the minimum point
222 [pmedian, pseq] = exp_conv_rate(k, xseq);
223
224 disp ('***************')
225
226 %% TRUNCATED NEWTON METHOD WITH PRECONDITIONING
227 disp ('***** START OF TRUNCATED NEWTON METHOD WITH PRECONDITIONING FOR
       PROBLEM 28 FUNCTION *****')
228 tic ;
229 [xk , fk , gradfk_norm , k, xseq , btseq ] = ...
230      truncated_newton_pre(x0 , f28, gradf28 , Hf28 , kmax , gradftol ,
           c1 , rho , btmax );
231 elapsed_time = toc ;
232 disp ('*****   FINISHED *****')
233 disp ('***** TRUNCATED NEWTON METHOD WITH PRECONDITIONING RESULTS *****
       ')
234 disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
235 disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
236 disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])
237
238 % experimental rate of convergence without knowing the minimum point
239 [pmedian, pseq] = exp_conv_rate(k, xseq);
240
241 disp ('***************')
242
243 clear
244 close all
245 clc
246
247 %% PROBLEM 5: Generalized Broyden tridiagonal function
248
249 % parameters
250 n=1e3; %dimension of the R^n space
251 c1 = 1e-4; % used for the Armijo condition
252 rho = 0.8; %works better with 0.8 rather than 0.5
253 btmax = 10; %max numer of iterations in the backtracking cycle
254 gradftol = 1e-8; % tolerance on gradient norm (stopping criterion)
255 kmax = 5e3; %max number of outer iterations
256
257 % starting points
258 x0=zeros(n,1); %first point
259 %x0=-ones(n,1); %second point
```

```
%x0=(-1).^(1:n)'; %third point
%x0 = zeros(n,1); %fourth point
%x0(10:10:end) = -1;


% function handles for f(x), its gradient and its Hessian matrix

f_5 = @(x) (abs((3-2*x(1))*x(1) - x(2)+1)).^(7/3) + ...
sum(abs((3-2*x(2:n-1)).*x(2:n-1) -x(1:n-2)- x(3:n)+1).^(7/3)) + ...
(abs((3-2*x(n))*x(n)-x(n-1)+1)).^(7/3);

gradf5 = @(x) [ ...

    7/3 * (abs((3 - 2*x(1)) * x(1) - x(2) + 1)).^(4/3) .* sign((3 - 2*x
       (1)) * x(1) - x(2) + 1) * (3 - 4*x(1)) + ...
    7/3* (abs((3 - 2*x(2)) * x(2) - x(1) - x(3) + 1)).^(4/3) .* sign((3
        - 2*x(2)) * x(2) - x(1) - x(3) + 1) * (-1); ...

    7/3 * (abs((3 - 2*x(1)) * x(1) - x(2) + 1).^(4/3).*sign((3 - 2*x(1)
       ) * x(1) - x(2) +1)) * (-1) + ...
    7/3 * (abs((3 - 2*x(2)) * x(2) - x(1) - x(3) + 1).^(4/3).* sign((3
       - 2*x(2)) * x(2) - x(1) - x(3) + 1)) * (3 - 4*x(2)) + ...
    7/3 * (abs((3 - 2*x(3)) * x(3) - x(2) - x(4) + 1).^(4/3).* sign((3
       - 2*x(3)) * x(3) - x(2) - x(4) + 1)) * (-1);

    7/3 * (abs((3 - 2*x(2:end-3)) .* x(2:end-3) - x(1:end-4) - x(3:end
       -2) + 1)).^(4/3) .* sign((3 - 2*x(2:end-3)) .* x(2:end-3) - x(1:
       end-4) - x(3:end-2) + 1) .* (-1) + ...
    7/3 * (abs((3 - 2*x(3:end-2)) .* x(3:end-2) - x(2:end-3) - x(4:end
       -1) + 1)).^(4/3) .* sign((3 - 2*x(3:end-2)) .* x(3:end-2) - x(2:
       end-3) - x(4:end-1) + 1) .* (3 - 4*x(3:end-2)) + ...
    7/3 * (abs((3 - 2*x(4:end-1)) .* x(4:end-1) - x(3:end-2) - x(5:end)
        + 1)).^(4/3) .* sign((3 - 2*x(4:end-1)) .* x(4:end-1) - x(3:end
       -2) - x(5:end) + 1) .* (-1); ...

