Unconstrained Optimization Using Steepest Descent and Newton Method*

*Course: Numerical optimization for large scale problems and Stochastic Optimization

Davide Buoso
Politecnico di Torino
Student ID: s317660

Marco Castiglia
Politecnico di Torino
Student ID: s317381

***Abstract*—In this report we will discuss about three minimization problems that we try to solve using two numerical methods: Steepest Gradient Descent and Newton methods. In both cases we exploit the backtracking line-search strategy in order to find a feasible step-length for a sufficient descent towards the minimum.**

I. Problem overview

The objective of the following report is to describe methods, strategies and results of three different unconstrained minimization problems published in [1]. The best result is certainly marked by reaching the global minimum point, or a local minimum, depending on the different functions taken into account. This must be achieved by exploiting numerical methods in the case of large-scale problems, since both computational and memory issues arise in these cases given the large amount of data to be handled. Before finding the solution of those problems, we had to implement numerical methods and we chose MATLAB to do so. Their correctness is tested on one of the most famous unconstrained problem functions: the *Rosenbrock* function (1).

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

The following problems have been chosen and will be discussed:

- *Extended Rosenbrock* function:

$$\begin{aligned} &F(x) = 1/2 \sum_{k=1}^{n} f_k^2(x), \\ &f_k(x) = 10(x_k^2 - x_{k+1}), mod(k,2) = 1 \\ &f_k(x) = x_{k-1} - 1, mod(k,2) = 0 \end{aligned} \tag{2}$$

  starting point:
  $x_i = -1.2, mod(K,2) = 1$
  $x_i = 1, mod(K,2) = 0$
- *Generalized Broyden* tridiagonal function

$$\begin{aligned} &F(x) = 1/2 \sum_{k=1}^{n} f_k^2(x), \\ &f_k(x) = (3 - 2x_k)x_k + 1 - x_{k-1} - x_{k+1} \end{aligned} \tag{3}$$

  where $x_0 = x_{n+1} = 0$
  starting point: $x_i = -1$
- *Problem 79* (does not have an official name) function

$$\begin{aligned} &F(x) = 1/2 \sum_{k=1}^{n} f_k^2(x), \\ &f_k(x) = (3 - x_k/10)x_k + 1 - x_{k-1} - 2x_{k+1} \end{aligned} \tag{4}$$

  where $x_0 = x_{n+1} = 0$
  starting point: $x_i = -1$

According to the methods we decided to exploit, we had to derive the analytical form of the gradient and the hessian function of the different problems. All the computations are summarized in the table in the Appendix A.

II. Description of the optimization algorithms

In the following section the optimization methods, their advantages and disadvantages will be introduced. After the implementation, they have been tested on the Rosenbrock function, showing good results and a satisfying convergence, as described in Section III.

A. Steepest Gradient Descent

The basic idea of the Steepest Descent is well summed up in its name: follow the most rapid descent direction. This does not ensure that the minimum point to which we are trying to converge is a global minimum point, since the descent direction will be the fastest, but only in local terms.

The gradient of the function represents the direction of the steepest increase of the function. Therefore, in each iteration of the algorithm, the new points are updated in the direction of the negative gradient, see Eq.5, so as to gradually decrease the value of the function and converge to the minimum.

$$p_k^{SD} = -\nabla F(x) \tag{5}$$

The magnitude of the update of the new points is determined by a step-length α, which is often chosen using a line search strategy such that it gives the largest decrease in the function value. The algorithm continues to make updates until the magnitude of the gradient is small enough, indicating that a local minimum has been reached, or until any stopping criteria is satisfied.

B. Newton method

In Section I, we have mentioned that it was also necessary to derive the hessian function in addition to the gradient. This is due to the fact that the Newton method is a second-order optimization method. The objective is always to find the minimum of a function, but the Newton method exploits both the gradient and the Hessian function. The first one is computed to obtain the steepest descent direction while the second order information represents the curvature of the function.

In this case, new points are updated using a direction that can be obtained as in Eq.6.

$$p_k^N = -(\nabla^2 F(x))^{-1}(\nabla F(x)) \tag{6}$$

The main advantage of this method is the quadratic local rate of convergence, however to ensure this, there are some requirements including that the Hessian function must be symmetric positive definite (SPD). The Modified Newton method can sometimes be a solution in case the SPD condition is not satisfied, but it will not be part of these experiments.

