Stochastic Optimization: Project 1

Vittoria Civiero, Filippo Grobbo

June 2022

Introduction to the problem and theoretical aspects

Let us consider the problem:

$$\min_{\forall x \in X} f(x) \quad \text{with} \quad f : R^n \rightarrow R \tag{1}$$

where f in this case is the De Jong's function $f(x) = \sum_{i=1}^{n} x_i^2$. A possible way to solve the problem is using the Projected Gradient method applied to the steepest descent. This algorithm finds the minimum of a general function, into a feasible set, through different type of projection. The main idea of this process could be summarised below:

Algorithm 1 constr steepest desc bcktrck

function constr steepest desc bcktrck
Input
- x0: starting point
- f: function to minimize
- gradf: gradient of f
- kmax: maximum number of iterations
- tolgrad: tolerance to stop the method with respect to gradf
- c1: parameter for Armijo condition
- rho: fixed factor to reduce alpha in the Armijo condition
- btmax: maximum number of steps in the backtracking strategy
- γ: fixed factor for the descent direction
- tolx: tolerance to stop the method with respect to x
- Pix: projection function

Output
- xk: last x computed
- fk: value of f(xk)
- gradfk norm: value of the norm of gradf(xk)
- deltaxk norm: $||x_k - x_{k-1}||$ in the last iteration
- k: number of iteration done
- xseq: vector of x calculated during the process
- btseq: vector of backtracking's number of iterations

Initialize xk, fk, gradfk, grafk norm and deltaxk norm with respect to x0
while k<kmax & gradfk norm≥tolgrad & deltaxk norm≥tolx **do**
- Compute the steepest descent direction $p_k = -$ gradf(x_k)
- Compute $\hat{x}_k =$ Pix $(x_k + \gamma p_k)$
- Compute the feasible direction $\pi_k = (\hat{x}_k - x_k)$
- Obtain the optimal steplength α for x_{k+1} applying the Backtracking strategy
- Compute the next step $x_{k+1} = x_k + \alpha \pi_k$

end while
end function

In order to compute the steepest descent direction we considered three alternatives:

- Exact derivatives: $p_k = -\nabla f(x_k)$
- Forward finite differences (type='fw'): $p_{k_i} = -\frac{\partial f(x_k)}{\partial x_i} \approx \frac{f(x_k+he_i)-f(x_k)}{h} \quad \forall i$
- Centered finite differences (type='c'): $p_{k_i} = -\frac{\partial f(x_k)}{\partial x_i} \approx \frac{f(x_k+he_i)-f(x_k-he_i)}{2h} \quad \forall i$

where h = $10^{-k}||x_k||$ with k = 2,4,6,8,10,12, n=10^d, d=3,4,5 and e_i is an n-dimensional vector with all zeros entries except for the i-th one. The feasible set of our problem was $X = \{x \in R^n : -5.12 \leq x_i \leq 5.12 \ \forall i = 1, ..., n\}$, so the projection function is:

$$\text{Pix}(x)_i = \begin{cases} \text{x}_i, & \text{if } -5.12 \leq x_i \leq 5.12 \\ -5.12, & \text{if } y_i < -5.12 \\ +5.12, & \text{if } y_i > 5.12 \end{cases}$$

Analysis and Results

In this section we compare the behaviour of three distinct projected gradient method based on different kinds of derivatives.

In order to speed up the findiff grad's algorithm we directly computed the finite difference with respect to function 1. This helped us to show results also in the highest dimension n=100000 since without this change matlab couldn't show results in reasonable time.

In details, the gradient became:

- type $\neq$'fw' and type $\neq$'c': $\nabla f(x) = 2ix$
- type='fw': $\nabla f(x) = 2ix + hi$
- type='c': $\nabla f(x) = 2ix$

The lack of "h" in the 'c' case justifies the fact that table 1 is a 6x3 matrix while the other two are 2x2 matrix. In addition, we omitted results in the exact case because they correspond to the values of the 'c' case.

