Numerical optimization for large scale problems and Stochastic Optimization

Mattia Delleani
s288854

February 11, 2020

Abstract

This report describes a possible approach to Problem 1 of *Constrained Optimization*.

1 Introduction and Overview

The problem concerns the minimization of a given function through the implementation of the projected gradient method, in order to respect the constraints. The gradient of the function has been computed, both using finite differences and exact derivatives.
Considering the loss function:

$$\min_{x \in R^n} \sum_{i=1}^{n} \frac{x_i^4}{4} + \frac{x_i^2}{2} - x_i \tag{1}$$

$$s.t. \quad 1 \leq x_i \leq 2 \quad \forall i \tag{2}$$

$$h = 10^{-k}\|x\| \quad k = 2, 4, 6, 8, 10, 12 \tag{3}$$

$$n = 10^4, 10^6 \tag{4}$$

1.1 Proposed solution

The results are achieved using the steepest descent algorithm, which is an iterative method that at each iteration computes the descent direction. This direction p_k is represented by the gradient of the function that can be computed directly with the exact derivative, only if the function is known (like in this case), or estimated through function evaluations (e.g. finite differences approach). The descent direction:

$$p_k = -\nabla f_k \tag{5}$$

At each iteration the point x_k is updated by summing itself with the product between the gradient and a component γ, that multiplies the descent direction before the (possible) projection:

$$\bar{x}_k = x_k - \gamma \nabla f_k \tag{6}$$

During the update of points towards the optimum, the constraints may be not satisfied, so a box projection function has been implemented in order to satisfy the constraints at each iteration of the algorithm.

$$\Pi(x_i) = \begin{cases} L_i & \text{if } x_i < L_i \\ x_i & \text{if } L_i \leq x_i \leq U_i \\ U_i & \text{if } \quad x_i > U_i \end{cases} \tag{7}$$

$$L \leq x \leq U \tag{8}$$

where L is the lower bound and U is the upper bound.
After computing the projection $\hat{x_k} : \Pi(\bar{x}_k)$ and the direction $\pi_k := (\hat{x_k} - x_k)$ we are able to compute the next step of the optimization method:

$$x_{k+1} = x_k + \alpha_k \pi_k \tag{9}$$

where α is the step length factor.
Note that, not all the choices of α are suitable for the problem, because (with respect to the minimization problem) if a too long step is taken, the value of the function may increase, on the contrary if a too small step is taken too many iterations are needed and the algorithm becomes very slow. In order to avoid that the Armijo's conditions with backtracking has been implemented.

$$f(x_k + \alpha p_k) \leq f(x_k) + \alpha c_1 \nabla f_k p_k \tag{10}$$

This allow to start from large values of α and reducing it only if the condition is not satisfied, in this way we ensure to take large steplengths whenever possible.
The algorithm stops whenever a given tolerance or the maximum number of iterations are reached (see Section 2.1).

1.1.1 Computing gradient

As mentioned above, in order to choose the descent direction we need to compute the gradient. In this case since the loss function F: $\mathbb{R}^n \to \mathbb{R}$, is known the computation of the gradient can be done in different ways:

- Exact derivatives: partial derivatives are computed for the gradient.

$$\frac{\partial F(x)}{\partial x_i} \quad \forall i = 1, .., n \tag{11}$$

$$\tag{12}$$

- Finite difference method: the gradient is estimated through function evaluation, without computing exact derivatives. This method is used for large problem with complex derivative computation. For this problem we implemented the *forward* and *centered* methods:

$$\text{forward:} \quad \frac{\partial F(x)}{\partial x_i}(x) = \frac{F(x + he_i) - F(x)}{h} \quad \forall i = 1, .., n \tag{13}$$

$$\text{centered:} \quad \frac{\partial F(x)}{\partial x_i}(x) = \frac{F(x + he_i) - F(x - he_i)}{2h} \quad \forall i = 1, .., n \tag{14}$$

$$\tag{15}$$

  The difference, considering F: $\mathbb{R}^n \to \mathbb{R}$, between this two is that *forward* is faster because does $n + 1$ function evaluation , in fact the error is $\mathcal{O}(h)$, whereas for *centered* the error is $\mathcal{O}(h^2)$, but does $2n$ function evaluations and it is slower.

For both method, in our problem we can exploit separability since our Loss function is separable, and avoid computing each component of the gradient, in fact the components of the gradient can be approximated simultaneously with a single increment. Exploiting separability allows our algorithm to strongly reduce the computational impact and also avoid to allocate memory for huge sparse matrices.
The exact derivative of one component of the loss function can be easily computed:

$$\frac{\partial f(x)}{\partial x} = x^3 + x - 1 \tag{16}$$

2 Results

2.1 Parameters

For the implementation:

- γ: factor that multiplies the descent direction before the possible projection.

