Constrained Optimization: Exercise 1

Beatrice Macchia, Riccardo Bosio
Politecnico di Torino
Student ids: s245844, s291299
s245844@studenti.polito.it, s291299@studenti.polito.it

I. Problem overview

The problem consists of implementing the projected gradient method to solve $min_{x \in R^n} \sum_{i=1}^{n} (\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 - x_i)$ s.t. $1 \leq x_i \leq 2 \ \forall i$. We solve this constrained optimization problem with $n = 10^4$ and $n = 10^6$, both using exact derivatives and using finite differences to approximate the gradient. Then, a comparison between the behaviors of the two implementations must be made, using these values for the increment h: $h = 10^{-k}\hat{x}$, $k = 2, 4, 6, 8, 10, 12$. $\hat{x}$ is the point at which the derivatives have to be approximated.

II. Proposed approach

To solve this constrained optimization problem we first create $function\_to\_optimize.m$ to introduce the function that we have to optimize and in $gradient.m$ we build the gradient function using exact derivatives. Then we notice that the function is separable [1], so we create $separated\_function\_to\_optimize.m$, which is a function $F : R^n \to R^n$ with each component of the sum as a row.

$$F(x_1, x_2 \dots x_n) = \begin{bmatrix} g(x_1) \\ g(x_2) \\ \vdots \\ g(x_n) \end{bmatrix} \tag{1}$$

where $g(x) = \frac{1}{4}x^4 + \frac{1}{2}x^2 - x$. Calling f the function to optimize, we get $f(x) = 1_n^T \cdot F(x)$, with 1_n vector of n ones.

Then we notice that:

$$f(x + he_1) - f(x) = 1_n^T \cdot F(x + he_1) - 1_n^T \cdot F(x) =$$

$$= 1_n^T \cdot (F(x + he_1) - F(x)) = g(x_1 + h) - g(x).$$

At this point we can create $finite\_difference\_gradient.m$ to implement the finite differences to compute the gradient.

In particular we exploit the property just discovered to approximate the i-th component of the gradient in the forward method, considering only the increment of the i-th component of F:

$$\nabla f(x) \simeq \begin{bmatrix} \frac{g(x_1+h)-g(x_1)}{h} \\ \frac{g(x_2+h)-g(x_2)}{h} \\ \vdots \\ \frac{g(x_n+h)-g(x_n)}{h} \end{bmatrix} = \frac{F(x+h) - F(x)}{h} \tag{2}$$

With a similar approach we get that we can approximate the gradient in the centered method in the following way:

$$\nabla f(x) \simeq \frac{F(x+h) - F(x-h)}{2 \cdot h}$$

The feasible set is $X = \{x \in R^n | 1 \leq x_i \leq 2 \ \forall x_i\}$ and we need a projection function to implement the projected gradient method. Since the feasible set is in the form $X = \{x \in R^n | L \leq x_i \leq U \ \forall x_i\}$, we use the following projection:

$$[\pi_X(y)]_i = \begin{cases} y_i & L \leq y_i \leq U \\ \text{Ł} & y_i < L \\ U & y_i > U \end{cases}$$

This function is in $projection.m$. Then we can implement the projected gradient method in $projected\_gradient\_method.m$, with backtracking using the Armijo condition. Finally, in $C1.m$ we can run the projected gradient method and make comparisons.

III. Results

The projected gradient method that we have implemented always reaches the solution x_k equal to a vector of n ones at the k-th iteration. The corresponding value of the function is $f(x_k) = -2500$ for $n = 10^4$ and $f(x_k) = -250000$ for $n = 10^6$.

Backtracking iterations are always equal to zero for all the methods and for all values of h.

We tested our code with the following parameters:

Initializations	
Parameter	Value
alpha0	1
kmax	1000
tolgrad	1e-12
c1	1e-4
rho	0.8
btmax	50
gamma	0.1
tolx	1e-6

We chose $x0 = (1.5, 1.5, \dots 1, 5)^T$, which is inside the feasible set, as starting point.

