Numerical optimization for large scale problems and Stochastic Optimization Unconstrained Optimization Methods Analysis

Mustafa Omer Mohammed Elamin Elshaigi
s289815@studenti.polito.it

Farzad Imanpour Sardroudi
s289265@studenti.polito.it

Abstract

*In this report, we conduct a detailed analysis on the characteristics of some of the numerical optimization methods tested to the three problems from Test Problems for Unconstrained Optimization.More specifically, we compared the **Steepest Decent**, **Newton Method** and **Inexact Newton Method** applied to Problem 1 [Chained Rosenbrock function] , Problem 25 [Extended Rosenbrock function] , Problem 76 and problem 81. We tested the first problem in 2-D space and the remaining three problems in N-D space, where N = [10^3,10^4,10^5]. For each problem we calculated analytically the gradient and the hessian in order to apply the chosen methods. We optimize successfully each function with at least one method.*

1. Introduction

We aim to minimize the three functions, introduced in the abstract, setting some common parameters:

- gradient norm tolerance for the stopping criteria equals to 10^{-8};
- maximum number of iterations for the backtracking strategy equals to 50;
- the factor ρ used in the backtracking strategy for decreasing the step_length α equals to 0.5;
- the term c used in the Armijo condition equals to 10^{-4};
- the initial step_length α_0 equals to 1.

We fixed the notations for the inputs and outputs used in the script and the analysis to standardize the notations:

- **n** : represents the dimension of the problem .
- **k** : represents the number of iterations taking by each method to reach the solution. [when **k** == K_{max} : the method failed to achieve the desired tolerance in the specified maximum number of iteration]
- ρ the factor used in the backtracking strategy for decreasing the step_length α equals to 0.5;
- **c** : the term used in the Armijo condition equals to 10^{-4};
- α_0 : the initial step_length equals to 1.

We analyzed the performances of each method for all the problems, in terms of the speed of convergence and the computational cost of each iteration, the accuracy of the solution compared to the actual minimum of the function. We decide to compute the gradient and the hessian analytically, because using finite differences would increase the computational cost of each single iteration.Except for the Problem 1 [Chained Rosenbrock function] in 2D, we applied the finite difference approximation, we analyzed how accurate is the approximation compared to using the analytical gradient and hessian of the function. the A further consideration can be made on the implementation of the hessian matrix, which actually is a large matrix in $\mathbb{R}^{n\times n}$, computing this matrix does not apply a large penalty to the Newton and Inexact Newton methods , also both the computation of the value function and the gradient depend on the dimension of the problem linearly.

The functions definitions can be found in Appendix [A]. The Matlab codes can be found in [B].

1.1. Steepest descent with backtracking

The descent direction P_k, taken at each iteration, is the negated of the gradient. This method is an extension of the gradient method (steepest descent) where the optimal step_length α is computed by taking an iterative approach, performing a backtracking line search strategy At each iteration, starting from a value α_0 we seek for a step-length which guarantees a sufficient decrease of the function's value checking if the Armijo condition is satisfied. We expect a higher computational cost for one iteration with respect to the classic steepest descent method, due to the iterative approach that required some inner iterations to be performed in order to find the optimal step_length. Nevertheless, we expect that the method reaches the desired tolerance with less outer iterations than without backtracking. Furthermore, we expect the cost of a single iteration to be much faster than a Newton iteration, since computing the decent direction doesn't require the computation of the Hessian matrix , nor solving a complex linear system. In most cases we expect this method to have a low rate of convergence with respect to Newton and Inexact Newton methods as it uses only the gradient information to compute the decent direction. The implementation of this method is reported in appendix [B.0.1]

1.2. NEWTON METHOD WITH BACK TRACKING

The descent direction P_k taken at iteration k is computed solving the system :

$$H_f(x_k)p = -\nabla f(x_k) \tag{1}$$

Where :

- $H_f(x_k)$: the hessian matrix computed at x_K
- $-\nabla$: the negated gradient computed at the same point x_K .
- P : the decent direction we are solving for.

We used the generalized minimal residual method gmres algorithm to exploit the sparsity of the hessian. We choose this algorithm since it performs pretty well on all considered problems, reaching the convergence in very few inner iterations for each outer one. Note that . Under proper assumptions ($H_f(x_k)$) PD 'is Positive Definite',α= 1, x_0 is a good starting point) , We expect the Newton method to have a very fast (quadratic) rate of convergence. However, These assumptions are rarely to be true for complex functions. In fact, We will see that in problem 81 this is not true , so we cannot apply this method.
Last consideration we should take into account,this method can be computationally more expensive than Steepest Descent (computation/storage of $H_f(x_k)$ and computation of P_k solving [1]
The implementation is reported in appendix [B.0.2]

1.3. INEXACT NEWTON WITH BACK TRACKING

As mentioned section [1.2], Solving the linear system in [1] can be quite expensive,Therefore, in order to save computational efforts, instead of computing P_K^N, we compute an 'approximated descent direction' $P_K^{IN} \approx P_K^N$ using an iterative method to solve [1]. We used Preconditioned conjugate gradient (PCG). at the k_{th} step the iterative solver returns a vector P_K^* with residual such that :

$$\|H_f(x_k)p_k^* + \nabla f(x_k)\| \geq \epsilon$$

instead of using a small and fixed ϵ,the Inexact Newton consists of using an adaptive tolerance :

$$\epsilon_K := \eta_k \|\nabla f(x_k)\|$$

where :

- ϵ_K : The adaptive tolerance
- η_k : The forcing Term
- $\|\nabla f(x_k)\|$: The gradient norm computed at x_k

that is relatively large during the first optimization steps; i.e., we use a large tolerance when we assume to be far from the optimum.
Under local and regularity assumptions, we used tested three forcing term sequences as follows:

- η_k = 0.5 fixed (Linear convergence)
- η_k = min(0.5 , $\sqrt{\|\nabla f(x_k)\|}$) $\xrightarrow{k \to \infty}$0 (Super Linear Convergence)
- η_k = min(0.5 , $\sqrt{\|\nabla f(x_k)\|}$) $\in$ $\mathcal{O}(\|\nabla f(x_k)\|)$ (Quadratic convergence)

We must note that the rate of convergence depends also on the iterative solver. Since we are using an iterative method, not always the solver is able to meet the tolerance within the max iteration as we will notice.
The implementation is reported in appendix [B.0.3]

2. Experiments and Results

In this section we discuss the results we obtained applying the three methods. We will analyze each problem individually because the application of the method is highly dependant on the characteristics of the function we tend to minimize.

2.1. Problem 1 [Chained Rosenbrock function] in 2D

The function definition is in .It is trivial that minimal solution for this function is x = (1, 1) and the Actual minimum is 0. The function is defined in [1.3]

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

In our problem we had two initial points we will refer to as :x_{01} $[-1.2, 1]^T$ and x_{02} $[1.2, 1.2]^T$. First of all, it is crucial to understand the best as possible the behaviour of f(x) in 3-D. The following plots show the function and each method behavior at each iteration taking step towards the minimum .
Note: we reported only samples of the graphs due to space limitation, the numerical results are reported in table [**??**]

From the plots and the table of results we can make some general conclusions :

- **The performance of the methods** : It's obvious that due to the low dimensionality of the problem, We expect all algorithms will get to a good optimum , We noticed that the most accurate method is Newton with finite difference, although, we used an approximated gradient and hessian, we still obtained a very good results. In our case, taking the approximated gradient and hessian is not the best option, due to the fact that both can be computed analytically .
  Since the results for all methods has an almost negligible margin of accuracy difference, we compared the execution time. The Newton method had the minimum execution time with 3.08 ms with starting point x_{02} and 3.65 ms with the x_{01}.
- **The relevance of the starting point** x_0: We can notice that the starting point is very crucial for all the methods, starting far from the minimum will require taking more steps, or the first step_length α_0 should be relatively bigger and some other consideration should be taking into account. It is clear the effect of this on the execution time. From the results x_{02} $[1.2, 1.2]^T$ is a better starting points for all methods.e.g in the Inexact Newton starting from x_{02}, it took only 70% of the time to reach a solution with a minimum $f(x^*) \approx 3.01e^{-13}$ more accurate solution than starting from x_{01} $f(x^*) \approx 3.71e^{-11}$
- **Newton VS Inexact Newton**: both methods performed well in terms of accuracy, in some cases starting from x_{02} obtained more accurate results, but we consider it negligible when compared to the very high cost of computation and execution time, it took $\approx$ 10x the execution time of the newton method, This is probably due to ~~the inner iteration performed to find an approximation of the optimal direction p_k and~~ the choice of the linear solver.

2.2. Problem 25 - extended rosenbrock function in N_D

This problem is an extension of 'Rosenbrock' in the N_D, The function and gradient implementation can be found in appendix [12] and [A.1.1]. The code implementation of applying all the methods and visualizing the results can be found in [13] .
We compared all the methods and reported the results in table [2] . From the results we concluded the following :

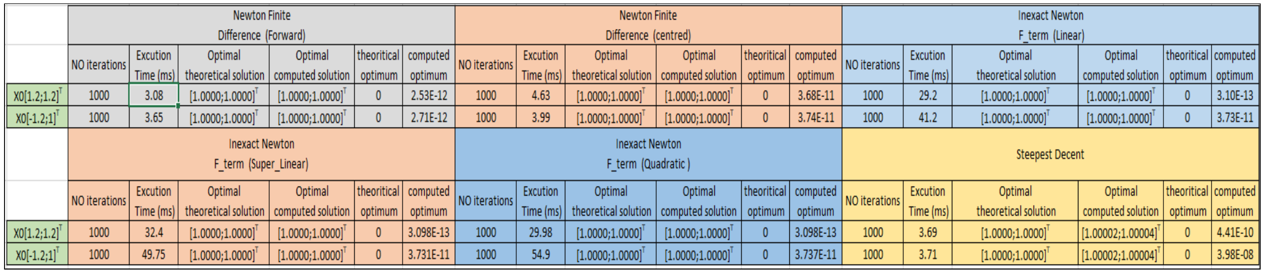
- Newton and Inexact newton performed very well and reached a solution very fast , both methods scaled very well with almost 1 iteration more for every increase in order of magnitude of the dimensional space.However, when the dimension space is getting larger (10^5), we can see that newton is starting to be more Superior in terms of execution time and accuracy of the solution.
- The steepest decent failed to find a solution in for all dimensions in 10k iterations.We expect it to need double the time to reach a local minimum. Which makes it not convenient for this problem.considering the very high execution time, applying this method for this problem in higher dimensional space can be unfeasible .
- From fig [4] the norm of the gradient oscillates in all methods in the first iterations. This is due to the fact that the Armijo condition, used in the backtracking strategy, finds a step-length which guarantees only a decrease for the function value, not for the gradient norm, so this behaviour was expected. Though it's more clear in the steepest decent case.

2.3. Problem 76 function in N_D

The function and gradient implementation can be found in appendix [14] and [A.2.1]. The code implementation of applying all the methods and visualizing the results is the same the only change is in the function handles for the function, gradient, and hessian implementations .
We compared all the methods and reported the results in table [3] .

From the results we concluded the following :

- The problem is totally in different from characteristics the previous one, in fact, all the methods performed very well with minor differences in terms of execution time and accuracy (all of the methods reached numerical zero) .Unlike the first problem, the steepest decent convergence forcing term overcome the other methods with significant margin in terms of time with small reduction in accuracy. This is due the factor α is always 1 and the cost for computing the descent direction is quite low.
- To investigate the first point more we plotted the number of inner iterations of backtracking for newton method in [2], it needed 3 iteration of backtracking twice to find optimal step length. while for the gradient method with α_0 = 1 does not perform the backtracking iterations since the Armijo condition is satisfied without decreasing the step-length. It guaranteed a sufficient decrease in the decent direction but still and reached a solution very fast but not the optimal.
- In lower dimension, the Inexact newton with super_linear rate of convergence performed the best in terms of execution time. Exploiting the trade off between the accuracy (strict forcing term quadratic convergence) and the execution time (less strict forcing term linear convergence rate).



	Newton Finite Difference (Forward)						Newton Finite Difference (centred)						Inexact Newton F_term (Linear)					
	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum
$X0[1.2;1.2]^T$	1000	3.08	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	2.53E-12	1000	4.63	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.68E-11	1000	29.2	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.10E-13
$X0[-1.2;1]^T$	1000	3.65	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	2.71E-12	1000	3.99	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.74E-11	1000	41.2	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.73E-11

	Inexact Newton F_term (Super_Linear)						Inexact Newton F_term (Quadratic)						Steepest Decent					
	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum	NO iterations	Excution Time (ms)	Optimal theoretical solution	Optimal computed solution	theoritical optimum	computed optimum
$X0[1.2;1.2]^T$	1000	32.4	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.098E-13	1000	29.98	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.098E-13	1000	3.69	$[1.0000;1.0000]^T$	$[1.00002;1.00004]^T$	0	4.41E-10
$X0[-1.2;1]^T$	1000	49.75	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.731E-11	1000	54.9	$[1.0000;1.0000]^T$	$[1.0000;1.0000]^T$	0	3.737E-11	1000	3.71	$[1.0000;1.0000]^T$	$[1.00002;1.00004]^T$	0	3.98E-08

Figure 1: Results of Rosenbrock 2D

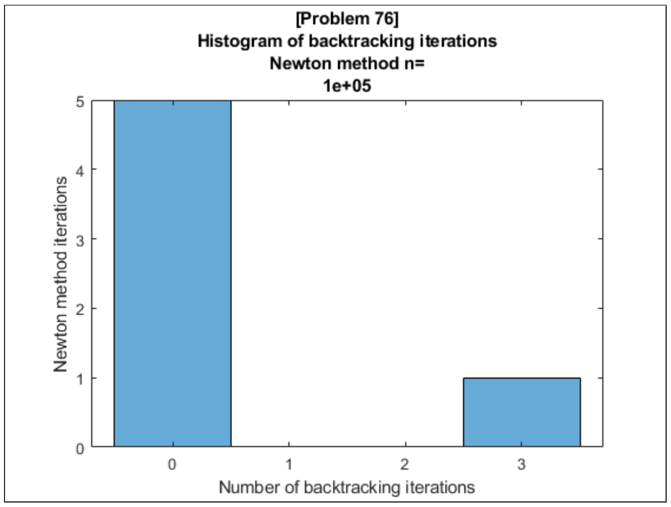


Figure 2: Backtracking iteration by ~~gmres~~ for newton in 10^5 dimensions

2.4. Problem 81 function in N_D

The function and gradient implementation can be found in appendix [15] and [A.3.1]. The code implementation of applying all the methods and visualizing the results can be found in [16] .
We compared steepest decent for 10^d Dimensional space where d $\in$ [3,4,5] and reported the results in table [1] .

