POLITECNICO
DI TORINO

Comparison of the performances of Newton method and Inexact Newton method on a large-scale problem

---

Course: Numerical optimization for large scale problems and Stochastic Optimization

Professors: Sandra Pieraccini, Enrico Bibbona

Students: Andrea Rubeis, Nicola Scarano

Academic year 2020-2021

Index

Abstract

In this report, we present how the Inexact Newton method with line-search works on a large-scale problem by varying the forcing terms in order to solve the problem with different rate of convergence. In addiction we compare the obtained results with the pure Newton method.

Theoretical background

In the **Newton method** the objective is to minimize a function $f(x)$ with $x \in \mathbb{R}^n$, $f: \mathbb{R}^n \to \mathbb{R}$. We start from an arbitrary starting point $x_k \in \mathbb{R}^n$ where $k = 0$, then we approximate the function f with the second order Taylor expansion of f around x_k

$$f(x_k + p) = f(x_k) + p^T \nabla f(x_k) + \frac{1}{2} p^T \nabla^2 f(x_k) p := m_k(p)$$

If we want to find the minimum of this local model the hessian $\nabla^2 f(x_k)$ has to be positive definite because in this way the model is convex so it has a unique minimum which is a stationary point as well.

$$\nabla m_k(p) = 0 + \nabla f(x_k) + \nabla^2 f(x_k) p_k^N = 0$$
$$\nabla^2 f(x_k) p_k^N = -\nabla f(x_k)$$
$$p_k^N = -\left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k)$$

Now we know x_k and p_k, the only unknown is α so we can write $f(x_k + \alpha p_k) = \phi(\alpha)$. Since the direction is a descent direction, α must be positive and in particular in the Newton method has to be equal to 1 because in this way we allow the method whenever possible to take the full step and has a quadratic convergence.

To provide a sufficient decrease we use the Armijo condition: we want to ask a decrease which is at least some fixed fraction of the initial rate of decrease in the direction, $\phi'(0) = {p_k}^T \nabla f(x_k)$, the derivative of $\phi(\alpha)$ with $\alpha = 0$.

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^T p_k$$

We take a small c_1 so we can easily satisfy the condition. To avoid taking steps too short, we use the *backtrack strategy*, so we start from $\alpha = 1$ and if we satisfy the condition we proceed, otherwise we reduce α by a small quantity ρ between 0 and 1 $\alpha_{j+1} = \rho \alpha_j$ until we satisfy the Armijo condition.

Since to solve exactly the linear system $\nabla^2 f(x_k) p_k^N = -\nabla f(x_k)$ may be too costly, we can use the **inexact Newton method** to reduce its cost by solving the linear system with an iterative method with a bigger tolerance. The tolerance ε become smaller and smaller as we converge to the minimum.

$$\|\nabla^2 f(x_k) p_k^{IN} + \nabla f(x_k)\| \leq \varepsilon$$

The norm of the gradient can tell us how far we are from the solution becoming smaller going towards the minimum and for this reason ε is defined as $\varepsilon = \eta_k \|(\nabla f(x_k)\|$, $\eta_k \in [0, 1)$. η_k is called *forcing term* and actually the rate of convergence towards the minimum depends on this term:

$\eta_k \leq \hat{\eta} < 1$: linear convergence

$\eta_k \xrightarrow[k \to \infty]{} 0$: super linear convergence

$\eta_k = O(\|\nabla f(x_k)\|)$: quadratic convergence

Problem and methods

Problem specification

We aim to compare the performances of Newton method and Inexact Newton method in finding the local minimum of the function below:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \sum_{i=1}^{n} \left(\frac{1}{4} x_i^4 + \frac{1}{2} x_i^2 + x_i \right)$$

We analyse the function for n equal to 10^4 and 10^5 by implementing two versions of the Newton method - pure (PN), conjugate gradient method implemented (PCGN) - and three versions of the inexact one which differs only in the rate of convergence: linear (INM-L), super-linear (INM-SL), quadratic (INM-Q).

