POLITECNICO DI TORINO

Corso di Laurea
in Ingegneria Matematica

Unconstrained optimization

Group
Lisa Gamarro s318426
Erika Spada s318375
Carlotta Agnese Trovati s318396

Contents

1 Theoretical introduction 2
- 1.0.1 Nelder-Mead method . . . 2
- 1.0.2 Fletcher-Reeves method . . . 3
- 1.0.3 Inexact Newton method . . . 3
- 1.0.4 Parameter values . . . 4

2 Test functions 5
- 2.0.1 Chained Rosenbrock function . . . 5
- 2.0.2 Banded Trigonometric . . . 7
- 2.0.3 Generalization of the Brown function 1 . . . 7
- 2.0.4 Chained Powel singular function . . . 8

3 Results 9

A Matlab code 10

Chapter 1

Theoretical introduction

The purpose of this paper is to implement different methods for unconstrained optimization and to compare their performances when applied to several test problems. The comparisons are also carried out with respect to the dimension of the problem, the starting point considered and the expected behaviour.
The paper is structured as follows: firstly we give an overview of the methods (the ones we chose are the Nelder-Mead method, the Fletcher-Reeves method and the Inexact Newton method), right after that we introduce the test functions, and eventually the results are shown.

1.0.1 Nelder-Mead method

The Nelder-Mead method is an optimisation algorithm which uses a geometric approach, constructing a non-singular simplex with a number of vertices equal to n+1, where n is the number of problem dimensions. The algorithm performs a series of iterations during which the simplex is modified in one point per iteration in order to approach the position of the minimum of the function f. It is assumed that the n+1 points at the k-th iteration $x_1^{(k)}$, $x_2^{(k)}$... $x_{n+1}^{(k)}$ are assigned such that $f(x_1^{(k)}) \leq f(x_2^{(k)}) \leq ... \leq f(x_{n+1}^{(k)})$, from which it follows that $x_1^{(k)}$ is the best point of the simplex S_k and $x_{n+1}^{(k)}$ the worst.
In particular, at each iteration k the following operations are performed:

1. Ordering phase: the value of f is calculated at $x_i^{(k)}$, i = 1,2..n+1 and the vertices are reordered as described above;

2. Reflection phase: the centre of gravity of the n best points $\bar{x}^{(k)}$ is calculated and then a reflection $x_R^{(k)}$ of the worst point $x_{n+1}^{(k)}$ is made with respect to the centre of gravity. The reflection is $x_R^{(k)} = \bar{x}^{(k)} + \rho(\bar{x}^{(k)} - x_{n+1}^{(k)})$, with $\rho > 0$.
   If $f(x_1^{(k)}) \leq f(x_R^{(k)}) \leq f(x_n^{(k)})$, $x_R^{(k)}$ is considered as the new point of the simplex S_{k+1} and the next iteration is immediately carried out;

3. Expansion phase: if $f(x_R^{(k)}) < f(x_1^{(k)})$ the expansion point $x_E^{(k)} = \bar{x}^{(k)} + \chi(x_R^{(k)} - \bar{x}^{(k)})$ is computed, with $\chi > 1$.

If $f(x_E^{(k)}) < f(x_R^{(k)})$, $x_E^{(k)}$ is considered as the new point of the simplex S_{k+1} and the next iteration is immediately carried out;

4. Contraction phase: if $f(x_R^{(k)}) > f(x_n^{(k)})$ the contraction point $x_C^{(k)} = \bar{x}^{(k)} - \gamma(\bar{x}^{(k)} - x_{n+1}^{(k)})$ is computed, with $0 < \gamma < 1$.
If $f(x_C^{(k)}) < f(x_{n+1}^{(k)})$, $x_C^{(k)}$ is considered as the new point of the simplex S_{k+1} and the next iteration is immediately carried out;

5. Shrinking phase: if all the former phases do not lead to significant improvements, a global vertex shortening is performed, bringing the simplex closer to the best vertex $x_1^{(k)}$.
Then, for each vertex $x_i^{(k)}$ different from the best vertex, the new shortened vertex $x_i^{(k+1)} = x_1^{(k)} + \sigma(x_i^{(k)} - x_1^{(k)})$ is calculated, with $0 < \sigma < 1$.