Apart from that, there is another point that may cause problems. The direction of the new step, in fact, requires that the inverse matrix of the hessian is computed, which can be highly demanding when dealing with large amounts of points. For this reason, we will exploit the properties of sparse matrices or algorithms, such as *gmres* or *pcg* in MATLAB, in order to obtain the inverse Hessian by solving a linear system.

As for Steepest Descent, also in this method the magnitude of the update of the parameters is determined by a step-length α, chosen using a line search strategy. The algorithm continues to make updates until the minimum point is reached or any stopping criteria is met.

C. Backtracking strategy

In the previous paragraphs we have mentioned the step-length α to set the magnitude of updates of the new points. The update is the following:

$$x_{k+1} = x_k + \alpha p_k \tag{7}$$

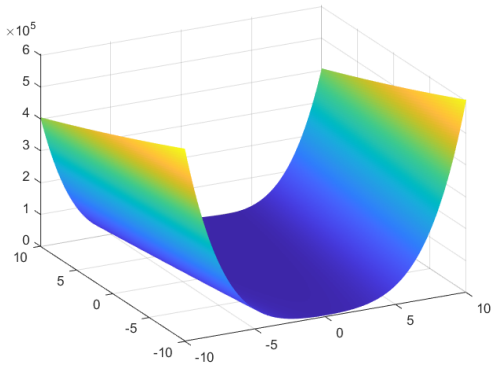
Thus, since p_k is a descent direction, the new point will be closer to the minimum at each iteration. However, the selection of a feasible step-length α is crucial. We want the function not to overshoot the minimum, therefore when the function increases instead of decreasing, we need to find an optimal step-length. Here the backtracking strategy comes to play. To check if the function has experienced a sufficient decrease we use the popular *Armijo condition*, Eq.8:

$$f(x_{k+1}) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^T p_k \tag{8}$$

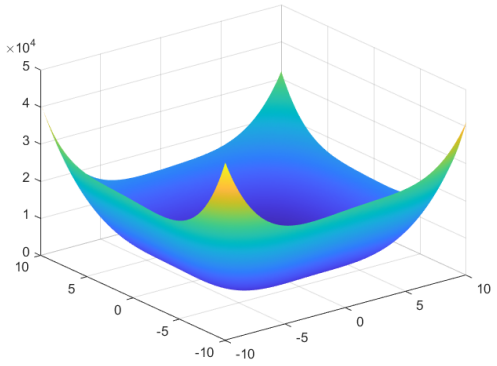
with $c_1 \in (0, 1)$, typically $c_1 = 10^{-4}$.

The algorithm follows two steps:

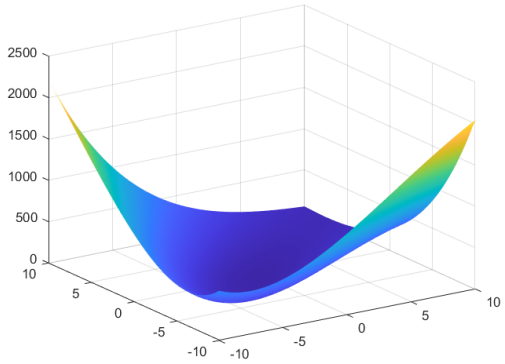
- choose an initial $\alpha_k^{(0)}$ (for Newton method the natural value is $\alpha_k^{(0)} = 1$);
- iteratively for $i \geq 0$, until the Armijo condition is satisfied, set $\alpha_k^{(i+1)} = \rho \alpha_k^{(i)}$,



(a) Extended Rosenbrock function in 3D



(b) Generalized Broyden tridiagonal function in 3D



(c) Problem 79 function in 3D

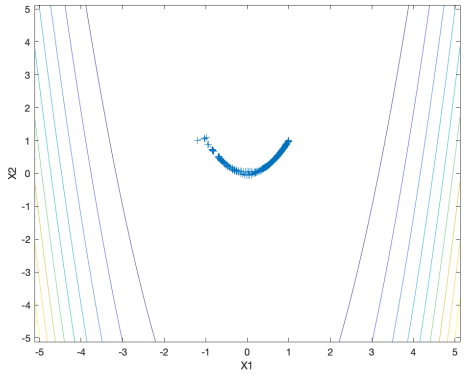
Fig. 1: 3D plots of the optimization problems

with $\rho \in]0, 1[$.