We compare our results in terms of time spent to reach the solution and the number of inner and outer iterations:

- Number of inner iterations (i.e., number of cycles of the iterative method used to find the descent direction, measured at each step k).
- Outer iterations (i.e., number of steps computed by the method before converging).
- Execution time.

Methods implementation

We start from an initial point x_0 of n random elements between 0 and 1 and we use the command `rng(1)` to keep always the same starting random vector for every execution of our code. Then we declare our function as a function handle, we compute its gradient and its hessian:

```
f = @(x)sum(x + (x.^4)/4 + (x.^2)/2),

gradf = @(x) (x.^3 + x + 1)

Hessf = @(x) sparse(idx,idx,1+3*(x.^2)).
```

We get this hessian formula by the fact that, for $i = 1$ we get a non-zero result just for x_1 and 0 for all the other components, for $i = 2$ the hessian is 0 respect to x_1, non-zero for x_2 and 0 for all the other components and so on. So, the hessian is a diagonal matrix, which has the only values strictly greater than 0 on the main diagonal. This is important because it is a symmetric positive definite matrix as well and all further algorithms can be executed on it. Since n is quite large, we apply the command *sparse* to store it in a more efficient way.

All our Newton method's versions implement the backtrack strategy and share the following inputs:

`alpha = 1;`	-in order to guarantee the quadratic convergence
`c1 = 1e-4;`	-we set c_1 equal to small value to easily satisfy the A. condition
`rho = 0.8;`	- fixed factor, lower than 1, used for reducing α
`btmax = 50;`	- max number of steps for updating α
`tollgrad = 10e-13;`	- max error's threshold accepted
`kmax = 100;`	- max number of steps allowed to find the next x_k
`fterms`	- only for inexact versions.

Once our inputs are ready, we can execute our algorithms, in every function related to a Newton version we declare the Armijo condition as:

```
farmijo = @(fk,alpha,xk,pk) fk+c1*alpha*gradf(xk)'*pk
```

Then, we initialize `xseq`, `btseq`, which are the vectors where we save our k steps and the number of iterations needed to find the correct α to satisfy the Armijo condition.

After, we write a while loop where it is computed the descent direction p_k by using the command "\" in MATLAB in case of direct pure Newton method and the `pcg()` command for all the other versions which receive as inputs the hessian's function handle, the gradient, the tolerance and a max number of steps allowed to converge. For the Newton version implemented with `pcg()`, the `tollgrad` is

fixed for every iteration, while in the inexact versions the tolerance is first computed at every iteration by using:

```
epsilon_k = fterms(gradf, xk, k) * norm(gradf(xk))
```

Where the forcing terms used are:

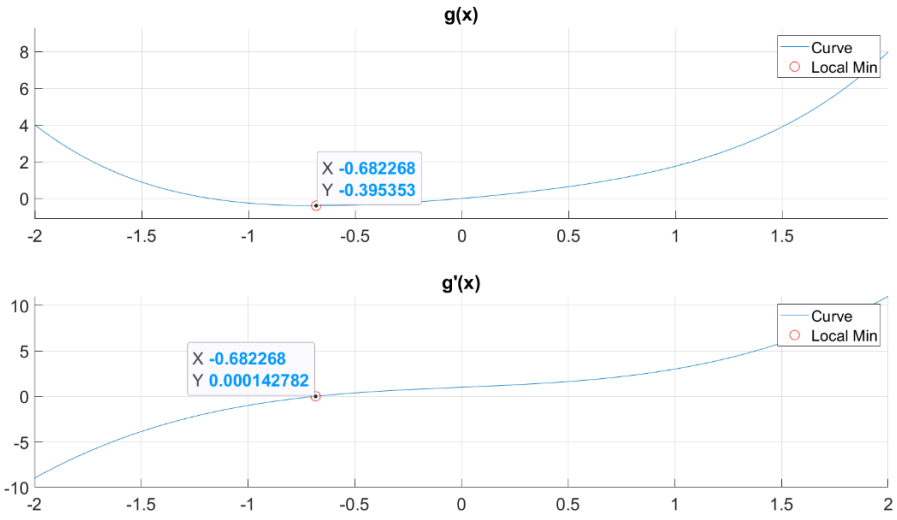
- $\eta_k = 0.5$: *linear convergence*
- $\eta_k = \min\left(0.5, \sqrt{\|\nabla f(x_k)\|}\right) \xrightarrow[k\to\infty]{} 0$: *super linear convergence*
- $\eta_k = min\,(0.5, \|\nabla f(x_k)\|) \in O(\|\nabla f(x_k)\|)$: *quadratic convergence*

