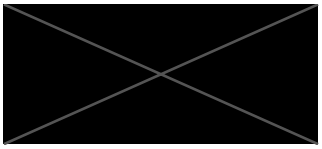
Constrained Opitimization Problem



Abstract: **Optimization is an important tool in decision science and in the analysis of physical systems. There are several types of these problems in which one is constrained optimization problems. In this report, we address optimization of De Jong function through Projected Gradient method using gradient-based methods: steepest-descent with backtracking strategy comparing with Newton method. We perform this method applying both exact derivatives and finite differences to approximate the gradient**.

I. PROBLEM DEFINITION:

A. Main Test Function:

The simplest of De Jong's functions is the so-called sphere function

$$f(x) = \sum_{i=1}^{n} x_i^2 \ , \quad -5.12 \leq x_i \leq 5.12 \qquad (1)$$

whose global minimum is obviously f∗ = 0 at (0,0, ..., 0). This function is unimodal and convex. A related function is the so-called weighted sphere function or hyper-ellipsoid function

$$f(x) = \sum_{i=1}^{n} i x_i^2 \ , \ -5.12 \leq x_i \leq 5.12 \qquad (2)$$

which is also convex and unimodal with a global minimum f∗ = 0 at x∗ = (0,0, ..., 0). The main function is equation 2 with related constrain.

B. Steepest Descent and Projected Gradient:

Let the function $f: R^n \to R$ be given that have to be minimized with respect to a set $X \in R^n$ of constrains, i.e. we want to find

$$argmin_{x \in R} f(x)$$

The Projected Gradient method applied to the steepest descent is characterized by the following main steps (at the k-th iteration):

- compute the steepest descent direction $p_k = - \nabla f(x_k)$
- given an arbitrary sequence of $\gamma_k > 0$, we check if $\bar{x} = (x_k + \gamma_k p_k)$ is in X or not; in particular, we compute

$$\widehat{x_k} = \Pi_X(\overline{x_k}) := \begin{cases} \overline{x_k}, & if\ \overline{x_k} \in X \\ \Pi_X(\overline{x_k}), & otherwise \end{cases}$$

- where $\Pi_X(\overline{x_k})$ is the projection of $\overline{x_k}$ on the boundary ∂X of X.
- compute the feasible direction $\pi_k = (\overline{x_k} - x_k)$.
- compute the next step of the optimization method: $x_{k+1} = x_k + \alpha_k \pi_k$, where $\alpha_k \in [0,1]$ is the sequence of step length factors.

It should be noted that since the minimum not necessarily corresponds to a point with null gradient, a second stopping criterium with respect to the incrementation step $||x_{k+1} - x_k||$ is necessary that shows the length of any step towards the optimum point.

C. Line Search and values of γ_k and α_k along the feasible direction π_k:

we assume a constant sequence $\gamma_k \equiv \gamma > 0$, for each k ∈ N, and we perform a line search method with respect to α_k along the direction π_k: (starting from the value α_k (0) = 1).

D. Projection Functions:

In this report, function Π_X with X is a n-dimensional box in R^n. Let X be the n-dimensional box [m1; M1] × · · · × [mn; Mn] ⊂ R^n. Then, it holds

$$\Pi_X(x) = [\Pi_{[m_1,M_1]}(x_1), \ldots, \Pi_{[m_n,M_n]}(xn)]^T$$

where

$$\Pi_{[m_i,Mi]}(x_i) = \begin{cases} m_i & if\ x_i < m_i \\ M_i & if\ x_i > M_i. \\ x_i & otherwise \end{cases}$$

E. Steepest Descen (SDM)t:

Let the function $f: R^n \to R$ be given. The steepest descent method is an iterative optimization method that, starting from a given vector $x_0 \in R^n$, computes a sequence of vectors $\{x_k\}_{k\in N}$characterized by equation 3, where the descent direction p_k is the steepest one, i.e. $p_k = -\nabla f(x_k)$, and the step length factor $\alpha \in R$ is arbitrarily chosen.

$$x_{k+1} = x_k + \alpha p_k \,, \forall\ k \geq 0 \qquad (3)$$

F. Newton Method (NM):

Let the function $f: R^n \to R$ be given. The Newton method is an iterative optimization method that, starting from a given vector $x_0 \in R^n$, computes a sequence of vectors $\{x_k\}_{k\in N}$characterized by again equation 4, where the descent direction p_k is the solution of the linear system $H_f(x_k)p_k = -\nabla f(x_k)$ and ∇f, H_f are the gradient and the Positive Definite (PD) Hessian matrix of f, respectively.

