Constrained Optimization homework (problem 1)

Tommaso Monopoli
s278727

Please refer to the appendix A at the end of this document for all the Matlab codes of this project

1 Problem introduction

Consider the following constrained optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) = \sum_{i=1}^{n} i x_i^2 \\ \text{subject to} \quad & -5.12 \leq x_i \leq 5.12 \quad (i = 1, \dots, n) \end{aligned} \tag{1}$$

where $n = 10^d$ and $d = 3, 4, 5$.

This problem is strictly convex, in fact:

- $f(x)$ is a strictly convex function (since it is a sum of n strictly convex functions)
- the feasible set X is convex (it is a n-dimensional cube)

therefore, it has a unique solution (in particular, global minimum $f(x^*) = 0$ at $x^* = (0, \cdots, 0)$).

One of the many constrained optimization techniques to solve this problem is the projected gradient method (sect. 2).

At each iteration, the projected gradient method computes a descent direction exploiting the gradient of the function $f(x)$, i.e. $\nabla f(x)$. There are two ways of computing the gradient:

- exactly, by computing $\partial f(x)/\partial x_i$ analitically for each i (sect. 3.1)
- numerically, with the so-called finite differences approximation (sect. 3.2)

2 Projected Gradient Method

The projected gradient method is a generalization of the gradient method (aka steepest descent method) to constrained optimization problems.

The main idea of the method is the **projection of a point y onto a set X**:

$$\Pi_X(y) = \underset{x \in X}{\arg\min} \|x - y\|^2$$

that is, the point $x \in X$ closest to y.

Computing the projection of a point $y \notin X$ onto X is not an easy task in general (it requires solving a constrained minimization problem), but there are cases in which the structure of X makes the problem easy. For example, if X is a box in $\overline{\mathbb{R}}^n$, i.e. $X = \{x \in \mathbb{R}^n \mid L_i \leq x_i \leq U_i \quad \forall i = 1, \cdots, n\}$, then

$$[\Pi_X(y)]_i = \begin{cases} y_i & \text{if } L_i \leq y_i \leq U_i \\ L_i & \text{if } y_i < L_i \\ U_i & \text{if } y_i > U_i \end{cases}$$

In our setting, X is a n-cube with $L_i = -5.12$ and $U_i = +5.12 \quad \forall i = 1, \cdots, n$.

The **projected gradient method** implemented for this project is described hereafter.

For $k \geq 0$ (and until a stopping criterion is satisfied):

1. Pick an initial point $x_0 \in X$[1]. Set the tolerances for the stopping criteria:
    - $\varepsilon_{\text{grad}}$ (tolerance on the norm of the gradient),
    - $\varepsilon_{\text{diff}}$ (tolerance on the norm of the difference between consecutive iterates),
    - $k_{\max}$ (max n. of iterations)
2. Loop until a stopping condition is met:
    (a) (Steepest Descent Step) pick $-\nabla f(x_k)$ as descent direction and move along it with a steplength $\gamma_k > 0$

    $$\hat{x}_k = x_k - \gamma_k \nabla f(x_k)$$

    (b) (Projection Step) compute

    $$\overline{x}_k = \Pi_X(\hat{x}_k)$$

    (c) (Update) move along the direction $\overline{x}_k - x_k$ with a steplength $\alpha_k > 0$

    $$x_{k+1} = x_k + \alpha_k(\overline{x}_k - x_k)$$

    (d) (Stopping criterion) END if at least one of the following conditions is met:
    - $\|\nabla f(x_{k+1})\| < \varepsilon_{\text{grad}}$
    - $\|x_{k+1} - x_k\| < \varepsilon_{\text{diff}}$
    - $k \geq k_{\max}$

    otherwise increment k by 1 and go back to step 2.

The stopping condition $\|x_{k+1} - x_k\| < \varepsilon_{\text{diff}}$ is needed because in a constrained optimization problem a local minimum of f can be attained also in a non-stationary point (i.e. on ∂X).