Following this, we compute `xnew = xk + alpha * pk`, we evaluate `f(xnew)` and check if it satisfies the Armijo condition, if not we write a second while loop where we iterate until we find the value of α such that `f(xnew)` can satisfy it and we save in the variable `bt` how many iterations we have computed. Once the Armijo condition is satisfied, we update `xk`, `fk` and `gradfk_norm`, we increment the number of iterations `k`, add the current `xk` to vector `xseq` and the variable `bt` to the vector `btseq`. If the error between the previous `fk` computed and the current one is lower than `tollgrad` or if the total number of iterations is greater than `kmax` we exit from the while loop and we cut the vectors `xseq`, `btseq` and `normgrad_seq` to the correct size.

Results

Expected outcome

If we write $f(x)$ in this way: $f(x) = \sum_{i=1}^{n} g(x_i)$, where $g(x) = \frac{1}{4}x^4 + \frac{1}{2}x^2 + x$, so if we find the minimum of g, the minum of f will be n times g's minimum. Moreover, the minimum point will be the same for each component since each component is described by the same function. Indeed, if we compute the derivative of g and put it equal to 0 we get that the point of minimum of g is $x_s = -0.6823$. This can be proven by a graph where we plot g, g':



So, we expect that our solution represented by the vector x_k, able to minimize our function f, will have all the components equal to x_s.

In terms of time execution, we expect better performances from Newton method with direct solver (PN) rather than the iterative one (PCGN), due to the backslash command which solves any given system with the most performing method. On the other hand, with respect to the number of outer iterations, we do not expect different behaviour among PN and PCGN because both should solve the descent direction approximately with the same precision, indeed the PN asks for a zero residual while the PCGN has a tolerance of 10^{-13} which is a very low tolerance.

Regarding the dimension n, we presume to see a higher number of iterations when $n = 10^5$ since the linear system to solve is larger than the linear system computed with $n = 10^4$.

Among all the methods implemented with pcg to compute the descent direction, we expect to see a higher number of inner iterations in the PCGN, since it has a smaller tolerance than any other version INM-X and a lower number of outer iterations.

More precisely, if we rank them according to the number of inner iterations we expect:

1. Newton method with pcg. (highest number of inner iterations)
2. Inexact Newton method with quadratic convergence.
3. Inexact Newton method with super-linear convergence.
4. Inexact Newton method with linear convergence. (lowest number of inner iterations)

Instead, regarding the outer iterations we expect the same ranking but with the inverse order:

1. Inexact Newton method with linear convergence. (highest number of outer iterations)
2. Inexact Newton method with super-linear convergence.
3. Inexact Newton method with quadratic convergence.
4. Newton method with pcg. (lowest number of outer iterations)

Obviously, these rankings are made only considering the tolerance used to solve the linear system of the descent direction.

Eventually looking at these rankings we cannot surely predict the results according to the time of execution, but we expect an improvement with the Inexact methods which avoid to over solving the linear system.

Obtained results

We report here the results obtained by running the methods with a fixed starting point (x_0), but we ensure that the following discussion is also valid for different x_0.