G. Backtracking:

Let the function $f: R^n \to R$ be given. The backtracking strategy for an iterative optimization method consists of a value α_k satisfying the Armijo condition at each step k of the method, i.e.

$$f(x_{k+1}) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T p_k$$
$$x_{k+1} = x_k + \alpha p_k \,, \forall\ k \geq 0$$

where $c_1 \in (0,1)$ (typically, the standard choice is $c_1 = 10^{-4}$). The backtracking strategy is an iterative process that looks for the value α_k. Given an arbitrary factor $\rho \in (0,1)$ and an arbitrary starting value $\alpha_k^{(0)}$ for α_k, we decrease iteratively $\alpha_k^{(0)}$, multiplying it by ρ, until the Armijo condition is satisfied. Then $\alpha_k = \rho^{t_k}\alpha_k^{(0)}$, for a $t_k \in N$, if it satisfies Armijo but $\alpha_k = \rho^{t_k-1}\alpha_k^{(0)}$, does not.

II. IMPLEMENTATION AND PARAMETERS:

To perform our methods, we need to define some parameters so that the generalization of the problem is satisfied. The evaluation of the methods has been performed for n-dimensional input with starting point $X = 7$ to be out of the box at the beginning. For this purpose, we consider three values: 10^3, $10^4$and 10^5 for the dimensions. There are two stopping criteria: maximum iteration = 3000 and tolerance = 10^{-12} (this is the threshold for gradient norm). There are also stopping criteria for backtracking strategy: maximum iteration = 50 and Armijo condition.

To do this, we take into account following considerations:

- The Armijo condition is often satisfied for very small values of α. Then, it is not enough to ensure that the algorithm makes reasonable progress; indeed, if α is too small, unacceptably short steps are taken.
- The choice of $\alpha_k^{(0)}$ is method-dependent, but it is crucial to choose $\alpha_k^{(0)} = 1$ in Newton method for (possibly) get the second-order rate of convergence, we also set α =1.
- To select best ρ in terms of generalization for high dimensions problems, we have performed with constant $c_1 = 10^{-4}$ and $\rho = 0.8$.

In addition, we use exact and finite differences methods to approximate the gradient. In exact derivates, we apply the precise gradient of the function ∇f to compute the derivatives. However, in finite differences, we consider here a C^1 loss function $f: R^n \to R$. Therefore, the gradient ∇f is a vectorial function

$$G: = \nabla f = \begin{bmatrix} f_{x_1} \\ \cdot \\ \cdot \\ \cdot \\ f_{x_n} \end{bmatrix} : R^n \to R^n$$

where f_{x_i} denotes the partial derivative of f with respect to x_i, for each i = 1,… n. Two types of finite different derivatives have been considered for SDM: forward and centered finite differences and centered one just for NM, in equations 4, 5 and 6 (just for diagonal), respectively.

$$f_{x_i}(x) = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x)}{h}, \quad \forall\, i = 1,\ldots n. \tag{4}$$

$$f_{x_i}(x) = \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x-he_i)}{2h}, \ \forall\, i = 1,\ldots n. \tag{5}$$

$$H_{f\,ii}(x) \approx \frac{f(x+he_i)-2f(x)+f(x-he_i)}{\mathrm{h}^2}, \ \forall\, i = 1,\ldots n. \tag{6}$$

Large values for h return obviously poor approximations, but too small values are characterized by numerical cancellation problems. We use following values for the increment h:

$$h = 10^{-k}\left|\left|\hat{x}\right|\right|, \qquad k = 2, 4, 6, 8, 10, 12$$

where x̂ is the point at which the derivatives have to be approximated. Furthermore, we simplified the equations using the structure of the function to reduce computation time and it can be seen that the gradient is independent from h for centered process and hessian is diagonal matrix which is also independent from h as below:

$$\frac{\partial f(x)}{\partial x_i} = \begin{cases} i(2x_i + h) & for\ forward \\ 2ix_i & for\ centered \end{cases}$$

