Homework Unconstrained Optimization

Simone Chiominto, Francesco Pappone, Giuseppe Intilla

December 2021

The aim of this work is to test the *steepest descent* and the *Fletcher and Reeves* methods for nonlinear unconstrained optimization problems. This text first provides the description of the problem, then explains the idea behind each of the two methods. The two methods are tested on the Rosenbrock function in a 2d setting and then on three different functions in higher dimension.

1 Problem setting

Definizione 1.1. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a deterministic scalar function, the problem

$$\min_{x \in \mathbb{R}^n} f(x)$$

is called an *unconstrained optimization problem.*

The goal of an unconstrained optimization problem is to find a global minimum for the function f. We recall now the definition of global minimum

Definizione 1.2. A vector x^* is a *global minimum* of f if $f(x^*) \leq f(x)$ for all feasible x. (In this case all $x \in \mathbb{R}$ are considered feasible).

For a generic f it is not assured the existence and the uniqueness of a global minimum in $\mathbb{R}^n$. If f is lower semicontinuous and coercive we have the following statement

Teorema 1.3. *Let $C \in \mathbb{R}^n$ be a closed set and $f : C \to \mathbb{R}$ a lower semicontinuous coercive function, i.e:*

- (Lower semicontinuity) *for every $x \in C$ and for every sequence $x_n \to x$ we have*
$$\liminf_n f(x_n) \geq f(x)$$
- (Coercive)
$$\lim_{||x|| \to \infty} f(x) = +\infty.$$

Then it exists x^* global minimum for $f(x)$ in C.

Another important theorem is a necessary condition for x to be a minimum for f.

Teorema 1.4. *Let x^* be a (local) minimum for f and $f : C \to \mathbb{R}$ a continuously differentiable function in a neighborhood of x^*, then $\nabla f(x^*) = 0$.*

2 Theoretical description of *steepest descent* and the *Fletcher and Reeves* methods

These two methods iterative methods aim to build a sequence x_n approaching the global minimum x^*. The sequence, given the first point x_0, is recursively built as

$$x_{n+1} = x_n + \alpha_n p_n,$$

where $\alpha_n > 0$ and $p_n \in \mathbb{R}^n$ a descent direction, i.e for small α_n, $f(x_n + \alpha_n p_n) > f(x_n)$. The choice of p_n is the main difference between the two methods.

2.1 Steepest descent method

The name is self explanatory. The idea is to choose p_n as the locally steepest descent direction, therefore it is easy to prove that $p_n = -\nabla f(x_n)$. The iteration can be written as

$$x_{n+1} = x_n - \alpha_n \nabla f(x_n),$$

where α_n need to be determined. At each iteration the main computational cost is the computation of the gradient, that when it is computed via finite differences is $(n + 1)$ times the cost of the computation cost of the function f.

2.2 Fletcher and Reeves method

The steepest descent method gives the *locally* steepest descent direction, but this greedy choice it is shown not to be the best in term of efficiency. In the Fletcher and Reeves method p_n is perturbed from the steepest descent direction tacking track of the direction of the previous iteration:

$$p_{n+1} = -\nabla f(x_{n+1}) + \beta_{n+1} p_n.$$

In the case of the Fletcher and Reeves method β_{n+1} is chosen as

$$\beta_{n+1} = \frac{||\nabla f(x_{n+1})||^2}{||\nabla f(x_n)||^2}.$$

The computational cost at each iteration is similar to the steepest descent method, but it usually converges to the minimum in less steps.

2.3 Backtracking line search

Regarding the choice of α_n there are two consideration that have to be done:

1. not all values of α_n are suitable. If α_n is too big $f(x_{n+1})$ might be bigger than $f(x_n)$;
2. choosing α_n too small might make x_n converge very slowly.