The results in terms of number of itarations and computing time are summarized in the following tables.

$n = 10^4$			
Number of iterations			
h	Exact derivatives	Finite differences (centered)	Finite differences (forward)
	3		
$10^{-2}\hat{x}$		2	2
$10^{-4}\hat{x}$		3	3
$10^{-6}\hat{x}$		3	3
$10^{-8}\hat{x}$		3	3
$10^{-10}\hat{x}$		3	3
$10^{-12}\hat{x}$		3	3

$n = 10^4$			
Computing time(seconds)			
h	Exact derivatives	Finite differences (centered)	Finite differences (forward)
	0.040461		
$10^{-2}\hat{x}$		0.034575	0.033844
$10^{-4}\hat{x}$		0,040235	0,037343
$10^{-6}\hat{x}$		0,037843	0,03349
$10^{-8}\hat{x}$		0,033699	0,033699
$10^{-10}\hat{x}$		0,035738	0,030167
$10^{-12}\hat{x}$		0,033545	0,028171

$n = 10^6$			
Number of iterations			
h	Exact derivatives	Finite differences (centered)	Finite differences (forward)
	3		
$10^{-2}\hat{x}$		2	2
$10^{-4}\hat{x}$		3	3
$10^{-6}\hat{x}$		3	3
$10^{-8}\hat{x}$		3	3
$10^{-10}\hat{x}$		3	3
$10^{-12}\hat{x}$		3	3

$n = 10^6$			
Computing time(seconds)			
h	Exact derivatives	Finite differences (centered)	Finite differences (forward)
	1,519397		
$10^{-2}\hat{x}$		2,827075	2,354431
$10^{-4}\hat{x}$		4,068107	3,396018
$10^{-6}\hat{x}$		3,985171	3,412225
$10^{-8}\hat{x}$		4,016851	3,539372
$10^{-10}\hat{x}$		3,834642	3,377349
$10^{-12}\hat{x}$		4,072665	3,463752

Also we tested our code changing the starting point from $x0 = (1.5, 1.5, \dots 1, 5)^T$ to $x0 = (0, 0, \dots 0)^T$, which is an external point with respect to the feasible set X. In terms of iterations, with the new $x0$ we got that all the methods need only 1 iteration both for $n = 10^4$ and $n = 10^6$. Talking about computing time, as for $x0 = (1.5, 1.5, \dots 1, 5)^T$, there are not significant differences with $n = 10^4$. Moreover for $n = 10^6$ the behavior is similar for both values of $x0$: the exact derivates method is the more efficient and the forward one performs better among the finite differences methods.

IV. Discussion

Looking at the results we can observe that the finite differences method with $h = 10^{-2}\hat{x}$ performs better in terms of number of iterations for both values of n. We get the result in 2 iterations, while in the other cases 3 iterations are needed.

Instead, in terms of computing time we can see that for $n = 10^4$ we get the fastest result using the forward finite differences method with $h = 10^{-12}\hat{x}$. However, comparisons between the computing times with $n = 10^4$ are not that meaningful, as they differ only in hundredths of a second.

For $n = 10^6$ we can note slightly more relevant differences in terms of computing time: in general the exact derivates method performs better with respect to the other options.

V. Appendix

A. *function_to_optimize.m*

```
%Create the function which has to be optimized

function y = function_to_optimize(x,n)
%
%INPUTS:
%x=column vector of length n;
%n=number of dimensions of x;
%OUTPUTS:
%y=real scalar value equal to f(x)
%
y=0;
for i=1:n
    y=y+(1/4*x(i)^4+1/2*x(i)^2-x(i));
end
end
```

B. *separated_function_to_optimize.m*

```
function Y = separated_function_to_optimize(x,n)
%
%INPUTS:
%x=column vector of length n;
%n=number of dimensions of x;
%OUTPUTS:
%Y=separated function R^n->R^n.
%
Y=zeros(n,1);
for i=1:n
    Y(i)=1/4*x(i)^4+1/2*x(i)^2-x(i);
end

end
```