$$Hessian\_f = \begin{cases} 2i & diagonal \\ 0 & otherwise \end{cases}$$

III. DISCUSSION:

By applying the SDM algorithm to our function using exact derivate method, we have reached to satisfactory values for dimension 10^3 but as the dimension increases, the approximation becomes worst in most aspects. Although, SDM stops due to iteration criteria and never reaches a gradient norm lower than threshold with 3000 iteration. In figure 1, the inner iteration for Armijo condition is 21 which means alpha is 0.8^{21} for outer iteration, besides, it shows that how the initial point is projected into the box (red dash-line) and then moves to the optimum point (blue star-line). It should be mentioned that the outcomes can be improved by changing in number of maximum iterations or starting point and other parameters but we have used these parameters due to the limitation of PC configuration.

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
SDM with Exact Derivative	10^3	3.504	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	50.44	1.6428	3000	31	8.4167	0.0007
	10^5	616.16	5.7668	3000	42	20.1414	0.0001

Table 1: Results of algorithm using exact derivate method

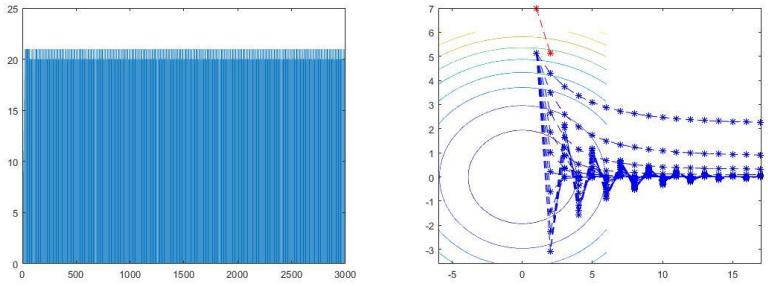


Figure 1: Backtracking iterations and contour plot of SDM

To show the differences in performance of exact and finite difference methods, we have performed forward method for all values of k, on the other hand, since centered finite difference does not depend on h, we have applied just for k = 2, 12 to illustrate that all results are the same.

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
SDM with Finite Difference-Forward K = 2	10^3	1.1872	0.0062	3000	50	0.1495	0.0004
	10^4	18.6154	2841.3078	3000	50	265.5132	0.0269
	10^5	283.9888	49859377573.52	3000	50	67008121.8	0.0088
SDM with Finite Difference-Forward K = 4	10^3	0.9502	2.9156e-05	3000	21	0.0323	2.9661e-05
	10^4	13.8049	1.1186	3000	34	8.2902	0.0007
	10^5	283.56	43.7322	3000	50	4.4034	6.3431e-06
SDM with Finite Difference-Forward K = 6	10^3	0.7988	2.9191e-05	3000	21	0.0325	2.9809e-05
	10^4	15.6921	1.6427	3000	31	8.4122	0.0007
	10^5	280.54	5.7353	3000	44	14.6829	0.0001
SDM with Finite Difference-Forward K = 8	10^3	0.7436	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	14.53	1.6428	3000	31	8.4166	0.0007
	10^5	259.22	5.7668	3000	42	20.1397	0.0001
SDM with Finite Difference-Forward K = 10	10^3	0.7325	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	14.48	1.6428	3000	31	8.4167	0.0007
	10^5	310	5.7668	3000	42	20.1414	0.0001
SDM with Finite Difference-Forward K = 12	10^3	0.7320	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	14.1512	1.6428	3000	31	8.4167	0.0007
	10^5	251.56	5.7668	3000	42	20.1414	0.0001

Table 2: Results of algorithm using forward finite difference method

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
SDM with Finite Difference-Centered K = 2	10^3	3.7940	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	52.2138	1.6428	3000	31	8.4167	0.0007
	10^5	590.23	5.7668	3000	42	20.1414	0.0001
SDM with Finite Difference-Centered K = 12	10^3	3.6218	2.9192e-05	3000	21	0.0325	2.9811e-05
	10^4	52.40	1.6428	3000	31	8.4167	0.0007
	10^5	619.08	5.7668	3000	42	20.1414	0.0001

Table 3: Results of algorithm using centered finite difference method

Regarding centered finite difference method, all results except computation time (slightly difference) coincide with exact gradient method as we expected before due to being independent from h. However, outputs of forward finite difference reach to the results of exact gradient method when we increase the value of k, and the reason is that we have small h for small values of k which results in poor performance (no guarantee for convergence when dimension is too high and k is small). Due to the simplified formula of forward method, computation time is better with respect to exact method while the other numbers are the same (after k=6).