Space Dimension	Iteration (k)	elapsed_times (sec)	grad_norm _last	f(Xk) _last
1k	42	0.014	5.9623E-09	1.0530E-17
10k	44	0.113	7.6251E-09	2.3471E-17
100k	42	0.973	6.1621E-09	1.4033E-17

Table 1: Problem 81 results

From the results we concluded the following :

- As we mentioned in section [1.2]. for both Newton and Inexact Newton p_k is a descent direction only if $H_f(x_k)$ is PD (Positive Definite), To prove that the hessian matrix is not always SPD we add to the function HessFx this check: at the end of the computation, we calculate the trace of the hessian (i.e. the sum of the elements on its main diagonal). Since this value is equal to the sum of the eigenvalues, if it is non-positive there should be some negative or zero eigenvalues, therefore the matrix is not positive definite. For these reasons we applied only Steepest Decent method for this problem.

- We tried to work with a "corrected" Hessian matrix, when $H_f(x_k)$ is non-PD; i.e., we tried to use SPD matrix $B_k := H_f(x_k) + Correction$. We tried to approximate the hessian using the Incomplete Cholesky factorization. But suffered from broken down.

- For previously mentioned reasons, we report only the results obtained with steepest descent method, which performs pretty well. In figure [3] we show the trend for both of the value function and the gradient norm, it is quite fast thanks to the high step-length regularized when necessary by the backtracking strategy. All the results are summarized in table [1].
the number of iterations k needed to reach the gradient norm tolerance of 10^{-8} is quite low and the overall time is pretty good of only 42 iterations.

3. Conclusion

We managed to optimize and find a minimum for all the problems with at least one method. We conclude that not always the theoretical assumptions can be true and the behaviour of the methods is dependent on several factors including:

- The Function characteristics : we know that some assumption need the function to have specific characteristics, i.e. Newton need the hessian of f to be PD in order to locally approximates f (sufficiently smooth) with a quadratic model (convex model for f around x_k) which is not always satisfied for complex non_linear functions; as we have encountered for problem 81.

- The importance of the starting point & the initial step_length : we examined the relevance of the starting point (x_0) ; in sec[2.1]; to the execution time of the methods, and the relation to the initial step_length (α_0). When (x_0) is far from the minimum we should take a big step in the decent direction. when we start near we don't want to take a big step in order not to over-shoot. In practice we assume (x_0) to be far from the minimum. i.e in Newton we take (α_0)= 1 and we use backtracking strategy to regularize the step_length when necessary during each of the following iterations.

- overall outcomes : from problem 25, we saw obtained results mostly coherent to the theoretical properties of all the methods. But noticed that the function was very easy to optimize.In problem 76, all methods reached a solution in few iterations , but due to the simplicity of computing the gradient, and the function value and gradient norm were monotonically decreasing, the steepest decent method was faster in reaching a minimum and we saw that for this specific problem, it could scale very well. In the third problem we face a hessian matrix not positive definite, so the only choice left is the gradient method, which in this case performs efficiently.

A. Appendix A
Functions, Gradient vectors

A.1. Problem 1 [Chained Rosenbrock function]

A.1.1 Function

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

$$f(x) = \begin{cases} 100(x_k^2 - x_{k+1})^2 \text{ if mod(k,2) = 1} \\ (x_{k-1} - 1)^2 \text{ if mod(k,2) = 0} \end{cases}$$

The starting point $x_0 \in \mathbb{R}^n$ is:

$$x_k^0 = \begin{cases} -1.2 \text{ if mod(k,2) = 1} \\ 1 \text{ if mod(k,2) = 0} \end{cases}$$

A.2. Problem 76

A.2.1 Function

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

$$f(x) = \begin{cases} (x_k - \frac{x_{k+1}^2}{10}) \text{ if } \quad 1 \le k < n \\ (x_k - \frac{x_1^k}{10})^2 \text{ if k = n} \end{cases}$$

The starting point $x_0 \in \mathbb{R}^n$ is:

$$x_k^0 = 2, \text{ if } 1 \le k < n$$

A.3. Problem 81

A.3.1 Function

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

$$f(x) = \begin{cases} (x_k^2 - 2) \text{ if } K = 1 \\ (x_{k-1}^2 - \log X_K - 1)^2 \text{ if} 1 < k \le n \end{cases}$$

The starting point $x_0 \in \mathbb{R}^n$ is:

$$x_k^0 = 0.5, k \ge 1$$

B. Appendix B
Matlab codes

B.0.1 STEEPEST DECENT WITH BACKTRACKING CODE IMPLEMENTATION

B.0.2 NEWTON WITH BACK TRACKING CODE IMPLEMENTATION

B.0.3 INEXACT NEWTON WITH BACK TRACKING CODE IMPLEMENTATION

Space Dimension	Method	Iteration (k)	elapsed_times (sec)	grad_norm_last	f(Xk)_last
1k	IN-Newton linear	21	0.031692	5.0021E-09	9.3671E-19
	IN-Newton super-inear	21	0.03021	5.0021E-09	9.3671E-19
	IN-Newton quadratic”	21	0.03068	5.0021E-09	9.3671E-19
	Newton exact	21	0.011951	5.0028E-09	9.3585E-19
	Steepest Decent	10000	6.72	2.5858E-04	6.7744E-08
10k	IN-Newton linear	22	0.175439	1.0049E-12	2.0226E-24
	IN-Newton super-inear	22	0.128115	1.0049E-12	2.0226E-24
	IN-Newton quadratic”	22	0.155903	1.0049E-12	2.0226E-24
	Newton exact	22	0.109098	2.0105E-12	3.3274E-29
	Steepest Decent	10000	93.2166	8.1771E-04	6.7744E-07
100k	IN-Newton linear	22	1.257572	1.4211E-11	1.2326E-25
	IN-Newton super-inear	22	1.124522	1.4211E-11	1.2326E-25
	IN-Newton quadratic”	22	1.148544	1.4211E-11	1.2326E-25
	Newton exact	23	0.8256239	3.1782E-12	2.0225E-23
	Steepest Decent	10000	787.8684	2.5858E-03	6.7744E-06

Table 2: Problem 25 results

Space Dimension	Method	Iteration (k)	elapsed_times (sec)	grad_norm_last	f(Xk)_last
1k	IN-Newton linear	6	0.022469	2.3687E-18	2.8054E-36
	IN-Newton super-inear	6	0.0078012	2.3687E-18	2.8054E-36
	IN-Newton quadratic”	6	0.010668	2.3687E-18	2.8054E-36
	Newton exact	7	0.01206	2.0539E-18	2.1092E-36
	Steepest Decent	7	0.0026824	1.8313E-15	1.6768E-30
10k	IN-Newton linear	6	0.03308	6.5881E-18	2.1702E-35
	IN-Newton super-inear	6	0.035787	6.5881E-18	2.1702E-35
	IN-Newton quadratic”	6	0.033025	6.5881E-18	2.1702E-35
	Newton exact	7	0.022988	6.4949E-18	2.1092E-35
	Steepest Decent	7	0.011428	5.7910E-15	1.6768E-29
100k	IN-Newton linear	6	0.32024	2.0568E-17	2.1152E-34
	IN-Newton super-inear	6	0.2456	2.0568E-17	2.1152E-34
	IN-Newton quadratic”	6	0.23262	2.0568E-17	2.1152E-34
	Newton exact	7	0.14586	2.0539E-17	2.1092E-34
	Steepest Decent	7	0.083735	1.8313E-14	1.6768E-28

Table 3: Problem 76 results

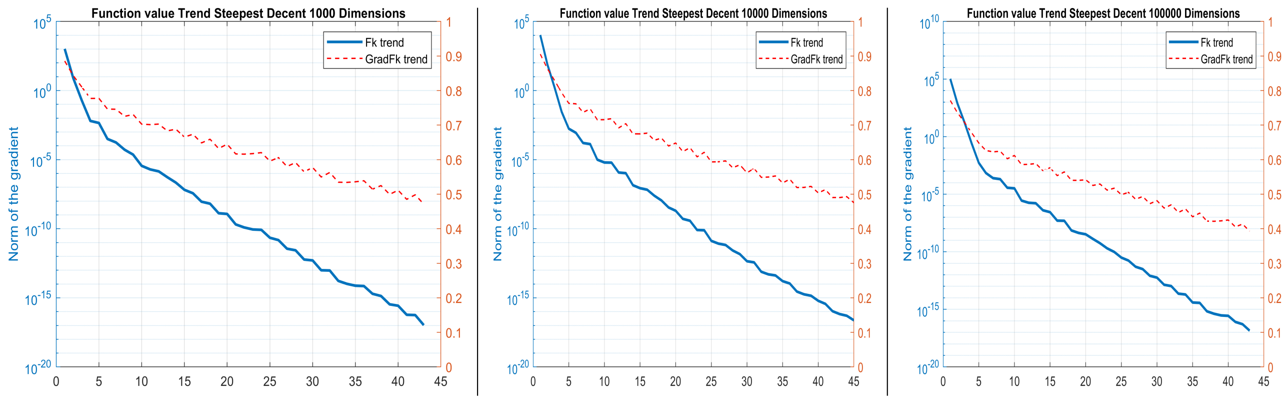


Figure 3: problem 81 10^3,10^4,10^5 dimensions results

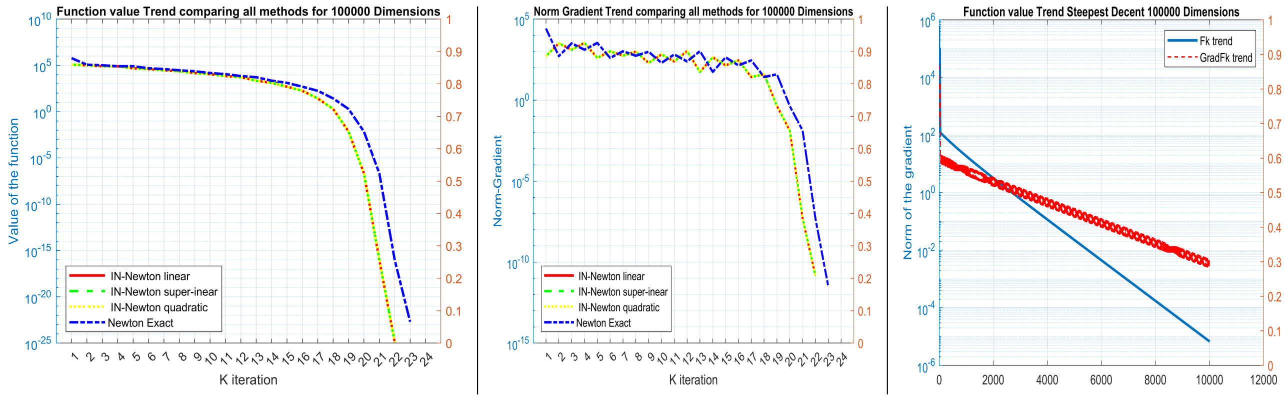


Figure 4: problem 25 10^5 dimensions results

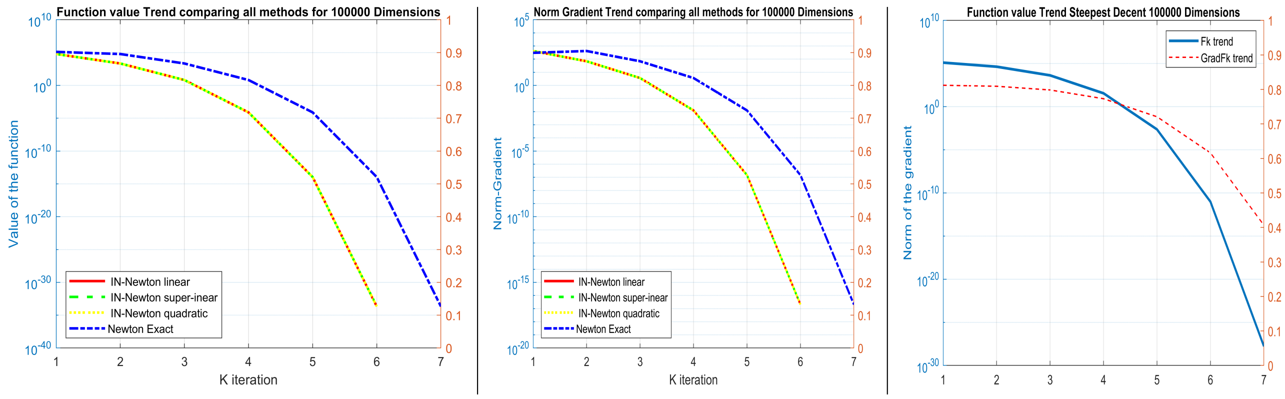


Figure 5: problem 76 10^5 dimensions results

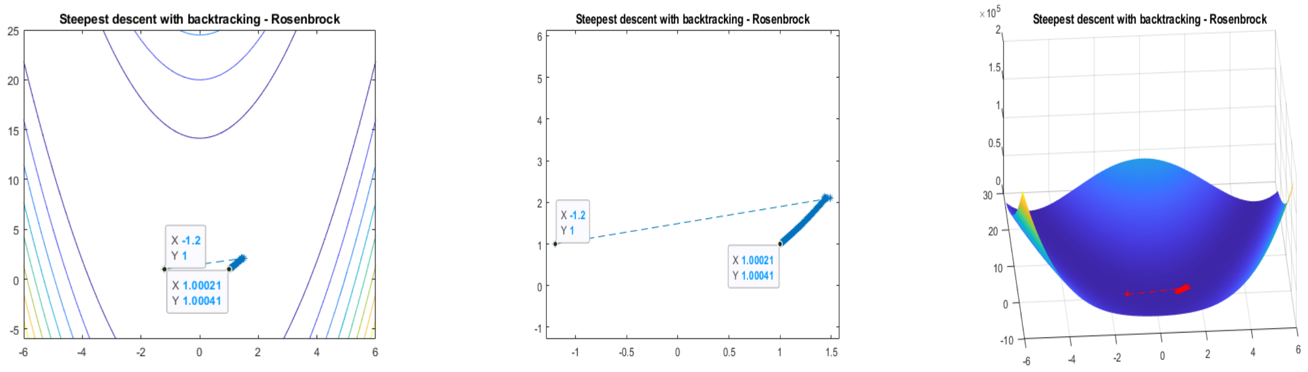


fig 1.1 STEEPEST DECENT X0 = [-1.2, 1]