    7/3 * (abs((3 - 2*x(end-2)) * x(end-2) - x(end-3) - x(end-1) + 1)
       .^(4/3) .* sign((3 - 2*x(end-2)) * x(end-2) - x(end-3) - x(end
       -1) +1)) * (-1) + ...
    7/3 * (abs((3 - 2*x(end-1)) * x(end-1) - x(end-2) - x(end) + 1)
       .^(4/3) .* sign((3 - 2*x(end-1)) * x(end-1) - x(end-2) - x(end)
       + 1)) * (3 - 4*x(end-1)) + ...
    7/3 * (abs((3 - 2*x(end)) * x(end) - x(end-1) + 1).^(4/3) .* sign
       ((3 - 2*x(end)) * x(end) - x(end-1) + 1)) * (-1);

    7/3 * (abs((3 - 2*x(end-1)) * x(end-1) - x(end-2) - x(end) + 1))
       .^(4/3) .* sign((3 - 2*x(end-1)) * x(end-1) - x(end-2) - x(end)
       + 1) * (-1) + ...
    7/3 * (abs((3 - 2*x(end)) * x(end) - x(end-1) + 1)).^(4/3) .* sign
       ((3 - 2*x(end)) * x(end) - x(end-1) + 1) * (3-4*x(end))];

H=zeros(n,n);

d = @(x) [ ...

    28/9 * abs((3 - 2*x(1)) .* x(1) - x(2) + 1).^(1/3) .* (3-4*x(1)).^2
        + ...
    7/3 * abs((3 - 2*x(1)) .* x(1) - x(2) +1).^(4/3) .* sign((3 - 2*x
       (1)) .* x(1) - x(2) +1) * (-4) + ...
```

```
    28/9 * abs((3 - 2 * x(2)) .* x(2) - x(1) - x(3) +1).^(1/3); ...

    28/9 * abs((3 - 2*x(1)) * x(1) - x(2) +1).^(1/3) + ...
    28/9 * abs((3 - 2*x(2)) * x(2) - x(1) - x(3) + 1).^(1/3) * (3-4*x
        (2)).^2 + ...
    7/3 * abs((3 - 2*x(2)) * x(2) - x(1) - x(3) + 1).^(4/3) .* sign((3
        - 2*x(2)) * x(2) - x(1) - x(3) + 1) * (-4) + ...
    28/9 * abs((3 - 2 * x(3)) * x(3) - x(2) - x(4) +1).^(1/3);

    28/9 * abs((3 - 2*x(2:end-3)) .* x(2:end-3) - x(1:end-4) - x(3:n-2)
         +1).^(1/3) + ...
    28/9 * abs((3 - 2*x(3:end-2)) .* x(3:end-2) - x(2:end-3) - x(4:end
        -1) + 1).^(1/3) .* (3-4*x(3:end-2)).^2 + ...
    7/3 * abs((3 - 2*x(3:end-2)) .* x(3:end-2) - x(2:end-3) - x(4:end
        -1) + 1).^(4/3) .* sign((3 - 2*x(3:end-2)) .* x(3:end-2) - x(2:
        end-3) - x(4:end-1) + 1) * (-4) + ...
    28/9 * abs((3 - 2 * x(4:end-1)) .* x(4:end-1) - x(3:end-2) - x(5:
        end) +1).^(1/3);

    28/9 * abs((3 - 2*x(end-2)) .* x(end-2) - x(end-3) - x(end-1) +1)
        .^(1/3) + ...
    28/9 * abs((3 - 2*x(end-1)) .* x(end-1) - x(end-2) - x(end) +1)
        .^(1/3) .* (3 - 4*x(end-1)).^2 + ...
    7/3 * abs((3 - 2*x(end-1)) .* x(end-1) - x(end-2) - x(end) +1)
        .^(4/3) .* sign((3 - 2*x(end-1)) .* x(end-1) - x(end-2) - x(end)
         +1) * (-4) + ...
    28/9 * abs((3 - 2*x(end)) .* x(end) - x(end-1) +1).^(1/3);