In order to analize the trend of the minima we plot tables' values, as shown in Figure 1 and Figure 2.

	n=1000	n=10000	n=100000
k=2	4.289450618017081e-01	2.206331815289144e+02	1.258005811025823e+11
k=4	1.392176015056341e-17	7.247839916827115e-04	1.536864570502553e+00
k=6	1.392360548060494e-17	8.660282707176550e-02	3.839905308451603e+00
k=8	1.392305798957194e-17	8.659704342485960e-02	3.859172802575781e+00
k=10	1.392305253198299e-17	8.659699627447327e-02	3.859163084787475e+00
k=12	1.392305247740916e-17	8.659699580403851e-02	3.859162991915454e+00

Table 1. Minimum with forward finite differences

	min f(x)
n=1000	1.392305247685789e-17
n=10000	8.659699579929085e-02
n=100000	3.859162990977660e+00

Table 2. Minimum with centered finite differences (same with exact derivatives)

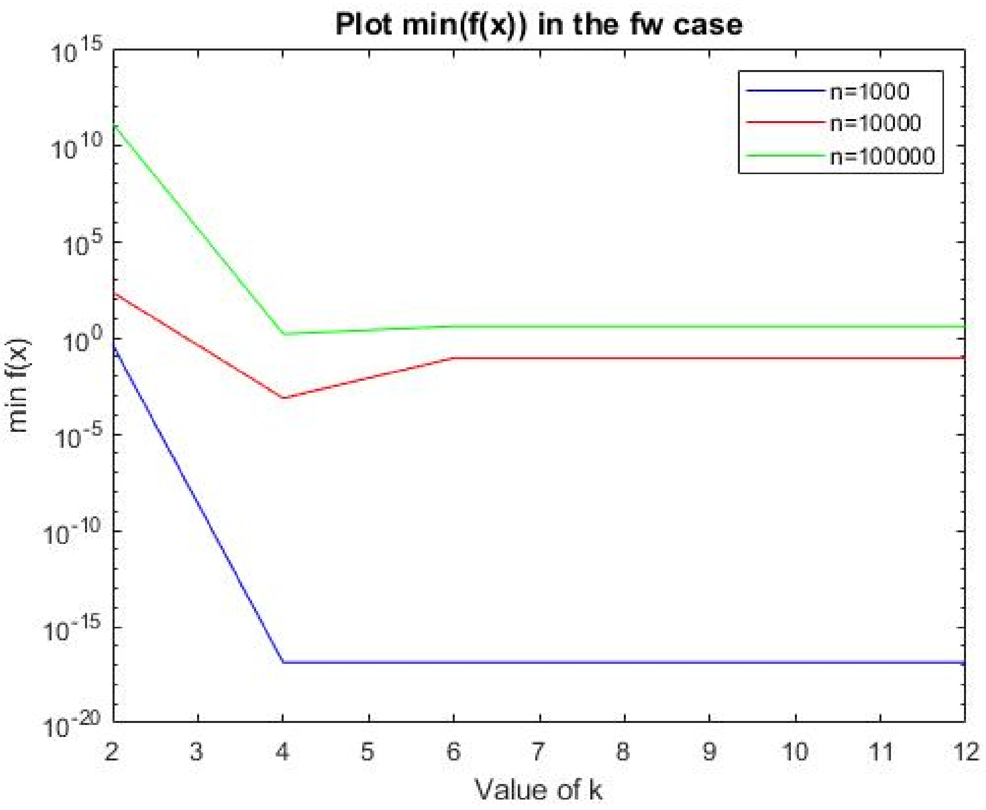

Figure 1. Plot of table 1's values, for Y-axes we use a logarithmic (base 10) scale.