As a consequence, we want to choose an α_n than guarantees a sufficient decrease for f. A popular condition is the following:

Definizione 2.1 (Armijo condition)**.**

$$f(x_{n+1}) = f(x_n + \alpha_n p_n) \leq f(n) + c_1 \alpha_n \nabla f(x_n)^T p_n.$$

The idea behind this condition is that we want $f(x_{n+1})$ to be at minimum smaller than $f(x_n)$ more than a fraction of the expected rate $f(x_n)^T p_n$ of decrease near x_n for the descent direction p_n. In actual implementations α_n is chosen using a backtracking strategy, i.e

- choose an initial step length $\alpha_n^{(0)}$;
- for $0 \leq j \leq N$ if Armijo condition is satisfied accept $\alpha_n^{(j)}$, otherwise $\alpha_n^{(j+1)} = \rho \alpha_n^{(j)}$ with a given $\rho \in [\rho_L, \rho_U]$, $\rho_U < 1$.

In fact choosing α_n this way, if $j < N$ it is guaranteed to have a sufficient decrease but it also guarantees to have $\alpha_n \geq \rho^N \alpha_n^{(0)}$.

3 Tests on the Rosebrock function

In both methods we hove chosen $||\nabla(f(n_n))|| < \varepsilon$ as stopping criterion with $\varepsilon = 10^{-4}$ and 1000 as maximum number of iteration permitted. Let's consider the *Rosebrock* function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

and test the method with $x_0 = (1.2, 1.2)$ and $x_0 = (-1.2, 1)$. The results are summarised in the table below:

Method	x_0	$n_{\max}$	$\|\|\nabla f(x_{n_{\max}})\|\|$	$\|\|f(x_{n_{\max}})\|\|$	$\|\|x_{n_{\max}} - x^*\|\|$
Steepest descent	$(1.2, 1.2)$	1000	0.1479	0.0032	0.1283
Fletcher and Reeves	$(1.2, 1.2)$	129	8.0690×10^{-5}	3.2220×10^{-9}	1.2699×10^{-4}
Steepest descent	$(-1.2, 1)$	1000	0.6103	0.0895	0.7498
Fletcher and Reeves	$(-1.2, 1)$	170	5.2133×10^{-5}	2.7635×10^{-9}	1.1763×10^{-4}

It is evident how much better the Fletcher and Reeves method performs in comparison to the steepest descent method. In fact in the case of the Fletcher and Reeves method the global minimum $x^* = (1, 1)$ is approximated very well with less than 200 iteration while the steepest descent totally failed to approach x^* when $x_0 = (-1.2, 1)$ even after the maximum number (1000) of iterations.

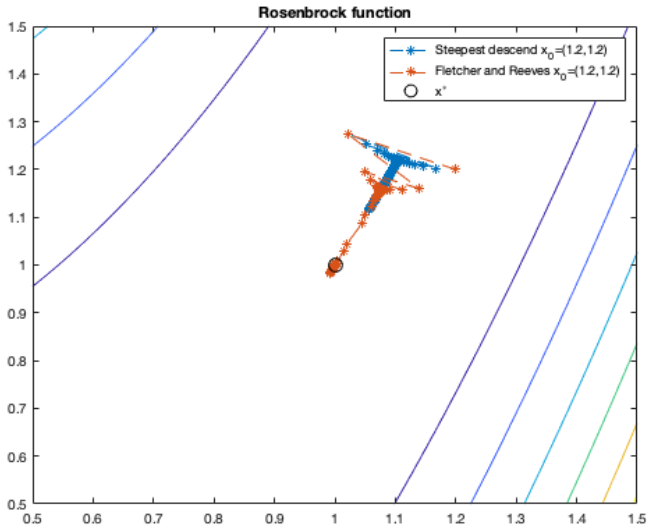


Figura 1: Contour plot with curve levels with sequence x_n for $x_0 = (1.2, 1.2)$