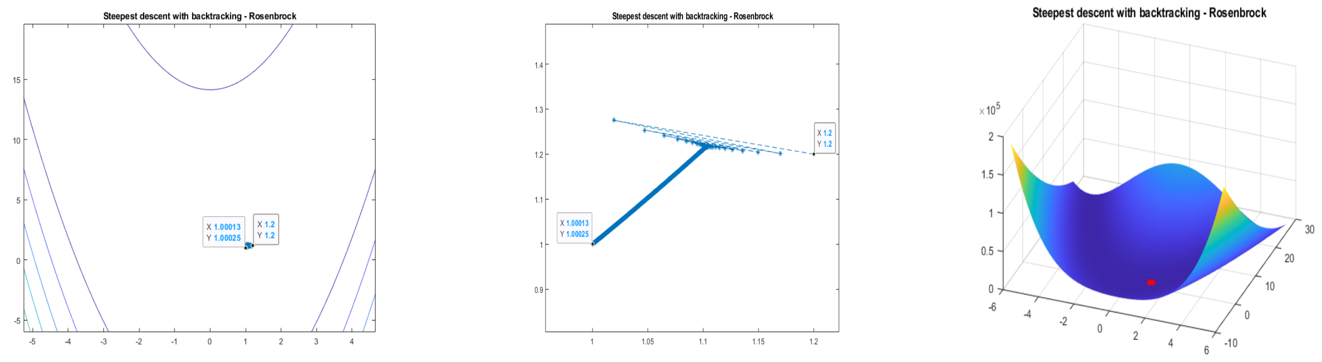


Figure 6: STEEPEST DECENT X0 = [1.2, 1.2]

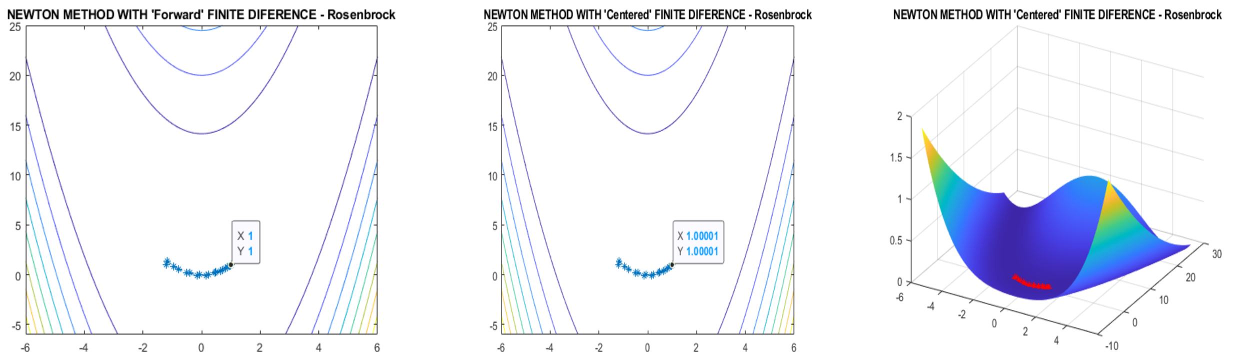


fig 1.1 NEWTON X0 = [-1.2, 1]

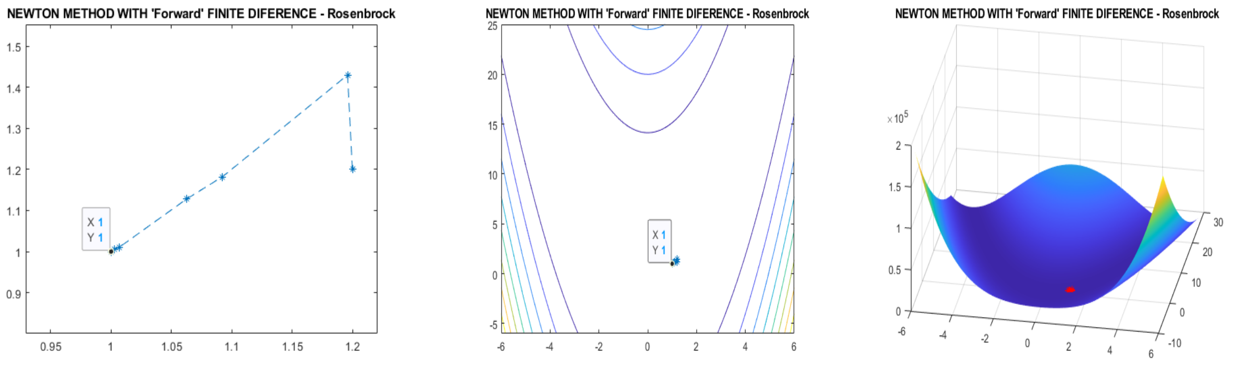


Figure 7: NEWTON X0 = [1.2, 1.2]

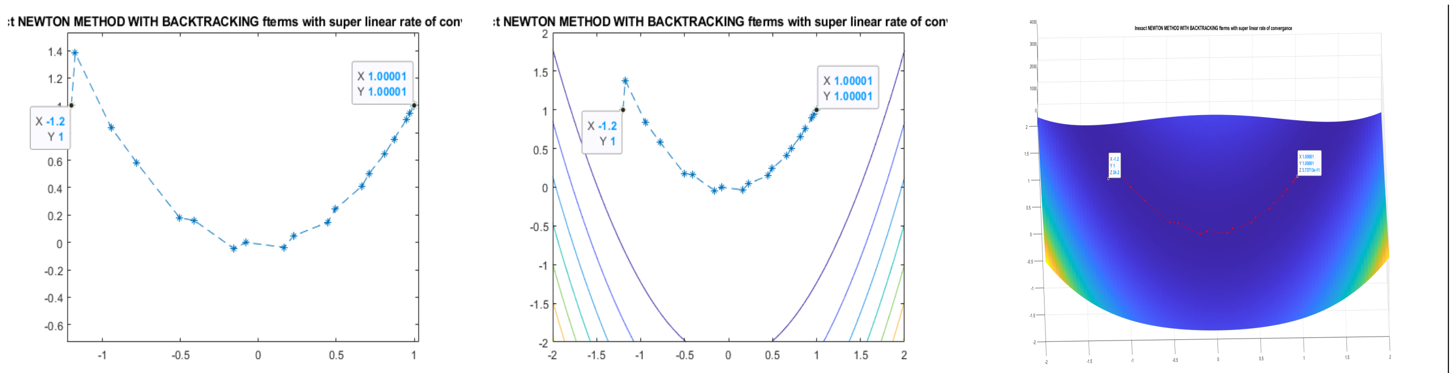


fig 1.1 Inexact NEWTON X0 = [-1.2, 1]

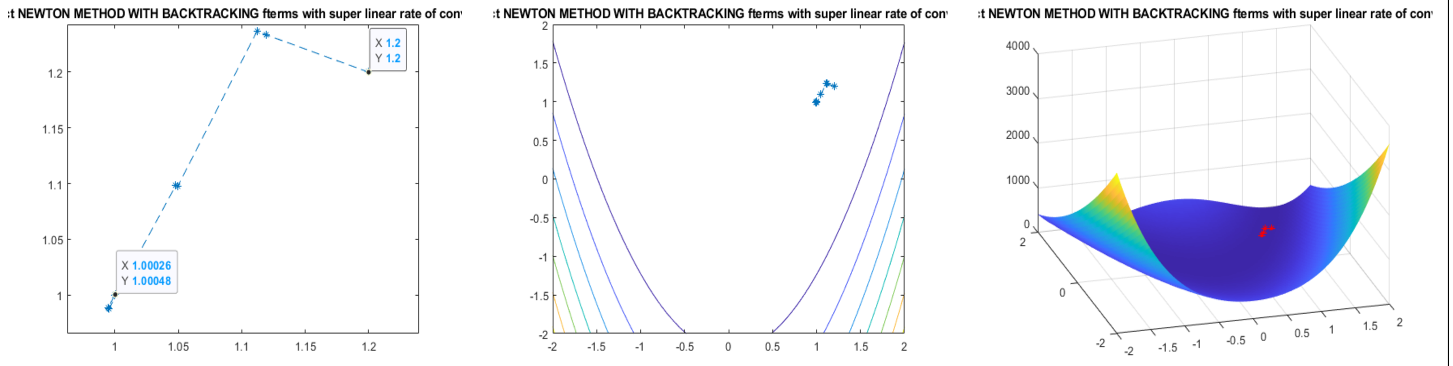
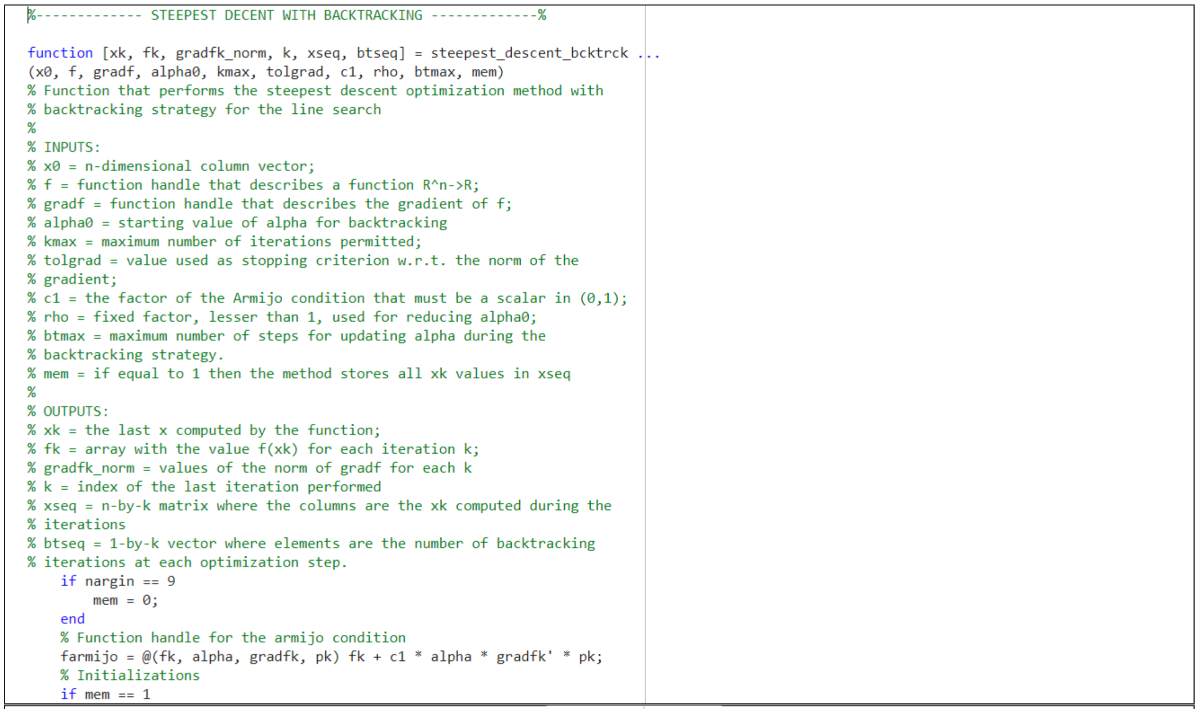
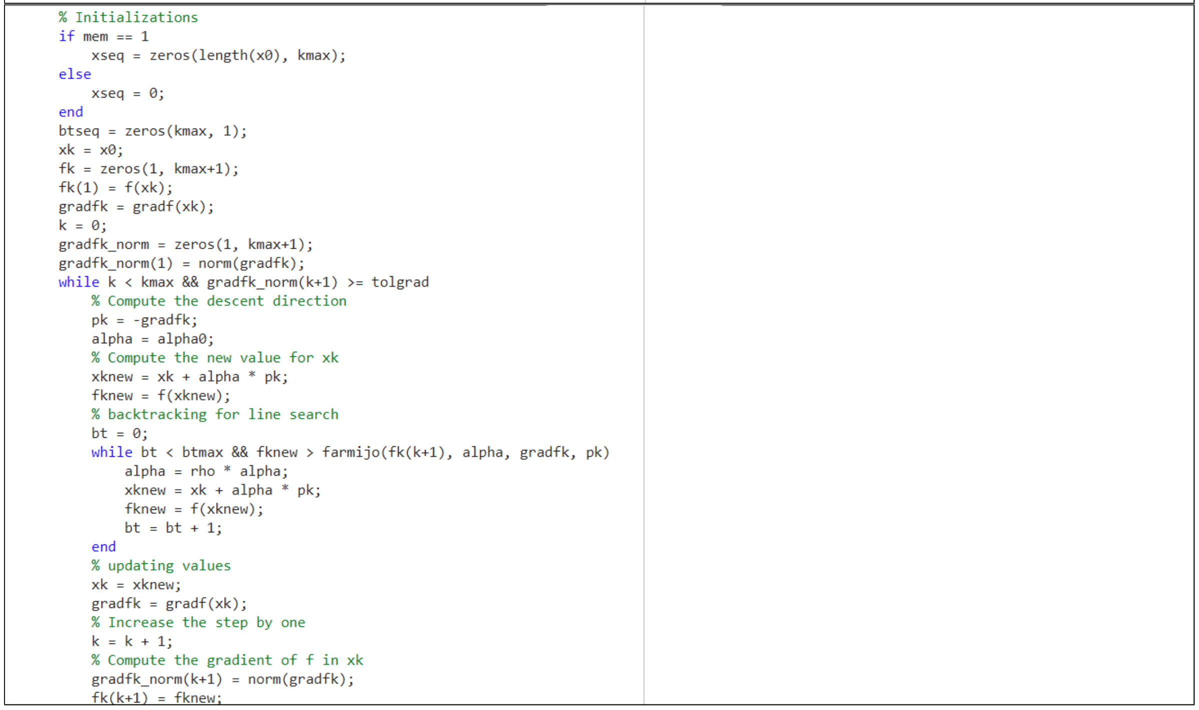


