Numerical optimization for large scale problems and stochastic optimization
Unconstrained optimization

Elisa Salvadori (302630), Sonia Vittone (302673)

Contents

1 Methods

The general problem is written in the form:

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

where $f : \mathbb{R}^n \to \mathbb{R}$.
We have chosen the steepest descent method and the Newton method for minimizing the function $f(x)$.

1.1 Steepest descent method

The steepest descent method is an iterative method that, starting from an initial point $x_0 \in \mathbb{R}^n$, computes a sequence $\{x_k\}_{k \in \mathbb{N}}$ characterized by:

$$x_{k+1} = x_k + \alpha_k p_k, \quad \forall k \in \mathbb{N},$$

where the descent direction p_k is the steepest one i.e. $-\nabla f(x_k)$ and α_k is the steplength found with the backtracking strategy.

1.2 Newton method

The Newton method is based on a local approximation of the function with a quadratic model (Taylor expansion):

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

and

$$f(x_k + p) \simeq m_k(p).$$

If $\nabla^2 f(x_k)$ is positive definite, otherwise it can be transformed by adding a correction matrix, then $m_k(p)$ is a convex model for f around x_k. The Newton method at each step minimizes $m_k(p)$ to find the descent direction, in particular the stationary point is computed as:

$$\nabla m_k(p) = \nabla f(x_k) + \nabla^2 f(x_k) p = 0,$$

so the descent direction p_k is the solution of the linear system:

$$\nabla^2 f(x_k) p = -\nabla f(x_k),$$

i.e. $p = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$.
Starting from an initial point $x_0 \in \mathbb{R}^n$, the Newton method computes a sequence $\{x_k\}_{k \in \mathbb{N}}$ characterized by:

$$x_{k+1} = x_k + \alpha_k p_k, \quad \forall k \in \mathbb{N},$$

where the descent direction p_k is the solution of the linear system above and α_k is the steplength found with the backtracking strategy.

1.3 Backtracking strategy

The backtracking strategy is implemented to find at each step the steplegth α_k which satisfyes the Armijo condition, i.e.

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

At each step k, starting from an iniatial steplegth $\alpha_k^{(0)}$, the steplength is updated as $\alpha_k^{(j+1)} = \rho \alpha_k^{(j)}$, with $\rho < 1$, until it satisfyes the Armijo condition.

2 Test functions

In this section the two methods are compared with four testing functions. Where it's not specified the choice of parameters is $\alpha_0 = 1$, $\rho = 0.5$, $c_1 = 10^{-4}$, $tol = 10^{-12}$, $kmax = 10000$, $btmax = 50$.

2.1 Rosenbrock function

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

the argmin and the minimum of this function are respectively $x^* = (1, 1)$, $f(x^*) = 0$. See Figure 2.
This problem is tested with the initial points $x_0 = (-1.2, 1)$ and $x_0' = (1.2, 1.2)$.
In the following table are shown the computational result of the test for the Rosenbrock function, in particular it can be seen the number of iterations of the method k, the time in seconds $Time$ (using tic toc in MATLAB), the value of the function at the last iterate $f(x)$ and the initial point x_0.

Rosenbrock							
Steepest descent				Newton			
x_0	k	$f(x)$	$Time$	x_0	k	$f(x)$	$Time$
(-1.2,1)	10000	2.7098e-10	0.047359	(-1.2,1)	22	3.7286e-29	0.003864
(1.2,1.2)	10000	8.1803e-11	0.045233	(1.2,1.2)	9	2.5559e-28	0.003818

Table 1: Comparison between the steepest descent method and Newton method for the Rosenbrock function.

We know from the theory that the Newton method has quadratic local rate of convergence.
We can see that, with the same choice of parameters and initial point, the Newton method is faster and more precise than the steepest descent, indeed the precision of the solution for the first method is in the order of 10^{-10} while the one of the second method is 10^{-28}.
We have also computed the counter plot of the function with the initial point equal to x_0 and the sequence $\{x_k\}_k$ as the next graphics shows.