This is our choice for the starting point:

```
rng(1);
x0 = rand(n,1);
```

In terms of number of outer iterations, all the methods have demonstrated high scalability with respect to the dimension of the linear system performing seven cycles both for $n = 10^4$ and 10^5. The number of outer iterations is so small because the starting point x_0 is close to the solution:

```
x* = [-0.6823;-0.6823;-0.6823;…]
```

Once we run our algorithms we get `xk_n`, `xk_pn`, `xk_lin`, `xk_slin`, `xk_q` and if we evaluate the function f over them we get `-3.9535e+03` for $n = 10^4$ and `-3.9535e+04` for $n = 10^5$. This proves our previous expectation; indeed, we get these results because we sum n times the minimum of g (-0.3953), which is also shown in the previous graph.
So, all the methods can converge to the solution.

We now show two graphs: the first one shows the inner iterations counter for each method in each iteration k, while the second one shows the methods' execution time.

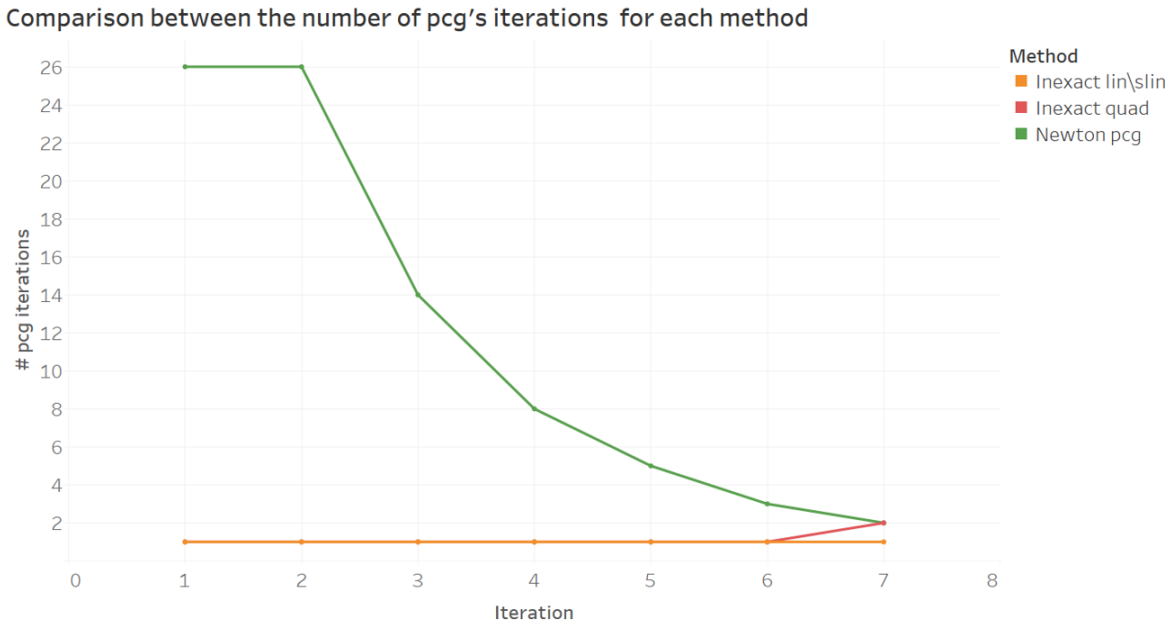


Figure 1: number of inner iterations performed by each method for $n = 10^4$ and $n = 10^5$