    28/9 * abs((3 - 2*x(end-1)) .* x(end-1) - x(end-2) - x(end) +1)
        .^(1/3) + ...
    28/9 * abs((3 - 2*x(end)) .* x(end) - x(end-1) +1).^(1/3) .* (3 -
        4*x(end)).^2 + ...
    7/3 * abs((3 - 2*x(end)) .* x(end) - x(end-1)+1).^(4/3) .* sign((3
        - 2*x(end)) .* x(end) - x(end-1) +1) * (-4)];

d_up = @(x) [

    28/9 * abs((3 - 2*x(1)) .* x(1) - x(2) +1).^(1/3) *(-1) * (3 - 4*x
        (1)) + ...
    28/9 * abs((3 - 2*x(2)) .* x(2) - x(1) - x(3) +1).^(1/3) *(-1) * (3
         - 4*x(2));

    28/9 * abs((3 - 2*x(2:n-2)) .* x(2:n-2) - x(1:n-3) - x(3:n-1) +1)
        .^(1/3) *(-1) .* (3 - 4*x(2:n-2)) + ...
    28/9 * abs((3 - 2*x(3:n-1)) .* x(3:n-1) - x(2:n-2) - x(4:n) +1)
        .^(1/3) *(-1) .* (3 - 4*x(3:n-1));

    28/9 * abs((3 - 2*x(end-1)) .* x(end-1) - x(end-2) - x(end) +1)
        .^(1/3) *(-1) * (3 - 4*x(end-1)) + ...
    28/9 * abs((3 - 2*x(end)) .* x(end) - x(end-1) +1).^(1/3) *(-1) *
        (3 - 4*x(end))];

d_up2 = @(x) [28/9 * abs((3 - 2*x(2:n-1)) .* x(2:n-1) - x(1:n-2) - x(3:
    n) +1).^(1/3)];


Hf5 = @(x) H + diag(d(x)) + diag(d_up(x),1) + diag(d_up2(x),2) + diag(
    d_up(x),-1) + diag(d_up2(x),-2);
```

```
333
334 %% MODIFIED NEWTON METHOD
335 disp ('******** START OF MODIFIED NEWTON METHOD FOR PROBLEM 5 FUNCTION
        ********')
336 tic ;
337 [xk, fk, k, gradfk_norm, xseq, btseq] =...
338     modifiednewton(x0, f_5, gradf5, Hf5, gradftol, kmax, c1, rho, btmax
          );
339 elapsed_time = toc ;
340 disp ('******** FINISHED ********')
341 disp ('******** MODIFIED NEWTON METHOD RESULTS ********')
342 disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
343 disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
344 disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])
345
346 % experimental rate of convergence without knowing the minimum point
347 [pmedian, pseq] = exp_conv_rate(k, xseq);
348
349 disp ('**************************')
350 %% TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING
351 disp ('******** START OF TRUNCATED NEWTON METHOD WITHOUT
        PRECONDITIONING FOR PROBLEM 5 FUNCTION ********')
352 tic ;
353 [xk , fk , gradfk_norm , k, xseq , btseq ] = ...
354      truncated_newton(x0 , f_5, gradf5 , Hf5 , kmax , gradftol , c1 ,
           rho , btmax);
355 elapsed_time = toc ;
356 disp ('******** FINISHED ********')
357 disp ('******** TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING RESULTS
        ********')
358 disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
359 disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
360 disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])
361
362 % experimental rate of convergence without knowing the minimum point
363 [pmedian, pseq] = exp_conv_rate(k, xseq);
364
365 disp ('**************************')
366 %% TRUNCATED NEWTON METHOD WITH PRECONDITIONING
367 disp ('******** START OF TRUNCATED NEWTON METHOD WITH PRECONDITIONING
        FOR PROBLEM 5 FUNCTION ********')
368 tic ;
369 [xk , fk , gradfk_norm , k, xseq , btseq ] = ...
370      truncated_newton_pre(x0 , f_5, gradf5 , Hf5 , kmax , gradftol , c1
           , rho , btmax );
371 elapsed_time = toc ;
372 disp ('******** FINISHED ********')
373 disp ('******** TRUNCATED NEWTON METHOD WITH PRECONDITIONING RESULTS
        ********')
374 disp ([ 'norm ( gradf(x)):', num2str(gradfk_norm) ])
375 disp(['N. of iterations: ', num2str(k), '/', num2str(kmax), ';'])
376 disp(['Elapsed Time: ', num2str(elapsed_time), ' seconds'])
377
378 % experimental rate of convergence without knowing the minimum point
379 [pmedian, pseq] = exp_conv_rate(k, xseq);
380
381 disp ('**************************')
382 %% FUNCTIONS IMPLEMENTED TO DEAL WITH THE ASSIGNMENT
```

Listing 1: Matlab Code of the project