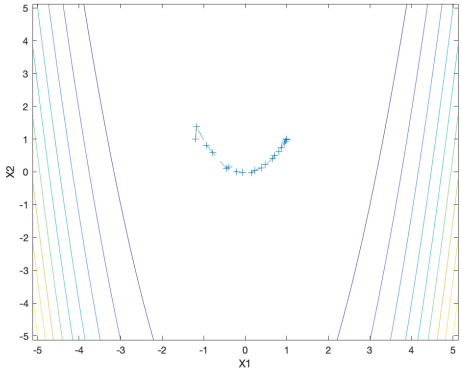
In summary, this way we decrease the step-length to ensure that the function assumes always smaller values.

III. Results

The tests on the Rosenbrock function were successful as both methods converged. The function was tested in two dimensions starting $x_0 = [-1.2, 1]$, while its minimum is located at $x_* = [1, 1]$. The results of these tests, including contour plots and step computations towards the minimum,

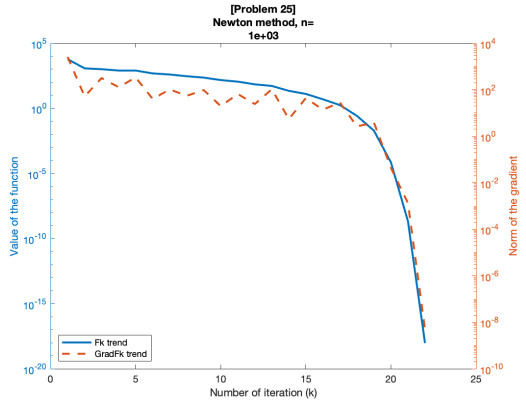


(a) Rosenbrock function contour plot and Steepest Descent steps



(b) Rosenbrock function contour plot and Newton Method steps

Fig. 2: Rosenbrock function contour plots



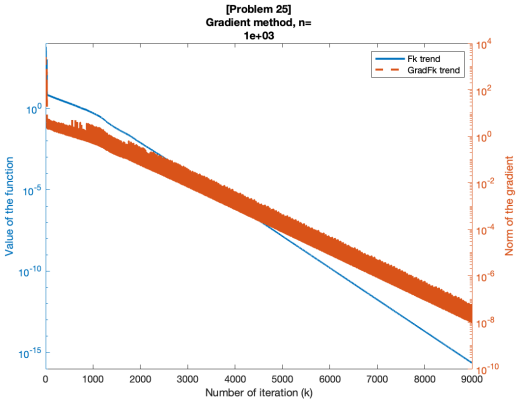


Fig. 3: Methods performances for $n = 10^3$

can be seen in Figure 2. Note that this function was tested with a limited number of dimensions.

Setting $\rho = 0.5$ and $c = 10^{-4}$ was an optimal choice for the convergence of Newton Method. It reached the minimum in $f(x_N) = 3.74 \cdot 10^{-21}$ with a norm of the gradient equal to $4.47 \cdot 10^{-10}$, obtained with 21 outer iterations and an average of 2 inner iterations for each outer, required by the iterative solver.

On the contrary, we noticed the steepest descent was doing really small steps and requiring lots of backtracking iterations to get to a value small enough to satisfy the Armijo condition. Hence $\rho = 0.1$ improved the iterations needed from 13000 to 932. It reached the minimum in $f(x_{SD}) = 9.21 \cdot 10^{-17}$ with a norm of the gradient equal to $8.58 \cdot 10^{-9}$.

All the other problems were tackled using $n = 10^k$ with $k = [3, 4]$.

