Constrained Numerical Optimization

Constrained assignment: exercise 1

Edoardo Fantolino s286008
Data Science and Engineering
Politecnico di Torino
Torino, Italia
s286008@studenti.polito.it

***Abstract*—This document contains my ideas about the first exercise of the constrained assignment. Here, I try to solve a minimization problem. Like everywhere else, some issues arose during the execution and you can find how I dealt with them. This document is addressed to Professor Sandra Pieraccini.**

***Index Terms*—constraints, finite differences, minimization, Projected Gradient Method, optimization.**

I. Introduction

The main objective of this homework is to deal with the following problem

$$\min_{x\in\mathbb{R}^n} \sum_{i=1}^{n} \left(\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 - x_i\right)$$
$$s.t. \quad 1 \leq x_i \leq 2 \quad \forall i$$

To solve this minimization problem it will be implemented the Projected Gradient Method method, both using the exact derivatives and using finite differences to approximate the gradient and the Hessian matrix.

We need to study the function with dimensions n equal to 10^4 and 10^6.

II. The firsts steps toward the understanding of the problem

A. The function and its characteristics

The function to be analyzed is:

$$f(x) = \sum_{i=1}^{n} \left(\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 - x_i\right) \tag{1}$$

Consequently, the gradient of f(x) is given by:

$$\nabla f(x) = \begin{bmatrix} x_1^3 + x_1 - 1 \\ x_2^3 + x_2 - 1 \\ \vdots \\ x_n^3 + x_n - 1 \end{bmatrix} \tag{2}$$

The next natural step is the calculation of the Hessian matrix.

$$H_f = \begin{bmatrix} 3x_1^2 + 1 & & & \\ & 3x_2^2 + 1 & & \\ & & \ddots & \\ & & & 3x_n^2 + 1 \end{bmatrix} \tag{3}$$

As we can see, the Hessian of the function is a diagonal matrix. The eigenvalues are the elements on the diagonal. We can easily show that these elements are all positive and always positive. In fact:

$$3x_i^2 + 1 > 0, \quad \forall x \in \mathbb{R}, \quad i = 1, \cdots, n$$

We can say that H is a symmetric positive definite matrix so there will be a unique and global minimum. This function has the same behaviour of the function that we analyzed for the unconstrained assignment, so we will not go deeper in the analysis because it is not necessary.

B. The constraint

The constrain for this problem are two boundaries for each dimension. If we consider a case in 1 dimension we will have a thick line. In two dimension, these boundaries are a square, in three dimension we can imagine a cube, while for higher dimension we will obtain a multidimensional hypercube. I realized a picture in two dimension in order to give an idea of the situation that we have to face. You can see the result in Fig.1.

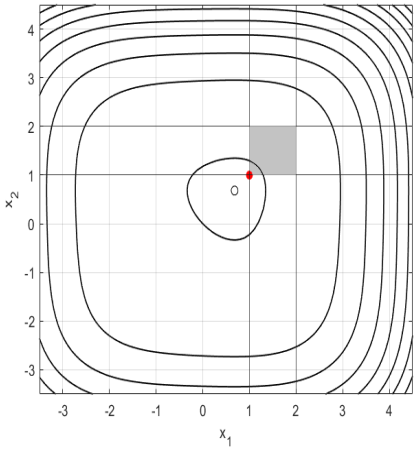

Fig. 1. Contour line of the function in two dimension. The gray space is the region of feasible points. The red dot is the expected result, while the black circle is the actual minimum

III. Projected Gradient Method

The basic idea behind the Projected Gradient method is the steepest descent adapted for the constraint. Basically if we are outside the feasible area we will perform a projection in order to be pushed in the proper space region.

After we choose a starting point, the main steps are:

- Compute $\bar{x}_k = \prod_x (x_k - \gamma_k \nabla f(x_k))$
- Find the new point $x_{k+1} = x_k + \alpha_k(\bar{x}_k - x_k)$

where γ is a constant greater than 0 and $\alpha \in [0, 1]$. The symbol $\prod$ represent the function that project the eventual external points into the feasible region.

To find the value of α we can exploit different techniques.

1) Limited Minimization rule
2) Armijo condition

In this document we will use the Armijo condition. The Armijo condition will be present in the algorithm in the form of a cycle that will reduce the value of α when needed.

A. Exact derivatives

The result obtained with the exact derivative are shown in Tab.1. We obtain a counterintuitive result. We just need 2 function evaluation and the algorithm seems to perform better than the situation without the constraints. This is due to the shape of the constraint and the position of our starting point.

In Fig.2 I highlighted a special region of the space. If we take a starting point that belongs to that region, to find the solution of the problem we only need to make a single projection and we will automatically reach the solution.

Instead, if we start from a point that do not lie int that peculiar region, even if we are far away from the solution, we just need a projection that allow us to reach the feasible area and than with few iteration we will be able to find the actual suitable minimum.