1.0.2 Fletcher-Reeves method

The Fletcher-Reeves method (FR-CG) is an iterative optimization algorithm used for unconstrained optimization problems. It is a variant of the Conjugate Gradient method designed to carry out the minimization of a generic nonlinear function. The key concept behind the algorithm is that the research of the optimal solution is computed using A-conjugate directions; this enables the convergence to be faster. The differences with the simple Conjugate Gradient method lay in the choice of the steplength, which in this case is computed with a line search, and in the residual, substituted by the gradient of the function.
The algorithm is structured as follows:

1. given x_0, compute the value of the function and of its gradient in the starting point, $f_0 = f(x_0)$ and $\nabla f_0 = \nabla f(x_0)$, and set the first descent direction to $p_0 = -\nabla f_0$;

2. for each iteration $k \geq 0$:

   - compute α_k with a line-search strategy;
   - compute the next solution $x_{k+1} = x_k + \alpha_k p_k$ and the associated gradient ∇f_{k+1};
   - compute the parameter $\beta_{k+1} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k} = \frac{\|\nabla f_{k+1}\|^2}{\|\nabla f_k\|^2}$ and the next conjugate direction $p_{k+1} = -\nabla f_{k+1} + \beta_{k+1} p_k$.

The second step is iterated until a stopping criterion is met, such as reaching a certain tolerance level for the residual or a maximum number of iterations.

1.0.3 Inexact Newton method

The Inexact Newton Method is an optimization algorithm that combines the Newton's method with the idea of using approximations to reduce the computational effort. It is especially useful for large-scale problems, since computing the exact Newton step might

be computationally expensive and lead to oversolving.
At each iteration the solution is computed as $x_{k+1} = x_k + p_k^{IN}$, where p_k^{IN} satisfies

$$\left\|\nabla^2 f(x_k)p + \nabla f(x_k)\right\| \leq \eta_k \left\|\nabla f(x_k)\right\|, \quad \eta_k \in [0, \hat{\eta}),\ \hat{\eta} < 1$$

The values η_k are called forcing terms and they determine the rate of convergence. In practice, the residual is not set to zero, but it is controlled through the gradient in order for it to be a small quantity. In order to obtain a good global convergence, in addition to a local one, it is possible to introduce a backtracking line search strategy with $a_k^{(0)} = 1$.
It is possible to further reduce the cost by using some approximations. Among all potential implemented combinations we agreed to use the exact gradient and approximate the hessian through the use of centered finite differences for the Jacobian matrix.

1.0.4 Parameter values

The problems were solved in two different dimensions, specifically $n = 10^3$ and $n = 10^4$. In all the methods, except for Nelder-Mead, a backtracking strategy for the line search has been applied. We chose to use as a reference the parameters previously selected in the practical lessons since they had already been tested.
In particular:

- Maximum number of iterations kmax= 10000;
- Factor that multiplies the descent direction at each iteration $\gamma = 0.9$;
- Initial step length alpha0 = 1;
- Fixed factor used for reducing alpha0 in the Armijo condition, rho = 0.8;
- Maximum number of steps for updating alpha during the backtracking strategy btmax = 50;
- Factor of the Armijo condition c1= 1e-4;
- Stopping criterion w.r.t. the norm of the gradient tollgrad = 1e-7;
- Stopping criterion w.r.t. the norm of x tolx = 1e-6 ;
- Parameters for the Nelder-Mead: rho = 1, chi = 2, gamma = 0.5, sigma = 0.5.

