Unconstrained Numerical Optimization

Unconstrained assignment: exercise 1

Edoardo Fantolino s286008
Data Science and Engineering
Politecnico di Torino
Torino, Italia
s286008@studenti.polito.it

***Abstract*—This document contains my ideas about the first exercise of the unconstrained assignment. Here, I try to solve a minimization problem. Like everywhere else, some issues arose during the execution and you can find how I dealt with them. This document is addressed to Professor Sandra Pieraccini.**

***Index Terms*—finite differences, minimization, Newton method, optimization.**

I. Introduction

The main objective of this homework is to deal with the following problem

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

To solve this minimization problem it will be implemented the Newton method with line-search, both using the exact derivatives and using finite differences to approximate the gradient and the Hessian matrix.

We need to study the function with dimensions n equal to 10^4 and 10^5.

II. The firsts steps toward the understanding of the problem

A. The function and its characteristics

The function to be analyzed is:

$$f(x) = \sum_{i=1}^{n} \left(\frac{1}{4}x_i^4 + \frac{1}{2}x_i^2 + x_i\right) \tag{1}$$

Consequently, the gradient of f(x) is given by:

$$\nabla f(x) = \begin{bmatrix} x_1^3 + x_1 + 1 \\ x_2^3 + x_2 + 1 \\ \vdots \\ x_n^3 + x_n + 1 \end{bmatrix} \tag{2}$$

The next natural step is the calculation of the Hessian matrix.

$$H_f = \begin{bmatrix} 3x_1^2+1 & 0 & \cdots & 0 \\ 0 & 3x_2^2+1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 3x_n^2+1 \end{bmatrix} \tag{3}$$

As we can see, the Hessian of the function is a diagonal matrix. The eigenvalues are the elements on the diagonal. We can easily show that these elements are all positive and always positive. In fact:

$$3x_i^2 + 1 > 0, \quad \forall x \in \mathbb{R}, \quad i = 1, \cdots, n$$

In conclusion we can say that H is a symmetric positive definite matrix. H symmetric positive definite tell us that the function is convex in its domain and that it will have a minimum that is unique and global.

B. Going deeper into the function behaviour

To practically approach the problem with MATLAB, the firsts steps was to divide the big problem into smaller ones.

Let's give a look to g(x) defined below:

$$g(x) = \frac{1}{4}x^4 + \frac{1}{2}x^2 + x$$

We can get the first and second derivative

$$g'(x) = x^3 + x + 1$$

$$g''(x) = 3x^2 + 1$$

with these information, we can say that $g(x)$ is a continuous convex function in $\mathbb{R}$. Next, we can analytically find the stationary point setting the derivative equal to zero, and we find a global minimum at $\hat{x} \approx -0.6823$.

Then we can analyze the two dimensions case. So, we define $g_2(x_1, x_2)$:

$$g_2(x_1, x_2) = \frac{1}{4}x_1^4 + \frac{1}{2}x_1^2 + x_1 + \frac{1}{4}x_2^4 + \frac{1}{2}x_2^2 + x_2$$

The contour lines of $g_2(x_1, x_2)$ are shown in fig.1. You can also see the path of the Newton method (NM) and the Steepest Descent (SD).

The shape of the two dimension function is like a bowl. In the region $[-1, 0] \times [-1, 0]$ the function is quite flat, while if we move away from that interval the function tend to increase faster. If we apply SD and NM to $g_2(x_1, x_2)$ starting from the point $[1, 2]^T$, the newton method is faster and require fewer iteration. In the table I you can see the obtained results.

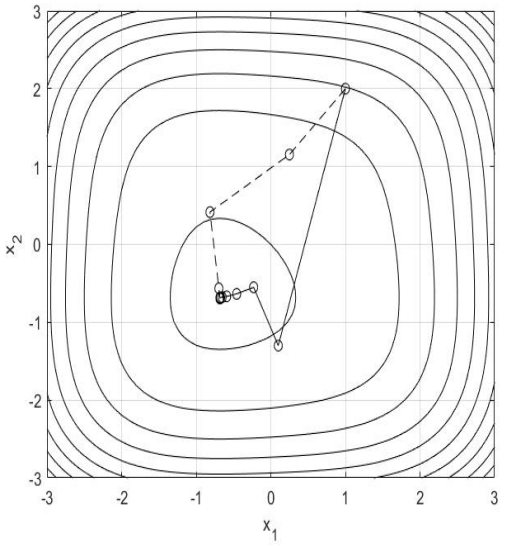

Fig. 1. Contour lines of $g_2(x)$. The full line represent the zig-zag path of SD while the dashed line is the path of the NM.

	NM	**SD**
number of iterations	6	21
Time in seconds	8.971×10^{-4}	1.146×10^{-3}

TABLE I
PERFORMANCES OF NM AND SD.