A. *Problem 25 - Extended Rosenbrock*

In this subsection we present the results obtained minimizing the Extended Rosenbrock function reported in the equation 2. A 3D plot of the function can be seen in Fig.1a. The analytical gradient and the analytical hessian matrix is reported in Appendix A. We observed that the Newton method perform much better than the steepest descent in terms of number of iterations. This is noticeable, since Newton method requires only twenty iterations to reach a local minimum, while Steepest Descent requires about twenty thousands iterations. The latter method was then tested using a smaller ρ as before and the improvements were significant since, in this scenario, the number of iterations required is 9510.
Another crucial difference between the methods is the time required to converge. The Newton Method takes less than 1 second in both the tested dimensions number while the steepest takes respectively 4 seconds and 58 seconds. In figure 3 , the decrease of the function and the gradient norm is reported for both methods.

We observed an oscillating behaviour of the gradient in both the figures. This is due to the fact that the Armijo condition, used in the backtracking strategy, finds a suitable step-length which guarantees only a decrease for the function value, not for the gradient norm, so this behaviour was expected. The Steepest Descent oscillating behaviour is more pronounced since the algorithm takes lots of small steps of the gradient direction to get closer to the minimum. Finally, from our experiment we can affirm that both methods are suitable for this optimization problem since they both converged without

method	n	k	f(xk)	grad. norm	time(s)
NM	10^3	6	$7.16 \cdot 10^{-27}$	$5.3 \cdot 10^{-13}$	0.03
	10^4	6	$7.4 \cdot 10^{-26}$	$1.71 \cdot 10^{-12}$	0.03
SD	10^3	1339	$7.49 \cdot 10^{-18}$	$8.58 \cdot 10^{-9}$	0.2
	10^4	12398	$6.8 \cdot 10^{-18}$	$7.62 \cdot 10^{-9}$	8.7

TABLE I: Results for problem 32

method	n	k	f(xk)	grad. norm	time(s)
NM	10^3	10000	405	8.46	0.3
	10^4	10000	4050	19.05	1.83
SD	10^3	1043	$5.92 \cdot 10^{-17}$	$8.4 \cdot 10^{-9}$	0.055
	10^4	1075	$6.20 \cdot 10^{-17}$	$8.54 \cdot 10^{-9}$	0.886

TABLE II: Results for problem 79

failures, however the Newton method is more promising.

B. Problem 32 - Generalized Broyden tridiagonal

In this subsection we present the results obtained for the problem 32 as defined in equation 3. A 3D plot of the function can be seen in Fig.1b. The recommended starting point $x_0 = [-1, -1, ..., -1]$ was used for this problem. The analytical gradient and the Hessian were calculated and can be found in Appendix A. The Hessian computed resulted to be symmetric positive definite for every iteration so the full potential of Newton was exploited. On the other hand the Steepest Descent was again slower and failed to converge in the scenario of $n = 10^4$ wih kmax=10^4. However increasing kmax=10^5 allowed the method to reach the minimum. The results are reported in Table I.

C. Problem 79

In this subsection we present the results obtained for the problem 79, as defined in equation 4. A 3D plot of the function can be seen in Fig.1c. The recommended starting point $x_0 = [-1, -1, ..., -1]$ was used for this problem. The analytical gradient and the Hessian were calculated and can be found in Appendix A. The Newton method is known to converge to a local minimum only if the function is locally convex near the current iterate. This requires the second derivative of the function (the Hessian matrix) to be positive definite at that point. When the Hessian is positive definite, it provides an accurate approximation of the local curvature, allowing the Newton method to make informed steps towards the minimum. Unfortunately, during the experiment, the Newton Method failed to converge. Upon further analysis, it was found that some of the eigenvalues of the Hessian were not positive at any step, resulting in the failure of the method.
The results concerning both methods are reported in Table II.

The data reported shows that the Steepest Descent outperforms the Newton method in this case. The Newton Method was unable to converge in either of the dimensions tested, due to the lack of positive definiteness of the Hessian. This condition is crucial for the effectiveness of the Newton method and its ability to converge quickly to a solution. Despite tuning the hyperparameters, such as changing ρ, gradient tolerance, and maximum number of iterations, the method was still unable to reach the minimum.

IV. Discussion

We demonstrated how, when its requirements are satisfied, the Newton Method outperforms the Steepest Descent. This was predictable since it uses also higher order derivative information. We tested in problem 25 the ratio of convergence of the methods using the formula:

$$ratio = \frac{f^{(k-2)} - f^{(k-1)}}{f^{(k-1)} - f^{(k)}} \tag{9}$$

We were expecting a quadratic rate of convergence for the Newton Method (ratio=2) and linear for the Steepest Descent (ratio=1). Computing the median of ratios stored during the algorithms runs we obtained:

- *Ratio NM:* 2.24.
- *Ratio SD:* 1.13.