Figure 8: Inexact NEWTON X0 = [1.2, 1.2]

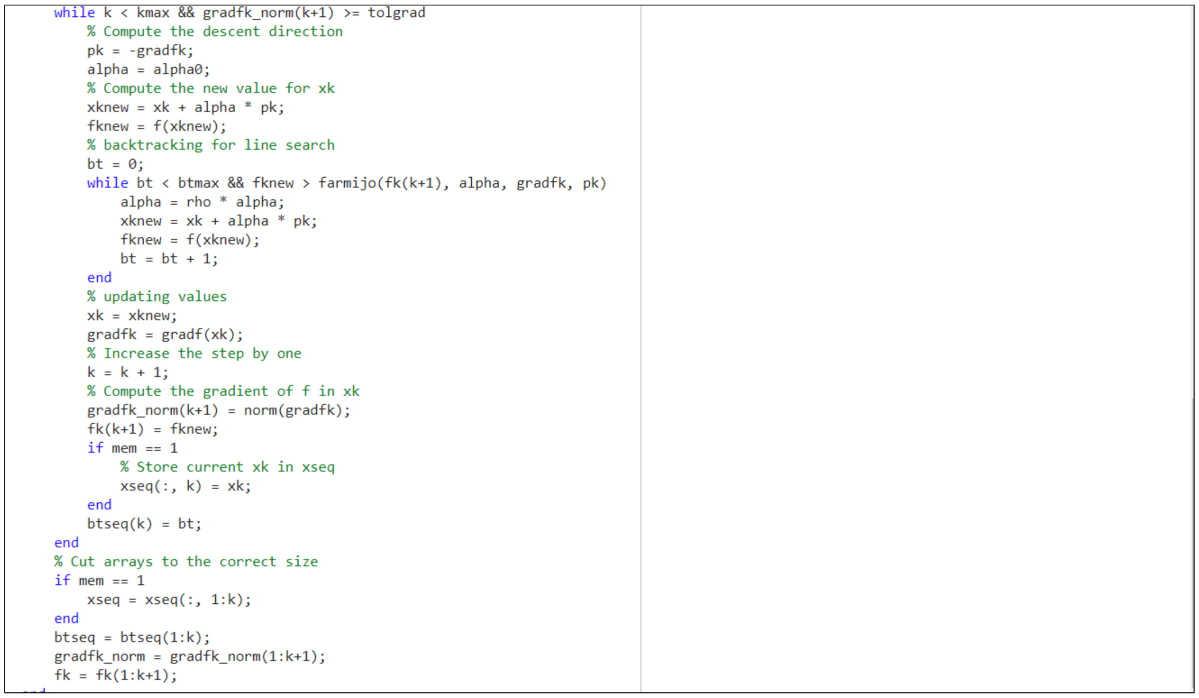
Figure 9: STEEPEST DECENT WITH BACKTRACKING



```
%------------- STEEPEST DECENT WITH BACKTRACKING -------------%

function [xk, fk, gradfk_norm, k, xseq, btseq] = steepest_descent_bcktrck ...
(x0, f, gradf, alpha0, kmax, tolgrad, c1, rho, btmax, mem)
% Function that performs the steepest descent optimization method with
% backtracking strategy for the line search
%
% INPUTS:
% x0 = n-dimensional column vector;
% f = function handle that describes a function R^n->R;
% gradf = function handle that describes the gradient of f;
% alpha0 = starting value of alpha for backtracking
% kmax = maximum number of iterations permitted;
% tolgrad = value used as stopping criterion w.r.t. the norm of the
% gradient;
% c1 = the factor of the Armijo condition that must be a scalar in (0,1);
% rho = fixed factor, lesser than 1, used for reducing alpha0;
% btmax = maximum number of steps for updating alpha during the
% backtracking strategy.
% mem = if equal to 1 then the method stores all xk values in xseq
%
% OUTPUTS:
% xk = the last x computed by the function;
% fk = array with the value f(xk) for each iteration k;
% gradfk_norm = values of the norm of gradf for each k
% 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.
    if nargin == 9
        mem = 0;
    end
    % Function handle for the armijo condition
    farmijo = @(fk, alpha, gradfk, pk) fk + c1 * alpha * gradfk' * pk;
    % Initializations
    if mem == 1
```

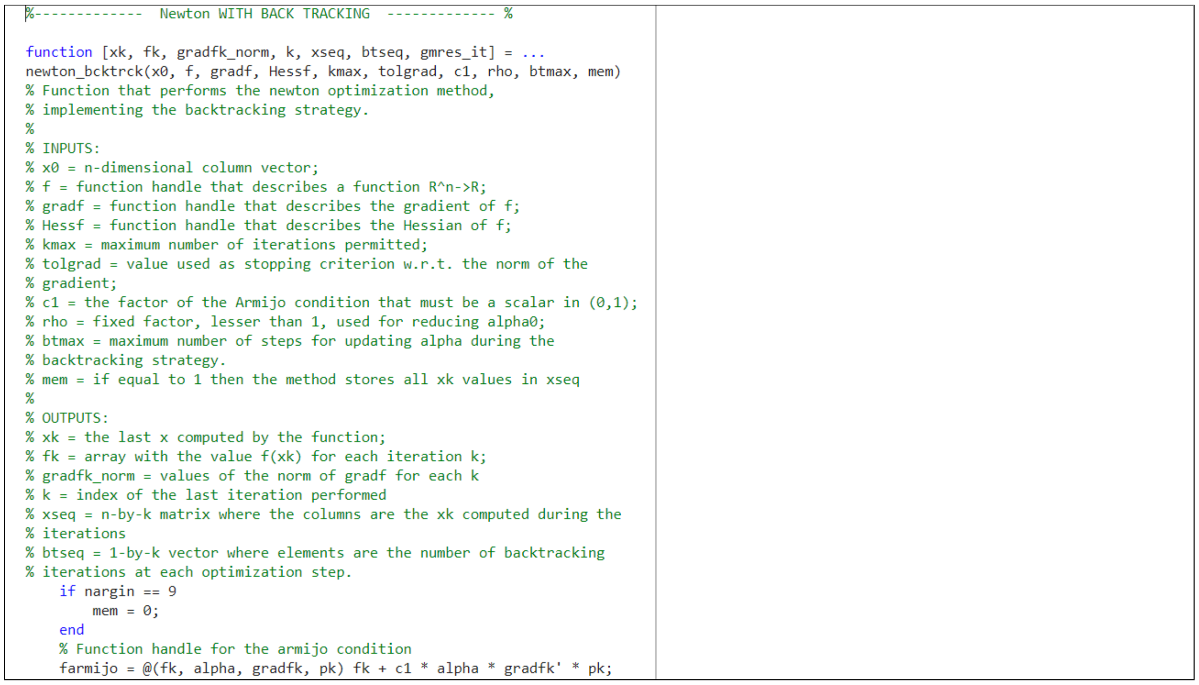


```
    % Initializations
    if mem == 1
        xseq = zeros(length(x0), kmax);
    else
        xseq = 0;
    end
    btseq = zeros(kmax, 1);
    xk = x0;
    fk = zeros(1, kmax+1);
    fk(1) = f(xk);
    gradfk = gradf(xk);
    k = 0;
    gradfk_norm = zeros(1, kmax+1);
    gradfk_norm(1) = norm(gradfk);
    while k < kmax && gradfk_norm(k+1) >= tolgrad
        % Compute the descent direction
        pk = -gradfk;
        alpha = alpha0;
        % Compute the new value for xk
        xknew = xk + alpha * pk;
        fknew = f(xknew);
        % backtracking for line search
        bt = 0;
        while bt < btmax && fknew > farmijo(fk(k+1), alpha, gradfk, pk)
            alpha = rho * alpha;
            xknew = xk + alpha * pk;
            fknew = f(xknew);
            bt = bt + 1;
        end
        % updating values
        xk = xknew;
        gradfk = gradf(xk);
        % Increase the step by one
        k = k + 1;
        % Compute the gradient of f in xk
        gradfk_norm(k+1) = norm(gradfk);
        fk(k+1) = fknew;
```



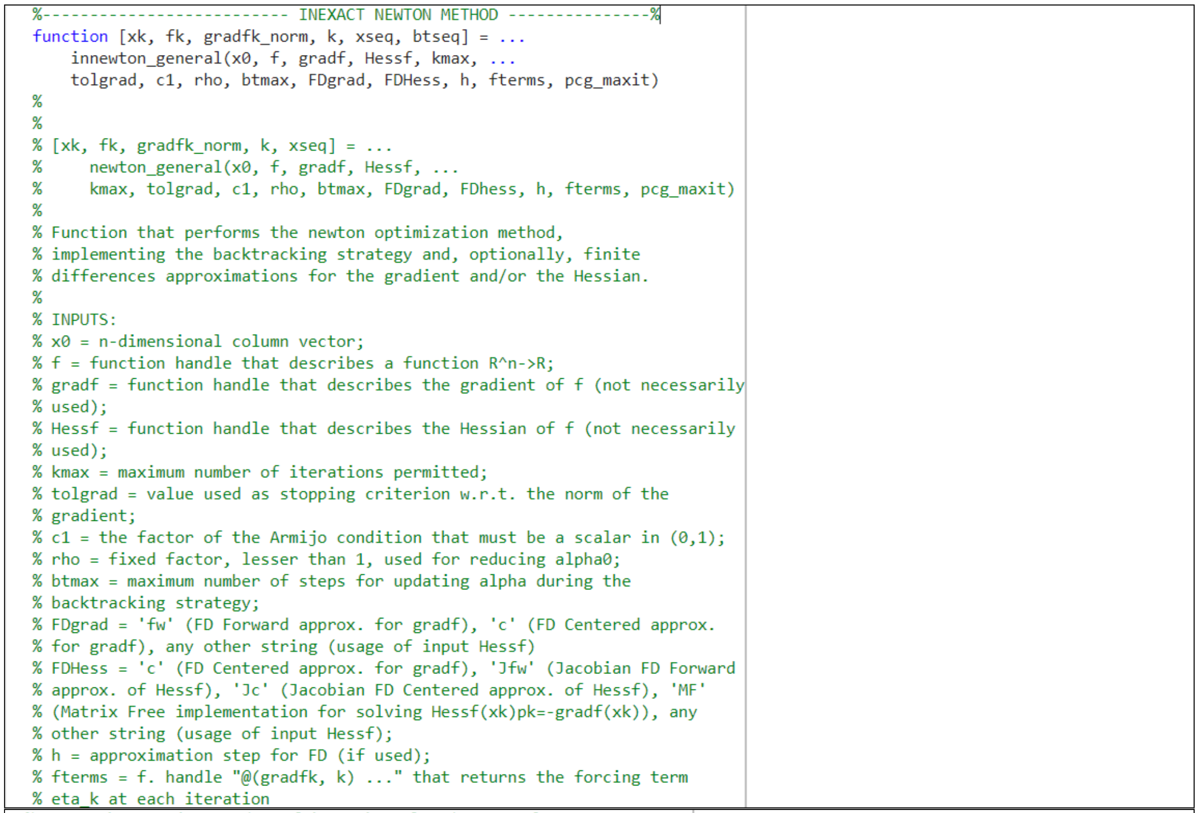
```
    while k < kmax && gradfk_norm(k+1) >= tolgrad
        % Compute the descent direction
        pk = -gradfk;
        alpha = alpha0;
        % Compute the new value for xk
        xknew = xk + alpha * pk;
        fknew = f(xknew);
        % backtracking for line search
        bt = 0;
        while bt < btmax && fknew > farmijo(fk(k+1), alpha, gradfk, pk)
            alpha = rho * alpha;
            xknew = xk + alpha * pk;
            fknew = f(xknew);
            bt = bt + 1;
        end
        % updating values
        xk = xknew;
        gradfk = gradf(xk);
        % Increase the step by one
        k = k + 1;
        % Compute the gradient of f in xk
        gradfk_norm(k+1) = norm(gradfk);
        fk(k+1) = fknew;
        if mem == 1
            % Store current xk in xseq
            xseq(:, k) = xk;
        end
        btseq(k) = bt;
    end
    % Cut arrays to the correct size
    if mem == 1
        xseq = xseq(:, 1:k);
    end
    btseq = btseq(1:k);
    gradfk_norm = gradfk_norm(1:k+1);
    fk = fk(1:k+1);
```