The stopping criteria parameters used for the algorithm are:

- *kmax*: maximum number of iterations
- *tollgrad*: value used as stopping criterion w.r.t. the norm of the gradient
- *tolx*: a real scalar value characterizing the tolerance with respect to the norm of $\|x_{k+1} - x_k\|$ in order to stop the method

The parameters for the Armijo's condition are:

- α: the initial factor that multiplies the descent direction at each iteration.
- ρ: fixed factor, lesser than 1, used for reducing α;
- *btmax*: maximum number of steps for updating alpha during the backtracking strategy
- c_1: the factor of the Armijo condition that must be a scalar in (0,1);

2.2 Case 1

For the implementation: $\gamma : 0.1$ as step-length, *tollgrad*:10^{-12}, *tolx*: 10^{-6}
For the Armijo's condition: $\alpha : 1$, $\rho : 0.8$, *btmax*: 50 (max number of backtracking operations for Armijo), $c_1 : 10^{-4}$.

n	k	Fmin	Time	iterations
10^4	2	-2500	0.0084	2
	4	-2500	0.0057	3
	6	-2500	0.0057	3
	8	-2500	0.0062	3
	10	-2500	0.0053	3
	12	-2500	0.0034	3
10^6	2	-250000	0.2988	2
	4	-250000	0.4276	3
	6	-250000	0.4596	3
	8	-250000	0.4072	3
	10	-250000	0.4001	3
	12	-250000	0.4814	3

Table 1: Forward finite difference

n	k	Fmin	Time	iterations
10^4	2	-2500	0.0084	3
	4	-2500	0.0077	3
	6	-2500	0.0057	3
	8	-2500	0.0062	3
	10	-2500	0.0055	3
	12	-2500	0.0057	3
10^6	2	-250000	0.2081	3
	4	-250000	0.1978	3
	6	-250000	0.1934	3
	8	-250000	0.1971	3
	10	-250000	0.2078	3
	12	-250000	0.2154	3

Table 2: Exact derivatives

As expected time increase as n increases in both cases. Both approaches use almost the same number of iterations, and get the same results. The difference between the two approach can be made with respect to computational time, in fact exact method derivatives, in this particular case where the partial derivative is the same for each component of the gradient, is faster because we can exploit the separability of the function, whereas the finite difference needs more function evaluations (2 instead of 1) even exploiting the separability. As a result, for $n = 10^6$ is easily notable that the time is almost doubled between the two methods, while for $n = 10^4$ this difference is not visible.
Both approaches leads to the optimum.
Centered finite difference obtains results comparable to Forward finite difference method.

2.3 Case 2

For the implementation: $\gamma : 0.1$ as step-length, *tollgrad*:10^{-12}, *tolx*: 10^{-6}
For the Armijo's condition: $\alpha : 0.5$, $\rho : 0.8$, *btmax*: 50 (max number of backtracking operations for Armijo), $c_1 : 10^{-4}$.

n	k	Fmin	Time	iterations
10^4	2	-2500	0.0372	26
	4	-2500	0.0412	27
	6	-2500	0.0399	27
	8	-2500	0.0315	27
	10	-2500	0.0453	27
	12	-2500	0.0401	27
10^6	2	-2500	3.9186	29
	4	-2500	3.8126	29
	6	-2500	3.7891	30
	8	-2500	3.6651	30
	10	-2500	3.9072	30
	12	-2500	3.8865	30

Table 3: Forward finite difference

n	k	Fmin	Time	iterations
10^4	2	-2500	0.0211	27
	4	-2500	0.0277	27
	6	-2500	0.0157	27
	8	-2500	0.0192	27
	10	-2500	0.0223	27
	12	-2500	0.0187	27
10^6	2	-250000	2.0218	30
	4	-250000	1.9823	30
	6	-250000	2.1118	30
	8	-250000	2.0911	30
	10	-250000	2.2823	30
	12	-250000	2.0181	30

Table 4: Exact derivatives

Results are similar to Case 1. In this case γ is lower than in Case 1 and as a result, the algorithm converge towards the optimum with an higher number of iterations that also increases computational time. The difference between the two approaches (FD and exact derivatives) is now visible even for $n = 10^4$, since the number of iteration increased.

2.4 Final comment

Obviously, precise values change at each trial, since there are variables (e.g. x0) that are initialized as random. However there are clearly some behaviours that always occurs.
First of all, the computing time increase a lot while passing from n $= 10^4$ to n $= 10^6$, whereas the number of iterations is almost the same. In seems that the h factor affects has little influence on time and number of iterations. For the choices of parameters in Case 1 and Case 2 the optimum, which is a vector of ones, is always reached.