The choice of the steplengths γ_k and α_k can be done in the following way:

- $\gamma_k = \gamma$ constant $\forall k$

[1] if x_0 is given, and it is *outside* X, then initialize $x_0 \longleftarrow \Pi_X(x_0)$

- $\alpha_k \in [0,1]$ chosen with a backtracking line search strategy: with fixed $\rho \in (0,1)$ and $c_1 \in (0,1)$
    1. start with $\alpha_k^{(0)} = 1$
    2. reduce $\alpha_k^{(j)} = \rho\alpha_k^{(j-1)}$ until you satisfy the Armijo condition

$$f\left(x_k + \alpha_k^{(j)}(\overline{x}_k - x_k)\right) \leq f(x_k) + c_1\alpha_k^{(j)}\nabla f(x_k)^\top(\overline{x}_k - x_k)$$

In all experiments, the maximum number of (inner) iterations of the backtracking line search is bounded to a fixed value $k_{\text{max}}^{\text{BT}}$.

3 Experiment and results

We can clearly see that $f(x)$ is additively separable, in the sense that $f(x) = f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n)$, and moreover X is defined by a set of n one-variable inequality constraints, $-5.12 \leq x_i \leq +5.12$, so we could easily solve this optimization problem by dividing it in n smaller constrained optimization problems, which can be solved independently and yield one component x_i^* of the global minimizer x^*.

However, I decided not to solve this problem in such a trivial way, for the sake of generality. Instead, I applied the projected gradient method directly to f.

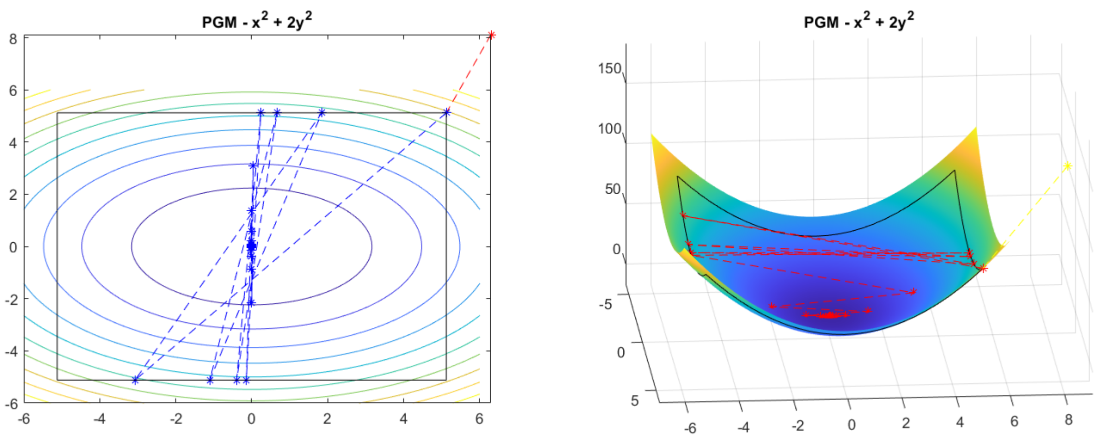
In all my experiments, I fixed these parameters:

- $c_1 = 10^{-4}$
- $\rho = 0.8$
- $k_{\text{max}}^{\text{BT}} = 100$
- $\varepsilon_{\text{grad}} = 10^{-5}$
- $\varepsilon_{\text{diff}} = 10^{-5}$
- $k_{\text{max}} = 3000$

The initial point x_0 is chosen randomly: each component $x_{0,i}$ is chosen uniformly at random in the interval $[-10, 10]$; therefore, for large n it is very likely that we will need to project x_0 onto X in order for the algorithm to begin. I decided to fix a random seed, for replicability purposes; this means that, given the dimension n, the initial point is always the same.

As for the Projection Step steplength γ, the choice can be done by manual tuning (several values are tried: $\gamma = 1,\ 0.9,\ 0.8$).

To visualize how the method behaves, I applied it in the very simple case $n = 2$, with $\gamma = 0.8$. In fig. 1 a 2D plot and 3D plot of f is reported, along with the sequence of iterates leading to the minimum of f.