Figure 10: Newton WITH BACK TRACKING

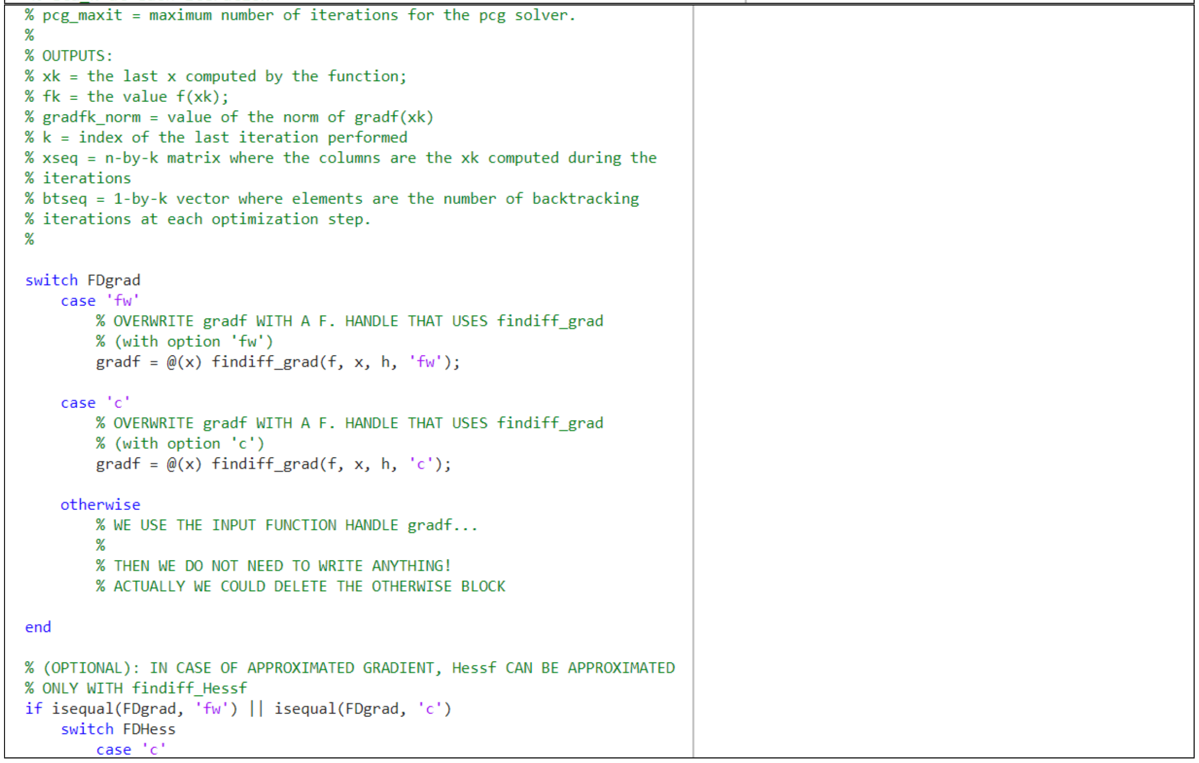


```
%-------------  Newton WITH BACK TRACKING  ------------- %

function [xk, fk, gradfk_norm, k, xseq, btseq, gmres_it] = ...
newton_bcktrck(x0, f, gradf, Hessf, kmax, tolgrad, c1, rho, btmax, mem)
% Function that performs the newton optimization method,
% implementing the backtracking strategy.
%
% INPUTS:
% x0 = n-dimensional column vector;
% f = function handle that describes a function R^n->R;
% gradf = function handle that describes the gradient of f;
% Hessf = function handle that describes the Hessian of f;
% kmax = maximum number of iterations permitted;
% tolgrad = value used as stopping criterion w.r.t. the norm of the
% gradient;
% c1 = the factor of the Armijo condition that must be a scalar in (0,1);
% rho = fixed factor, lesser than 1, used for reducing alpha0;
% btmax = maximum number of steps for updating alpha during the
% backtracking strategy.
% mem = if equal to 1 then the method stores all xk values in xseq
%
% OUTPUTS:
% xk = the last x computed by the function;
% fk = array with the value f(xk) for each iteration k;
% gradfk_norm = values of the norm of gradf for each k
% 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.
    if nargin == 9
        mem = 0;
    end
    % Function handle for the armijo condition
    farmijo = @(fk, alpha, gradfk, pk) fk + c1 * alpha * gradfk' * pk;
    % Initializations
    if mem == 1
        xseq = zeros(length(x0), kmax);
    else
        xseq = 0;
    end
    btseq = zeros(1, kmax);
    gmres_it = zeros(1, kmax);
    xk = x0;
    fk = zeros(1, kmax+1);
    fk(1) = f(xk);
    k = 0;
    gradfk = gradf(xk);
    gradfk_norm = zeros(kmax+1, 1);
    gradfk_norm(1) = norm(gradfk);
    while k < kmax && gradfk_norm(k+1) >= tolgrad
        % Compute the descent direction as solution of
        % Hessf(xk) pk = -gradfx
        HessFx = Hessf(xk);
        [pk, flag, ~, iter] = gmres(HessFx, -gradfk);

        if flag ~= 0
            disp(flag)
            warning("gmres did not converge");
        end
        % Reset the value of alpha
        alpha = 1;
        % Compute the candidate new xk
        xnew = xk + alpha * pk;
        % Compute the value of f in the candidate new xk
        fnew = f(xnew);
        bt = 0;

        % Backtracking strategy:
        % 2nd condition is the Armijo condition not satisfied
        while bt < btmax && fnew > farmijo(fk(k+1), alpha, gradfk, pk)
            % Reduce the value of alpha
            alpha = rho * alpha;
            % Update xnew and fnew w.r.t. the reduced alpha
            xnew = xk + alpha * pk;
            fnew = f(xnew);
            % Increase the counter by one
            bt = bt + 1;
        end
        % Update xk, fk, gradfk_norm
        xk = xnew;
        gradfk = gradf(xk);
        % Increase the step by one
        k = k + 1;
        gradfk_norm(k+1) = norm(gradfk);
        fk(k+1) = fnew;
        if mem == 1
            % Store current xk in xseq
            xseq(:, k) = xk;
        end
        % Store bt iterations in btseq
        btseq(k) = bt;
        gmres_it(k) = iter(2);
    end
    % "Cut" arrays to the correct size
    if mem == 1
        xseq = xseq(:, 1:k);
    end
    btseq = btseq(1:k);
    gradfk_norm = gradfk_norm(1:k+1);
    gmres_it = gmres_it(1:k);
    fk = fk(1:k+1);
end
```

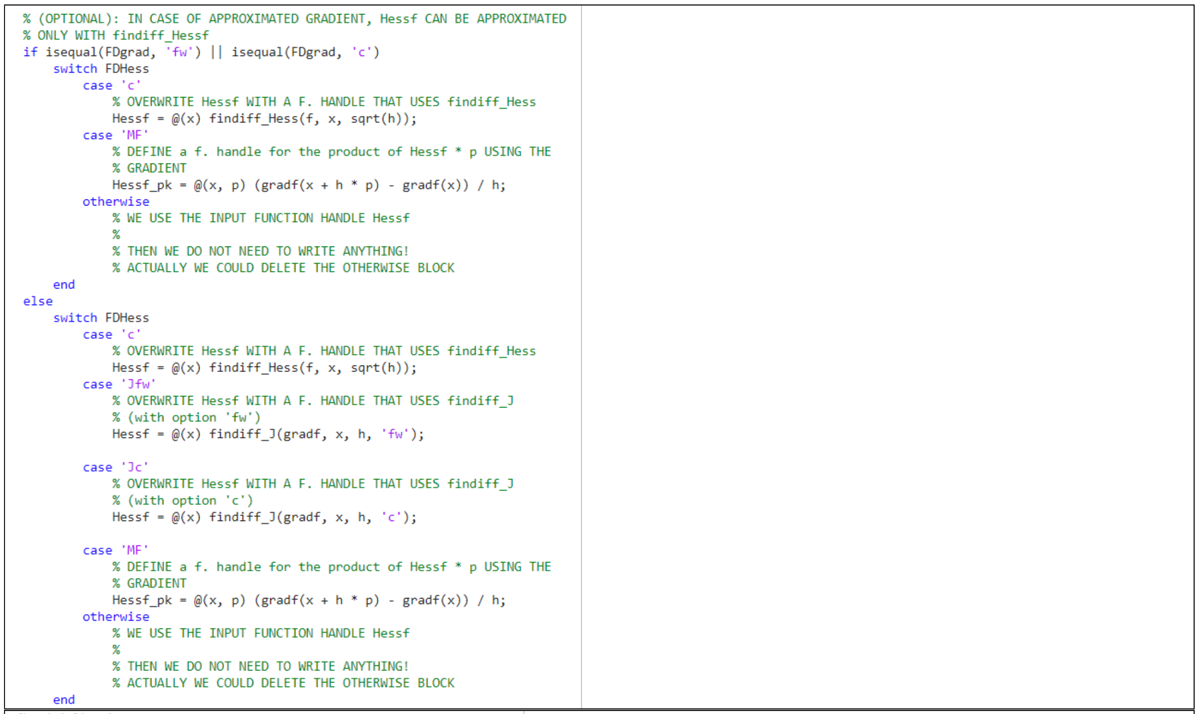
Figure 11: INEXACT NEWTON WITH BACK TRACKING



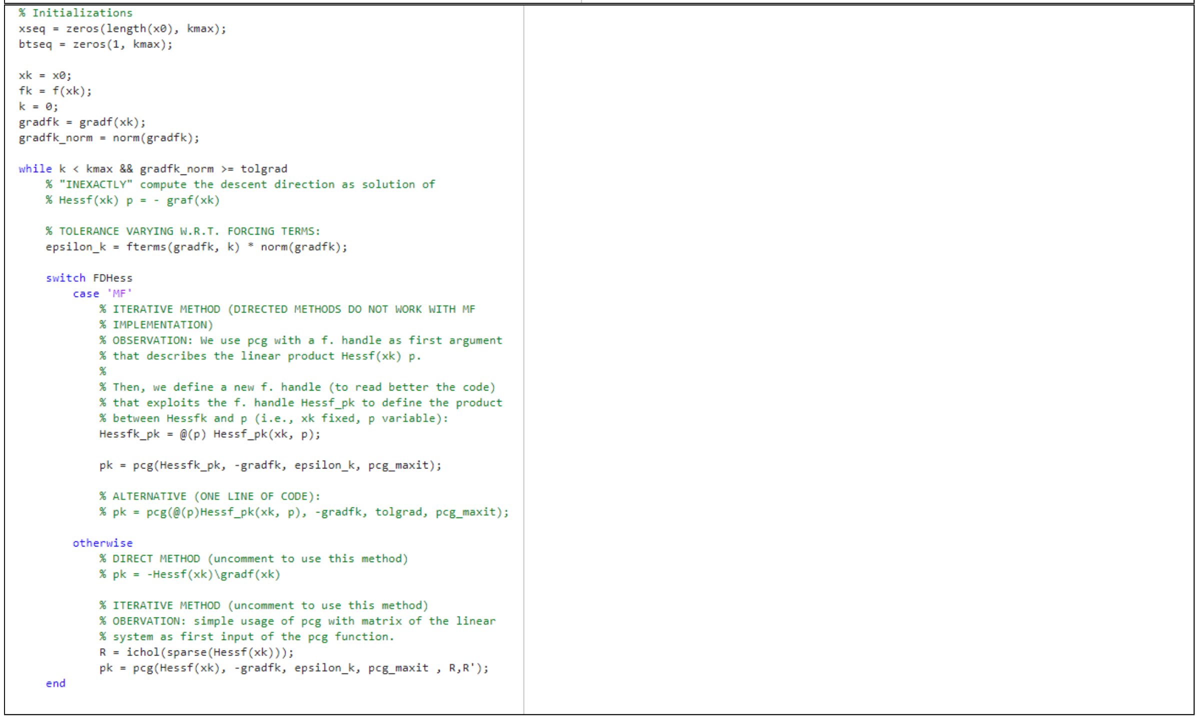
```
%-------------------------- INEXACT NEWTON METHOD ---------------%
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    innewton_general(x0, f, gradf, Hessf, kmax, ...
    tolgrad, c1, rho, btmax, FDgrad, FDHess, h, fterms, pcg_maxit)
%
%
% [xk, fk, gradfk_norm, k, xseq] = ...
%     newton_general(x0, f, gradf, Hessf, ...
%     kmax, tolgrad, c1, rho, btmax, FDgrad, FDhess, h, fterms, pcg_maxit)
%
% Function that performs the newton optimization method,
% implementing the backtracking strategy and, optionally, finite
% differences approximations for the gradient and/or the Hessian.
%
% INPUTS:
% x0 = n-dimensional column vector;
% f = function handle that describes a function R^n->R;
% gradf = function handle that describes the gradient of f (not necessarily
% used);
% Hessf = function handle that describes the Hessian of f (not necessarily
% used);
% kmax = maximum number of iterations permitted;
% tolgrad = value used as stopping criterion w.r.t. the norm of the
% gradient;
% c1 = the factor of the Armijo condition that must be a scalar in (0,1);
% rho = fixed factor, lesser than 1, used for reducing alpha0;
% btmax = maximum number of steps for updating alpha during the
% backtracking strategy;
% FDgrad = 'fw' (FD Forward approx. for gradf), 'c' (FD Centered approx.
% for gradf), any other string (usage of input Hessf)
% FDHess = 'c' (FD Centered approx. for gradf), 'Jfw' (Jacobian FD Forward
% approx. of Hessf), 'Jc' (Jacobian FD Centered approx. of Hessf), 'MF'
% (Matrix Free implementation for solving Hessf(xk)pk=-gradf(xk)), any
% other string (usage of input Hessf);
% h = approximation step for FD (if used);
% fterms = f. handle "@(gradfk, k) ..." that returns the forcing term
% eta_k at each iteration
```