Chapter 2

Test functions

Once all the methods have been implemented, the first step is to test them using a two-dimensional function. The one which is often chosen in these cases is the Rosenbrock function, a non-convex function, defined as follows:

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

with two distinct starting points: $x_0 = (1.2, 1.2)$ and $x_0 = (-1.2, 1)$. This function has a global minimum of 0 at the point $x^* = (1, 1)$. Using the first starting point $x_0 = (1.2, 1.2)$, all three methods converge to the actual minimum x^* very quickly and the error is not relevant. We have a different situation with the the second starting point $x_0 = (-1.2, 1)$, in fact Nelder-Mead and Fletcher and Reeves present good results, but Inexact Newton Method reaches the point $(-0.97, 0.96)$, which is different from the right solution. It is possible that the Hessian matrix is not PD, so the method can not converge.

Method	k	Time	$f(x_k)$	$\|\nabla f(x_k)\|$
NM	31	0.010	1.2075×10^{-7}	0.0125
FR	92	0.018	0.00017	0.0125
INM	1000	4.4849	0.0083	3.7411×10^{-5}

Table 2.1. Results with starting point $x_0 = (1.2, 1.2)$

All tables included represent the best results obtained using the recommended ones as starting points.
For the subsequent testing we selected four functions, listed below.

2.0.1 Chained Rosenbrock function

The function $F(x)$ is defined as:

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

where $\bar{x}_i = -1.2$ if $i \equiv 1 \pmod 2$, and $\bar{x}_i = 1.0$ if $i \equiv 0 \pmod 2$.

Applying the Nelder-Mead algorithm unfortunately did not yield the expected results, as run times were very high, we considered these attempts as failed. This is because this method does not make use of the exact gradient; rather, it performs several transformations to find the minimum. While this can be advantageous in certain scenarios, it might result in slower convergence compared to gradient-based methods, especially for large-scale problems. In high dimensions, the algorithm may struggle to converge efficiently and accurately, leading to slow convergence or getting stuck in local optima. However, we verified that the method itself worked. In fact, trying it with $n = 10^2$ we observe that good results are obtained, so probably with longer runs convergence would have been achieved. This behavior affects not only this function, but also all of the following.
Analyzing the results of the Fletcher and Reeves and Inexact Newton Method, it can be seen that they show different behavior depending on which starting point was chosen. In fact, if one considers the recommended point, neither of the two methods turns out to converge since the point is "far" from the point of minimum and no descent direction is identified.
A different situation arises for the intial points $\bar{x} = 0$ and $\bar{x} = 10$. For the first one, Fletcher and Reeves Method converges for $n = 10^3$, although it performs all iterations, the solution still turns out to be accurate, for $n = 10^4$ the results are even better; the Inexact Newton Method, on the other hand, does not converge in either dimension, after a few iterations the hessian matrix is no longer positive definite (for $n = 10^3$ is not already from the first).
Exactly the opposite behavior is observed for the second chosen point: the Inexact Newton Method converges in a few iterations (57) even with $n = 10^4$.

n	**Method**	k	Time	$f(x_k)$	$\|\nabla f(x_k)\|$
10^3	FR	10000	3.769654	1.1154e-10	0.00033001
10^4	FR	4057	21.769698	9.6966e-18	9.9554e-08

Table 2.2. Results with starting point $\bar{x}_i = 0$ for $i \geq 1$

n	**Method**	k	Time	$f(x_k)$	$\|\nabla f(x_k)\|$
10^3	INM	54	5.336269	2.1536e-18	5.0157e-08
10^4	INM	57	312.716622	2.2154e-18	5.0836e-08

Table 2.3. Results with starting point $\bar{x}_i = 10$ for $i \geq 1$

2.0.2 Banded Trigonometric

The function $F(x)$ is defined as:

$$F(x) = \sum_{i=1}^{n} i\left[(1 - \cos x_i) + \sin x_{i-1} - \sin x_{i+1}\right]$$

where $\bar{x}_0 = \bar{x}_{n+1} = 0$ and $\bar{x}_i = 1$ for $i \geq 1$.