(a) Contour lines of f (b) Surface of f

Figure 1: 2D and 3D plots showing how the method behaves when $n = 2$

3.1 Analytic $\nabla f(x_k)$

As mentioned in the introduction, we can apply the projected gradient method to our problem by computing the exact gradient of $f(x)$ analytically

$$[\nabla f(x)]_i = 2\,i\,x_i \quad \forall i \qquad \Longrightarrow \qquad \nabla f(x) = (2x_1,\ 4x_2,\ 6x_3,\ \cdots)$$

I have run the projected gradient method with exact gradient for $n = 10^3, 10^4, 10^5$ and for several values of γ. Results are reported in table 1.

n	γ	#iterations	#projections	$\|\nabla f(x_K)\|$	$\|x_K - x_{K-1}\|$	$f(x_K)$
10^3	1	359	34	0.0097034	9.8045e-06	2.4539e-08
	0.9	68	39	0.0066817	7.724e-06	4.8023e-08
	0.8	362	33	0.0098027	9.9055e-06	2.6611e-08
10^4	1	76	59	0.059506	7.3575e-06	4.7095e-07
	0.9	105	43	0.09199	9.7045e-06	4.0859e-07
	0.8	79	62	0.0723	8.9619e-06	5.0719e-07
10^5	1	85	60	0.75082	8.4509e-06	1.9125e-06
	0.9	104	93	0.64921	8.5169e-06	1.6418e-06
	0.8	86	60	0.7542	8.4832e-06	2.234e-06

Table 1: PGM with exact gradient

It is evident that the speed of convergence of the algorithm is heavily dependent on the choice of γ. However, with a lucky choice of γ, the algorithm converges to the minimum in less than 100 iterations, for all n.

It is interesting to note that the number of (inner) iterations of the backtracking strategy to find α_k is increasing with the number k of outer iterations, and then stabilizes towards the end of the algorithm. In fig. 2 this behaviour is clearly shown (for $n = 10^3$ and $\gamma = 0.9$). This is because the function is steeper far from the minimum point (where early iterates live), and flatter near to the minimum point (and therefore the Armijo condition is satisfied only with smaller α_k, until this need of having a smaller α_k is counterbalanced by $\|\nabla f(x_k)\|$ too becoming very small).

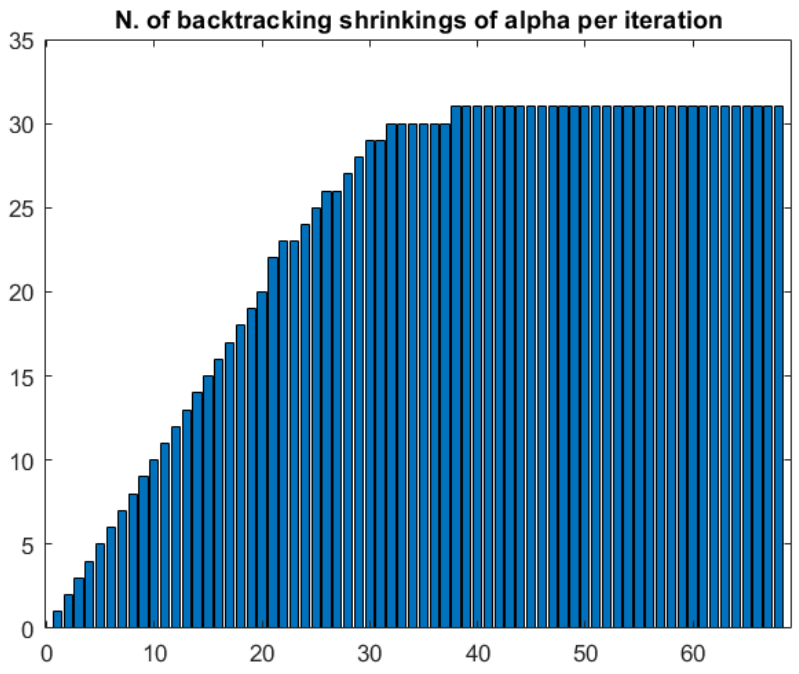


Figure 2: Number of backtracking iterations per outer iteration (for $n = 10^3$ and $\gamma = 0.9$)

3.2 Numerical $\nabla f(x_k)$ (finite differences approximation)

Instead of computing the exact $\nabla f(x)$ analytically, we can approximate it using finite differences approximations of the derivatives.