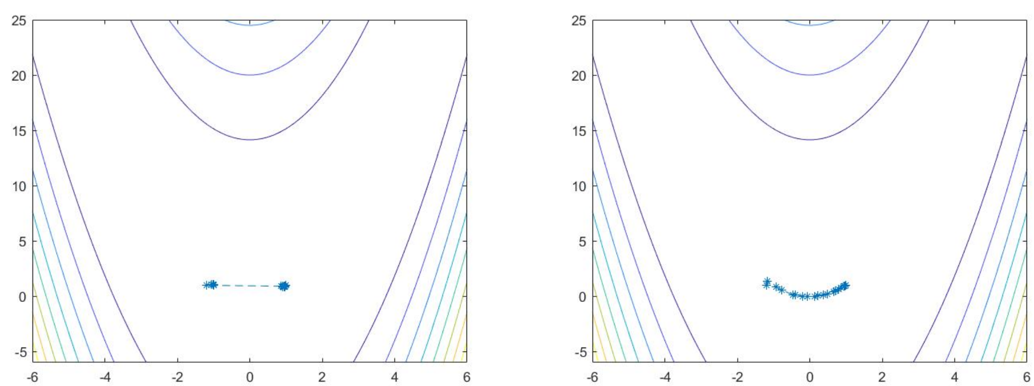

Figure 1: Counter plot of Rosenbrock function for the steepest descent method with $x_0 = (-1.2, 1)$ and counter plot of the Rosenbrock function for the Newton method with $x_0 = (-1.2, 1)$.

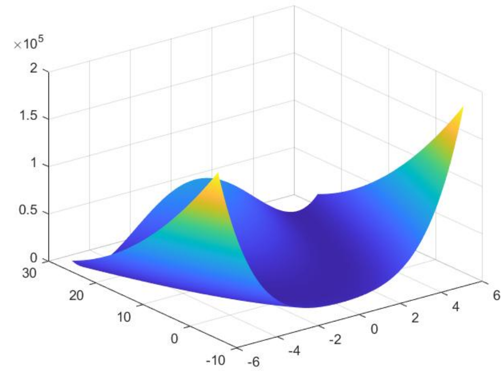

Figure 2: Plot of the Rosenbrock function.

2.2 Chained Rosenbrock function

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

like the previous function the argmin and the minimum are respectively $x^* = (1, ..., 1)$, $f(x^*) = 0$. This problem is tested with the initial points $x_0 = (-1.2, 1, ..., -1.2, 1)$ and $x_0' = (1.2, ..., 1.2)$.
In the following table are shown the computational result of the test for the Chained Rosenbrock function, in particular it can be seen the number of iterations of the method k, the time in seconds $Time$ (using tic toc in MATLAB), the value of the function at the last iterate $f(x)$, the initial point x_0 and the dimension n.

Chained Rosenbrock									
Steepest descent					Newton				
n	x_0	k	$f(x)$	$Time$	n	x_0	k	$f(x)$	$Time$
4	x_0'	10000	2.9763e-08	0.483628	4	x_0'	9	1.4249e-29	0.069364
10	x_0'	10000	1.2054e-07	0.611025	10	x_0'	8	1.3806e-28	0.065981
50	x_0'	10000	1.174e-07	1.826207	50	x_0'	8	3.1204e-28	0.435096
100	x_0'	10000	1.1816e-07	3.419164	100	x_0'	8	8.3619e-29	3.312603
4	x_0	10000	2.479e-08	0.627656	4	x_0	10000	3.7081	2.088726
10	x_0	10000	8.6887e-07	0.585803	10	x_0	10000	9.6058	2.503903
50	x_0	10000	0.001246	1.627978	50	x_0	10000	49.1568	7.967060
100	x_0	10000	23.6025	3.125786	100	x_0	10000	98.6516	26.935982

Table 2: Comparison between the steepest descent method and Newton method for the Chained Rosenbrock function.

We can see that considering x_0', the initial point, the steepest descent method reaches the maximum number of iterations and as the dimension increases also the time increases. Instead the Newton method converges with just few iterations and the time at different n is quite the same. While if we consider x_0, both methods do not converge to the minimum. This point is not "good" because it is too distant from the solution. We have also computed the relative error of the solution as the graphics below show, in this case it can be seen the relative error for $n = 100$ only for the second initial point.