```
1
2  %% 1) NEWTON METHOD
3
4  function [xk, fk, k, normgradfk, xseq, btseq] =...
5      modifiednewton(x0, f, gradf, Hf, gradftol, kmax, c1, rho, btmax)
6  % input:
7  % x0=starting point of the sequence
8  % f=function to minimize
9  % gradf=gradient of f
10 % Hf=hessian matrix of f
11 % gradftol=tolerance for gradf under which we assume convergence to min
12 % kmax=max number of iterations for the sequence xk
13 % c1=parameter of Armijo condition
14 % rho=reduction parameter for backtracking
15 % btmax=max number of iterations for backtracking
16
17 % output:
18 % xk=approximation of argmin f
19 % fk=approximation of min f
20 % k=number of iterates for convergence
21 % normgradfk=norm of gradf(xk) at convergence
22 % xseq=sequence produced by the method
23 % btseq=sequence produced by backtracking
24
25
26 % Function handle for the armijo condition%
27 farmijo = @(fk, alpha, gradfk, pk) ...
28       fk + c1 * alpha * (-sum((gradfk*pk).^2, 1));
29
30 %initialization
31 xseq = zeros(length(x0), kmax);
32 btseq = zeros(1, btmax);
33
34 xk = x0;
35 fk = f(xk);
36 gradfk = gradf(xk);
37 gradfk=gradfk';
38 hessfk = Hf(xk);
39 normgradfk = norm(gradf(xk));
40 Ek = eye(size(hessfk));
41 Bk = hessfk;
42
43 k = 0;
44 while k <= kmax && normgradfk >= gradftol
45     %build Bk=Hf(xk)+Ek
46     beta = min( norm ( gradfk , 'fro') , norm ( hessfk ,'fro') ) ;
47     tau = 0;
48     for i = 1:size(hessfk, 1)
49         if hessfk(i,i) <= 0
50             tau = beta/2;
51             break
52         end
53     end
54     flag = 1;
55     while flag ~= 0
```

```
        Bk = hessfk + tau*Ek;
        %check if Bk positive definite by trying the Cholesky
        %decomposition
        [~, flag] = chol(Bk);
        %if flag~=0 %if Bk not positive definite, increase tau
            if tau > 0
                tau = 2*tau;
            else
                tau = beta/2;
            end
        %end
    end
    Bk = hessfk + tau*Ek;

    %solve the linear system Bk*pk=-gradf(xk)
    pk = Bk \ (-gradfk');

    alpha = 1;
    xnew = xk + alpha * pk;
    fnew = f(xnew);

    %find alpha with backtracking
    bt = 0;
    while bt < btmax && fnew > farmijo(fk, alpha, gradfk, pk)
        %while Armijo condition is not satisfied, reduce the value of
            alpha
        alpha = rho * alpha;
        xnew = xk + alpha * pk;
        fnew = f(xnew);
        bt = bt + 1;
    end

    xk = xnew;
    fk = f(xk);
    gradfk = gradf(xk);
    gradfk=gradfk';
    normgradfk = norm(gradfk);
    hessfk = Hf(xk);
    k = k+1;

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

% remove the extra columns
xseq = xseq (:, 1:k);
btseq = btseq (1:k);
end

%% 2) TRUNCATED NEWTON METHOD WITHOUT PRECONDITIONING

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

% INPUTS :
```

```
% x0 = n- dimensional column vector ;
% f = function handle for the function R^n->R;
% gradf = function handle for the gradient of f;
% Hessf = function handle for the Hessian of f;
% kmax = maximum number of outer iterations ;
% tolgrad = value used as stopping criterion for the norm of the
% gradient ;
% c1 = factor of the Armijo condition that must be a scalar in (0 ,1);
% rho = fixed factor used for reducing alpha0 ;
% btmax = maximum number of steps for updating alpha during the
% backtracking strategy .
%
% OUTPUTS :
% xk = the last x computed by the function ;
% fk = the value f(xk);
% gradfk_norm = value of the norm of gradf (xk)
% k = index of the last iteration performed
% xseq = n-by -k matrix where the columns are the xk computed during
    the
% iterations
% btseq = 1-by -k vector where elements are the number of backtracking
% iterations at each optimization step .