For our experiment, three versions of the finite differences approximation were employed:

- Forward differences: $\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x)}{h} \quad \forall i = 1, \dots, n$
- Backward differences: $\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x)-f(x-he_i)}{h} \quad \forall i = 1, \dots, n$
- Centered differences: $\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x-he_i)}{2h} \quad \forall i = 1, \dots, n$

The choice of the increment h is of critical importance:

- h too large $\implies$ high analytical error in the estimation of the gradient
- h too small $\implies$ numerical cancellation issues (because the differences $f(x+he_i) - f(x)$, $f(x) - f(x - he_i)$, $f(x + he_i) - f(x - he_i)$ will be very small)

For our problem, we will experiment with an increment $h = 10^{-k}\|x\|$, where $k = 2, 4, 6, 8, 10, 12$.

One issue I encountered, especially when applying the method in a high dimensional space ($n = 10^4, 10^5$), is that computing the gradient with these formulas is very computationally expensive: gradient computation would require n (or $2n$, in the centered differences case) function evaluations for each iteration of the PGM. Therefore, I decided to exploit the particularly simple structure of $f(x)$ to rewrite these formulas in a simpler, more computationally-friendly way.

For the forward difference formula:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x)}{h} = \frac{\left(\sum\limits_{j=1}^{i-1} jx_j^2 + i(x_i+h)^2 + \sum\limits_{j=i+1}^{n} jx_j^2\right) - \sum\limits_{j=1}^{n} jx_j^2}{h} =$$
$$= \frac{i(x_i+h)^2 - ix_i^2}{h} = i\frac{2x_ih+h^2}{h} = \mathbf{i(2x_i+h)}$$

Similarly, for the backward difference formula:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x)-f(x-he_i)}{h} = \frac{\sum\limits_{j=1}^{n} jx_j^2 - \left(\sum\limits_{j=1}^{i-1} jx_j^2 + i(x_i-h)^2 + \sum\limits_{j=i+1}^{n} jx_j^2\right)}{h} =$$
$$= \frac{ix_i^2 - i(x_i-h)^2}{h} = i\frac{2x_ih-h^2}{h} = \mathbf{i(2x_i-h)}$$

Finally, for the centered difference formula:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x-he_i)}{2h} =$$
$$= \frac{\left(\sum\limits_{j=1}^{i-1} jx_j^2 + i(x_i+h)^2 + \sum\limits_{j=i+1}^{n} jx_j^2\right) - \left(\sum\limits_{j=1}^{i-1} jx_j^2 + i(x_i-h)^2 + \sum\limits_{j=i+1}^{n} jx_j^2\right)}{2h} =$$
$$= \frac{i(x_i+h)^2 - i(x_i-h)^2}{2h} = i\frac{4x_ih}{2h} = \mathbf{2ix_i}$$

With these three formulas[2], the numerical computation of the gradient is almost as fast as the analytic computation. Surprisingly, in the centered differences case the numerical gradient corresponds exactly to the analytic one, whatever h we choose. Therefore, approximating the gradient with the centered differences formula is (theoretically) equivalent to computing it analytically, and so we would obtain the same results already reported in table 1.

Moreover, note that with these new formulas any numerical cancellation problem caused by a very low value of the increment h is avoided.

Nonetheless, for didactic reasons (i.e. for showing off the typical numerical problems arising when choosing h too low), I have employed the usual finite differences formulas whenever computationally feasible, i.e. for $n = 10^3$, and used the new formulas only for the problematic cases $n = 10^4, 10^5$.

[2]Obviously, this mathematical "trick" is problem-dependent (I exploited the nice structure of $f(x)$) and in no way generalizable to any $f(x)$.

Results are reported in tables 2, 3 and 4 (see page 7). This time, I fixed $\gamma = 0.9$ once and for all, and reported only the n. of iterations and n. of projections for reasons of brevity.