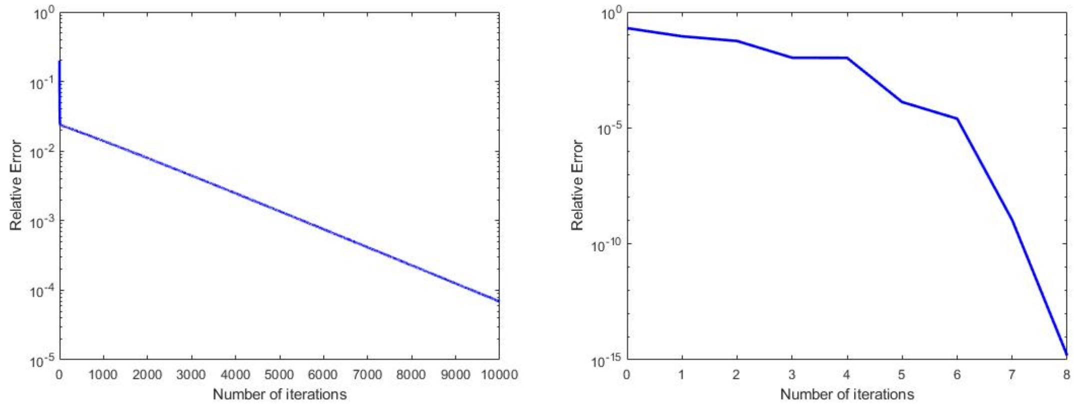

Figure 3: Relative error for the steepest descent method with initial point x_0' and $n = 100$ and relative error for the Newton method with initial point x_0' and $n = 100$.

For the steepest descent method, the error is linear and for the Newton method, the error appears to be quadratic.

2.3 Chained Wood function

$$f(x) = \sum_{j=1}^{k}[100(x_{i-1}^2 - x_i)^2 + (x_{i-1}-1)^2 + 90(x_{i+1}^2 - x_{i+2})^2 + (x_{i+1}-1)^2 + 10(x_i + x_{i+2} - 2)^2 + (x_i - x_{i+2}^2)/10],$$

with $i = 2j$ and $k = \frac{n-2}{2}$.
The argmin and the minimum are respectively $x^* = (1, ..., 1)$, $f(x^*) = 0$. This problem is tested with the initial points $x_0 = (-3, -1, -3, -1, -2, 0, ..., -2, 0)$ and $x_0' = (1.5, ..., 1.5)$.
In the following table are shown the computational result of the test for the Chained Wood function, in particular it can be seen the number of iterations of the method k, the time in seconds $Time$ (using tic toc in MATLAB), the value of the function at the last iterate $f(x)$, the initial point x_0 and the dimension n.

Chained Wood									
Steepest descent					Newton				
n	x_0	k	$f(x)$	$Time$	n	x_0	k	$f(x)$	$Time$
4	x_0'	10000	1.3146e-17	0.568880	4	x_0'	8	4.1195e-29	0.036118
10	x_0'	10000	8.6581e-13	0.670495	10	x_0'	7	8.6725e-29	0.037303
50	x_0'	10000	1.8772e-15	1.908060	50	x_0'	7	1.0539e-27	0.293487
100	x_0'	10000	5.7018e-16	4.282330	100	x_0'	7	2.6025e-27	2.978958
4	x_0	10000	1.0333e-14	0.427432	4	x_0	10000	7.8765	1.951431
10	x_0	10000	6.8207e-07	0.547268	10	x_0	37	8.2452e-29	0.032606
50	x_0	10000	19.7608	1.208807	50	x_0	10000	189.0729	9.471323
100	x_0	10000	330.2601	2.243474	100	x_0	10000	386.0023	35.558603

Table 3: Comparison between the steepest descent method and Newton method for the Chained Wood function.

Again, one point is a good point while the other is not. This is because both methods are quite sensitive to the initial point. So if we take a bad initial point i.e. a point too far from the solution, the algorithms don't reach the minima.
On the other hand, if we start with a good point, both the first method and the second behave as we expect. Indeed, the Newton method converges faster than the steepest descent method and in

addition, the Newton method needs only few iterations and the other method reaches the maximum number of iterations. We have also computed the relative error of the solution as the graphics below show, in this case it can be seen the relative error for $n = 100$ only for the second initial point.