% function handle for the armijo condition
%farmijo = @(fk , alpha , gradfk , pk)fk + c1 * alpha * gradfk' * pk;
    %QUI HO MESSO I PUNTI, AURO NO

% Function handle for the armijo condition%
farmijo = @(fk, alpha, gradfk, pk) ...
      fk + c1 * alpha * (-sum((gradfk*pk).^2, 1));

% initializations
xseq = zeros(length(x0), kmax );
btseq = zeros(1, kmax );

xk = x0;
fk = f(xk);
gradfk = gradf (xk);
gradfk=gradfk';
hessfk = Hessf (xk);

gradfk_norm = norm ( gradfk );

k = 0; % counter

%the second one is the stopping criterion
while k < kmax && gradfk_norm > tolgrad
    z = zeros ( length (x0) ,1);
    etak = min (0.5 , ( norm ( gradfk )));
    pk = cg_curvtrun_newt ( hessfk , -gradfk, z, etak ); % cg +
        curvature condition with no preconditioning

    % Backtracking with armijo condition
    % start with alpha = 1
    alpha = 1;

    % the new candidate xk
```

```
    xnew = xk + alpha * pk;
    %the value of f in the candidate new xk
    fnew = f(xnew);

    bt = 0;

    % Backtracking strategy :
    % 2nd condition holds if the Armijo condition is not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, gradfk, pk)

        % reduce alpha
        alpha = rho * alpha ;

        % update xnew and fnew w.r.t. the reduced alpha
        xnew = xk + alpha * pk;
        fnew = f( xnew );

        % increase the counter by one
        bt = bt + 1;

    end

    % update the values
    xk = xnew ;
    fk = fnew ;
    gradfk = gradf(xk);
    gradfk=gradfk';
    gradfk_norm = norm( gradfk );
    hessfk = Hessf(xk);

    % increase the step of the outer cycle by one
    k = k + 1;

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

% remove the extra columns
xseq = xseq (:, 1:k);
btseq = btseq (1:k);

end

%% 3) CONJUGATE GRADIENT METHOD WITH NEGATIVE CURVATURE CONDITION

function xk = cg_curvtrun_newt ( A, b, x0 , tol)

% INPUTS :
% A = matrix of the coefficients of the lienar system
% b = vector of constants
% x0 = n- dimensional column vector
% tol = tolerance on the norm of the relative residual

% OUTPUT :
% xk = the last x computed by the function ;
%b=b';
```

```
% initializations
xk = x0;
b=b';
rk = b - A * xk; % residual
pk = rk;
norm_b = norm (b);
relres = norm (rk) / norm_b ; % norm of the relative residual
kmax = 100;
k = 0;
% compute cg iterations until the relative residual gets smaller
% than the tolerance or you reach the max number of iterations

while relres > tol && k < kmax
    if pk' * A * pk <= 0 %if the negative curvature condition is
        satisfied
        if k == 0 % if it   s   the first iteration , compute the new xk
            and stop
            zk = A * pk;
            alphak = (rk' * pk )/ (pk' * zk);
            xk = xk + alphak * pk;
        end
        % stop computing xk and return it
        break
    end
    % continue the cycle if not
    zk = A * pk;
    alphak = (rk' * pk )/ (pk' * zk);
    xk = xk + alphak * pk;
    rk = rk - alphak * zk;
    betak = -(rk' * zk) / (pk' * zk);
    pk = rk + betak * pk;

    relres = norm (rk) / norm_b ;

    k = k + 1;
end
end

%% 4) TRUNCATED NEWTON METHOD WITH PRECONDITIONING

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

% function that performs the modified newton method ,
% implementing the backtracking strategy with preconditioning on the cg
% iterations .