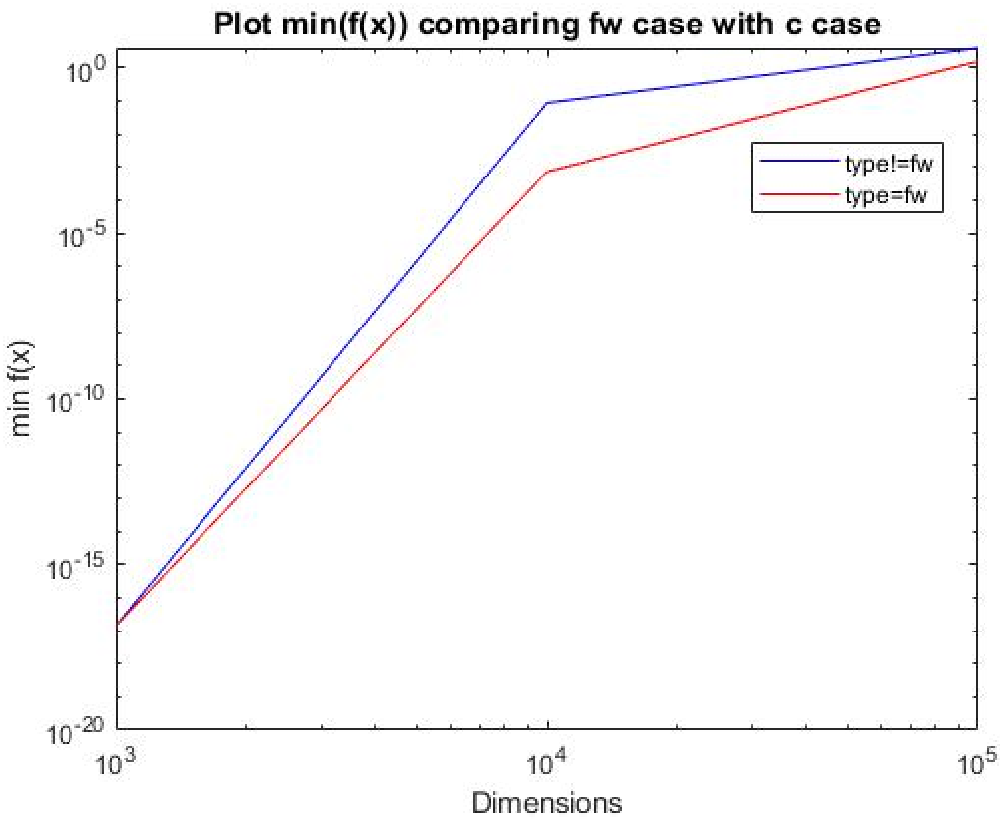

Figure 2. In this plot we compare values of table 1 and table 2 to understand which derivatives' approximation gives better results. In this case we use a logarithmic scale for both X- and Y- axes.

Figure 1 shows that, in the 'fw' case, the more k increases the more values of minimum decrease. However, due to the curse of dimensionality, by increasing the dimension n we have a decay in the results. In general, the best k is 4 which corresponds to the number closest to 0 in each column of Table 1.

To make plot 2 we computed the minimum among k for every dimension. This helped us to compare the three different methods (type!= fw includes 'c' case and exact derivatives case). In conclusion, although finite differences are only an approximations of the gradient they revealed to be optimal to solve the starting problem 1.

Appendix

```
%Main script

clear
close all
clc
format long e

%Initializations
c1 = 1e-4;
rho = 0.8;
btmax = 50;
gamma = 1e-1;
tolx = 1e-12;
ftosave_fw = ones(6,3);
ftosave_grad = ones(3,1);
ftosave_c = ones(3,1);
type = '0';
disp('***********************************')

for d = 3:5
    n = 10^d; %dimension
    d = d-2;
    f = @(x) sum((x.^2).*(1:length(x))'); %De Jong's function
    kmax = 10000;
    tolgrad = 1e-12;

    x0 = -5*ones(n,1);
    box_mins = -5.12*ones(n,1);
    box_maxs = +5.12*ones(n,1);
    Pi_X = @(x) box_projection(x, box_mins, box_maxs); %projection ...
        function

```

```
    switch type
        case "fw"
            for t = 12:12 %2:2:12
                gradf = @(x) findiff_grad(f, x, t, type);

                %RUN THE NEWTON METHOD ON De Jong function
                disp('**** CONSTR. STEEPEST DESCENT: START De Jong ...
                    function *****')
                disp('                         ...')
                [xk_n, fk_n, gradfk_norm_n, ∆xk_norm_n, k_n, xseq_n, ...
                    btseq_n] = ...
                    constr_steepest_desc_bcktrck(x0, f, gradf, ...
                    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X);
                disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
                disp('************************************')
                ftosave_fw(t/2,d)=fk_n;
            end
        case "c"
            gradf = @(x) findiff_grad(f, x, 1, type); %t=1 because ...
                it isn't used in findiff_grad

            %RUN THE NEWTON METHOD ON De Jong function
            disp('**** CONSTR. STEEPEST DESCENT: START De Jong ...
                function *****')
            disp('                         ...')
            [xk_n, fk_n, gradfk_norm_n, ∆xk_norm_n, k_n, xseq_n, ...
                btseq_n] = ...
                constr_steepest_desc_bcktrck(x0, f, gradf, ...
                kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X);
            disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
            disp('************************************')
            ftosave_c(d)=fk_n;
        otherwise
            gradf = @(x) 2*x.*(1:length(x))';