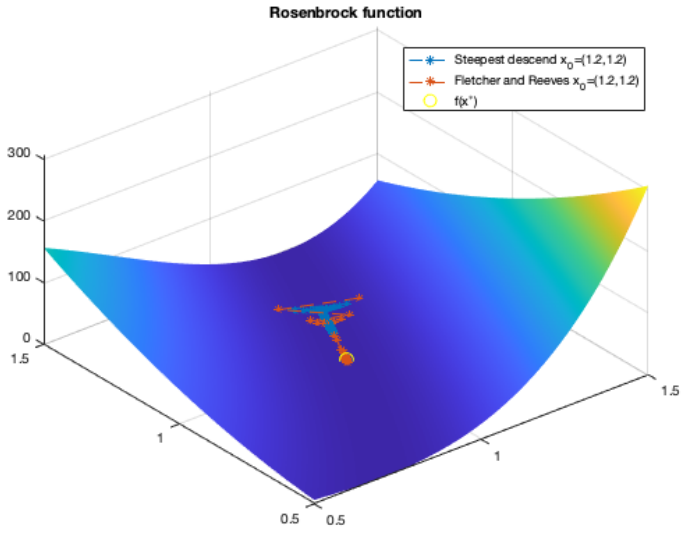


Figura 2: Surf plot with the sequence $f(x_n)$ for for $x_0 = (1.2, 1.2)$

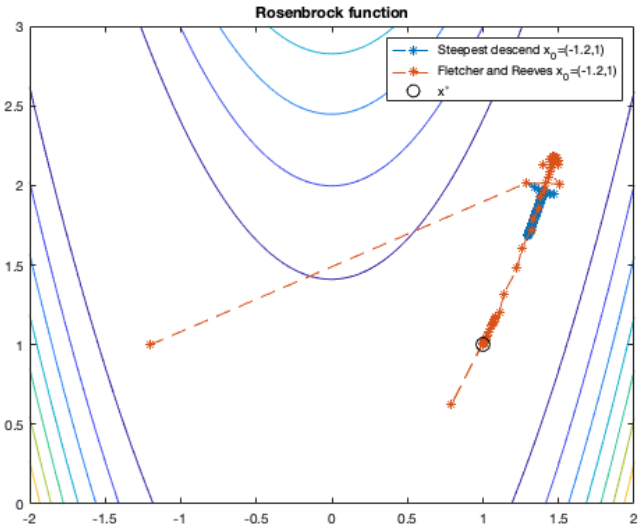


Figura 3: Contour plot with curve levels with sequence x_n for $x_0 = (-1.2, 1)$

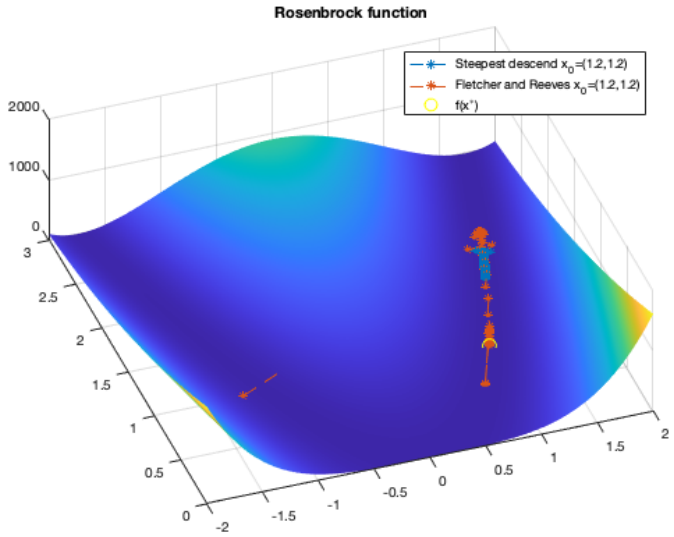


Figura 4: Contour plot with curve levels with sequence x_n for $x_0 = (-1.2, 1)$

4 Tests on high dimension functions

We have tested the two methods onto these three functions

Function 1: Extended Powell singular function

$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

$$f_k(x) = \begin{cases} x_k + 10x_{k+1} & \mod(k,4) = 1 \\ \sqrt{5}(x_{k+2} - x_{k+3}) & \mod(k,4) = 2 \\ (x_{k+1} - 2x_{k+2})^2 & \mod(k,4) = 3 \\ \sqrt{10}(x_k - x_{k+3})^2 & \mod(k,4) = 0 \end{cases}$$