C. *gradient.m*

```
%Gradient function of the function to be optimized
function gradfx=gradient(x,n)
%
%INPUTS:
%x=column vector of size n;
%n=number of dimensions;
%
%OUTPUTS:
%gradf=column vector where i-th component is the partial derivative of f
%with respect to the i-th component of x.
%
gradfx=zeros(n,1);
for i=1:n
    gradfx(i)=x(i)^3+x(i)-1;
end
end
```

D. *finite_difference_gradient.m*

```
%Compute the gradient with the finite difference method

function gradfx=finite_difference_gradient(F,x,n,k,type)
%
%function gradfx=finite_difference_gradient(f,x,n,h,type)
%
%INPUTS:
%F=separated function R^n->R^n, of which we want to find the gradient;
%x=column vector of dimension n;
%n=dimension of x;
%k=integer between 2 and 12 used to find h, the parameter of the finite
%difference method;
%type='fw' or 'c' to choose the forward/centered finite difference method
%respectively.
%
%OUTPUTS:
%gradfx=column vector of dimension n which approximates the gradient of f
%in x.
%

gradfx=zeros(n,1);

h=10^(-k)*norm(x);

switch type
    case 'fw'
        xh=x+h;
        gradfx=(F(xh)-F(x))/h;
    case 'c'
        xh_plus=x+h;
        xh_minus=x-h;
        gradfx=(F(xh_plus)-F(xh_minus))/(2*h);
    otherwise %we do forward one
        xh=x+h;
        gradfx=(F(xh)-F(x))/h;
end
end
```

E. *projection.m*

```
%Function to project xk_bar into the feasible set
%
%
%
function x_hat=projection(x,lower,upper)
%
%function x_hat=projection(x,lower,upper)
%
%INPUTS:
%x is a column vector containing n components
%lower is a column vector where the i-th element is the lower bound of the
%i-th dimension of the feasible set
%lower is a column vector where the i-th element is the upper bound of the
%i-th dimension of the feasible set
%
%OUTPUT:
%x_hat is equal to x if x is already in the feasible set otherwise it
%is updated to the projection of x in theboundary of the feasible set
%
x_hat=max(min(x,upper),lower);
end
```

F. *projected_gradient_method.m*

```
%Projected gradient method function
function [xk, fk, gradfk_norm, Δxk_norm, k, xseq, btseq]=...
    projected_gradient_method(x0, f, gradf, alpha0, kmax, tolgrad, c1,...
    rho, btmax, gamma, tolx, Pi_X)
%
%function [xk, fk, gradfk_norm, Δxk_norm, k, xseq, btseq]=
%projected_gradient_method(x0, f, gradf, alpha0, kmax, tolgrad,
%c1, rho, btmax, gamma, tolx, Pi_X)
%
%INPUTS:
%x0=n-dimensional column vector;
%f=function R^n->R;
%gradf=function that describes the gradient of f;
%alpha0=first constant that multiplies the descent direction;
%kmax=maximum number of iterations;
%tolgrad=tolerance for the gradient (used as a stopping criterion);
%c1=Armijo condition factor that must be a scalar in (0,1);
%rho=fixed factor lesser than one used for reducing alpha0 at each
%iteration;
%btmax=maximum number of steps of the backtracking strategy to update
%alpha;
%gamma=constant that multiplies the descent direction before projection;
%tolx=tolerance for the norm of the difference between two consecutive
%xk (used as a stopping criterion);
%Pi_X=function that handles the projection;
%
%OUTPUTS:
```

```
%xk=the last x computed by the function;
%fk=f(xk);
%gradfk_norm=norm of gradf(xk);
%k=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.
%

%Initializations
xseq=zeros(length(x0),kmax);
btseq=zeros(1,kmax);
xk=Pi_X(x0);
fk=f(xk);
k=0;
gradfk_norm=norm(gradf(xk));
Δxk_norm=tolx+1; %to be sure of entering the ...
    while

%create function to handle armijo conditions
f_armijo=@(fk,alpha,xk,pk) ...
    fk+c1*alpha*gradf(xk)'*pk;

while k<kmax && gradfk_norm≥tolgrad && Δ...
    xk_norm≥tolx
    %Find the descent direction
    pk=-gradf(xk);
    xk_bar=xk+gamma*pk;
    xk_hat=Pi_X(xk_bar);

    %Reset alpha
    alpha=alpha0;

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

    %Compute f in xnew
    fnew=f(xnew);

    bt=0;

    %Start backtracking
    while bt<btmax && ...
        fnew>f_armijo(fk,alpha,xk,pik)
        %Reduce alpha
        alpha=rho*alpha;

        %Update fnew and xnew
        xnew=xk+alpha*pik;
        fnew=f(xnew);

        bt=bt+1;
    end

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

    %Increase iteration
    k=k+1;

    %Store xk in k-th column of xseq
    xseq(:,k)=xk;

    %Store bt in btseq
    btseq(k)=bt;
end

%Resize xseq and btseq
xseq=xseq(:,1:k);
btseq=btseq(1:k);

end
```