Considering NM, all outcomes of exact and finite differences derivates are fairly similar for all values of k even for combination with forward finite differences in which the gradient depends on h. it can be seen that we have obtained very good results for all cases even for small k with few iterations comparing with SDM. It should be noted that in all scenarios stopping criteria for function value and length of step length are totally satisfied without entering backtracking loop. Numerical results are shown as follows:

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
NM with Exact Derivative	10^3	0.6112	3.7897e-20	290	0	1.0055e-08	9.6684e-13
	10^4	12.1917	3.7286e-19	301	0	9.9717e-08	9.5945e-13
	10^5	1385.65	3.6715e-18	312	0	9.8948e-07	9.5212e-13

Table 4: Results of algorithm using exact method

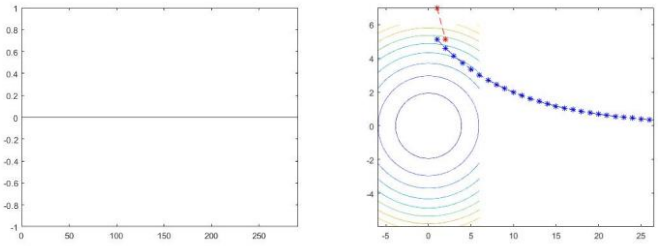


Figure 2: Backtracking iterations and contour plot of NM

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
NM with Finite Difference-Centered	10^3	0.4558	3.7897e-20	290	0	1.0055e-08	9.6684e-13
	10^4	12.47	3.7286e-19	301	0	9.9717e-08	9.5945e-13
	10^5	1331.5	3.6715e-18	312	0	9.8948e-07	9.5212e-13

Table 5: Results of algorithm using centered finite difference method

Method	*dimension*	*CPU time(s)*	*f_value*	*N. of iteration*	*N. of backtrack_iter*	*Gradient norm*	*Length of last step*
SDM with Finite Difference-Forward&Centered K = 2	10^3	1.965	2.457e-20	250	0	9.3766e-09	9.1771e-13
	10^4	126.5	1.4704e-19	198	0	9.3931e-08	9.5695e-13
	10^5	512.9	3.5843e-19	114	0	7.9799e-07	9.3152e-13
SDM with Finite Difference-Forward&Centered K = 4	10^3	0.5702	3.4225e-20	290	0	9.5708e-09	9.2043e-13
	10^4	12.89	3.298e-19	300	0	9.4251e-08	9.0737e-13
	10^5	1352.0	3.5771e-18	307	0	9.9212e-07	9.5634e-13
SDM with Finite Difference-Forward&Centered K = 6	10^3	0.4151	3.7858e-20	290	0	1.005e-08	9.6637e-13
	10^4	13.32	3.7161e-19	301	0	9.9555e-08	9.5790e-13
	10^5	1401.7	3.6314e-18	312	0	9.8422e-07	9.4708e-13
SDM with Finite Difference-Forward&Centered K = 8	10^3	0.4331	3.7896e-20	290	0	1.0055e-08	9.6684e-13
	10^4	12.87	3.7285e-19	301	0	9.9715e-08	9.5944e-13
	10^5	1463.01	3.6434e-18	312	0	9.8936e-07	9.5112e-13
SDM with Finite Difference-Forward&Centered K = 10	10^3	0.4254	3.7897e-20	290	0	1.0055e-08	9.6684e-13
	10^4	14.44	3.7286e-19	301	0	9.9717e-08	9.5945e-13
	10^5	1484.49	3.6514e-18	310	0	9.8946e-07	9.5210e-13
SDM with Finite Difference-Forward&Centered K = 12	10^3	0.4208	3.7897e-20	290	0	1.0055e-08	9.6684e-13
	10^4	12.58	3.7286e-19	301	0	9.9717e-08	9.5945e-13
	10^5	1567.28	3.6715e-18	312	0	9.8948e-07	9.5212e-13

Table 6: Results of algorithm using centered_hessian & forward_gradient method

IV. CONCLUSION:

To conclude, we have introduced two algorithms to find the minimum point of the weighted sphere function using exact gradient method that generated reasonable approximation of real minimum of the function which is 0. In addition, finite differences method has been applied to estimate the exact gradient ensuing the outcomes totally match with the main methods that indicates the estimation works perfectly.