There are several strategies which can enhance the performances of these algorithms:

- Using a more sophisticated line search method to find the best step size at every iteration, even if this will increase the computational power and time needed.
- Using preconditioners when using PCG method to solve the linear system in Newton Method. This can avoid problems with ill-conditioned Hessians and bring better overall achievements.

References

[1] Ladislav Luksan, Jan Vlcek, "Test Problems for Unconstrained Optimization" Nov 2003.
[Online]. Available: https://www.researchgate.net/publication/325314497_Test_Problems_for_Unconstrained_Optimization

APPENDIX A
GRADIENTS AND HESSIANS IN N DIMENSIONS

A. Problem 25 - Extended Rosenbrock function

1) Gradient vector:

$$\nabla F(x) = \begin{bmatrix} \nabla_{x_1} F(x) \\ \vdots \\ \nabla_{x_n} F(x) \end{bmatrix} \qquad \nabla_{x_k} F(x) = \begin{cases} 200x_k^3 - 200x_k x_{k+1} + x_k - 1, & if mod(k,2) = 1 \\ 100x_k - 100x_{k-1}^2, & if mod(k,2) = 0 \end{cases} \tag{10}$$

2) Hessian matrix:

$$\nabla^2 F(x) = \begin{bmatrix} 600x_1^2 + 1 & -200x_1 & 0 & 0 & \dots & 0 & 0 \\ -200x_1 & 100 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 600x_3^2 - 200x_4 + 1 & -200x_3 & \dots & 0 & 0 \\ 0 & 0 & -200x_3 & 100 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 600x_{n-1}^2 - 200x_n + 1 & -200x_{n-1} \\ 0 & 0 & 0 & 0 & \dots & -200x_{n-1} & 100 \end{bmatrix} \tag{11}$$

B. Problem 32 - Generalized Broyden tridiagonal function

1) Gradient vector:

$$\nabla F(x) = \begin{bmatrix} \nabla_{x_1} F(x) \\ \vdots \\ \nabla_{x_k} F(x) \\ \vdots \\ \nabla_{x_n} F(x) \end{bmatrix} = \begin{bmatrix} (3 - 4x_1) f_1(x) - f_2(x) \\ \vdots \\ -f_{k-1}(x) + (3 - 4x_k) f_k(x) - f_{k+1}(x) \\ \vdots \\ -f_{n-1}(x) + (3 - 4x_n) f_n(x) \end{bmatrix} \tag{12}$$

where f_k is described in Eq.3.

2) Hessian matrix:

$$\nabla^2 F(x) = \begin{bmatrix} (3-4x_1)^2 - 4f_1(x) + 1 & -6 + 4x_1 + 4x_2 & 1 & 0 & 0 & \dots & 0 & 0 \\ -6 + 4x_1 + 4x_2 & (3-4x_2)^2 - 4f_2(x) + 2 & -6 + 4x_2 + 4x_3 & 1 & 0 & \dots & 0 & 0 \\ 1 & -6 + 4x_2 + 4x_3 & (3-4x_3)^2 - 4f_3(x) + 2 & -6 + 4x_3 + 4x_4 & 1 & 0 & \dots & 0 \\ 0 & 1 & -6 + 4x_3 + 4x_4 & (3-4x_4)^2 - 4f_4(x) + 2 & -6 + 4x_4 + 4x_5 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 1 & -6 + 4x_{n-1} + 4x_n & (3-4x_n)^2 - 4f_n(x) + 1 \end{bmatrix} \tag{13}$$

where f_k is described in Eq.3.

C. Problem 79 - Function

1) Gradient vector:

$$\nabla F(x) = \begin{bmatrix} \nabla_{x_1} F(x) \\ \vdots \\ \nabla_{x_k} F(x) \\ \vdots \\ \nabla_{x_n} F(x) \end{bmatrix} = \begin{bmatrix} (3 - \frac{x_1}{5}) f_1(x) - f_2(x) \\ \vdots \\ -2f_{k-1}(x) + (3 - \frac{x_k}{5}) f_k(x) - f_{k+1}(x) \\ \vdots \\ -2f_{n-1}(x) + (3 - \frac{x_n}{5}) f_n(x) \end{bmatrix} \tag{14}$$

where f_k is described in Eq.4.