G. C1.m

```
clear all;
%
%
%Constrained optimization
%1)Projected gradient method
%
%
%Select a value for n
n=10^6; %We run C1 also with n=10^4

%Set parameters for the projected gradient ...
    method
x0=1.5*ones(n,1); %starting point
alpha0=1;
kmax=1000;
tolgrad=1e-12;
c1=1e-4;
rho=0.8;
btmax=50;
gamma=0.1;
tolx=1e-6;

%Create the function which has to be optimized
f=@(x) function_to_optimize(x,n);

%Create the separated function which has to ...
    be optimized
F=@(x) separated_function_to_optimize(x,n);

%Create the projection function
Pi_X=@(x) projection(x,ones(n,1),2*ones(n,1));

%Run the projected gradient method using the ...
    exact derivatives
%Create the gradient function
gradf=@(x) gradient(x,n);

tic
[xk, fk, gradfk_norm, Δxk_norm, k, xseq, ...
    btseq]=...
    projected_gradient_method(x0, f, gradf, ...
        alpha0, kmax, tolgrad, c1,...
    rho, btmax, gamma, tolx, Pi_X);
toc

disp('Number of iterations with exact ...
    derivatives:')
disp(k)

disp('Value of the function in xk:')
disp(fk)

%We have to try for different values of h
for j=2:2:12
    j %print j

    %Create the gradient computed with the ...
        finite difference method
    tic
    fd_gradf_c=@(x) ...
        finite_difference_gradient(F,x,n,j,'c'); ...
        %centered
    [xk, fk, gradfk_norm, Δxk_norm, k, xseq, ...
        btseq]=...
```

```
55          projected_gradient_method(x0, f, ...
                fd_gradf_c, alpha0, kmax,...
56          tolgrad, c1, rho, btmax, gamma, ...
                tolx, Pi_X);
57      toc
58
59      disp('Number of iterations with centered ...
            finite differences:')
60      disp(k)
61
62      disp('Value of the function in xk:')
63      disp(fk)
64
65      tic
66      fd_gradf_fw=@(x) ...
            finite_difference_gradient(F,x,n,j,'fw'); ...
            %forward
67      [xk, fk, gradfk_norm, Δxk_norm, k, xseq, ...
            btseq]=...
68          projected_gradient_method(x0, f, ...
                fd_gradf_fw, alpha0, kmax,...
69          tolgrad, c1, rho, btmax, gamma, ...
                tolx, Pi_X);
70      toc
71
72      disp('Number of iterations with forward ...
            finite differences:')
73      disp(k)
74
75      disp('Value of the function in xk:')
76      disp(fk)
77  end
```

REFERENCES

[1] S. J.Nocedal, *Numerical Optimization*. Springer, 1999.