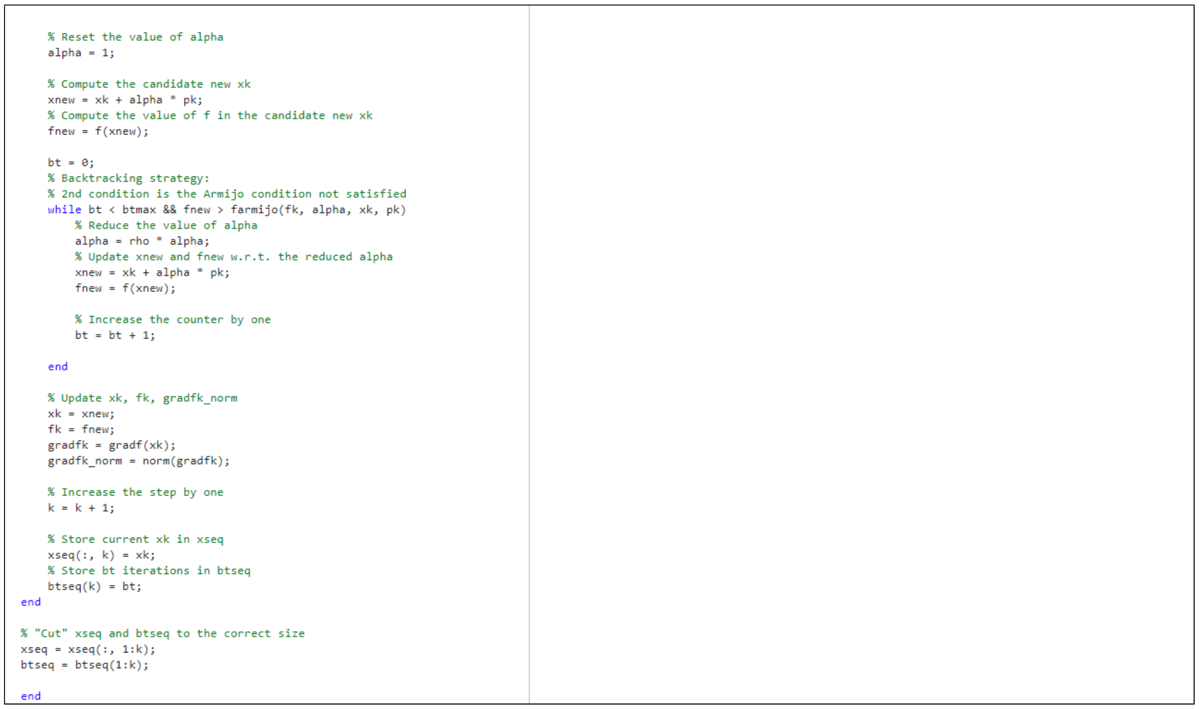
```
% pcg_maxit = maximum number of iterations for the pcg solver.
%
% OUTPUTS:
% xk = the last x computed by the function;
% fk = the value f(xk);
% gradfk_norm = value of the norm of gradf(xk)
% k = index of the last iteration performed
% xseq = n-by-k matrix where the columns are the xk computed during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

switch FDgrad
    case 'fw'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'fw')
        gradf = @(x) findiff_grad(f, x, h, 'fw');

    case 'c'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'c')
        gradf = @(x) findiff_grad(f, x, h, 'c');

    otherwise
        % WE USE THE INPUT FUNCTION HANDLE gradf...
        %
        % THEN WE DO NOT NEED TO WRITE ANYTHING!
        % ACTUALLY WE COULD DELETE THE OTHERWISE BLOCK

end

% (OPTIONAL): IN CASE OF APPROXIMATED GRADIENT, Hessf CAN BE APPROXIMATED
% ONLY WITH findiff_Hessf
if isequal(FDgrad, 'fw') || isequal(FDgrad, 'c')
    switch FDHess
        case 'c'
```



```
% (OPTIONAL): IN CASE OF APPROXIMATED GRADIENT, Hessf CAN BE APPROXIMATED
% ONLY WITH findiff_Hessf
if isequal(FDgrad, 'fw') || isequal(FDgrad, 'c')
    switch FDHess
        case 'c'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_Hess
            Hessf = @(x) findiff_Hess(f, x, sqrt(h));
        case 'MF'
            % DEFINE a f. handle for the product of Hessf * p USING THE
            % GRADIENT
            Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x)) / h;
        otherwise
            % WE USE THE INPUT FUNCTION HANDLE Hessf
            %
            % THEN WE DO NOT NEED TO WRITE ANYTHING!
            % ACTUALLY WE COULD DELETE THE OTHERWISE BLOCK
    end
else
    switch FDHess
        case 'c'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_Hess
            Hessf = @(x) findiff_Hess(f, x, sqrt(h));
        case 'Jfw'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_J
            % (with option 'fw')
            Hessf = @(x) findiff_J(gradf, x, h, 'fw');

        case 'Jc'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_J
            % (with option 'c')
            Hessf = @(x) findiff_J(gradf, x, h, 'c');

        case 'MF'
            % DEFINE a f. handle for the product of Hessf * p USING THE
            % GRADIENT
            Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x)) / h;
        otherwise
            % WE USE THE INPUT FUNCTION HANDLE Hessf
            %
            % THEN WE DO NOT NEED TO WRITE ANYTHING!
            % ACTUALLY WE COULD DELETE THE OTHERWISE BLOCK
    end
```

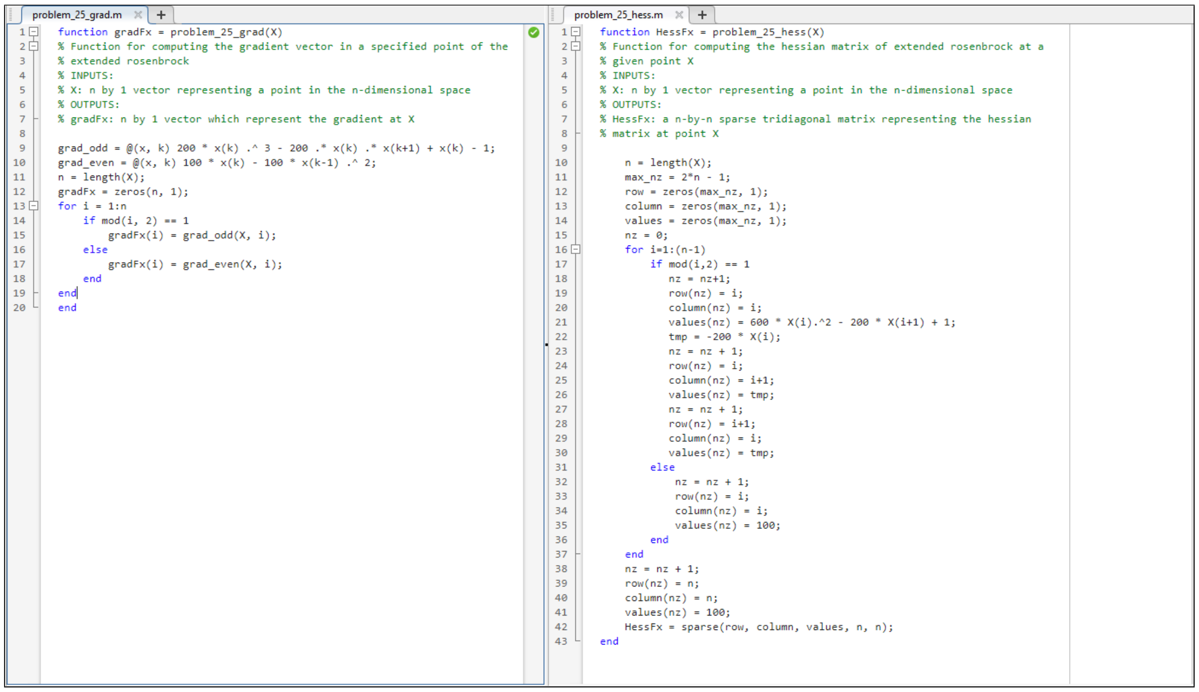


```
% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);

xk = x0;
fk = f(xk);
k = 0;
gradfk = gradf(xk);
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    % "INEXACTLY" compute the descent direction as solution of
    % Hessf(xk) p = - graf(xk)

    % TOLERANCE VARYING W.R.T. FORCING TERMS:
    epsilon_k = fterms(gradfk, k) * norm(gradfk);

    switch FDHess
        case 'MF'
            % ITERATIVE METHOD (DIRECTED METHODS DO NOT WORK WITH MF
            % IMPLEMENTATION)
            % OBSERVATION: We use pcg with a f. handle as first argument
            % that describes the linear product Hessf(xk) p.
            %
            % Then, we define a new f. handle (to read better the code)
            % that exploits the f. handle Hessf_pk to define the product
            % between Hessfk and p (i.e., xk fixed, p variable):
            Hessfk_pk = @(p) Hessf_pk(xk, p);

            pk = pcg(Hessfk_pk, -gradfk, epsilon_k, pcg_maxit);

            % ALTERNATIVE (ONE LINE OF CODE):
            % pk = pcg(@(p)Hessf_pk(xk, p), -gradfk, tolgrad, pcg_maxit);

        otherwise
            % DIRECT METHOD (uncomment to use this method)
            % pk = -Hessf(xk)\gradf(xk)

            % ITERATIVE METHOD (uncomment to use this method)
            % OBERVATION: simple usage of pcg with matrix of the linear
            % system as first input of the pcg function.
            R = ichol(sparse(Hessf(xk)));
            pk = pcg(Hessf(xk), -gradfk, epsilon_k, pcg_maxit , R,R');
    end
```



```
    % Reset the value of alpha
    alpha = 1;

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

    bt = 0;
    % Backtracking strategy:
    % 2nd condition is the Armijo condition not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, xk, pk)
        % Reduce the value of alpha
        alpha = rho * alpha;
        % Update xnew and fnew w.r.t. the reduced alpha
        xnew = xk + alpha * pk;
        fnew = f(xnew);

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

    end

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

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

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

% "Cut" xseq and btseq to the correct size
xseq = xseq(:, 1:k);
btseq = btseq(1:k);

end
```

Figure 12: Problem 25 gradient & Hessian



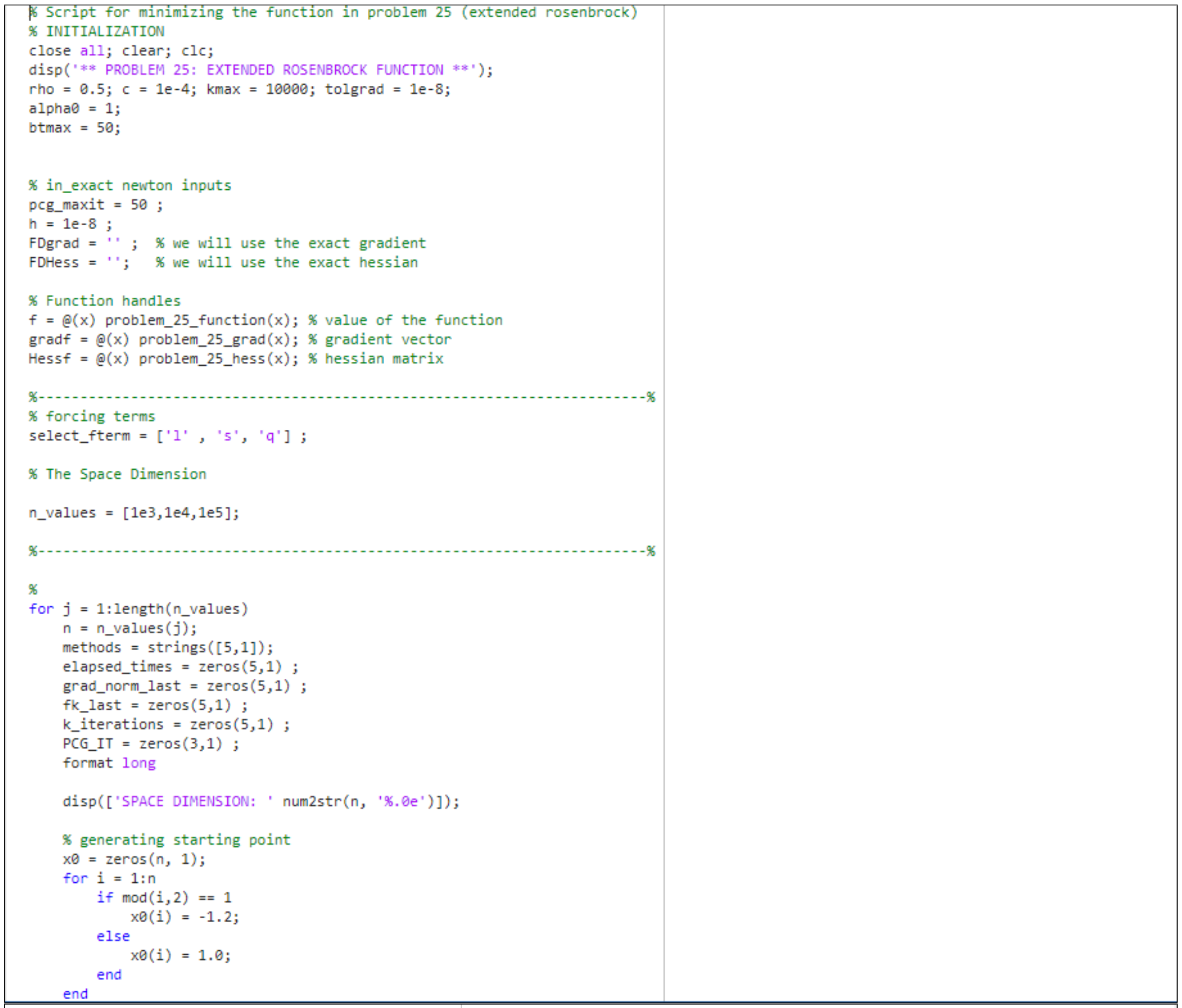
problem_25_grad.m

```
function gradFx = problem_25_grad(X)
% Function for computing the gradient vector in a specified point of the
% extended rosenbrock
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% gradFx: n by 1 vector which represent the gradient at X

grad_odd = @(x, k) 200 * x(k) .^ 3 - 200 .* x(k) .* x(k+1) + x(k) - 1;
grad_even = @(x, k) 100 * x(k) - 100 * x(k-1) .^ 2;
n = length(X);
gradFx = zeros(n, 1);
for i = 1:n
    if mod(i, 2) == 1
        gradFx(i) = grad_odd(X, i);
    else
        gradFx(i) = grad_even(X, i);
    end
end
end
```