V. MATLAB CODE:

A. *TEST:*

```
%% LOADING THE VARIABLES FOR THE TEST

clear
close all
clc

load('jong_test(10^3).mat')

c1 = 1e-4;
rho = 0.8;
btmax = 50;

gamma = 1e-1;
tolx = 1e-12;

domain = 'box';

box_mins = zeros(n,1);
box_mins(:) = -5.12;

box_maxs = zeros(n,1);
box_maxs(:) = 5.12;

Pi_X = @(x) box_projection(x, box_mins, box_maxs);

FDgrad = 'c';
FDHess = 'c';

k_h = 4;
%% RUN THE CONSTR. STEEPEST DESCENT ON f

disp('**** CONSTR. STEEPEST DESCENT: n=10^3 *****')
switch FDgrad
    case ''
        disp('//////THE METHOD FOR DERIVATIVE IS EXACT//////')
    case 'fw'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
FORWARD//////')
    case 'c'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
CENTERED//////')
end
tic
[xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, FDgrad, k_h);
toc
disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
disp('**** CONSTR. STEEPEST DESCENT: RESULTS *****')
disp('************************************')
disp(['xk_min: ', mat2str(min(xk)), 'xk_max: ', mat2str(max(xk)) ])
disp(['f(xk): ', num2str(fk), ' (actual min. value: 0);'])
disp(['N. of Iterations: ', num2str(k),'/',num2str(kmax), ';'])
disp(['N. of backtrackingIterationMax: ', num2str(max(btseq)), ';'])
disp(['gradient norm: ', num2str(gradfk_norm), ';'])
```

```
disp(['length of last step: ', num2str(deltaxk_norm), ';'])
disp('*****************DONE*******************')

%% PLOTS

t = linspace(0, 1, 25);


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

% Creation of the meshgrid for the contour-plot
[X1, Y1] = meshgrid(linspace(-6, 6, 500), linspace(-6, 6, 500));

% Creation of the meshgrid for the contour-plot
[X2, Y2] = meshgrid(linspace(-6, 6, 500), linspace(-6, 25, 500));
% Computation of the values of f for each point of the mesh

% Computation of the values of f for each point of the mesh
Z1 = X1.^2 + 4*Y1.^2 + 5;
Z2 = 100*(Y2-X2.^2).^2+(1-X2).^2;
Z3 = (X1.^2+Y1-11).^2+(X1+Y1.^2-7).^2;

% Simple Plot
fig1_n = figure();
% Contour plot with curve levels for each point in xseq
[C1, ~] = contour(X1, Y1, Z1);
hold on
% plot of the points in xseq
for i = 1:n
    plot([x0(i), Pi_X_x0(i)], 'r--*')
end

for i = 1:n
    plot([Pi_X_x0(i) xseq(i, :)], 'b--*')
end
%title('h for k= ',num2str(k_h))
hold off


% Barplot of btseq
fig1_bt = figure();
bar(btseq)

%% LOADING THE VARIABLES FOR THE TEST

load('jong_test(10^4).mat')

c1 = 1e-4;
rho = 0.8;
btmax = 50;

gamma = 1e-1;
tolx = 1e-12;

domain = 'box';

box_mins = zeros(n,1);
```

```
box_mins(:) = -5.12;

box_maxs = zeros(n,1);
box_maxs(:) = 5.12;

Pi_X = @(x) box_projection(x, box_mins, box_maxs);

FDgrad = 'c';
FDHess = 'c';

k_h = 4;
%% RUN THE CONSTR. STEEPEST DESCENT ON f

disp('**** CONSTR. STEEPEST DESCENT: n=10^4 *****')
switch FDgrad
    case ''
        disp('//////THE METHOD FOR DERIVATIVE IS EXACT//////')
    case 'fw'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
FORWARD//////')
    case 'c'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
CENTERED//////')
end
tic
[xk1, fk1, gradfk_norm1, deltaxk_norm1, k1, xseq1, btseq1] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, FDgrad, k_h);
toc
disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
disp('**** CONSTR. STEEPEST DESCENT: RESULTS *****')
disp('************************************')
disp(['xk_min: ', mat2str(min(xk1)), 'xk_max: ', mat2str(max(xk1)) ])
disp(['f(xk): ', num2str(fk1), ' (actual min. value: 0);'])
disp(['N. of Iterations: ', num2str(k1),'/',num2str(kmax), ';'])
disp(['N. of backtrackingIterationMax: ', num2str(max(btseq1)), ';'])
disp(['gradient norm: ', num2str(gradfk_norm1), ';'])
disp(['length of last step: ', num2str(deltaxk_norm1), ';'])
disp('*****************DONE*******************')

%% LOADING THE VARIABLES FOR THE TEST

load('jong_test(10^5).mat')

c1 = 1e-4;
rho = 0.8;
btmax = 50;

gamma = 1e-1;
tolx = 1e-12;

domain = 'box';

box_mins = zeros(n,1);
box_mins(:) = -5.12;

box_maxs = zeros(n,1);
box_maxs(:) = 5.12;
```

```
Pi_X = @(x) box_projection(x, box_mins, box_maxs);

FDgrad = 'c';
FDHess = 'c';

k_h = 4;
%% RUN THE CONSTR. STEEPEST DESCENT ON f

disp('**** CONSTR. STEEPEST DESCENT: n=10^5 *****')
switch FDgrad
    case ''
        disp('//////THE METHOD FOR DERIVATIVE IS EXACT//////')
    case 'fw'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
FORWARD//////')
    case 'c'
        disp('//////THE METHOD FOR DERIVATIVE IS FINITE DEFFIRENCE,
CENTERED//////')
end
tic
[xk2, fk2, gradfk_norm2, deltaxk_norm2, k2, xseq2, btseq2] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, FDgrad, k_h);
toc
disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
disp('**** CONSTR. STEEPEST DESCENT: RESULTS *****')
disp('************************************')
disp(['xk_min: ', mat2str(min(xk2)), 'xk_max: ', mat2str(max(xk2)) ])
disp(['f(xk): ', num2str(fk2), ' (actual min. value: 0);'])
disp(['N. of Iterations: ', num2str(k2),'/',num2str(kmax), ';'])
disp(['N. of backtrackingIterationMax: ', num2str(max(btseq2)), ';'])
disp(['gradient norm: ', num2str(gradfk_norm2), ';'])
disp(['length of last step: ', num2str(deltaxk_norm2), ';'])
disp('*****************DONE*******************')
```