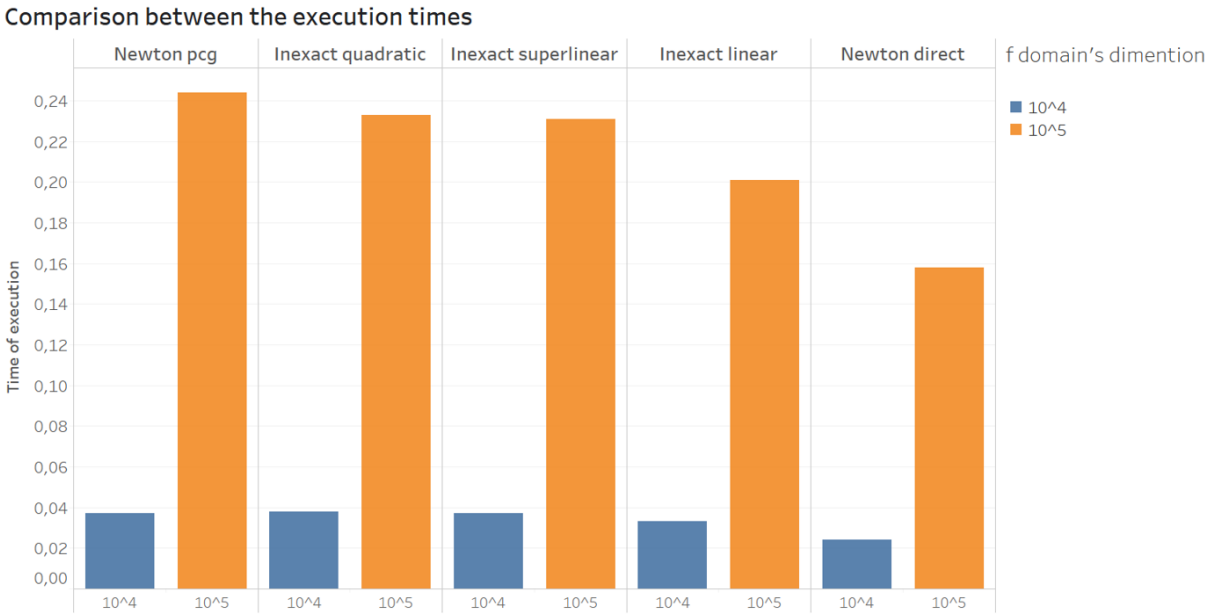


Figure 2: time spent from our methods to converge to the solution

Regarding the first figure, we can see how our previous expectation related to the number of inner iterations for every method is partly satisfied. Indeed, the PCGN version at the beginning has the highest number of inner iterations then they decrease at every iteration. INM-L and INM-SL converge to the solution with just a single iteration at every outer iteration k. INM-Q behaves as the INM-L and INM-LS except for the last iteration where makes 2 inner iterations instead of 1.

Looking at the second graph, we can see that PN is faster than PCGN even if the number of outer iterations computed is the same for both of them, this is probably due to the fact that PN is able to compute the descent direction more efficiently and as a consequence has a higher gain in time cost.

Among the three versions of the Inexact Newton method, the number of internal and external iterations is almost the same. So, the factor that most of all influences the time performance is the calculation of the forcing terms. Indeed, INM-L is the fastest among the other inexact versions since it does not have to computes this value which is fixed.

The time spent by PCGN, despite does not has to compute the tolerance because is fixed it, is mainly affected by the number of pcg iterations. Indeed, they are greater than all the other inner iterations in the inexact methods INM-X, and this is why PCGN is the slowest one.