Both the Fletcher and Reeves and Inexact Newton Method converge and return the same function value, considering $n = 10^3$ and $n = 10^4$. The main difference is that the former method performs all iterations, as opposed to the latter, so the solution is less accurate. If one were to increase the number of iterations, a better solution could be achieved; however, this would affect the execution time.
As an alternative point, we chose $\bar{x} = 10$, but in the case of this function, it did not yield positive results. In fact, both methods not only do not converge (sometimes remaining fixed at the initial point), but they "explode", returning very large values. This phenomenon occurs for both dimensions. We also tried using $x = 0$ as the starting point, and here as well we can observe fairly accurate results.

n	**Method**	k	Time	$f(x_k)$	$\|\nabla f(x_k)\|$
10^3	FR	10000	9.096217	-427.4045	0.00021376
10^3	Inexact Newton	1753	88.491947	-427.4045	4.2378×10^{-08}
10^4	FR	10000	99.795359	-4159.9324	0.012283
10^4	Inexact Newton	33	92.353145	-4159.9324	8.8991×10^{-08}

Table 2.4. Results with starting point $\bar{x}_i = 1$ for $i \geq 1$

2.0.3 Generalization of the Brown function 1

The function $F(x)$ is defined as:

$$F(x) = \sum_{j=1}^{k} \left[\frac{(x_{i-1} - 3)^2}{1000} - (x_{i-1} - x_i) + \exp(20(x_{i-1} - x_i))\right]$$

$$+ \left(\sum_{j=1}^{k} x_{i-1} - 3\right)^2,$$

where $i = 2j$, $k = \frac{n}{2}$

$\bar{x}_i = 0$ if $i \mod 2 = 1$, and $\bar{x}_i = -1$ if $i \mod 2 = 0$.

Since it is a function with many irregularities, from none of the chosen points can be obtained a result that converges. We also tried using other parameters and starting

points, but in none of these cases were the results obtained good. It emerges that for a large-scale problem, with a function that has many irregularities, these algorithms make it difficult to converge. In fact, trying to solve the problem in smaller dimensions, they all turn out to converge at the same point.

2.0.4 Chained Powel singular function

The function $F(x)$ is defined as:

$$F(x) = \sum_{j=1}^{k}[(x_{i-1} + 10x_i)^2 + 5(x_{i+1} - x_{i+2})^2 + (x_i - 2x_{i+1})^4$$

$$+10(x_{i-1} - x_{i+2})^4],$$

where $i = 2j$, $k = \frac{n-2}{2}$

where $\bar{x}_i = 3$ if $i \equiv 1 \pmod 4$, $\bar{x}_i = -1$ if $i \equiv 2 \pmod 4$, $\bar{x}_i = 0$ if $i \equiv 3 \pmod 4$, and $\bar{x}_i = 1$ if $i \equiv 0 \pmod 4$.

Of the test functions used, this is the one that generally presents the best results, both with the recommended point and the one chosen later. In fact, using point $\bar{x}$, it is observed that Fletcher and Reeves converges before it completes all iterations reaching the minimum in 0, for $n = 10^3$. The result obtained with $n = 10^4$ is not optimal because after a few iterations it can be observed that the method does not find a descent direction. Since inexact method instead completes all iterations, one might therefore think that it does not converge, but comparing the results with that of the previous method we conclude that it is a good approximation of the solution. Even using $\bar{x} = 10$ as the starting point, the same behavior is observed by all methods.

n	**Method**	k	Time	$f(x_k)$	$\|\nabla f(x_k)\|$
10^3	FR	8078	20.567657	7.7841e-14	8.9259e-08
10^3	INM	10000	631.718286	7.7194e-11	1.9267e-07
10^4	INM	10000	44503.9201	1.5033e-10	3.1764e-07

Table 2.5. Results with starting point $\bar{x}_i = 3$ if $i \equiv 1 \pmod 4$, $\bar{x}_i = -1$ if $i \equiv 2 \pmod 4$, $\bar{x}_i = 0$ if $i \equiv 3 \pmod 4$, and $\bar{x}_i = 1$ if $i \equiv 0 \pmod 4$