Function 2

$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

$$f_k(x) = \begin{cases} x_k - 1 & k = 1 \\ 10(k-1)(x_k - x_{k-1})^2 & 1 < k \leq n \end{cases}$$

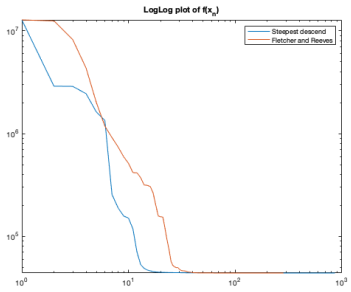
Function 3: Chained Rosenbrock function

$$F(x) = \sum_{i=2}^{n}[100(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2]$$

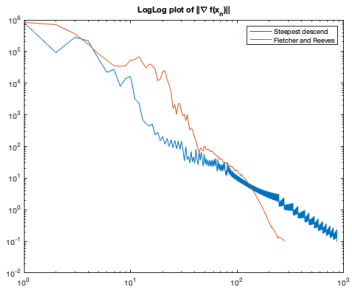
in dimension $n = 10^4$. We have tested the methods from 10 different initial points x_0 chosen randomly as $x_n^{(i)} \sim Normal(0, 9)$. The results are summarised in the following table for function 1,2,3.

Function	Method	$n_{\max}$ (mean)	$\|\|\nabla f(x_{n_{\max}})\|\|$ (mean, min, max)	$\|\|f(x_{n_{\max}})\|\|$ (mean, min, max)
1	Steepest descent	897.4	0.0973 , 0.0929 , 0.0995	44450 , 42220 , 46305
1	Fletcher and Reeves	385.6	0.0979 , 0.0951 , 0.0994	44450 , 42220 , 46304
2	Steepest descent	5000	40780 , 21000 , 103770	10976 , 52270 , 260390
2	Fletcher and Reeves	5000	52.026 , 15.315 , 138.07	1.509 , 0.540 , 4.191
3	Steepest descent	5000	3.5341 , 1.0968 , 5.4591	9998.5 , 9996.1 , 9999.7
3	Fletcher and Reeves	5000	282.49 , 11.198 , 673.92	10200 , 9992 , 10622

In figure 5 it is showed respectively the decay of $||f(x_n)||$ and $||\nabla f(x_n)||$ against n, in figure 6 for function 2 and in figure 7 for function 3 in the last choice of x_0.

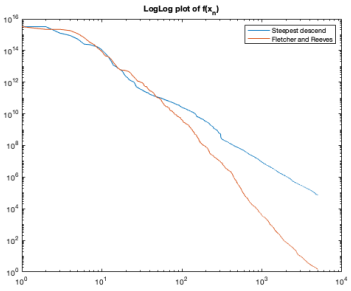


(a) $||f(x_{n_{\max}})||$

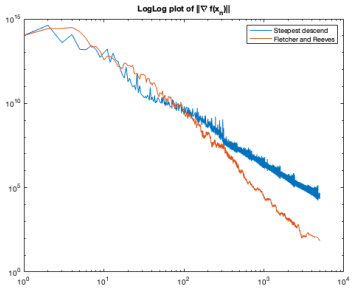


(b) $||\nabla f(x_{n_{\max}})||$

Figura 5: LogLog plot of the decay of of $||f(x_n)||$ and $||\nabla f(x_n)||$ against n for function 1