2) Hessian matrix:

$$\nabla^2 F(x) = \begin{bmatrix} (3-\frac{x_1}{5})^2 - \frac{f_1(x)}{5} + 1 & -9 + \frac{2x_1}{5} + \frac{x_2}{5} & 2 & 0 & 0 & \dots & 0 & 0 \\ -9 + \frac{2x_1}{5} + \frac{x_2}{5} & (3-\frac{x_2}{5})^2 - \frac{f_2(x)}{5} + 5 & -9 + \frac{2x_2}{5} + \frac{x_3}{5} & 2 & 0 & \dots & 0 & 0 \\ 2 & -9 + \frac{2x_2}{5} + \frac{x_3}{5} & (3-\frac{x_3}{5})^2 - \frac{f_3(x)}{5} + 5 & -9 + \frac{2x_3}{5} + \frac{x_4}{5} & 2 & 0 & \dots & 0 \\ 0 & 2 & -9 + \frac{2x_3}{5} + \frac{x_4}{5} & (3-\frac{x_4}{5})^2 - \frac{f_4(x)}{5} + 5 & -9 + \frac{2x_4}{5} + \frac{x_5}{5} & 2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 2 & -9 + \frac{2x_{n-1}}{5} + \frac{x_n}{5} & (3-\frac{x_n}{5})^2 - \frac{f_n(x)}{5} + 4 \end{bmatrix} \tag{15}$$

where f_k is described in Eq.4.

APPENDIX B
SNIPPETS OF CODE

A. General code to apply the optmization algorithms and visualization (e.g. problem 25)

```
% 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 = 100; tolgrad = 1e-8;
btmax = 50; n_values = [1e3, 1e4];
alpha0=1;
% 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

disp('**** NEWTON METHOD WITH BACKTRACKING *****');
for j = 1:length(n_values)
   n = n_values(j);
   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
tic;
[xk, fk, gradfk_norm, k, xseq, btseq, gmres_it, ratio] = newton_bcktrck(x0, f, gradf,
    Hessf, ...
   kmax, tolgrad, c, rho, btmax,0);

elapsed_time = toc;
disp('************** RESULTS ***************');
disp(['f(xk): ', num2str(fk(end))]);
disp(['gradfk_norm: ', num2str(gradfk_norm(end))]);
disp(['N. of Iterations: ', num2str(k),'/',num2str(kmax), ';']);
disp(['ratio: ', num2str(median(ratio))]);
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 25]' 'Newton method, n=', num2str(n, '%.0e')});
   legend('Fk trend', 'GradFk trend', 'Location', 'southwest');
   xlabel('Number of iteration (k)');
   disp('---');
   % histogram plot of btseq values
```

```
   figure();
   histogram(btseq);
   title({'[Problem␣25]' 'Histogram␣backtracking␣iterations'...
      'Newton␣method␣n=', num2str(n, '%.0e')});
   xlabel('Number␣of␣backtracking␣iterations');
   ylabel('Newton␣method␣iterations');
   xticks(0:3);
   % bar plot of gmres number of iterations
   figure();
   bar(1:k, gmres_it);
   ylim([0, 5]);
   title({'[Problem␣25]' 'PCG␣iterations'});
   xlabel('Iterations␣of␣the␣Newton␣method');
   ylabel('Number␣of␣PCG␣iterations')
end



disp('***␣STEEPEST␣DESCENT␣WITH␣BACKTRACKING␣**');
kmax = 100000; alpha0 = 1; n_values = [1e3, 1e4];
rho=0.2;
for j = 1:length(n_values)
   n = n_values(j);
   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
tic;
[~, fk, gradfk_norm, k, ~, btseq, ratio] = ...
      steepest_desc_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(['ratio:␣', num2str(median(ratio))]);
   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␣25]' '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 25]' 'Histogram backtracking iterations,' ...
      'gradient method n=', num2str(n, '%.0e')});
   xlabel('Number of backtracking iterations');
   ylabel('Gradient method iterations');
   xticks(0:10)
end
```