% INPUTS :
% x0 = n- dimensional column vector ;
% f = function handle for the function R^n->R;
% gradf = function handle for the gradient of f;
% Hessf = function handle for the Hessian of f;
% kmax = maximum number of outer iterations ;
% tolgrad = value used as stopping criterion for the norm of the
% gradient ;
% c1 = factor of the Armijo condition that must be a scalar in (0 ,1);
% rho = fixed factor used for reducing alpha0 ;
```

```
% btmax = maximum number of steps for updating alpha during the
% backtracking strategy .
%
% OUTPUTS :
% xk = the last x computed by the function ;
% fk = the value f(xk);
% gradfk_norm = value of the norm of gradf (xk)
% k = index of the last iteration performed
% xseq = n-by -k matrix where the columns are the xk computed during
    the
% iterations
% btseq = 1-by -k vector where elements are the number of backtracking
% iterations at each optimization step .

% function handle for the armijo condition
%farmijo = @(fk , alpha , gradfk , pk) fk + c1 * alpha * gradfk' * pk;

% Function handle for the armijo condition%
farmijo = @(fk, alpha, gradfk, pk) ...
      fk + c1 * alpha * (-sum((gradfk*pk).^2, 1));


% initializations
xseq = zeros( length(x0), kmax );
btseq = zeros(1, kmax );

xk = x0;
fk = f(xk);
gradfk = gradf (xk);
gradfk=gradfk';
hessfk = Hessf (xk);

gradfk_norm = norm ( gradfk );

k = 0; % counter

%the second one is the stopping criterion
while k < kmax && gradfk_norm > tolgrad
    z = zeros( length (x0) ,1);
    etak = min (0.5 , gradfk_norm);
    pk = cg_curvtrun_newt_pre( hessfk , -gradfk, z, etak ); %cg +
       negative curvature condition with preconditioning

    % Backtracking with armijo condition
    % start with alpha = 1
    alpha = 1;

    % the new candidate xk
    xnew = xk + alpha * pk;



    %the value of f in the candidate new xk
    fnew = f( xnew );

    bt = 0;

    % Backtracking strategy :
```

```
336     % 2nd condition holds if the Armijo condition is not satisfied
337
338     while bt < btmax && fnew > farmijo (fk , alpha , gradfk , pk)
339
340         % reduce alpha
341         alpha = rho * alpha ;
342
343         % update xnew and fnew w.r.t. the reduced alpha
344         xnew = xk + alpha * pk;
345         fnew = f( xnew );
346
347         % increase the counter by one
348         bt = bt + 1;
349     end
350
351     % update the values
352     xk = xnew ;
353     fk = fnew ;
354     gradfk = gradf (xk);
355     gradfk=gradfk';
356     gradfk_norm = norm ( gradf(xk) );
357     hessfk = Hessf (xk);
358
359     % increase the step of the outer cycle by one
360     k = k + 1;
361
362     % store current xk in xseq
363     xseq (:, k) = xk;
364     % store bt iterations in btseq
365     btseq (k) = bt;
366 end
367
368 % remove the extra columns
369 xseq = xseq (:, 1:k);
370 btseq = btseq (1:k);
371
372 end
373
374 %% 5) PRECONDITIONED CONJUGATE GRADIENT METHOD WITH NEGATIVE CURVATURE
        CONDITION
375
376 function xk = cg_curvtrun_newt_pre( A, b, x0 , tol)
377
378 % INPUTS :
379 % A = matrix of the coefficients of the lienar system
380 % b = vector of constants
381 % x0 = n- dimensional column vector ;.
382 % tol = tolerance on the norm of the relative residual
383
384
385 % OUTPUT :
386 % xk = the last x computed by the function ;
387
388
389 % preconditioning operations
390 %either use the ichol when possible or diag ( diag (A)) as
        preconditioner
391 try
```