Chapter 3

Results

Different problems emerge from the discussion. The main observation can be made about the initial points, as they significantly influence the convergence of the different methods. It is evident how different results are obtained, in particular in the Inexact Newton Method according to the chosen point a positive definite Hessian matrix can not be obtained, which implies that a descent direction is not found and therefore the method does not converge. Dimensionality issues also arise. Applying the Fletcher and Reeves Method we use the exact gradient, as opposed to the inexact which uses finite differences, this leads in many calculations having to be performed at each iteration. A similar problem occurs with Nelder-Mead algorithm, which has to perform many transformations at each iteration.
In summary, while the Nelder-Mead algorithm can work for large-scale problems in some cases, it's important to consider its limitations and explore other optimization techniques that might be better suited to your specific problem's characteristics. In fact, in general, despite very high computational times, even if all the simulation are performed with Matlab, using the parallel pool, the best results were obtained with Inexact Newton method.

Appendix A

Matlab code

```
%% LOADING THE VARIABLES FOR THE TEST

load('generalized_brown.mat'); %different for each run
load('forcing_terms.mat');


% Problem dimension
% n= 2;
% n = 10^3;
n = 10^4;

% Starting point
% x0 = ones(n,1); % starting point banded trigonometric
% x0 = 1.2*ones(n,1); % starting point Rosenbrock 2d
% x0 = 10*ones(n,1); % different starting point

for i=1:n  % starting point generalized brown
    if mod(i,2)==1
        x0(i)=0;
    else
        x0(i)=-1;
    end
end

% for i=1:n  % starting point Rosenbrock
%     if mod(i,2)==1
%         x0(i)=-1.2;
%     else
%         x0(i)=1;
%     end
% end

% for i=1:n  % starting point chained powel
%     if mod(i,4)==1
```

```
%         x0(i)=3;
%     elseif mod(i,4)==2
%         x0(i)=-1;
%     elseif mod(i,4)==3
%         x0(i)=0;
%     else
%         x0(i)=1;
%     end
% end

alpha0 = 1;
rho = 0.8;
btmax = 50;
gamma = 0.9;
tolx = 1e-6;
tollgrad = 1e-7;
kmax = 10000;
c1= 1e-4;


% gradf = @(x) banded_trigonometric_grad(x0);
% gradf = @(x) chainedpowell_grad(x0);
% gradf = @(x) generalized_brown_grad(x0);



%% RUN FLETCHER AND REEVES

disp('**** FLETCHER AND REEVES: START ****')

tic
[xk_fr, fk_fr, gradfk_norm_fr, k_fr, xseq_fr, btseq_fr] = FR_CG_bcktrck(x0, f, ...
 gradf, alpha0, kmax, tollgrad, c1, rho, btmax);
toc

disp('**** FLETCHER AND REEVES: FINISHED ******' )
disp('**** FLETCHER AND REEVES: RESULTS *****' )
disp('************************************')
disp( ['f(xk_fr): ' , num2str(fk_fr)])   %' (global min. value: 0);'])
disp( ['grad(xk_fr): ' , num2str(norm(gradf(xk_fr),2))])
disp(['N. of Iterations: ', num2str(k_fr), '/', num2str(kmax), ';'])
disp('************************************')

%% RUN INEXACT NEWTON METHOD

Hessf = @(x) 0;
FDgrad = 1; % exact gradient
FDHess = 'Jc'; %hessian approximated through the jacobian
pcg_maxit = 50;
```

```
fterms = fterms_suplin;
h =sqrt(eps)*norm(x0);

disp('**** INEXACT NEWTON METHOD: START ****')

tic
[xk_n, fk_n, gradfk_norm_n, k_n, xseq_n] = innewton_general(x0, f, gradf, Hessf, ...
 kmax, tollgrad, c1, rho, btmax, FDgrad, FDHess, h, fterms, pcg_maxit);
toc

disp('**** INEXACT NEWTON METHOD: FINISHED ******' )
disp('**** INEXACT NEWTON METHOD: RESULTS *****' )
disp('****************************************')
disp( ['f(xk_n): ' , num2str(fk_n)])   %' (global min. value: 0);'])
disp( ['grad(xk_n): ' , num2str(norm(gradf(xk_n),2))])
disp(['N. of Iterations: ', num2str(k_n), '/', num2str(kmax), ';'])
disp('****************************************')

%% RUN THE NELDER MEAD


disp('**** NELDER MEAD: START ****')

tic
[xk_nm, fk_nm, k_nm] = nelder_mead(x0, f, tolx);
toc

disp('**** NELDER MEAD: FINISHED ******' )
disp('**** NELDER MEAD: RESULTS *****' )
disp('****************************************')
disp( ['f(xk_nm): ' , num2str(fk_nm)])   %' (global min. value: 0);'])
disp( ['grad(xk_nm): ' , num2str(norm(gradf(xk_nm),2))])
disp(['N. of Iterations: ', num2str(k_nm), '/', num2str(kmax), ';'])
disp('****************************************')
```