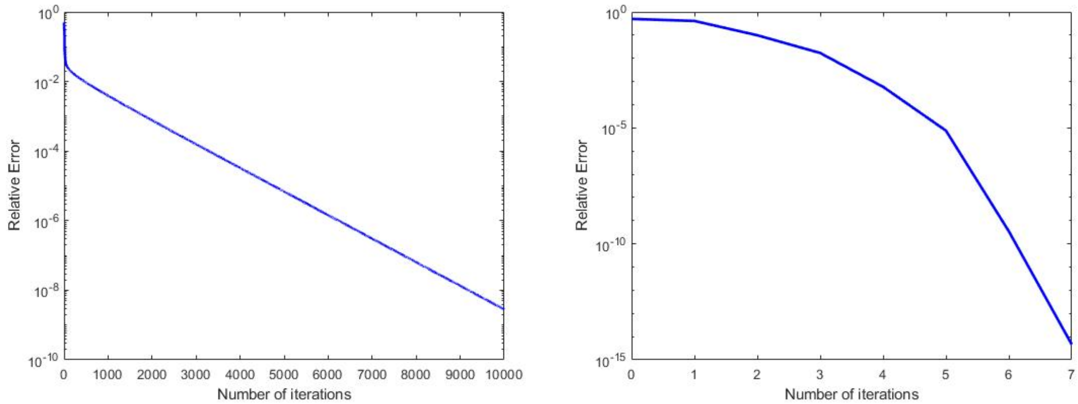

Figure 4: Relative error for the steepest descent method with initial point x_0' and $n = 100$ and relative error for the Newton method with initial point x_0' and $n = 100$.

Also the relative error behaves as we expect, for the steepest descent is linear with respect to the iterations and for the Newton method is quadratic.

2.4 Chained Powel function

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

where $i = 2j$ and $k = \frac{n-2}{2}$. The argmin and the minimum are respectively $x^* = (0, ..., 0)$, $f(x^*) = 0$. This problem is tested with the initial points $x_0 = (3, -1, 0, 1, ...)$ and $x_0' = (-1, 1, ..., -1, 1)$.
In the following table are shown the computational result of the test for the Chained Powel function, in particular it can be seen the number of iterations of the method k, the time in seconds $Time$ (using tic toc in MATLAB), the value of the function at the last iterate $f(x)$, the initial point x_0 and the dimension n.

Chained Powel									
Steepest descent					Newton				
n	x_0	k	$f(x)$	$Time$	n	x_0	k	$f(x)$	$Time$
4	x_0'	10000	2.9694e-07	0.095843	4	x_0'	28	4.5687e-18	0.002437
10	x_0'	10000	3.3849e-07	0.200602	10	x_0'	28	3.7148e-18	0.010520
50	x_0'	10000	3.3849e-07	1.189545	50	x_0'	28	3.7154e-18	0.168846
100	x_0'	10000	3.3849e-07	2.501956	100	x_0'	28	3.7154e-18	0.948954
4	x_0	10000	1.4367e-06	0.102204	4	x_0	28	3.0521e-18	0.002804
10	x_0	10000	3.5581e-07	0.218348	10	x_0	29	2.8211e-18	0.003465
50	x_0	10000	2.7098e-10	0.059792	50	x_0	29	2.8321e-18	0.015635
100	x_0	10000	3.6387e-07	2.295142	100	x_0	28	6.3049e-18	0.890951

Table 4: Comparison between the steepest descent method and Newton method for the Chained Powel function.

We can easily see from the table that the two methods reach the minimum of the function starting from x_0 and x_0'. The computational time increases with the dimension, despite the dimension gets very high and the cost to compute the hessian and to solve the linear system is higher, the Newton

method finished very fast, also in this case the second method is more precise and finished within less iterations (almost constant), the steepest descent needs more iterations due to zig-zag behaviour.
We have also computed the relative error of the solution as the graphics below show, in this case it can be seen the relative error for $n = 100$ only for the first initial point.

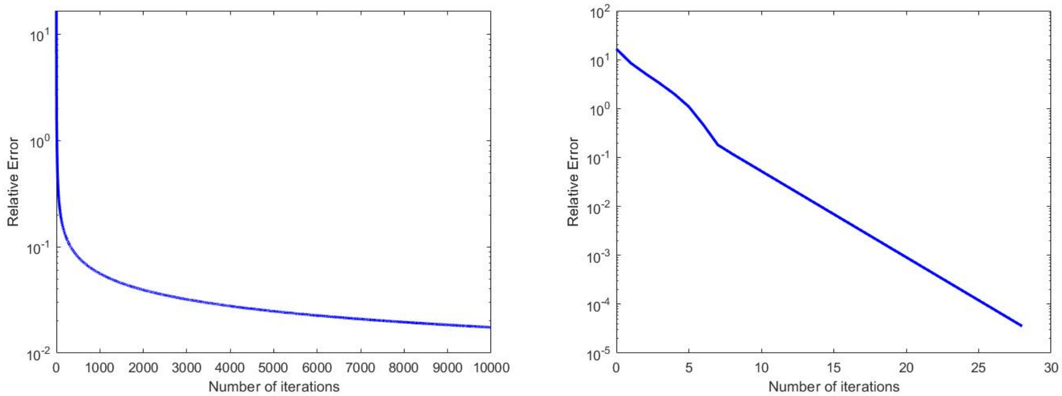

Figure 5: Relative error for the steepest descent method with initial point x_0 and $n = 100$ and relative error for the Newton method with initial point x_0 and $n = 100$.