```
392     L = ichol (sparse(A));
393 catch exception
394     L = diag ( diag(A));
395 end
396 %L = diag ( diag (A));
397 L=L'*L;
398 b=b';
399 L_i = L \eye( size (L ,1) ); % compute the inverse
400 A = L_i * A; %new matrix
401 b = L_i * b; % new vector
402
403 % initilizations
404 xk = x0;
405 rk = b - A * xk;
406 pk = rk;
407 norm_b = norm (b);
408 relres = norm (rk) / norm_b ; % relative residual
409 kmax = 1000;
410 k = 0;
411
412 % compute pcg iterations until the relative residual gets smaller
413 % than the tolerance or you reach the max number of iterations
414
415 while relres > tol && k < kmax
416     if pk' * A * pk <= 0 %if the negative curvature condition is
        satisfied
417         if k == 0 % if its the first iteration , compute the new xk and
              stop
418             zk = A * pk;
419             alphak = (rk' * pk )/ (pk' * zk);
420             xk = xk + alphak * pk;
421         end
422         break
423         % stop computing xk and return it
424     end
425     zk = A * pk;
426     alphak = (rk' * pk )/ (pk' * zk);
427     xk = xk + alphak * pk;
428     rk = rk - alphak * zk;
429     betak = -(rk' * zk) / (rk' * zk);
430     pk = rk + betak * pk;
431     relres = norm (rk) / norm_b ;
432     k = k + 1;
433 end
434
435 end
436
437 %% 6) EXPERIMENTAL RATE OF CONVERGENCE WITH MINIMUN POINT COMPUTATION
438
439 function [p,pseq] = exp_conv_rate_min(x_min, k, xseq)
440 % function to compute the experimental rate of convergence of a method,
441 % given:
442 % x_min the minimum of the function
443 % k = number of iterations of the method until convergence
444 % xseq = sequence of points produced by the iterative method
445 % The function returns:
446 % pseq = the sequence of experimental rates computed for 3 consecutive
447 %   values of the sequence x(k-1), x(k), x(k+1)
```

```
% p = the median of the convergence rate vector
if k<=100
    pseq=zeros(k,1);
    for i=2:length(xseq)-1
        ekm=norm(xseq(:,i-1)-x_min);
        ek=norm(xseq(:,i)-x_min);
        ekp=norm(xseq(:,i+1)-x_min);
        pseq(i-1)= log( ekp/ek ) / log( ek/ekm ) ;
    end
    p=median(pseq)
else
    xseq_new=xseq(:,length(xseq)-100:length(xseq));
    pseq=zeros(100,1);
    for i=2:length(xseq_new)-1
        ekm=norm(xseq_new(:,i-1)-x_min);
        ek=norm(xseq_new(:,i)-x_min);
        ekp=norm(xseq_new(:,i+1)-x_min);
        pseq(i-1)= log( ekp/ek) / log( ek/ekm );
    end
    p=median(pseq)
end

end

%% 7) EXPERIMENTAL RATE OF CONVERGENCE WITHOUT MINIMUN POINT
   COMPUTATION

function [p,pseq] = exp_conv_rate(k, xseq)
% function to compute the experimental rate of convergence of a method,
% given:
% x_min the minimum of the function
% k = number of iterations of the method until convergence
% xseq = sequence of points produced by the iterative method
% The function returns:
% pseq = the sequence of experimental rates computed for 3 consecutive
%   values of the sequence x(k-1), x(k), x(k+1)
% p = the median of the convergence rate vector
if k<=100
    pseq=zeros(k,1);
    for i=3:size(xseq,2)-1
        ek=norm( xseq(:,i)-xseq(:,i-1) ); % xk-x(k-1)
        ekm=norm( xseq(:, i-1)-xseq(:,i-2) ); %x(k-1)-x(k-2)
        ekp=norm(xseq(:,i+1)-xseq(:,i)); %x(k+1)-x(k)
        pseq(i-2)= log( ekp/ek ) / log( ek/ekm ) ;
    end
    p=mean(pseq)
else
    xseq_new=xseq(:,size(xseq,2)-100:size(xseq,2));
    pseq=zeros(100,1);
    for i=3:size(xseq_new,2)-1
        ek=norm( xseq_new(:,i)-xseq_new(:,i-1) );
        ekm=norm( xseq_new(:,i-1)-xseq_new(:,i-2) );
        ekp=norm( xseq_new(:,i+1)-xseq_new(:,i) );
        pseq(i-2)= log( ekp/ek ) / log( ek/ekm );
    end
    p=mean(pseq)
end

```

```
505 end
```

Listing 2: Definition of the functions used in the main code