B. ***CONSTR_STEEPEST_DESC_BCKTRCK:***

```
function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, FDgrad, k_h)
%
% function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq] = ...
%     constr_steepest_desc_bcktrck(x0, f, gradf, alpha0, ...
%     kmax, tollgrad, c1, rho, btmax, gamma, tolx, Pi_X)
%
% 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, lesser 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;
% Pi_X = projection function
%
% 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 'fw'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'fw')
        gradf = @(x) findiff_grad(f, x, k_h, 'fw');

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

% 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);

xk = Pi_X(x0); % Project the starting point if outside the constraints
fk = f(xk);
gradfk = gradf(xk);

k = 0;
gradfk_norm = norm(gradfk);
deltaxk_norm = tolx + 1;

while k < kmax && gradfk_norm >= tolgrad && deltaxk_norm >= tolx
    % Compute the descent direction
    pk = -gradf(xk);

    xbark = xk + gamma * pk;
    xhatk = Pi_X(xbark);
```

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

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

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

    bt = 0;
    % Backtracking strategy:
    % 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;
end

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

end
```

C. ***CONSTR_NEWTON_BCKTRCK:***

```
function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq] = ...
    constr_newton_bcktrck(x0, f, gradf, Hessf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X, FDgrad, FDHess, k_h)
%
% function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq, btseq] = ...
%     constr_steepest_desc_bcktrck(x0, f, gradf, alpha0, ...
%     kmax, tollgrad, c1, rho, btmax, gamma, tolx, Pi_X)
%
% 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, lesser 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;
% Pi_X = projection function
%
% 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 'fw'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'fw')
        gradf = @(x) findiff_grad(f, x, k_h, 'fw');

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

switch FDHess
    case 'c'
        % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_Hess
        Hessf = @(x) findiff_Hess(f, x, k_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
```