```

```
            %RUN THE NEWTON METHOD ON De Jong function
            disp('**** CONSTR. STEEPEST DESCENT: START De Jong ...
                function *****')
            disp('                    ...')
            [xk_n, fk_n, gradfk_norm_n, Δxk_norm_n, k_n, xseq_n, ...
                btseq_n] = ...
                constr_steepest_desc_bcktrck(x0, f, gradf, ...
                kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X);
            disp('**** CONSTR. STEEPEST DESCENT: FINISHED *****')
            disp('************************************')
            ftosave_grad(d) = fk_n;
    end
end

%Results
switch type
    case 'fw'
        [(2:2:12)', ftosave_fw]
    case 'c'
        [(3:5)', ftosave_c]
    otherwise
        [(3:5)', ftosave_grad]
end

%Plotting results
vec1k = ...
    [4.289450618017081e-01,1.392176015056341e-17,1.392360548060494e-17,...
    1.392305798957194e-17,1.392305253198299e-17,...
    1.392305247740916e-17]; %vector of minima in fw case n=1000
vec10k = [2.206331815289144e+02,7.247839916827115e-04,...
    8.660282707176550e-02,8.659704342485960e-02,...
    8.659699627447327e-02,8.659699580403851e-02]; %vector of minima ...
        in fw case n=10000
vec100k = [1.258005811025823e+11,1.536864570502553e+00,...
    3.839905308451603e+00,3.859172802575781e+00,...
```

```
    3.859163084787475e+00,3.859162991915454e+00];%vector of minima ...
        in fw case n=100000
vec_c = ...
    [1.392305247685789e-17,8.659699579929085e-02,3.859162990977660e+00]; ...
    %vector of minima in c case (the same in the exact derivatives case)

semilogy(2:2:12,vec1k,'b',2:2:12,vec10k,'r',2:2:12,vec100k,'g')
title('Plot min(f(x)) in the fw case','FontSize',12);
legend('n=1000','n=10000','n=100000');
xlabel('Value of k')
ylabel('min f(x)')

%Second plot
loglog([1000,10000,100000],vec_c,'b',...
    [1000,10000,100000],[min(vec1k),min(vec10k),min(vec100k)],'r')
title('Plot min(f(x)) comparing fw case with c case','FontSize',12);
legend('type!=fw ', 'type=fw')
xlabel('Dimensions')
ylabel('min f(x)')
```

```
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
```

```
function [gradfx] = findiff_grad(f, x, t, 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.

    %Initializations
    h=(10^(-t))*norm(x);
    gradfx = zeros(size(x));

    switch type
        case 'fw'
```

```
            for i=1:length(x)
                %xh = x;
                %xh(i) = xh(i) + h;
                %gradfx(i) = (f(xh) - f(x))/ h; original implementation
                gradfx(i)=2*x(i)*i+h*i; %adapted implementation
            end
        case 'c'
            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); ...
                    original
                %implementation
                gradfx(i)=2*x(i)*i; %adapted implementation
            end
        otherwise % repeat the 'fw' case
            for i=1:length(x)
                %xh = x;
                %xh(i) = xh(i) + h;
                %gradfx(i) = (f(xh) - f(x))/h; original implementation
                gradfx(i)=2*x(i)*i+h*i; %adapted implementation
            end
    end
end
```

```
function [xk, fk, gradfk_norm, Δxk_norm, k, xseq, btseq] = ...
    constr_steepest_desc_bcktrck(x0, f, gradf, ...
    kmax, tolgrad, c1, rho, btmax, gamma, tolx, Pi_X)

    % Projected gradient method (steepest descent) for constrained ...
        optimization.

```

```
    % 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);
    ∆xk_norm = tolx + 1;

    while k < kmax & gradfk_norm ≥ tolgrad & ∆xk_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, Δxk_norm
        Δxk_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
```