With the following algorithms:

INM

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    innewton_general(x0, f, gradf, Hessf, ...
    kmax, tolgrad, c1, rho, btmax, FDgrad, FDHess, h, fterms, pcg_maxit)


% 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 (not necessarily
% used);
% Hessf = function handle that describes the Hessian of f (not necessarily
% used);
% 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;
% FDgrad = 'fw' (FD Forward approx. for gradf), 'c' (FD Centered approx.
% for gradf), any other string (usage of input Hessf)
% FDHess = 'fw' (FD approx. for Hessf), 'Jfw' (Jacobian FD Forward
% approx. of Hessf), 'Jc' (Jacobian FD Centered approx. of Hessf), 'MF'
% (Matrix Free implementation for solving Hessf(xk)pk=-gradf(xk)), any
% other string (usage of input Hessf);
% h = approximation step for FD (if used);
% fterms = f. handle "@(gradfk, k) ..." that returns the forcing term
% eta_k at each iteration
% pcg_maxit = maximum number of iterations for the pcg solver.
%
% 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 where elements are the number of backtracking
% iterations at each optimization step.
%
switch FDgrad
    case 'fw'
        % Calculating the gradient with forward finite differences
        gradf = @(x) findiff_grad(f, x, h, 'fw');

    case 'c'
        %Calculating the gradient with central finite differences
        gradf = @(x) findiff_grad(f, x, h, 'c');

    otherwise
        % Use the exact gradient as in inpunt gradf


end
```

```
if isequal(FDgrad, 'fw') || isequal(FDgrad, 'c')
    switch FDHess
        case 'fw'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_Hess
            Hessf = @(x) findiff_Hess(f, x, sqrt(h));
        case 'MF'
            % DEFINE a f. handle for the product of Hessf * p USING THE
            % GRADIENT
            Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x)) / h;
        otherwise
            % Use the Hessf as in input

    end
else
    switch FDHess
        case 'fw'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_Hess
            Hessf = @(x) findiff_Hess(f, x, sqrt(h));
        case 'Jfw'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_J
            % (with option 'fw')
            Hessf = @(x) findiff_J(gradf, x, h, 'fw');

        case 'Jc'
            % OVERWRITE Hessf WITH A F. HANDLE THAT USES findiff_J
            % (with option 'c')
           Hessf = @(x) findiff_J(gradf, x, h, 'c');

        case 'MF'
            % DEFINE a f. handle for the product of Hessf * p USING THE
            % GRADIENT
            Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x)) / h;
        otherwise
            % Use the Hessf as in input

    end
end

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

n = length(x0);