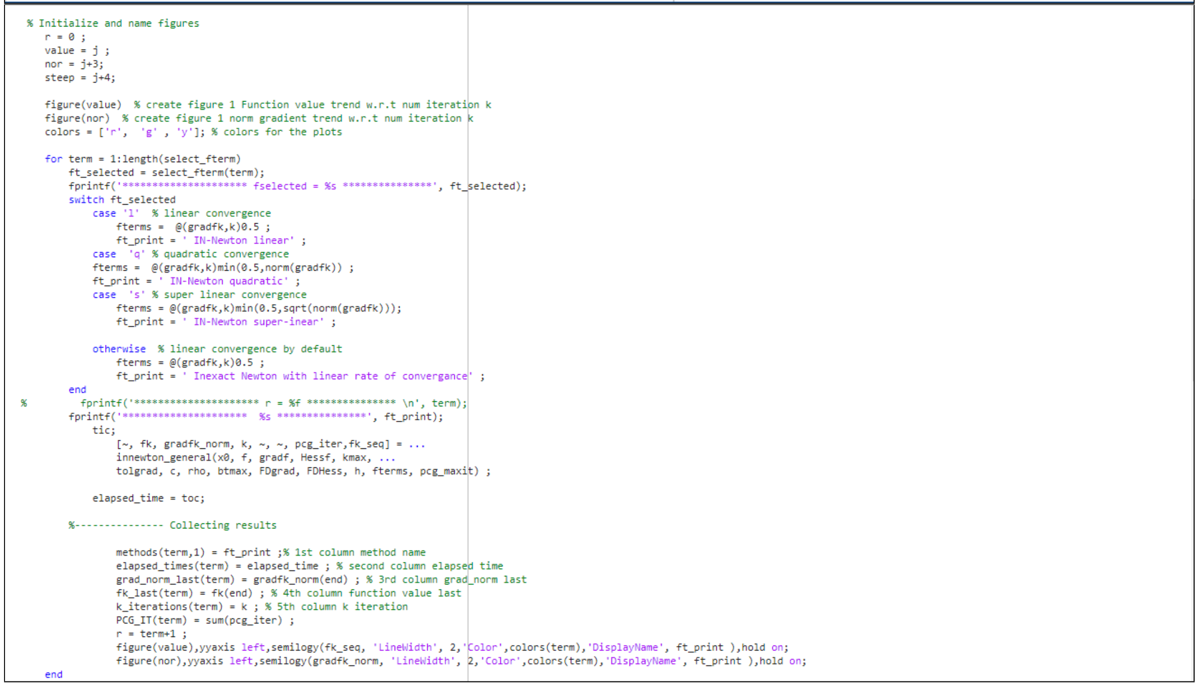
problem_25_hess.m

```
function HessFx = problem_25_hess(X)
% Function for computing the hessian matrix of extended rosenbrock at a
% given point X
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% HessFx: a n-by-n sparse tridiagonal matrix representing the hessian
% matrix at point X

    n = length(X);
    max_nz = 2*n - 1;
    row = zeros(max_nz, 1);
    column = zeros(max_nz, 1);
    values = zeros(max_nz, 1);
    nz = 0;
    for i=1:(n-1)
        if mod(i,2) == 1
           nz = nz+1;
           row(nz) = i;
           column(nz) = i;
           values(nz) = 600 * X(i).^2 - 200 * X(i+1) + 1;
           tmp = -200 * X(i);
           nz = nz + 1;
           row(nz) = i;
           column(nz) = i+1;
           values(nz) = tmp;
           nz = nz + 1;
           row(nz) = i+1;
           column(nz) = i;
           values(nz) = tmp;
        else
            nz = nz + 1;
            row(nz) = i;
            column(nz) = i;
            values(nz) = 100;
        end
    end
    nz = nz + 1;
    row(nz) = n;
    column(nz) = n;
    values(nz) = 100;
    HessFx = sparse(row, column, values, n, n);
end
```

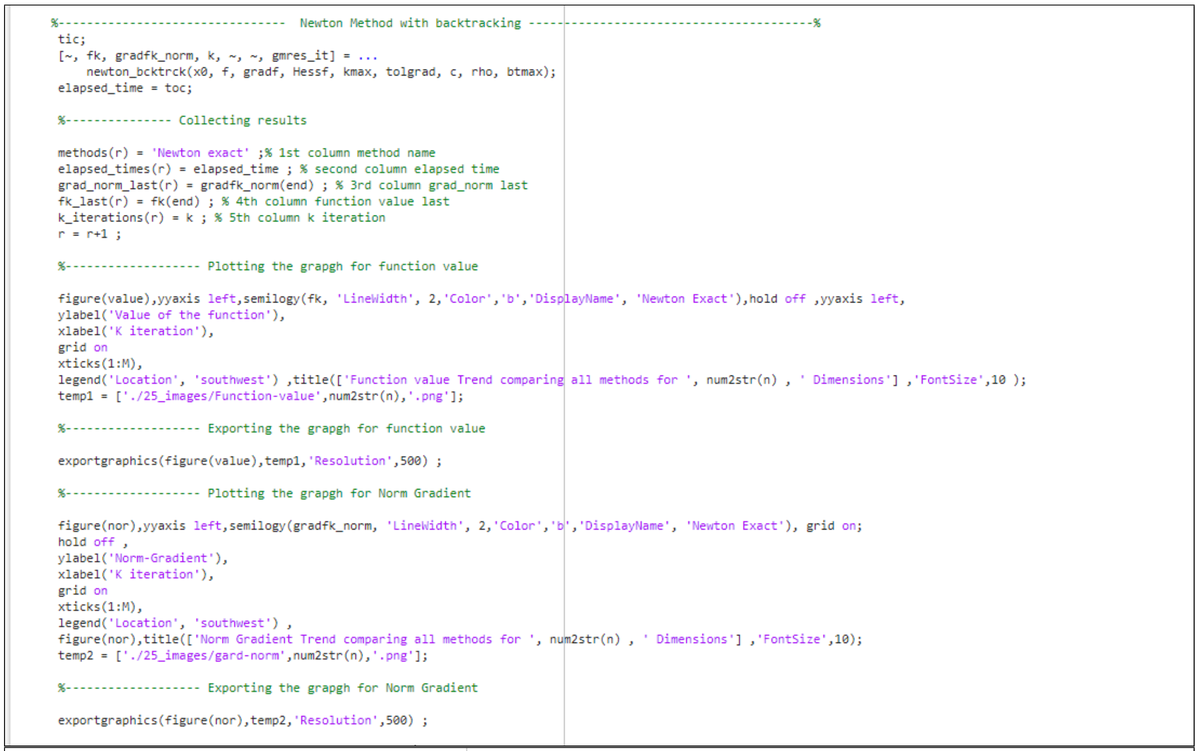
Figure 13: Problem 25 main code



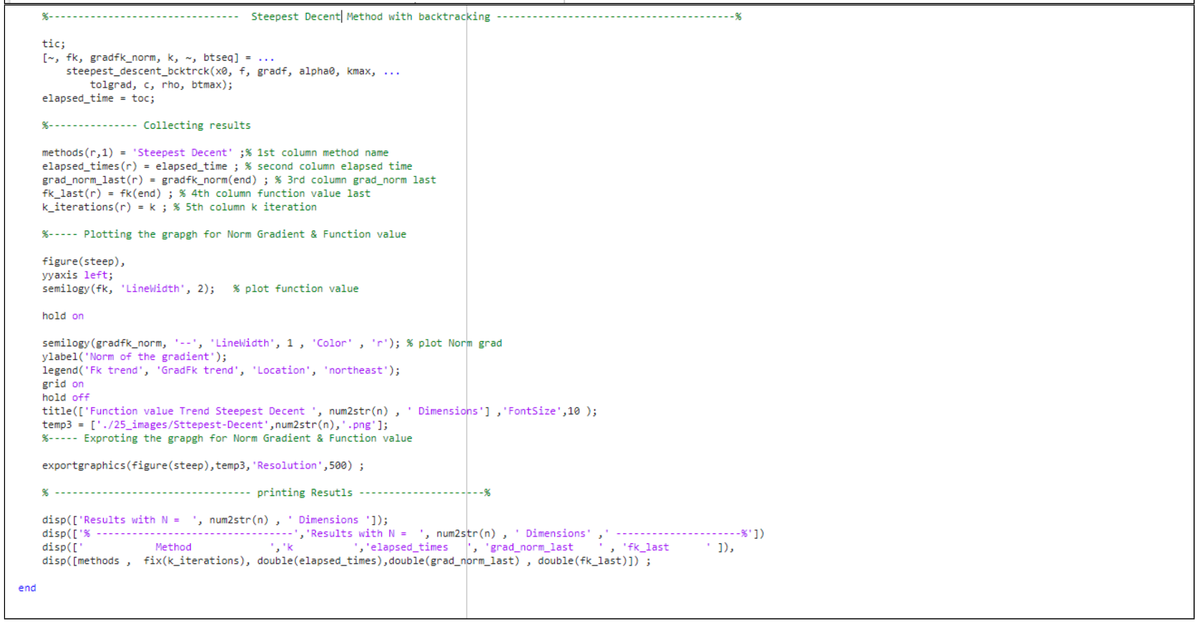
```
% Script for minimizing the function in problem 25 (extended rosenbrock)
% INITIALIZATION
close all; clear; clc;
disp('** PROBLEM 25: EXTENDED ROSENBROCK FUNCTION **');
rho = 0.5; c = 1e-4; kmax = 10000; tolgrad = 1e-8;
alpha0 = 1;
btmax = 50;


% in_exact newton inputs
pcg_maxit = 50 ;
h = 1e-8 ;
FDgrad = '' ;  % we will use the exact gradient
FDHess = '';   % we will use the exact hessian

% Function handles
f = @(x) problem_25_function(x); % value of the function
gradf = @(x) problem_25_grad(x); % gradient vector
Hessf = @(x) problem_25_hess(x); % hessian matrix

%-------------------------------------------------------------------------%
% forcing terms
select_fterm = ['l' , 's', 'q'] ;

% The Space Dimension

n_values = [1e3,1e4,1e5];

%-------------------------------------------------------------------------%

%
for j = 1:length(n_values)
    n = n_values(j);
    methods = strings([5,1]);
    elapsed_times = zeros(5,1) ;
    grad_norm_last = zeros(5,1) ;
    fk_last = zeros(5,1) ;
    k_iterations = zeros(5,1) ;
    PCG_IT = zeros(3,1) ;
    format long

    disp(['SPACE DIMENSION: ' num2str(n, '%.0e')]);

    % generating starting point
    x0 = zeros(n, 1);
    for i = 1:n
        if mod(i,2) == 1
            x0(i) = -1.2;
        else
            x0(i) = 1.0;
        end
    end
```



```
 % Initialize and name figures
   r = 0 ;
   value = j ;
   nor = j+3;
   steep = j+4;

   figure(value)  % create figure 1 Function value trend w.r.t num iteration k
   figure(nor)  % create figure 1 norm gradient trend w.r.t num iteration k
   colors = ['r',  'g' , 'y']; % colors for the plots

   for term = 1:length(select_fterm)
       ft_selected = select_fterm(term);
       fprintf('********************* fselected = %s ***************', ft_selected);
       switch ft_selected
           case 'l'  % linear convergence
               fterms =  @(gradfk,k)0.5 ;
               ft_print = ' IN-Newton linear' ;
           case  'q' % quadratic convergence
           fterms =  @(gradfk,k)min(0.5,norm(gradfk)) ;
           ft_print = ' IN-Newton quadratic' ;
           case  's' % super linear convergence
               fterms = @(gradfk,k)min(0.5,sqrt(norm(gradfk)));
               ft_print = ' IN-Newton super-inear' ;

           otherwise  % linear convergence by default
               fterms = @(gradfk,k)0.5 ;
               ft_print = ' Inexact Newton with linear rate of convergance' ;
       end
%        fprintf('********************* r = %f *************** \n', term);
       fprintf('*********************  %s ***************', ft_print);
           tic;
               [~, fk, gradfk_norm, k, ~, ~, pcg_iter,fk_seq] = ...
               innewton_general(x0, f, gradf, Hessf, kmax, ...
               tolgrad, c, rho, btmax, FDgrad, FDHess, h, fterms, pcg_maxit) ;

           elapsed_time = toc;

       %-------------- Collecting results

               methods(term,1) = ft_print ;% 1st column method name
               elapsed_times(term) = elapsed_time ; % second column elapsed time
               grad_norm_last(term) = gradfk_norm(end) ; % 3rd column grad_norm last
               fk_last(term) = fk(end) ; % 4th column function value last
               k_iterations(term) = k ; % 5th column k iteration
               PCG_IT(term) = sum(pcg_iter) ;
               r = term+1 ;
               figure(value),yyaxis left,semilogy(fk_seq, 'LineWidth', 2,'Color',colors(term),'DisplayName', ft_print ),hold on;
               figure(nor),yyaxis left,semilogy(gradfk_norm, 'LineWidth', 2,'Color',colors(term),'DisplayName', ft_print ),hold on;
   end
```