3 Matlab code

3.1 File: constrained_bcktrck_test.m

```
%% LOADING THE VARIABLES FOR THE TEST

clear
clc
% Init. Armijo's parameters
 alpha0 = 1;
 c1 = 1e-4;
 rho = 0.8;
 btmax = 50;
 disp('**** PARAMETERS: alpha c1 rho btmax *****')
 format short
 [alpha0 c1 rho btmax]

for n = [1e+4 , 1e+6]
    % Variables for data visualization
    iterations = zeros(6,1);
    time = zeros(6,1);
```

```
    res = zeros(6,1);
    fres = zeros(6,1);
    minx = zeros(6,1);
    i = 1;
   for a = [2, 4, 6, 8, 10, 12]

       tic
        x0 = rand(1,n)'+3*rand(1,n)'; % starting point outside the constraint

        kmax = 1000;


        tollgrad = 1e-12;
        h = 10^(-a);%*norm(x0);
        f = @(x)sum(1/4*x.^4 +1/2*x.^2-x);

        f_component = @(x) (1/4*x.^4 +1/2*x.^2-x);


        %finite difference
        gradf = @(x) findiff_grad(f_component, x, h, 'c');% c: centered, fw: forward, None: exact der



        %set constraints
        mins= ones(n,1);
        maxs= ones(n,1)*2;

        %% RUN THE STEEPEST DESCENT
        % steepest descent params
        gamma = 0.1;
        tolx = 1e-6;

        % Projection function
        Pi_X = @(x) box_projection(x,mins,maxs);

        [xk, fk, gradfk_norm, deltaxk_norm, k] = ...
            constr_steepest_desc_bcktrck(x0, f,  gradf, alpha0, kmax, ...
            tollgrad, c1, rho, btmax, gamma, tolx, Pi_X);

        % output
        time(i) = toc;
        iterations(i) = k;
        fres(i) = fk;
        minx(i) = min(xk);
        i = i+1;

   end
    disp(['**** STEEPEST DESCENT N:',num2str(n),' *****'])
    format short

   [time iterations minx fres/1e4]
end
```

3.2 File: box_projection_.m

```
function xhat = box_projection(x, mins, maxs)
%
% function [xhat] = box_projection(x, mins, maxs)
%
% Function that performs the projection of a vector on the boundaries of an
% n-dimensional box, if the vector is not inside it.
%
% INPUTS:
% x = n-dimensional vector;
% mins = n-dimensional vector where the i-th element is the left boundary
% of the i-th interval chararacterizing the box;
% maxs = n-dimensional vector where the i-th element is the right boundary
% of the i-th interval chararacterizing the box;
%
% OUTPUTS:
% xhat = it is x if x is in the box, otherwise it is the projection
% of x on the boundary of the box.
%

xhat = max(min(x, maxs), mins);


end
```

3.3 File: findiff_grad.m

```
function [gradfx] = findiff_grad(f, x, h, type)
%
% function [gradf] = findiff_grad(f, x, h, type)
%
% Function that approximate the gradient of f in x (column vector) with the
% finite difference (forward/centered) method.
%
% 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 gradf
% type = 'fw' or 'c' for choosing the forward/centered finite difference
% computation of the gradient.
%
% OUTPUTS:
% gradfx = column vector (same size of x) corresponding to the approximation
% of the gradient of f in x.

gradfx = zeros(size(x));

 h = h * norm(x);

switch type
    case 'fw'
%       Without separability
%         for i=1:length(x)
%            xh = x;
```

```
%             xh(i) = xh(i) + h;
%             prova = (f(xh) - f(x))/ h;
%             prova
%             gradfx(i) = sum((f(xh) - f(x))/ h);
%
%         end

%      EXPLOIT SEPARABILITY
        xh = x+h;
        gradfx = ((f(xh)-f(x))/h);
    case 'c'
%          Without separability
%         for i=1:length(x)
%             xh_plus = x;
%             xh_minus = x;
%             xh_plus(i) = xh_plus(i) + h;
%             xh_minus(i) = xh_minus(i) - h;
%             gradfx(i) = (f(xh_plus) - f(xh_minus))/(2 * h);
%         end
%      EXPLOIT SEPARABILITY
            xh_plus = x+h;
            xh_minus = x-h;
            gradfx = ((f(xh_plus)-f(xh_minus))/(2*h));
    otherwise %
          % exact derivatives
         gradf = @(x) (x.^3 + x -1);
         gradfx = gradf(x);
end


end
```

3.4 File: constr_steepest_desc_bcktrck.m

```
function [xk, fk, gradfk_norm, deltaxk_norm, k] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, alpha0, ...
    kmax, tollgrad, c1, rho, btmax, gamma, tolx, Pi_X)
%
% 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;
% alpha0 = the initial factor that multiplies the descent direction at each
% iteration;
% 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 = factor that multiplies the descent direction before the (possible) projection;
% tolx = a real scalar value characterizing the tolerance with respect to the norm of  |xk+1 - xk| in o
% 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)
% k = index of the last iteration performed


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

% Initializations

xk = Pi_X(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
    % Compute the descent direction
    pk = -gradf(xk);

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

    % Reset the value of alpha
    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);

    bt = 0;
    % Backtracking strategy:
    % 2nd condition is the Armijo (w.r.t. pik) condition not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, xk, 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_norm = norm(gradf(xk));

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

end

end
```