Code

Main:

```
%% MAIN

clear; clc; close all;

alpha0 = 1;
c1 = 1e-4;
rho = 0.8;
btmax = 50;
tollgrad = 10e-13;
kmax = 100;
n = 1e4;              %10^4

%-------DEFINITION OF THE FUNCTIONAL TO BE ANALYZED,
%-------ITS HESSIAN, ITS GRADIENT;
%-------AND X0: STARTING POINT OF OUR NEWTON METHOD

f = @(x)sum(x + (x.^4)/4 + (x.^2)/2);
gradf = @(x) (x.^3 + x + 1);
idx = [1:1:n];
Hessf = @(x) sparse(idx,idx,1+3*(x.^2));

rng(1)
x0 = rand(n,1);

%% PURE NEWTON

tic
disp('**** PURE_NEWTON: START *****')

[xk_n, fk_n, gradfk_norm_n, k_n, xseq_n, btseq] = ...
    pureNewton(x0, f, gradf, Hessf, alpha0, kmax, ...
    tollgrad, c1, rho, btmax);

toc
disp('**** PURE_NEWTON: FINISHED *****')
disp('**** PURE_NEWTON: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk_n)])
disp(['f(xk): ', num2str(fk_n)])
disp(['N. of Iterations: ', num2str(k_n),'/',num2str(kmax),
';'])
disp('************************************')


%% NEWTON METHOD WITH PCG

tic
```

```
disp('**** NEWTON_PCG: START *****')
[xk_pn, fk_pn, gradfk_norm_pn, k_pn, xseq_pn, btseq_pn,
norm_grad_seq_pn] = ...
    newton_pcg(x0, f, gradf, Hessf, alpha0, kmax, ...
    tollgrad, c1, rho, btmax);
toc
disp('**** NEWTON_PCG: FINISHED *****')
disp('**** NEWTON_PCG: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk_pn)])
disp(['f(xk): ', num2str(fk_pn)])
disp(['N. of Iterations: ', num2str(k_n),'/',num2str(kmax),
';'])
disp('************************************')

% MAX PCG ITERATIONS
max_pcgiters = 50;

%% RUN THE INEXACT NEWTON (LINEAR) ON f

% OPTIONS FOR FORCING TERMS
fterms = @(gradf, x, k) 0.5;
disp('**** IN_NEWTON_LINEAR: F.T. OPTIONS *****')
disp(['Forcing Terms: ', char(fterms)])

disp('**** IN_NEWTON_LINEAR: START *****')
tic
[xk_lin, fk_lin, gradfk_norm_lin, k_lin, xseq_lin,
btseq_lin,norm_grad_seq_lin] = ...
innewton(x0, f, gradf, Hessf, alpha0, kmax, ...
    tollgrad, c1, rho, btmax, ...
    fterms, max_pcgiters);
toc
disp('**** IN_NEWTON_LINEAR: FINISHED *****')
disp('**** IN_NEWTON_LINEAR: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk_lin), ';'])
disp(['f(xk): ', num2str(fk_lin), ';'])
disp(['N. of Iterations: ', num2str(k_lin),'/',num2str(kmax),
';'])

disp('************************************')

% RUN THE INEXACT NEWTON (SUPERLINEAR) ON f

% OPTIONS FOR FORCING TERMS
fterms = @(gradf, x, k) min(0.5, sqrt(norm(gradf(x))));

disp('**** IN_NEWTON_SUPERLINEAR: F.T. OPTIONS *****')
disp(['Forcing Terms: ', char(fterms)])
```

```
disp('**** IN_NEWTON_SUPERLINEAR: START *****')
tic
[xk_slin, fk_slin, gradfk_norm_slin, k_slin, xseq_slin,
btseq_slin, norm_grad_seq_slin] = ...
    innewton(x0, f, gradf, Hessf, alpha0, kmax, ...
    tollgrad, c1, rho, btmax, ...
    fterms, max_pcgiters);
toc
disp('**** IN.NEWTON_SUPERLINEAR: FINISHED *****')
disp('**** IN.NEWTON_SUPERLINEAR: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk_slin), ';'])
disp(['f(xk): ', num2str(fk_slin), ';'])
disp(['N. of Iterations: ',
num2str(k_slin),'/',num2str(kmax), ';'])
disp('************************************')


%% RUN THE INEXACT NEWTON (QUADRATIC) ON f

%OPTIONS FOR FORCING TERMS
fterms = @(gradf, x, k) min(0.5, norm(gradf(x)));
disp('**** IN.NEWTON_QUADRATIC: F.T. OPTIONS *****')
disp(['Forcing Terms: ', char(fterms)])

disp('**** IN.NEWTON_QUADRATIC: START *****')
tic
[xk_q, fk_q, gradfk_norm_q, k_q, xseq_q, btseq_q,
norm_grad_seq_q] = ...
    innewton(x0, f, gradf, Hessf, alpha0, kmax, ...
    tollgrad, c1, rho, btmax, ...
    fterms, max_pcgiters);
toc
disp('**** IN.NEWTON_QUADRATIC: FINISHED *****')
disp('**** IN.NEWTON_QUADRATIC: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk_q), ';'])
disp(['f(xk): ', num2str(fk_q), ';'])
disp(['N. of Iterations: ', num2str(k_q),'/',num2str(kmax),
';'])
disp('************************************')

x = linspace(-2,2,n);
g = (x + (x.^4)/4 + (x.^2)/2);
idx = islocalmin(g);

figure
subplot(2,1,1)
hold on
plot(x,g)
```

```
plot(x(idx),g(idx),'ro', 'MarkerSize',10)
legend('Curve','Local Min')
hold off
grid on
title("g(x)")
set(gca,'FontSize',18);
fprintf('Min located at %0.2f\n',x(idx))


subplot(2,1,2)
hold on
x = linspace(-2,2,n);
dg = x.^3 + x + 1;
idx = islocalmin(g);
plot(x, dg)
plot(x(idx),dg(idx),'ro', 'MarkerSize',10)
legend('Curve','Local Min')
hold off
grid on
title("g'(x)")

set(gca,'FontSize',18);
fprintf('Min located at %0.2f\n',x(idx))
```