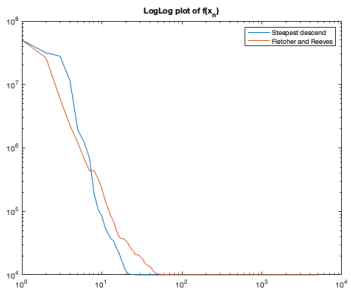


(a) $||f(x_{n_{\max}})||$

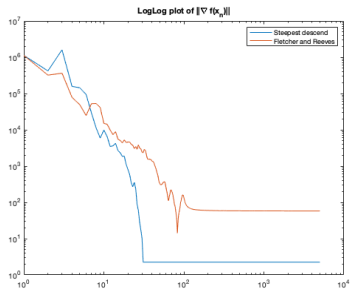


(b) $||\nabla f(x_{n_{\max}})||$

Figura 6: LogLog plot of the decay of of $||f(x_n)||$ and $||\nabla f(x_n)||$ against n for function 2



(a) $||f(x_{n_{\max}})||$



(b) $||\nabla f(x_{n_{\max}})||$

Figura 7: LogLog plot of the decay of of $||f(x_n)||$ and $||\nabla f(x_n)||$ against n for function 3

5 Conclusions

It is noticeable how both methods get to satisfy our stopping criterion of $||\nabla f(x_n)|| < 0.1$ only when solving the first problem and in this case Fletcher and Reeves method reaches the condition in less than half the steps taken by steepest descent methods as we expected from the theory. The results are comparable in both methods and in both cases they all are are far from optimal since the function evaluated at the origin is equal to 0. Speaking about problem 2, both methods fail to reach the stopping criterion of $||\nabla f(x_n)|| < 0.1$ starting from all x_0 taken in consideration after 5000 iterations. Despite this, Fletcher and Reeves seems to perform much better in term of $||f(x_n)||$, since it gets very close to 0 from all x_0 considering how high is the dimension. In problem 3 both methods perform quite bad, none of them seems to approach the optimal solution in any of the ten different runs. Steepest descent method in this case seems to perform sightly better.

6 Appendix

Listing 1: Steepest descent method with backtracking

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    steepest_desc_bcktrck(x0, f, gradf, FDgrad, h, alpha0, ...
    kmax, tolgrad, c1, rho, btmax)

% Function that performs the steepest descent optimization method,
% for a given function for the choice of the step length alpha.
%
% 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;
% FDgrad = 'fw' (FD Forward approx. for gradf), 'c' (FD Centered
% approx. for gradf), any other string (usage of input Hessf);
% h = approximation step for FD (if used);
% 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 = 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.
%
% 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
% xseq = n-by-k matrix where the columns are the xk computed during
% the iterations
% btseq = 1-by-k vector whose 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, h, 'fw');

    case 'c'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'c')
        gradf = @(x) findiff_grad(f, x, h, 'c');
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 = x0(:);
fk = f(xk);
gradfk = gradf(xk);
k = 0;
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    % Compute the descent direction
    pk = -gradfk;

    % Reset the value of alpha
    alpha = alpha0;

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

    bt = 0;
    % Backtracking strategy:
    % 2nd condition is the Armijo condition not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, gradfk, pk)
        % Reduce the value of alpha
        alpha = rho * alpha;
        % Update xnew and fnew w.r.t. the reduced alpha
        xnew = xk + alpha * pk;
        fnew = f(xnew);