B. Problem 25: Function, Gradient, Hessian

```
function Fx = problem_25_function(X)
% Function for computing the value F(x) where F is the extended rosenbrock
%
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% Fx: the value of the extended rosenbrock function at X
f_k_odd = @(x, k) 100 * (x(k).^2 - x(k+1)) .^ 2;
f_k_even = @(x, k) (x(k-1) - 1) .^ 2;
n = length(X);
Fx = 0;
for i=1:n
   if mod(i, 2) == 1
      Fx = Fx + f_k_odd(X, i);
   else
      Fx = Fx + f_k_even(X, i);
   end
end
Fx = Fx ./ 2;
end

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

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
```

C. Problem 32: Function, Gradient, Hessian

```
Fx = @(x) 0.5*sum(small_fx_32(x).^2);
gradf = @(x) problem_32_grad(x);
Hessf = @(x) problem_32_hess(x);

function small_fx_array = small_fx_32(x)
% creating the small f_k function
   n = length(x);
   small_fx_array = zeros(n,1);
   for k = 1:n
      small_fx_array(k) = (3*x(k) - 2*x(k).^2 + 1);
      if(k==1)
```

```
         small_fx_array(k) = small_fx_array(k) - x(k+1);
      elseif (k==n)
         small_fx_array(k) = small_fx_array(k) - x(k-1);
      else
         small_fx_array(k) = small_fx_array(k) - x(k-1) - x(k+1);
      end
   end
end

function gradFx = problem_32_grad(x)
% Function for computing the gradient vector at a specified point of the
% function reported in problem 32
% 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
   small_fx_array = small_fx_32(x);
   n = length(x);
   gradFx = zeros(n,1);
   gradFx(1) = (3 - 4*x(1))*small_fx_array(1) - small_fx_array(2);
   gradFx(n) = (3 - 4*x(n))*small_fx_array(n) - small_fx_array(n-1);
   for i = 2:(n-1)
      gradFx(i) = -small_fx_array(i-1) + (3 - 4*x(i))*small_fx_array(i) -
         small_fx_array(i+1);
   end
end

function HessFx = problem_32_hess(x)
% Function for computing the hessian matrix at a specified point of the
% function reported in problem 32
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% HessFx: n by n matrix which represent the Hessian at X
% We will exploit the sparsity of the matrix
   small_fx_array = small_fx_32(x);
   n = length(x);

   %Initialize vectors for sparse matrix:
   rows=zeros(5*n-6,1);
   cols=zeros(5*n-6,1);
   vals=zeros(5*n-6,1);
   count=3;
   % Compute the diagonal first and last element
   % HessFx(1,1) = (3 - 4*x(1)).^2 -4*small_fx_array(1) + 1;
   rows(1) = 1; cols(1) = 1; vals(1) = (3 - 4*x(1)).^2 - 4*small_fx_array(1) + 1;
   % HessFx(n,n) = 4 + (3 - 4*x(n)).^2 -4*small_fx_array(n);
   rows(2) = n; cols(2) = n; vals(2) = 1 + (3 - 4*x(n)).^2 - 4*small_fx_array(n);
   for i = 1:(n-1)
      if(i~=1)
         % Compute the diagonal
         % HessFx(i,i) = 5 + (3 - 4*x(i)).^2 -4*small_fx_array(i);
         rows(count) = i; cols(count) = i; vals(count) = 2 + (3 - 4*x(i)).^2 -4*
            small_fx_array(i);
         count = count+1;
      end
```

```
      % Compute the "second" diagonal
      % HessFx(i,i+1) = -2*(3 - 4*x(i))-3 - 4*x(i+1);
      rows(count) = i; cols(count) = i+1; vals(count) = -(3 - 4*x(i))-(3 - 4*x(i+1));
      count = count+1;
      % HessFx(i+1,i) = HessFx(i,i+1);
      rows(count) = i+1; cols(count) = i; vals(count) = -(3 - 4*x(i))-(3 - 4*x(i+1));
      count = count+1;

      % Compute the "third" diagonal
      if(i~=n-1)
         % HessFx(i,i+2) = 2;
         rows(count) = i; cols(count) = i+2; vals(count) = 1;
         count = count+1;
         % HessFx(i+2,i) = 2;
         rows(count) = i+2; cols(count) = i; vals(count) = 1;
         count = count+1;
      end
   end
   HessFx = sparse(rows,cols,vals,n,n);
end
```