The results are interesting in a number of ways:

- with an adequate choice of h (not too large nor to small), the algorithm converges in a number of iterations very similar to the case with the analytic gradient (despite taking significantly more computational time)
- results in **red** show a very high n. of iterations to converge; this is most likely caused by the value of h being too large, so the numerical gradient at each step is inaccurate w.r.t. the analytic one (so the algorithm is not taking the direction of steepest descent)
- results in **violet** show a very high n. of iterations to converge; since this happens only for $n = 10^3$, this is most likely caused by numerical cancellation issues, which make the numerical gradient again very inaccurate w.r.t. the analytic one
- results in **blue** show a surprisingly low n. of iterations to converge; I verified that in all these cases, the stopping criterion on $\|x_{k+1} - x_k\|$ is met, even though $f(x_k)$ is far from the minimum. This strange behavior might be explained by the fact that the numerical gradient is so inaccurate (due to the very high h) that its opposite is a very "mild" descent direction or not a descent direction at all (in both cases, the backtracking line search yields a very small α_k, therefore $x_{k+1} \approx x_k$)

n	k	#iterations	#projections
10^3	2	**2071**	38
	4	69	39
	6	68	39
	8	68	39
	10	**164**	39
	12	**2573**	39
10^4	2	**3000**	1140
	4	105	43
	6	105	43
	8	105	43
	10	105	43
	12	105	43
10^5	2	**2**	3
	4	**151**	138
	6	98	87
	8	102	91
	10	104	93
	12	104	93

Table 2: PGM with numerical gradient (forward differences approximation)

n	k	#iterations	#projections
10^3	2	**1987**	43
	4	68	39
	6	68	39
	8	68	39
	10	**433**	39
	12	**2117**	39
10^4	2	**53**	51
	4	106	43
	6	105	43
	8	105	43
	10	105	43
	12	105	43
10^5	2	**2**	3
	4	**167**	152
	6	104	93
	8	104	93
	10	104	93
	12	104	93

Table 3: PGM with numerical gradient (backward differences approximation)

n	k	#iterations	#projections
10^3	2	68	39
	4	68	39
	6	68	39
	8	68	39
	10	68	39
	12	**624**	39
10^4	2	105	43
	4	105	43
	6	105	43
	8	105	43
	10	105	43
	12	105	43
10^5	2	104	93
	4	104	93
	6	104	93
	8	104	93
	10	104	93
	12	104	93

Table 4: PGM with numerical gradient (centered differences approximation)