        % Increase the counter by one
        bt = bt + 1;
    end

    % Update xk, fk, gradfk_norm
    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
```

Listing 2: Fletcher and Reeves method with backtracking

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    fletcher_reeves(x0, f, gradf, FDgrad,h, alpha0, ...
    kmax, tolgrad, c1, rho, btmax)

% Function that performs Fletcher and Reeves optimization method,
% for a given function for the choice of the step length alpha.
%
% 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;
% FDgrad = 'fw' (FD Forward approx. for gradf), 'c' (FD Centered
% approx. for gradf), any other string (usage of input gradf);
% h = approximation step for FD (if used);
% 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 = 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.
%
% 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
% xseq = n-by-k matrix where the columns are the xk computed during
% the iterations
% btseq = 1-by-k vector whose 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, h, 'fw');

    case 'c'
        % OVERWRITE gradf WITH A F. HANDLE THAT USES findiff_grad
        % (with option 'c')
        gradf = @(x) findiff_grad(f, x, h, 'c');
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 = x0(:);
fk = f(xk);
gradfk = gradf(xk);
k = 0;
gradfk_norm = norm(gradfk);
pk=-gradf(xk);

while k < kmax && gradfk_norm >= tolgrad

    % Reset the value of alpha
    alpha = alpha0;

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

    bt = 0;
    % Backtracking strategy:
    % 2nd condition is the Armijo condition not satisfied
    while bt < btmax && fnew > farmijo(fk, alpha, gradfk, pk)
        % Reduce the value of alpha
        alpha = rho * alpha;
        % Update xnew and fnew w.r.t. the reduced alpha
        xnew = xk + alpha * pk;
        fnew = f(xnew);
        % Increase the counter by one
        bt = bt + 1;
    end

    % Compute the descent direction
    gradfnew=gradf(xnew);
    gradfnew_norm=norm(gradfnew);
    betanew=gradfnew_norm^2/gradfk_norm^2;
    pk =-gradfnew+ betanew* pk;

    % Update xk, fk, gradfk_norm
    xk = xnew;
    fk = fnew;
    gradfk = gradfnew;
```

```
    gradfk_norm = gradfnew_norm;

    % 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
```

Listing 3: Function to evaluate gradient via finite differences

```
function gradfx = findiff_grad(f,x,h,type)
%FINDIFF_GRAD compute the appossimation of the gradient via finite
%differences
%
% INPUTS:
% f = function handle that describes a function R^n->R;
% x = n-dimensional column vector;
% h = approximation step for FD (if used);
% gradf = function handle that describes the gradient of f;
% type = 'fw' (FD Forward approx. for gradf), 'c' (FD Centered
% approx. for gradf);
%
% OUTPUTS:
% gradfx=the appossimation of the gradient in x via finite differences
%

gradfx=zeros(length(x),1);
ei=zeros(length(x),1); ei(1)=1;
if type=="c"
    for i=1:length(x)
        if i>1
            ei(i-1)=0; ei(i)=1;
        end
        gradfx(i)= (f(x+h*ei)-f(x-h*ei))/(2*h);
    end

else
    fx=f(x);
    for i=1:length(x)
        if i>1
            ei(i-1)=0; ei(i)=1;
        end
        gradfx(i)= (f(x+h*ei)-fx)/h;
    end
end

end
```