We can see different behaviours, the error for the first method goes asymptotically to zero, but the relevant observation is that in this case the Newton method doesn't have a quadratic convergence, actually the rate is linear.

Appendix

Steepest descent with backtracking strategy

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    steepest_desc_bcktrck(x0, f, gradf, alpha0, ...
    kmax, tolgrad, c1, rho, btmax)
%
% [xk, fk, gradfk_norm, k, xseq] = steepest_descent(x0, f, gradf, alpha0,
    kmax,
% tollgrad)
%
% 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;
% 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, 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 = -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);

    % 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
```

Newton with backtracking strategy

```
function [xk, fk, gradfk_norm, k, xseq, btseq] = ...
    newton_bcktrck(x0, f, gradf, alpha0, Hessf, kmax, ...
    tolgrad, c1, rho, btmax)
%
% [xk, fk, gradfk_norm, k, xseq] = ...
% newton_bcktrck(x0, f, gradf, alpha0, Hessf, kmax, ...
% tolgrad, 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;
% alpha0 = the initial factor that multiplies the descent direction at
   each
% iteration;
% Hessf = function handle that describes the Hessian 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.
%
% 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, 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);

try chol(Hessf(xk))
    disp('The matrix is positive definite')
catch return
    disp('Error: The matrix isnt positive definite')
end
while k < kmax && gradfk_norm >= tolgrad
    % Compute the descent direction as solution of
    % Hessf(xk) p = - graf(xk)
    pk = -Hessf(xk)\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
```

Initialization and testing example

```
%Chained Wood
%symbolic expression
n = 10;
kk = (n-2)/2;

x = sym('x', [n 1]);

f = 0;
for j = 1:kk
    i = 2*j;
    f = f + 100*(x(i-1)^2-x(i))^2 + (x(i-1)-1)^2 + 90*(x(i+1)^2-x(i+2))^2 ...
    + (x(i+1)-1)^2 + 10*(x(i)+x(i+2)-2)^2 + ((x(i)-x(i+2))^2)/10;
end

g(x) = gradient(f,x);
h(x) = hessian(f,x);

f = matlabFunction(f, 'Vars', {x});
gradf = matlabFunction(g, 'Vars', {x});
Hessf = matlabFunction(h, 'Vars', {x});

%starting points Chained Wood
x0 = zeros(n,1);
for i = 1:n
    if mod(i,2) == 1 & i <= 4
        x0(i) = -3;
    end
    if mod(i,2) == 0 & i <= 4
        x0(i) = -1;
    end
    if mod(i,2) == 1 & i > 4
        x0(i) = -2;
    end
    if mod(i,2) == 0 & i > 4
        x0(i) = 0;
    end
end

%starting points Chained Wood
x0 = zeros(n,1);

kmax = 10000;
```

```
tolgrad = 1.0e-12;
alpha0 = 1;
c1 = 1e-4;
rho = 0.5;
btmax = 50;
%% RUN THE STEEPEST DESCENT f3 (Chained Wood)

disp('**** STEEPEST DESCENT: START *****')

tic
[xk, fk, gradfk_norm, k, xseq, btseq] = ...
    steepest_desc_bcktrck(x0, f, gradf, alpha0, kmax, ...
    tolgrad, c1, rho, btmax);
toc

disp('**** STEEPEST DESCENT: FINISHED *****')
disp('**** STEEPEST DESCENT: RESULTS *****')
disp('************************************')
disp(['xk: ', mat2str(xk), ' (actual minimum: [1;1;...;1]);'])
disp(['f(xk): ', num2str(fk), ' (actual min. value: 0);'])
disp(['N. of Iterations: ', num2str(k),'/',num2str(kmax), ';'])
disp('************************************')

argmin = ones(n,1);
err = zeros(k+1,1);
err(1) = norm(x0-argmin)/norm(argmin);

for i = 1:k
err(i+1) = norm(xseq(:,i)-argmin)/norm(argmin);
end

fig1 = figure();
ax_x = linspace(0,k,k+1) ;
semilogy(ax_x,err','b','LineWidth',2)
xlabel('Number of iterations')
ylabel('Relative Error')
```