A Matlab code

In this section is reported the Matlab code produced for this project

A.1 pgm_test.m

```
clear
close all
clc

%% DEFINE THE PROBLEM

n = 10^3;   % n. of variables
f = @(x) [1:n] * (x.^2);     % f. handle for the objective function
gradf = @(x) [1:n]' .* (2*x);    % f. handle for the gradient of the ...
    objective function
box_mins = -5.12 * ones(n,1);   % upper boundaries of the feasible box
box_maxs = 5.12 * ones(n,1);    % lower boundaries of the feasible box



%% DEFINE PARAMETERS FOR THE PROJECTED GRADIENT METHOD

rng(0);    % set seed (for replicability)
x0 = unifrnd(-10,10,n,1);

% for the projection step
gamma = 0.9; % 0.8, 0.9, 1
Pi_X = @(x) max(min(x, box_maxs), box_mins);  % f. handle for the ...
    box projection function

% For the purpose of comparing the projected gradient method and the
% unconstrained gradient method, here is a function handle for an ...
    identity
% projection function
%Pi_X = @(x) x;

% for the backtracking line search
c1 = 1e-4;
rho = 0.8;
btmax = 100;

% for the stopping criterion
tolgrad = 1e-5;      % iterate until ||gradf(x(k))|| < tolgrad
tolx = 1e-5;         % iterate until ||x(k+1)-x(k)|| < tolx
kmax = 3000;

% Choose how to compute the gradient
% - 'forw'  --> approximate grad(f) with the forward difference ...
    approximation
% - 'backw' --> approximate grad(f) with the backward difference ...
    approximation
% - 'centr' --> approximate grad(f) with the centered difference ...
    approximation
% - 'exact' (or just every other string) --> use the exact grad(f)
```

```
FDgrad = 'exact';
k_FDgrad = 2; % the exponent which appears in the expression of the ...
    increment h=10^(-k)*||x||

verbose = true;      % if true, print the current iteration every 10



%% RUN THE PGM

disp('**** CONSTR. STEEPEST DESCENT: START *****')
tic     % start timer

[xk_final, fk_final, gradfk_norm_final, deltaxk_norm_final, k_final, ...
    xseq_final, btseq_final, projection_count] = ...
    projected_gradient_method(x0, f, gradf, kmax, tolgrad, c1, rho, ...
    btmax, gamma, tolx, Pi_X, verbose, FDgrad, k_FDgrad);

disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
disp('')

disp('**** CONSTR. STEEPEST DESCENT: RESULTS *****')
disp('************************************')
toc     % stop timer
if n==2
    disp(['xk: ', mat2str(xk_final), ' (actual minimizer: vector of ...
        0s);'])
end
disp(['norm(xk): ', mat2str(norm(xk_final)), ' (actual norm of the ...
    minimizer: 0; NB: this is also the norm of the error, ||xk-x*||);'])
disp(['f(xk): ', num2str(fk_final), ' (actual min. value: 0);'])
disp(['N. of iterations: ', num2str(k_final),'/',num2str(kmax), ';'])
disp(['N. of projections: ', num2str(projection_count), ';'])
disp(['gradient norm: ', num2str(gradfk_norm_final), ';'])
disp(['length of last step: ', num2str(deltaxk_norm_final), ';'])
disp('************************************')



%% PLOTS

% Barplot of btseq
fig_bt_iters = figure();
bar(btseq_final)
title('PGM - x^2 + 2y^2')

if n==2     % in this very simple case, we can visualize the ...
    objective function in 3D

    % Creation of the data to plot the domain boundaries
    t = linspace(0, 1, 25);
    dom_xy_1 = box_mins + t .* ([box_mins(1); box_maxs(2)] - box_mins);
    dom_xy_2 = [box_mins(1); box_maxs(2)] + t .* (box_maxs - ...
        [box_mins(1); box_maxs(2)]);
    dom_xy_3 = box_maxs + t .* ([box_maxs(2); box_mins(1)] - box_maxs);
    dom_xy_4 = [box_maxs(2); box_mins(1)] + t .* (box_mins - ...
        [box_maxs(2); box_mins(1)]);

    dom_xy = [dom_xy_1, dom_xy_2, dom_xy_3, dom_xy_4];
    f_z = f(dom_xy);

    % Projection of the starting point
    Pi_X_x0 = Pi_X(x0);

    % Creation of the meshgrid for the contour-plot
    [X, Y] = meshgrid(linspace(-6, 6, 500), linspace(-6, 6, 500));

    % Computation of the values of f for each point of the mesh
    Z = X.^2 + 2*Y.^2;

    % Simple Plot
    fig_contour = figure();
    % Contour plot with curve levels for each point in xseq
    [C1, ~] = contour(X, Y, Z);
    hold on
    % plot of the points in xseq
    plot([x0(1), Pi_X_x0(1)], [x0(2), Pi_X_x0(2)], 'r--*')
    plot([Pi_X_x0(1) xseq_final(1, :)], [Pi_X_x0(2) xseq_final(2, ...
        :)], 'b--*')
    plot(dom_xy(1, :), dom_xy(2, :), 'k')
    hold off
    title('PGM - x^2 + 2y^2')

    % Much more interesting plot
    fig_surface = figure();
    surf(X, Y, Z,'EdgeColor','none')
    hold on
    plot3([x0(1) Pi_X_x0(1)], [x0(2) Pi_X_x0(2)], [f(x0), ...
        f(Pi_X_x0)], 'y--*')
    plot3([Pi_X_x0(1) xseq_final(1, :)], [Pi_X_x0(2) xseq_final(2, ...
        :)], [f(Pi_X_x0), f(xseq_final)], 'r--*')
    plot3(dom_xy(1, :), dom_xy(2, :), f_z, 'k')
    hold off
    title('PGM - x^2 + 2y^2')

end
```