Listing 4: Test script

```
% Homework uncostrained optimization

clear all
for i=1:10
% generate random starting point x0
x0(:,i)=randn(10000,1)*3;
h=0.01;
kmax=5000;
tolgrad=0.1;
c1=1e-4;
rho=0.5;
btmax=100;
alpha0=1;


% Optimize Function 1
f1=@(x) Problem_26(x);
gradf1=@(x) grad_Problem_26(x);
[xk1(:,i), fk1(i), gradfk_norm1(i), k1(i), xseq1, btseq1] = ...
    steepest_desc_bcktrck(x0(:,i), f1,gradf1, "",h, alpha0, ...
    kmax, tolgrad, c1, rho, btmax);

[xk_fr1(:,i), fk_fr1(i), gradfk_norm_fr1(i), k_fr1(i), ...
    xseq_fr1, btseq_fr1] = ...
    fletcher_reeves(x0(:,i), f1,gradf1, "",h, alpha0, kmax,...
    tolgrad, c1, rho, btmax);

% Optimize Function 2
f2=@(x) Problem_75(x);
gradf2=@(x) grad_Problem_75(x);
[xk2(:,i), fk2(i), gradfk_norm2(i), k2(i), xseq2, btseq2] = ...
    steepest_desc_bcktrck(x0(:,i), f2,gradf2, "",h, alpha0,...
    kmax, tolgrad, c1, rho, btmax);

[xk_fr2(:,i), fk_fr2(i), gradfk_norm_fr2(i), k_fr2(i),...
    xseq_fr2, btseq_fr2] = ...
    fletcher_reeves(x0(:,i), f2,gradf2, "",h, alpha0, kmax,...
    tolgrad, c1, rho, btmax);

%Optimize Function 3
f3=@(x) Problem_1(x);
gradf3=@(x) grad_Problem_1(x);

[xk3(:,i), fk3(i), gradfk_norm3(i), k3(i), xseq3, btseq3] = ...
    steepest_desc_bcktrck(x0(:,i), f3,gradf3, "",h, alpha0,....
    kmax, tolgrad, c1, rho, btmax);

[xk_fr3(:,i), fk_fr3(i), gradfk_norm_fr3(i), k_fr3(i),...
    xseq_fr3, btseq_fr3] = ...
    fletcher_reeves(x0(:,i), f3,gradf3, "",h, alpha0, kmax,...
    tolgrad, c1, rho, btmax);

end

%%
% Plot from the last iteration
```

```
% loglog plot of f1(x_i) and ||gradf1(x_i)||
for k=1:k1(i)
    grad_norm1(k)=norm(gradf1(xseq1(:,k)));
    y1(k)=f1(xseq1(:,k));
end
 for k=1:k_fr1(i)
     grad_norm1_fr(k)=norm(gradf1(xseq_fr1(:,k)));
     y1_fr(k)=f1(xseq_fr1(:,k));
 end

fig1=figure();
loglog(1:k1(i),y1)
hold on
y1_fr(k1(i))=0;
loglog(1:k1(i),y1_fr)
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of f(x_n)")
hold off

fig2=figure();
loglog(1:k1(i),grad_norm1)
hold on
grad_norm1_fr(k1(i))=0;
loglog(1:k1(i),grad_norm1_fr)
hold off
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of ||\nabla f(x_n)||")

% loglog plot of f2(x_i) and ||gradf2(x_i)||
for k=1:k2(i)
    grad_norm2(k)=norm(gradf2(xseq2(:,k)));
    y2(k)=f2(xseq2(:,k));
end

for k=1:k_fr2(i)
    grad_norm2_fr(k)=norm(gradf2(xseq_fr2(:,k)));
    y2_fr(k)=f2(xseq_fr2(:,k));
end

fig3=figure();
loglog(1:k2(i),y2)
hold on
y2_fr(k2(i))=0;
loglog(1:k2(i),y2_fr)
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of f(x_n)")
hold off
fig4=figure();
loglog(1:k2(i),grad_norm2)
hold on
grad_norm2_fr(k2(i))=0;
loglog(1:k2(i),grad_norm2_fr)
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of ||\nabla f(x_n)||")
hold off
```

```
% loglog plot of f3(x_i) and ||gradf3(x_i)||
 for k=1:k3(i)
     grad_norm3(k)=norm(gradf3(xseq3(:,k)));
     y3(k)=f3(xseq3(:,k));
 end

 for k=1:k_fr3
     grad_norm3_fr(k)=norm(gradf3(xseq_fr3(:,k)));
     y3_fr(k)=f3(xseq_fr3(:,k));
 end

fig5=figure();
loglog(1:k3(i),y3)
hold on
y3_fr(k3(i))=0;
loglog(1:k3(i),y3_fr)
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of f(x_n)")
hold off
fig6=figure();
loglog(1:k3(i),grad_norm3)
hold on
grad_norm3_fr(k3(i))=0;
loglog(1:k3(i),grad_norm3_fr)
legend("Steepest descend","Fletcher and Reeves")
title("LogLog plot of ||\nabla f(x_n)||")
hold off
```