% Initializations
xseq = zeros(n, kmax);
btseq = zeros(1, kmax);
```

```
xk = x0;
fk = f(xk);
k = 0;
gradfk = gradf(xk);
gradfk_norm = norm(gradfk);

while k < kmax && gradfk_norm >= tolgrad
    % "INEXACTLY" compute the descent direction as solution of
    % Hessf(xk) p = - gradf(xk)

    % TOLERANCE VARYING W.R.T. FORCING TERMS:
    eta_k = fterms(gradfk, k);
        % We will use directly eta_k as tolerance in the pcg because
    % this function looks at the RELATIVE RESIDUAL and not the RESIDUAL

    switch FDHess
        case 'MF'
            pk = pcg(Hessfk_pk, -gradfk, eta_k, pcg_maxit);


        otherwise
            % DIRECT METHOD
            % pk = -Hessf(xk)\gradf(xk)

            % ITERATIVE METHOD
            pk = pcg(Hessf(xk), -gradfk, eta_k, pcg_maxit);
    end


    % Reset the value of alpha
    alpha = 1;

    % 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, xk, 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
```

NM

```
function [xk, fk, k] = nelder_mead(x0, f, tol)


% [xk, fk] = nelder_mead(x0, f, tol)

% Function that performs the Nelder Mead method


% INPUTS:
% x0 = n-dimensional column vector;
% f = function handle that describes a function R^n->R;
% tol = tolerance for stopping criterion;

% OUTPUTS:
% xk = the last x computed by the function;
% fk = the value f(xk);

    %Definition of parameters
    rho = 1;        %rho > 0 (tipically 1);
    chi = 2;        %chi > 1 (tipically 2);
    gamma = 0.5;    %0 < gamma < 1 (tipically 0.5);
```

```
sigma = 0.5;     %0 < sigma < 1 (tipically 0.5);
k=0;

% Create initial simplex
n = length(x0);
X = zeros(n, n+1);
X(:,1) = x0;
for i = 2:n+1
    X(:,i) = x0;
    X(i-1,i) = x0(i-1) + 1;
end

err=1;

while err>tol
     k=k+1;
    % Sort simplex by function value (ordering phase)
    f_vect = zeros(1,n+1);
    for i = 1:n+1
        f_vect(i) = f(X(:,i));
    end
    [f_vect,idx] = sort(f_vect);
    X = X(:,idx);

    % Compute centroid
    x_bar = mean(X(:,1:n),2);

    % Reflection phase
    x_r = x_bar + rho*(x_bar - X(:,n+1));
    f_r = f(x_r);
    if f_vect(1) <= f_r && f_r < f_vect(n)
        X(:,n+1) = x_r;
        continue; %Returns to the beginning of the while (new simplex)
    end

    % Expansion phase
    if f_r < f_vect(1)
        x_e = x_bar + chi*(x_r - x_bar);
        f_e = f(x_e);
        if f_e < f_r
            X(:,n+1) = x_e;
        else
            X(:,n+1) = x_r;
        end
        continue;

    end
```

```
        % Contraction phase
        if f_r >= f_vect(n)
            x_c = x_bar - gamma*(x_bar - X(:,n+1));
            f_c = f(x_c);
            if f_c < f_vect(n+1)
                X(:,n+1) = x_c;
            else
                % Shrinking phase
                for i = 2:n+1
                    X(:,i) = X(:,1) + sigma*(X(:,i) - X(:,1));
                end
            end
        end

        % Stopping criterion
        err=norm(f_vect(1) - f_vect(n+1));

    end

    % Output
    xk = X(:,1);
    fk = f_vect(1);
end
```

FR

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

% 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;
% 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.

% 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);
k = 0;
gradfk = gradf(xk);
gradfk_norm = norm(gradfk);
pk = -gradf(xk);

while k < kmax && gradfk_norm >= tolgrad

    gradf_initial = gradf(xk);
    % 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);

    % Find b_k+1 and the descent direction
    beta = (gradfk_norm^2)/(norm(gradf_initial)^2);
    pk = -gradfk + beta*pk;

    % 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
```