A.2 projected_gradient_method.m

```
function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq, ...
    projection_count] = ...
    projected_gradient_method(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, verbose, ...
        FDgrad, k_FDgrad)
%
% Projected gradient method (steepest descent) for constrained ...
    optimization.
%
% 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;
% 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, less than 1, used for reducing alpha0;
% btmax = maximum number of steps for updating alpha during the
%         backtracking strategy.
% gamma = the initial factor that multiplies the descent direction ...
    at each
%         iteration;
% tolx = value used as stopping criterion w.r.t. the norm of the
%        steps (xnew - xk); infact, in constrained optimization the ...
    global
%        minimizer could also be a non-stationary point, so the stopping
%        criterion based on a tolerance on the norm of the gradient ...
    is not
%        sufficient attained in a non-stationary point
% Pi_X = projection function
% verbose = if true, print the current iteration every 10
% FDgrad = choose how to compute the gradient (exactly or approximately)
% k_FDgrad = the exponent which appears in the expression of the ...
    increment
%            h=10^(-k_FDgrad)*||x||
%
% OUTPUTS:
% xk = the last x computed by the function;
% fk = the value f(xk);
% gradfk_norm = value of the norm of gradf(xk)
% deltaxk_norm = length of the last step of the sequence
% 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 'forw'
        % overwrite gradf with a f. handle that uses the forward ...
            difference
        % approximation
        h = @(x) (10^(-k_FDgrad))*norm(x);
        n = size(x0,1);

        if n < 10^4
            % general-purpose forw. diff. approx.
            gradf = @(x) (f(x + h(x)*eye(n)) - f(x))' / h(x);
        else
            % problem-dependent forw. diff. approx.
            gradf = @(x) [1:n]' .* (2*x + h(x));
        end

    case 'backw'
        % overwrite gradf with a f. handle that uses the backward ...
            difference
        % approximation
        h = @(x) (10^(-k_FDgrad))*norm(x);
        n = size(x0,1);

        if n < 10^4
```

```
            % general-purpose backw. diff. approx.
            gradf = @(x) (f(x) - f(x - h(x)*eye(n)))' / h(x);
        else
            % problem-dependent backw. diff. approx.
            gradf = @(x) [1:n]' .* (2*x - h(x));
        end

    case 'centr'
        % overwrite gradf with a f. handle that uses the centered ...
           difference
        % approximation
        h = @(x) (10^(-k_FDgrad))*norm(x);
        n = size(x0,1);

        if n < 10^4
            % general-purpose centr. diff. approx.
            gradf = @(x) (f(x + h(x)*eye(n)) - f(x - h(x)*eye(n)))' ...
               / (2*h(x));
        else
            % problem-dependent centr. diff. approx.
            gradf = @(x) [1:n]' .* (2*x);   % exact gradient!
        end

    otherwise
        % we use the input function handle gradf
end

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

% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);
projection_count = 0;

xk = Pi_X(x0); % Project the starting point if outside the constraints
if ~isequal(xk, x0)
    projection_count = projection_count + 1;
end
fk = f(xk);

% compute the gradient (exactly or approximately, depending on FDgrad)
gradfk = gradf(xk);

k = 0;
gradfk_norm = norm(gradfk);
deltaxk_norm = tolx + 1;    % this is to ensure that at least a ...
   first iteration is performed

if verbose
    disp(['iteration: ', num2str(k)])
    disp(['norm(xk): ', num2str(norm(xk))])
    disp(['gradient norm: ', num2str(gradfk_norm)])
    if strcmp(FDgrad,'forw') || strcmp(FDgrad,'backw') || ...
       strcmp(FDgrad,'centr')
        disp(['h(xk): ', num2str(h(xk))])
    end
end
```

```

while k < kmax && gradfk_norm >= tolgrad && deltaxk_norm >= tolx
    % Compute the descent direction (exactly or approximately, ...
        depending on FDgrad)
    pk = -gradf(xk);

    % Take a step in the descent direction and project the resulting ...
        vector
    % onto the feasible set X
    xhatk = xk + gamma * pk
    xbark = Pi_X(xhatk);
    if ~isequal(xbark, xhatk)
        projection_count = projection_count + 1;
    end


    % Backtracking line search

    % Reset the value of alpha
    alpha = 1;

    % Compute the candidate new xk
    pik = xbark - xk;
    xnew = xk + alpha * pik;

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

    bt = 0;      % counter of backtracking iterations

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

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

    end

    % Update xk, fk, gradfk_norm, deltaxk_norm
    deltaxk_norm = norm(xnew - xk);
    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;

    if verbose && mod(k,10) == 0
        disp(['iteration: ', num2str(k)])
        disp(['norm(xk): ', num2str(norm(xk))])
        disp(['gradient norm: ', num2str(gradfk_norm)])
        if strcmp(FDgrad,'forw') || strcmp(FDgrad,'backw') || ...
            strcmp(FDgrad,'centr')
            disp(['h(xk): ', num2str(h(xk))])
        end
    end
end

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

end
```