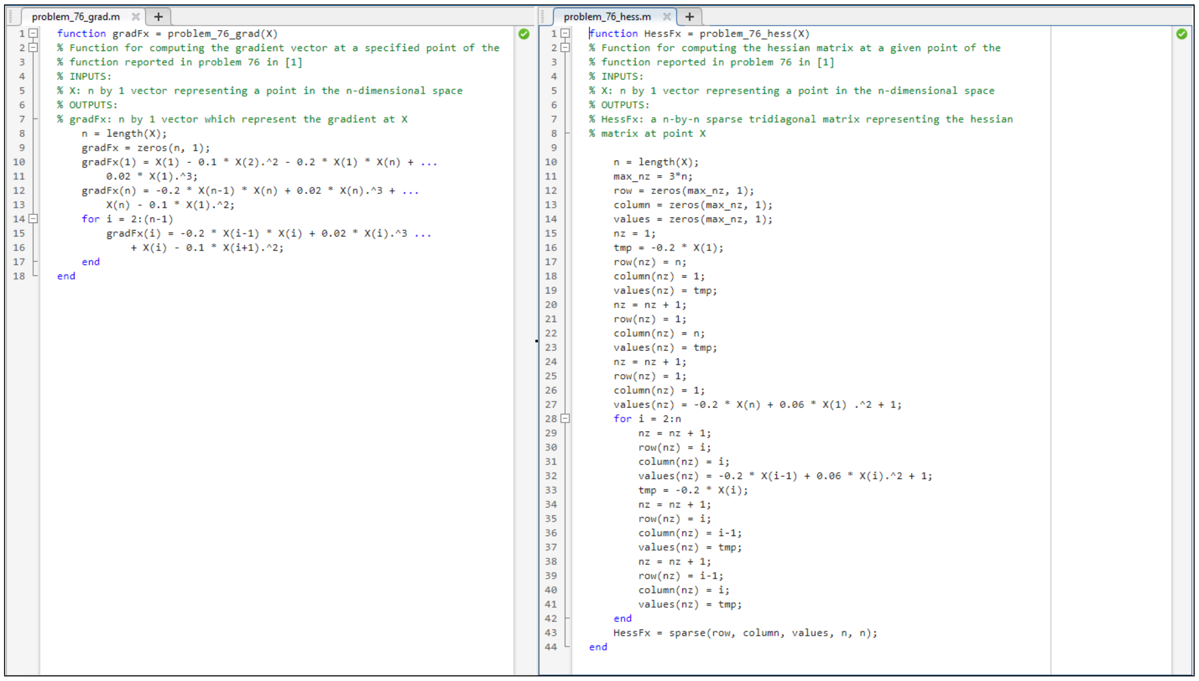


```
 %--------------------------------  Newton Method with backtracking -------------------------------------%
  tic;
  [~, fk, gradfk_norm, k, ~, ~, gmres_it] = ...
      newton_bcktrck(x0, f, gradf, Hessf, kmax, tolgrad, c, rho, btmax);
  elapsed_time = toc;

  %-------------- Collecting results

  methods(r) = 'Newton exact' ;% 1st column method name
  elapsed_times(r) = elapsed_time ; % second column elapsed time
  grad_norm_last(r) = gradfk_norm(end) ; % 3rd column grad_norm last
  fk_last(r) = fk(end) ; % 4th column function value last
  k_iterations(r) = k ; % 5th column k iteration
  r = r+1 ;

  %------------------ Plotting the grapgh for function value

  figure(value),yyaxis left,semilogy(fk, 'LineWidth', 2,'Color','b','DisplayName', 'Newton Exact'),hold off ,yyaxis left,
  ylabel('Value of the function'),
  xlabel('K iteration'),
  grid on
  xticks(1:M),
  legend('Location', 'southwest') ,title(['Function value Trend comparing all methods for ', num2str(n) , ' Dimensions'] ,'FontSize',10 );
  temp1 = ['./25_images/Function-value',num2str(n),'.png'];

  %------------------ Exporting the grapgh for function value

  exportgraphics(figure(value),temp1,'Resolution',500) ;

  %------------------ Plotting the grapgh for Norm Gradient

  figure(nor),yyaxis left,semilogy(gradfk_norm, 'LineWidth', 2,'Color','b','DisplayName', 'Newton Exact'), grid on;
  hold off ,
  ylabel('Norm-Gradient'),
  xlabel('K iteration'),
  grid on
  xticks(1:M),
  legend('Location', 'southwest') ,
  figure(nor),title(['Norm Gradient Trend comparing all methods for ', num2str(n) , ' Dimensions'] ,'FontSize',10);
  temp2 = ['./25_images/gard-norm',num2str(n),'.png'];

  %------------------ Exporting the grapgh for Norm Gradient

  exportgraphics(figure(nor),temp2,'Resolution',500) ;
```



```
    %------------------------------  Steepest Decent Method with backtracking --------------------------------------%

    tic;
    [~, fk, gradfk_norm, k, ~, btseq] = ...
        steepest_descent_bcktrck(x0, f, gradf, alpha0, kmax, ...
            tolgrad, c, rho, btmax);
    elapsed_time = toc;

    %-------------- Collecting results

    methods(r,1) = 'Steepest Decent' ;% 1st column method name
    elapsed_times(r) = elapsed_time ; % second column elapsed time
    grad_norm_last(r) = gradfk_norm(end) ; % 3rd column grad_norm last
    fk_last(r) = fk(end) ; % 4th column function value last
    k_iterations(r) = k ; % 5th column k iteration

    %----- Plotting the grapgh for Norm Gradient & Function value

    figure(steep),
    yyaxis left;
    semilogy(fk, 'LineWidth', 2);   % plot function value

    hold on

    semilogy(gradfk_norm, '--', 'LineWidth', 1 , 'Color' , 'r'); % plot Norm grad
    ylabel('Norm of the gradient');
    legend('Fk trend', 'GradFk trend', 'Location', 'northeast');
    grid on
    hold off
    title(['Function value Trend Steepest Decent ', num2str(n) , ' Dimensions'] ,'FontSize',10 );
    temp3 = ['./25_images/Sttepest-Decent',num2str(n),'.png'];
    %----- Exproting the grapgh for Norm Gradient & Function value

    exportgraphics(figure(steep),temp3,'Resolution',500) ;

    % -------------------------------- printing Resutls -------------------%

    disp(['Results with N =  ', num2str(n) , ' Dimensions ']);
    disp(['% --------------------------------','Results with N =  ', num2str(n) , ' Dimensions' ,' --------------------%'])
    disp(['            Method             ','k           ','elapsed_times   ', 'grad_norm_last    ' , 'fk_last      ' ]),
    disp([methods ,  fix(k_iterations), double(elapsed_times),double(grad_norm_last) , double(fk_last)]) ;

end
```

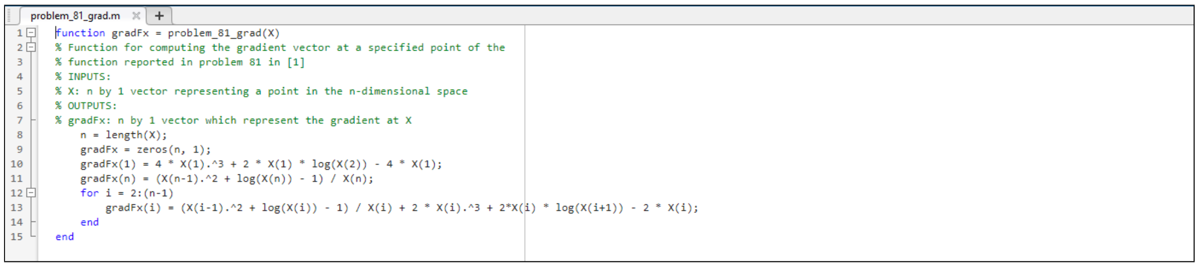
Figure 14: Problem 76 gradient & Hessian



```matlab
function gradFx = problem_76_grad(X)
% Function for computing the gradient vector at a specified point of the
% function reported in problem 76 in [1]
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% gradFx: n by 1 vector which represent the gradient at X
    n = length(X);
    gradFx = zeros(n, 1);
    gradFx(1) = X(1) - 0.1 * X(2).^2 - 0.2 * X(1) * X(n) + ...
        0.02 * X(1).^3;
    gradFx(n) = -0.2 * X(n-1) * X(n) + 0.02 * X(n).^3 + ...
        X(n) - 0.1 * X(1).^2;
    for i = 2:(n-1)
        gradFx(i) = -0.2 * X(i-1) * X(i) + 0.02 * X(i).^3 ...
            + X(i) - 0.1 * X(i+1).^2;
    end
end
```

```matlab
function HessFx = problem_76_hess(X)
% Function for computing the hessian matrix at a given point of the
% function reported in problem 76 in [1]
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% HessFx: a n-by-n sparse tridiagonal matrix representing the hessian
% matrix at point X

    n = length(X);
    max_nz = 3*n;
    row = zeros(max_nz, 1);
    column = zeros(max_nz, 1);
    values = zeros(max_nz, 1);
    nz = 1;
    tmp = -0.2 * X(1);
    row(nz) = n;
    column(nz) = 1;
    values(nz) = tmp;
    nz = nz + 1;
    row(nz) = 1;
    column(nz) = n;
    values(nz) = tmp;
    nz = nz + 1;
    row(nz) = 1;
    column(nz) = 1;
    values(nz) = -0.2 * X(n) + 0.06 * X(1) .^2 + 1;
    for i = 2:n
        nz = nz + 1;
        row(nz) = i;
        column(nz) = i;
        values(nz) = -0.2 * X(i-1) + 0.06 * X(i).^2 + 1;
        tmp = -0.2 * X(i);
        nz = nz + 1;
        row(nz) = i;
        column(nz) = i-1;
        values(nz) = tmp;
        nz = nz + 1;
        row(nz) = i-1;
        column(nz) = i;
        values(nz) = tmp;
    end
    HessFx = sparse(row, column, values, n, n);
end
```

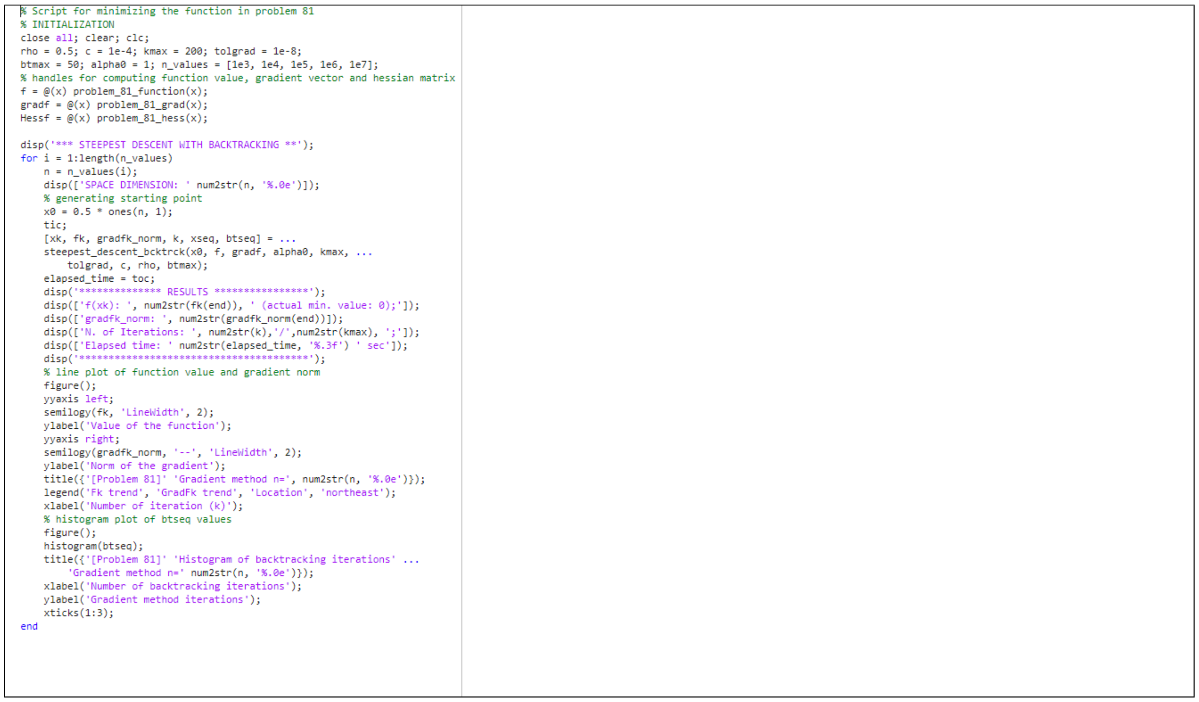
Figure 15: Problem 81 gradient



problem_81_grad.m

```matlab
function gradFx = problem_81_grad(X)
% Function for computing the gradient vector at a specified point of the
% function reported in problem 81 in [1]
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% gradFx: n by 1 vector which represent the gradient at X
    n = length(X);
    gradFx = zeros(n, 1);
    gradFx(1) = 4 * X(1).^3 + 2 * X(1) * log(X(2)) - 4 * X(1);
    gradFx(n) = (X(n-1).^2 + log(X(n)) - 1) / X(n);
    for i = 2:(n-1)
        gradFx(i) = (X(i-1).^2 + log(X(i)) - 1) / X(i) + 2 * X(i).^3 + 2*X(i) * log(X(i+1)) - 2 * X(i);
    end
end
```

Figure 16: Problem 81 main code



```matlab
% Script for minimizing the function in problem 81
% INITIALIZATION
close all; clear; clc;
rho = 0.5; c = 1e-4; kmax = 200; tolgrad = 1e-8;
btmax = 50; alpha0 = 1; n_values = [1e3, 1e4, 1e5, 1e6, 1e7];
% handles for computing function value, gradient vector and hessian matrix
f = @(x) problem_81_function(x);
gradf = @(x) problem_81_grad(x);
Hessf = @(x) problem_81_hess(x);

disp('*** STEEPEST DESCENT WITH BACKTRACKING **');
for i = 1:length(n_values)
    n = n_values(i);
    disp(['SPACE DIMENSION: ' num2str(n, '%.0e')]);
    % generating starting point
    x0 = 0.5 * ones(n, 1);
    tic;
    [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    steepest_descent_bcktrck(x0, f, gradf, alpha0, kmax, ...
        tolgrad, c, rho, btmax);
    elapsed_time = toc;
    disp('************** RESULTS ****************');
    disp(['f(xk): ', num2str(fk(end)), ' (actual min. value: 0);']);
    disp(['gradfk_norm: ', num2str(gradfk_norm(end))]);
    disp(['N. of Iterations: ', num2str(k),'/',num2str(kmax), ';']);
    disp(['Elapsed time: ' num2str(elapsed_time, '%.3f') ' sec']);
    disp('**************************************');
    % line plot of function value and gradient norm
    figure();
    yyaxis left;
    semilogy(fk, 'LineWidth', 2);
    ylabel('Value of the function');
    yyaxis right;
    semilogy(gradfk_norm, '--', 'LineWidth', 2);
    ylabel('Norm of the gradient');
    title({'[Problem 81]' 'Gradient method n=', num2str(n, '%.0e')});
    legend('Fk trend', 'GradFk trend', 'Location', 'northeast');
    xlabel('Number of iteration (k)');
    % histogram plot of btseq values
    figure();
    histogram(btseq);
    title({'[Problem 81]' 'Histogram of backtracking iterations' ...
        'Gradient method n=' num2str(n, '%.0e')});
    xlabel('Number of backtracking iterations');
    ylabel('Gradient method iterations');
    xticks(1:3);
end
```