And the functions with their gradients are the following:

ROSENBROCK

```
function f = rosenbrock(x)
    f = 0;
    n = length(x);
    for i=2:n
        f = f + 100*(x(i-1)^2-x(i))^2+(x(i-1)-1)^2;
    end
end

function grad = rosenbrock_grad(x)
  n = length(x);
  grad = zeros(n,1);

  for i=2:n
        grad(i-1) = grad(i-1)+ 400*x(i-1)*(x(i-1)^2-x(i))+2*(x(i-1)-1);
        grad(i) = -200*(x(i-1)^2-x(i));
  end
end
```

CHAINED POWEL

```
function f = chainedpowell(x)

    n = length(x);
    f = 0;
    for i = 1:(n-2)/2
        f = f + (x(2*i-1) +10*x(2*i))^2 + 5*(x(2*i+1)-x(2*i+2))^2 + (x(2*i)-2*x(2*i+1))^4 + ...
            10*(x(2*i-1)- x(2*i + 2))^4;
    end
end


function grad = chainedpowell_grad(x)
    n = length(x);
    grad = zeros(n,1);
    for i = 1:(n-2)/2
        grad(2*i-1) = grad(2*i-1) + 2*(x(2*i-1) + 10*x(2*i)) + 40*(x(2*i-1) - x(2*i + 2))^3;
        grad(2*i) = grad(2*i) + 20*(x(2*i-1) + 10*x(2*i)) + 4*(x(2*i) - 2*x(2*i+1))^3;
```

```
        grad(2*i+1) = grad(2*i+1) + 10*(x(2*i+1) - x(2*i+2)) -8*(x(2*i) - 2*x(2*i+1))^3;
        grad(2*i+2) = grad(2*i+2) -10*(x(2*i+1) - x(2*i+2)) - 40*(x(2*i-1) - x(2*i + 2))^3;
    end
end
```

BANDED TRIGONOMETRIC

```
function f = banded_trigonometric(x)
    n = length(x);
    f = 0;
    for i = 1:n
        if i-1 == 0
            f = i*(1-cos(x(i)) - sin(x(i+1)));
        elseif i+1 == n+1
            f = f + i*(1-cos(x(i)) + sin(x(i-1)));
        else
             f = f + i*(1-cos(x(i)) + sin(x(i-1)) - sin(x(i+1)));
        end
    end
end

function grad = banded_trigonometric_grad(x)
    n = length(x);
    grad = zeros(n,1);
    for i = 1:n
        if i-1 == 0
            grad(i) = grad(i) + i*sin((x(i)));
            grad(i+1) = -i*cos((x(i+1)));

        elseif i+1 == n+1
            grad(i-1) = grad(i-1) + i*cos((x(i-1)));
            grad(i) = grad(i) + i*sin((x(i)));
        else
            grad(i-1) = grad(i-1) + i*cos((x(i-1)));
            grad(i) = grad(i) + i*sin((x(i)));
            grad(i+1) = -i*cos((x(i+1)));
        end

    end
end
```

GENERALIZED BROWN

```
function f = generalized_brown(x)
    n = length(x);
    f = 0;
    k=0;
    for i = 1:n/2
        f = f + ((x(2*i-1)-3)^2)/1000 -(x(2*i-1)-x(2*i))+ exp(20*(x(2*i-1)-x(2*i)));
```

```
    end
    for j = 1:n/2
            k= k + x(2*j-1)-3;
    end
    f= f+ k^2;
end

function grad = generalized_brown_grad(x)
    n = length(x);
    grad = zeros(n,1);
    q = 0;
    for i = 1:n/2
        q = q+(x(2*i-1)-3);
        grad(2*i-1) = (x(2*i -1)-3)/500 -1 + 20*exp(20*(x(2*i -1)-x(2*i)))+2*q;
        grad(2*i) = 1-20*exp(20*(x(2*i -1)-x(2*i)));
    end
end
```