D. Problem 79: Function, Gradient, Hessian

```
Fx = @(x) 0.5*sum(small_fx_79(x).^2);
gradf = @(x) problem_79_grad(x);
Hessf = @(x) problem_79_hess(x);

function small_fx_array = small_fx_79(x)
% creating the small f_k function
   n = length(x);
   small_fx_array = zeros(n,1);
   for k = 1:n
      small_fx_array(k) = (3*x(k) - 0.1*x(k).^2 + 1);
      if(k==1)
         small_fx_array(k) = small_fx_array(k) - 2*x(k+1);
      elseif (k==n)
         small_fx_array(k) = small_fx_array(k) - x(k-1);
      else
         small_fx_array(k) = small_fx_array(k) - x(k-1) - 2*x(k+1);
      end
   end
end

function gradFx = problem_79_grad(x)
% Function for computing the gradient vector at a specified point of the
% function reported in problem 79
% 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
   small_fx_array = small_fx_79(x);
   n = length(x);
   gradFx = zeros(n,1);
   gradFx(1) = (3 - 0.2*x(1))*small_fx_array(1) - small_fx_array(2);
   gradFx(n) = (3 - 0.2*x(n))*small_fx_array(n) - 2*small_fx_array(n-1);
   for i = 2:(n-1)
```

```
      gradFx(i) = -2*small_fx_array(i-1) + (3 - 0.2*x(i))*small_fx_array(i) -
         small_fx_array(i+1);
   end
end

function HessFx = problem_79_hess(x)
% Function for computing the gradient vector at a specified point of the
% function reported in problem 79
% INPUTS:
% X: n by 1 vector representing a point in the n-dimensional space
% OUTPUTS:
% HessFx: n by n matrix which represent the Hessian at X
% We will exploit the sparsity of the matrix
   small_fx_array = small_fx_79(x);
   n = length(x);

   %Initialize vectors for sparse matrix:
   rows=zeros(5*n-6,1);
   cols=zeros(5*n-6,1);
   vals=zeros(5*n-6,1);
   count=3;
   % Compute the diagonal first and last element
   % HessFx(1,1) = (3 - 0.2*x(1)).^2 -0.2*small_fx_array(1) + 1;
   rows(1) = 1; cols(1) = 1; vals(1) = (3 - 0.2*x(1)).^2 - 0.2*small_fx_array(1) + 1;
   % HessFx(n,n) = 4 + (3 - 0.2*x(n)).^2 -0.2*small_fx_array(n);
   rows(2) = n; cols(2) = n; vals(2) = 4 + (3 - 0.2*x(n)).^2 - 0.2*small_fx_array(n);
   for i = 1:(n-1)
      if(i~=1)
         % Compute the diagonal
         % HessFx(i,i) = 5 + (3 - 0.2*x(i)).^2 -0.2*small_fx_array(i);
         rows(count) = i; cols(count) = i; vals(count) = 5 + (3 - 0.2*x(i)).^2 -0.2*
            small_fx_array(i);
         count = count+1;
      end
      % Compute the "second" diagonal
      % HessFx(i,i+1) = -2*(3 - 0.2*x(i))-3 - 0.2*x(i+1);
      rows(count) = i; cols(count) = i+1; vals(count) = -2*(3 - 0.2*x(i))-(3 - 0.2*x(i
         +1));
      count = count+1;
      % HessFx(i+1,i) = HessFx(i,i+1);
      rows(count) = i+1; cols(count) = i; vals(count) = -2*(3 - 0.2*x(i))-(3 - 0.2*x(i
         +1));
      count = count+1;

      % Compute the "third" diagonal
      if(i~=n-1)
         % HessFx(i,i+2) = 2;
         rows(count) = i; cols(count) = i+2; vals(count) = 2;
         count = count+1;
         % HessFx(i+2,i) = 2;
         rows(count) = i+2; cols(count) = i; vals(count) = 2;
         count = count+1;
      end
   end
   HessFx = sparse(rows,cols,vals,n,n);
end
```