Listing 5: Function 1

```
function y=Problem_26(x)
% Extend Powell singular function
%
% Taken from "Test Problems for Unconstrained Optimization"
% Link: https://www.researchgate.net/publication
%/325314497_Test_Problems_for_Unconstrained_Optimization
% Corrected, seeing the original paper [23]
if mod(length(x),4)
    error("length(x) must be a multiple of 4");
end
% I assure x a column vector
x=x(:);

    function z=f(k,x)
        switch mod(k,4)
            case 1
                z= x(k)+10*x(k+1);
            case 2
                z= 5^.5 *(x(k+1)-x(k+2));
            case 3
                z= (x(k-1)-2*x(k))^2;
            case 0
                z= 10^.5 *(x(k-3)-x(k))^2;
        end
    end

    z=zeros(1,length(x));
```

```
    for k=1:length(x)
        z(k)=f(k,x);
    end
    y= 0.5* sum((z).^2);
end
```

Listing 6: Exact gradient of Function 1

```
function y=grad_Problem_26(x)
n=length(x) ;
y=zeros(n,1);
    function z=f(k,x)
        switch mod(k,4)
            case 1
                z= x(k)+10*x(k+1);
            case 2
                z= 5^.5 *(x(k+1)-x(k+2));
            case 3
                z= (x(k-1)-2*x(k))^2;
            case 0
                z= 10^.5 *(x(k-3)-x(k))^2;
        end
    end

    for i=1:n
        switch mod(i,4)
            case 1
                y(i)=f(i,x)+40^0.5*(x(i)-x(i+3))*(f(i+3,x));
            case 2
                y(i)=10*f(i-1,x)+2*(x(i)-2*x(i+1))*f(i+1,x);
            case 3
                y(i)=5^0.5*f(i-1,x)-4*(x(i-1)-2*x(i))*f(i,x);
            case 4
                y(i)=-5^0.5*f(i-2,x)-40^0.5*(x(i-3)-x(i))*f(i,x);
        end

    end
end
```

Listing 7: Function 2

```
function y=Problem_75(x)
% Taken from "Test Problems for Unconstrained Optimization"
% Link: https://www.researchgate.net/publication
%/325314497_Test_Problems_for_Unconstrained_Optimization

    function z=f(k,x)
        if k==1
            z= x(k)-1;
        else
            z=10*(k-1)*((x(k)-x(k-1))^2);
        end
    end

z=zeros(1,length(x));
for k=1:length(x)
    z(k)=f(k,x);
```

```
end
y= 0.5* sum((z).^2);

end
```

Listing 8: Exact gradient of Function 2

```
function y=grad_Problem_75(x)
n=length(x) ;
y=zeros(n,1);
    function z=f(k,x)
        if k==1
            z= x(k)-1;
        else
            z=10*(k-1)*((x(k)-x(k-1))^2);
        end
    end

    for i=1:n
        if i==1
            y(i)=f(i,x)-20*i*(x(i+1)-x(i))*f(i+1,x);
        elseif i==n
            y(i)=20*(i-1)*(x(i)-x(i-1))*f(i,x) ;
        else
            y(i)=20*(i-1)*(x(i)-x(i-1))*f(i,x)-20*i*(x(i+1)-x(i))*f(i+1,x);
        end
    end
end
```

Listing 9: Function 3

```
function y=Problem_1(x)
% Chained Rosenbrock function
%
% Taken from "Test Problems for Unconstrained Optimization"
% Link: https://www.researchgate.net/publication
%/325314497_Test_Problems_for_Unconstrained_Optimization
%
x=x(:);
n=length(x);

    function y=f(k,x)
        y=(100*(x(k-1).^2-x(k)).^2 + (x(k-1)-1).^2);
        y=y(:);
    end

y=sum(f(2:n,x));
end
```

Listing 10: Exact gradient of Function 3

```
function y=grad_Problem_1(x)

x=x(:);
n=length(x);
y=zeros(n,1);
```

```
    function y=f(k,x)
        if k==1
            y=200*x(k)*(x(k)^2-x(k+1))+2*x(k);
        elseif k==n
            y=-200*(x(k-1)^2-x(k))+2*x(k);
        else
            y=-200*(x(k-1)^2-x(k))+2*x(k)+200*x(k)*(x(k)^2-x(k+1))+2*x(k);
        end
    end

for k=1:n
    y(k)=f(k,x);
end

end
```