Newton methods:

Pure Newton:

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    pureNewton(x0, f, gradf, Hessf, alpha0, ...
    kmax, tollgrad, c1, rho, btmax)
%
% Function that performs the newton optimization method,
% implementing the backtracking strategy.
%
% 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;
% Hessf = function handle that describes the Hessian 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.
%
% 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.
%


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

% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);

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

while k < kmax && gradfk_norm >= tollgrad
    % Compute the descent direction as solution of
    % Hessf(xk) p = - graf(xk)
    pk = -Hessf(xk)\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, 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_norm = norm(gradf(xk));

    % 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
```

PCG Newton:

```
function [xk, fk, gradfk_norm, k, xseq, btseq, norm_grad_seq]
= ...
    newton_pcg(x0, f, gradf, Hessf, alpha0, ...
    kmax, tollgrad, c1, rho, btmax)
%
% Function that performs the newton optimization method based
on pcg,
% implementing the backtracking strategy
%
% 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);
```

```
% 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;
%
% 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.
% norm_grad_seq = 1-by-k vector where elements are the value
of the norm of
% the gradient at each iteration

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

% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);
norm_grad_seq = zeros(1, kmax);

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

while k < kmax && gradfk_norm >= tollgrad
    % Compute the descent direction as solution of
    % Hessf(xk) p = - graf(xk)
    pk = pcg(Hessf(xk), -gradf(xk), tollgrad, 50);
```

```
    % 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, 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_norm = norm(gradf(xk));

    % Increase the step by one
    k = k + 1;
    %store value of current norm of the gradient
    norm_grad_seq(k) = gradfk_norm;
    % 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
```

Inexact Newton Method:

```
function [xk, fk, gradfk_norm, k, xseq, btseq, norm_grad_seq]
= ...
    innewton(x0, f, gradf, Hessf, alpha0, ...
    kmax, tollgrad, c1, rho, btmax, ...
    fterms, max_pcgiters)
%
% Function that performs the inexact newton optimization
method,
% implementing the backtracking strategy

% 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);
% 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;
% fterms = function handle that characterize the sequence of
forcing terms
% eta_k for the inexact newton method. This function handle
must be defined
% with input variables @(gradf, x, k). In this way, it is
possible
% to use both constant forcing terms and forcing terms that
depend from
% gradf(xk) value (for superlinear and quadratic
convergence).
% max_pcgiters = maximum number of iterations for the pcg
solver used to
% compute the inexact newton direction pk.
%
% 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.
%

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

% Initializations
xseq = zeros(length(x0), kmax);
btseq = zeros(1, kmax);
norm_grad_seq = zeros(1,kmax);

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

while k < kmax && gradfk_norm >= tollgrad
    % Compute the descent direction as solution of
    % Hessf(xk) * p = - graf(xk)
    % TOLERANCE VARYING W.R.T. FORCING TERMS:
    epsilon_k = fterms(gradf, xk, k) * norm(gradf(xk));

    % ITERATIVE METHOD
    pk = pcg(Hessf(xk), -gradf(xk), epsilon_k, max_pcgiters);

    % 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, 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_norm = norm(gradf(xk));

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

    %store value of current norm of the gradient
    norm_grad_seq(k) = gradfk_norm;
    % 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
```