```
% 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);

xk = Pi_X(x0); % Project the starting point if outside the constraints
fk = f(xk);
gradfk = gradf(xk);

k = 0;
gradfk_norm = norm(gradfk);
deltaxk_norm = tolx + 1;

while k < kmax && gradfk_norm >= tolgrad && deltaxk_norm >= tolx
    % Compute the descent direction
    pk = -Hessf(xk)\gradfk;

    xbark = xk + gamma * pk;
    xhatk = Pi_X(xbark);

    % Reset the value of alpha
    alpha = 1;

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

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

    bt = 0;
    % Backtracking strategy:
    % 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;
end

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

end
```

D. ***MAIN FUNCTION:***

```
function f = jong_f(n, x)

f=0;
for i = 1:n
    f = f + i*x(i)^2;
end
end
```

E. ***EXACT DERIVATIVE OF THE FUNCTION:***

```
function f = jong_g(n, x)

f = zeros(n,1);
for i = 1:n
    f(i) = 2*i*x(i);
end
end
```

F. ***FINDIFF_GRAD:***

```
function [gradfx] = findiff_grad(f, x, k_h, type)

h = 10^(-k_h)*norm(x);
gradfx = zeros(size(x));

switch type
    case 'fw'
        for i=1:length(x)
            gradfx(i) = i*(2*x(i) + h);
        end
    case 'c'
        for i=1:length(x)
            gradfx(i) = 2*i*x(i);
        end
end

end
```

G. ***FINDIFF_HESS:***

```
function [Hessfx] = findiff_Hess(f, x, k_h)
%
% [Hessf] = findiff_Hess(f, x, h)
%
```

```
% Function that approximate the Hessian of f in x (column vector) with the
% finite difference (centered) method.
% ATTENTION: we assume a regular f (C^2) s.t. Hessf is symmetric.
%
% INPUTS:
% f = function handle that describes a function R^n->R;
% x = n-dimensional column vector;
% h = the h used for the finite difference computation of Hessf
%
% OUTPUTS:
% Hessfx = n-by-n matrix corresponding to the approximation of the Hessian
% of f in x.

h = 10^(-k_h)*norm(x);
Hessfx = spdiags([],[], length(x), length(x));

for i=1:length(x)
   Hessfx(i,i) = 2*i;
end

% for j=1:length(x)
%     % Elements on the diagonal
%     %Hessfx(j,j) = (j*h)/2;
%     xh_plus = x;
%     xh_minus = x;
%     xh_plus(j) = xh_plus(j) + h;
%     xh_minus(j) = xh_minus(j) - h;
%     Hessfx(j,j) = (f(xh_plus) - 2*f(x) + f(xh_minus))/(h^2);
%     % ALTERNATIVELY (no xh_plus and xh_minus)
%     % Hessf(j,j) = (f([x(1:j-1); x(j)+h; x(j+1:end)]) - 2*f(x) + ...
%     %     f([x(1:j-1); x(j)-h; x(j+1:end)]))/(h^2);
%     for i=j+1:length(x)
%         xh_plus_ij = x;
%         xh_plus_ij([i, j]) = xh_plus_ij([i, j]) + h;
%         xh_plus_i = x;
%         xh_plus_i(i) = xh_plus_i(i) + h;
%         xh_plus_j = x;
%         xh_plus_j(j) = xh_plus_j(j) + h;
%         Hessfx(i,j) = (f(xh_plus_ij) - ...
%             f(xh_plus_i) - f(xh_plus_j) + f(x))/(h^2);
% %         Hessfx(i,j) = (i*x(i)^2 + j*x(j)*2)/h^2;
%         Hessfx(j,i) = Hessfx(i,j);
%         % ALTERNATIVELY (no xh_plus_i/j)
%         % Hessf(i,j) = (f(xh_plus_ij) - ...
%         %     f([x(1:i-1); x(i)+h; x(i+1:end)]) - ...
%         %     f([x(1:j-1); x(j)+h; x(j+1:end)]) + f(x))/(h^2);
%
%     end
% end


end
```

H. TEST FUNCTION FOR N = 10^5:

```
Hessf: @(x)(jong_h(n,x))
alpha: 1
    f: @(x)(jong_f(n,x))
```

```
  gradf: @(x)(jong_g(n,x))
   kmax: 3000
      n: 100000
tolgrad: 1.0000e-12
     x0: [100000×1 double]
```