In this case having $\alpha >> 1$ seems to be an advantage. This is because it increase the probability of ending in the special region of Fig.2, and as we said, if we go in that region, the next iteration will show us the solution of the problem.

n = 10^4	**time (s)**	2.118×10^{-3}
	iter	2
n = 10^6	**time (s)**	2.155×10^{-1}
	iter	2

TABLE I
Result of Project Gradient Method with exact derivatives

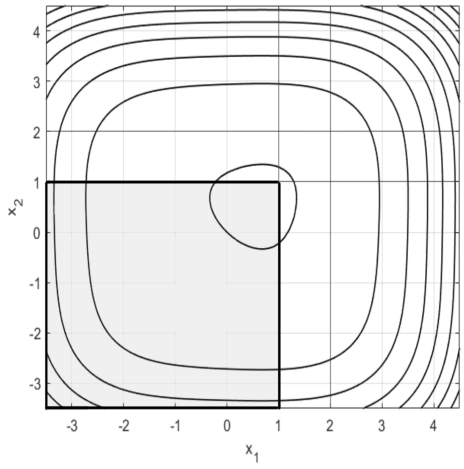

Fig. 2. Contour line of the function in two dimension. The gray space is the region that allow to find the solution with one single projection

B. Approximation of derivatives

I used the same strategies of the previous assignment (Jacobian with single perturbation), so I will not repeat the reasoning. I just present the results found and you can find them in Tab.2 and Tab.3

TABLE II
True gradient and Hessian approximated with Jacobian centered finite difference

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	2	2	2	2	2	2	2
	t (ms)	4.78	5.04	4.81	5.26	4.66	4.78	4.78
$n = 10^6$	**iter**	66	2	2	2	2	2	2
	t (ms)	522	509	491	487	491	489	484

TABLE III
True gradient and Hessian approximated with Jacobian centered finite difference

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	2	2	2	2	2	2	2
	t (ms)	4.80	5.05	5.24	5.26	5.01	4.82	4.84
$n = 10^5$	**iter**	2	2	2	2	2	2	2
	t (ms)	510	513	488	486	487	488	488

IV. Conclusion

The result seems to be reasonable and in line with the theory concepts. Surprisingly, we are able to obtain meaningful results even when h go beyond the safe threshold. The reason is that we have to make the approximation of the gradient just 2 times. In fact, in the unconstrained optimization problem we repeated the jacobian technique many times and so the probability to encounter a numerical cancellation increase, while in this case with just few function evaluation we reach the minimum and the probability to encounter the same issue are really low. We are able to obtain this result because of the structure of the problem and the shape of the constraints.

References

Jorge Nocedal, Stephen J. Wright. Numerical Optimization, Springer, 1999.

Sandra Pieraccini. Lecture on constrained optimization. 2020.

The initialization code

Here, there are the initialization of the variables like n, the actual minimum, the starting point x. There is the definition of the funciton handle f, gradf. Then we define the values of alpha, the maximum number of iterations allowed, the tolerance for the gradient stopping criteria. We even define the boundaries of the constrainet of the problem.

Typeg is the variable where we store the techniques that we want to use to approximate the derivative.

```
close all
clear
clc

n = 1e4;
x_min = -0.682327803828019*ones(n,1); % actual minumum
x = -2*rand(n,1)+2;
alpha0 = 1;
kmax = 50;
c1 = 1e-4;
rho = 0.8;
btmax = 2;
tollgrad = 1e-8;
gamma = 0.1;
tolx = 1e-6;
left = 1;
right = 2;
Pi_X = @(x) projection(x, left, right);

% function handle for the function and the gradient
f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);
gradf = @(x) [x.^3 + x + 1];
fg = @(x) [(1/4)*x.^4 + (1/2)*x.^2 + x];
```

Iteration and time analysis

In this page there is the code I used to make the analysis for time and the number of iterations. It is basic MATLAB code.

```
% for the iteration results
num_iter = 100;
times = zeros(num_iter,1);
steps = zeros(num_iter,1);

for kh=2:2:14
    h = 10^(-kh);
    for i=1:num_iter
        tic;
        [xk, fk, gradfk_norm, deltaxk_norm, k, xseq] = ...
        projected_gradient_method(x, f, gradf, alpha0, ...
        kmax, tollgrad, gamma, tolx, Pi_X, h, 'Jc');
        times(i) = toc;
        steps(i) = k;
        xseq = [x xseq];
    end
    kh
    result = xk(end-1)
    avg_steps = mean(steps)
    avg_time = mean(times)
end
```

Initialization of Projected Gradient Method

In the first part of the Projected Gradient method function I define a switch clause that allow to understand what kind of approximation we want to use. Here we just need a case fot the gradient approximation.

```
function [xk, fk, gradfk_norm, deltaxk_norm, k, xseq] = ...
    projected_gradient_method(x0, f, gradf, alpha0, ...
    kmax, tollgrad, gamma, tolx, Pi_X, h, typeg)

switch typeg
    case 'Jfw'
        gradf = @(x) fd_grad(f, x, h, 'Jfw');
%         disp('Gradient approximatino with forward method')
    case 'Jbw'
        gradf = @(x) fd_grad(f, x, h, 'Jbw');
%         disp('Gradient approximatino with backward method')
    case 'Jc'
        gradf = @(x) fd_grad(f, x, h, 'Jc');
%         disp('Gradient approximatino with centered method')
    otherwise
end
```

Core of Projected Gradient method

Here you can find the code that perform the Projected Gradient method. There are the main steps like the computation of the descent direction and the check over the norm of the gradient in order to understand if the solution is good enough.

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

xk = x0; % Project the starting point if outside the constraints
fk = f(xk);
k = 0;
gradfk_norm = norm(gradf(xk));
deltaxk_norm = tolx + 1;

while k < kmax && gradfk_norm >= tollgrad && deltaxk_norm >= tolx
    xk = Pi_X(xk);
    pk = -gradf(xk);

    xbark = xk + gamma * pk;
    xhatk = Pi_X(xbark);
    alpha = alpha0;

    % 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);

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

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

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

xseq = xseq(:, 1:k);
end
```

Gradient function

The code for the gradient approximation:

```
function [gradfx] = fd_grad(f, x, h, type)

h = h*norm(x);

switch type
    case 'Jfw'
        gradfx = (f(x+h*(ones))-f(x))/h;
    case 'Jbw'
        gradfx = (f(x)-f(x-h*(ones)))/h;
    case 'Jc'
        gradfx = (f(x+h*(ones))-f(x-h*(ones)))/(2*h);
end

end
```