III. NEWTON METHOD WITH LINE-SEARCH AND EXACT DERIVATIVES

The classical Newton method with line-search is an iterative method that starts from a given point and try to find a solution (in our case the minimum of the function). The NM algorithm is characterized by the following steps that are inside a cycle:

- Given a starting point x_0
- Repeat until $iter < iter_{max}$ and $||\nabla f(x_k)||_2 >= \epsilon$
  - Find the direction p_k, $p_k = H_f^{-1} \nabla f$
  - Update x_k, $x_k = x_k + \alpha p_k$

When we don't have or we don't want to use the exact ∇f and H_f we will need finite differences methods (FD) to approximate the gradient, the Hessian or both.

For the exact derivatives case, the analysis was performed using two strategies that manipulate in different way the step length:

- α with backtrack strategy
- α fixed

The tolerance of the gradient stopping criterion used for the following result will always be equal to 10^{-8}.

I tested the Newton method with backtrack strategy with an initial α = 1,2 and 3. The parameter for the Armijo conditions was $c_1 = 10^{-4}$, $\rho = 0.8$. When we use α equal to 1, we see that the Armijo condition is always satisfied at the first attempt, so in this case we never reduce the value of this parameter. With $\alpha = 2$ we already see an increase in number of iteration

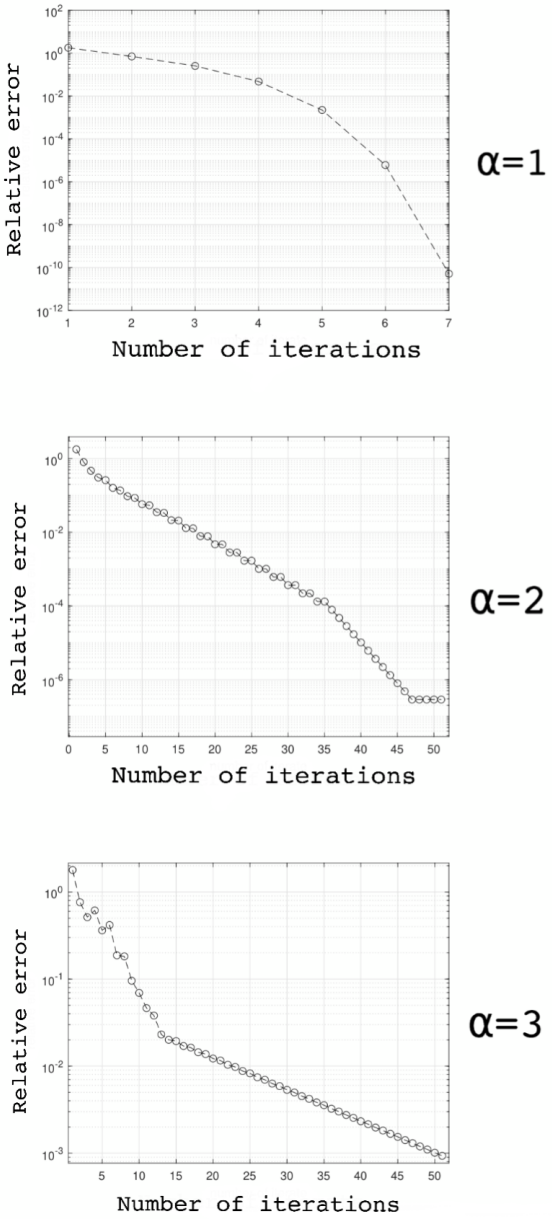

Fig. 2. Sequence of relative error with Newton method and backtrack strategy.

and time. The we can observe a particular behaviour with $\alpha = 3$ where at iteration number 4 and 6 we see an increase in the relative error. During this iteration we have the maxim number of inner iteration in the Armijo condition. In Tab.2 there are the actual result of this tests. (To obtain some reliable time data, the iteration of the same algorithm was performed 100 times.)

The NM with fixed $\alpha = 1$ allow us to have a simpler code and fast convergence. I say simpler code because we did not need to implement any Wolfe conditions. To reach the solution we need 6 iteration and 8.22×10^{-3} seconds in case of $n = 10^4$. For $n = 10^5$, the number of iteration is 7 and the time required is 8.49×10^{-2}. The solution found by

TABLE II
RESULT NEWTON METHOD WITH BACKTRACK STRATEGY

dimensions	10^4		
α	1	2	3
time in second	1.42×10^{-2}	1.32×10^{-1}	2.02×10^{-1}
outer iterations	6	MAX	MAX
avg Armijo iterations	0	0.6	2.1
Convergence	achieved	failed	failed

dimensions	10^5		
α	1	2	3
time in second	1.46×10^{-1}	1.20	1.75
outer iterations	6	MAX	MAX
avg Armijo iterations	0	0.6	2.1
Convergence	achieved	failed	failed

the algorithm was $x_i \approx -0.6823, \quad \forall i = 1, \cdots, n$. We could further increase the precision of the solution without disturb the robustness of the algorithm.

IV. NEWTON METHOD WITH LINE-SEARCH AND APPROXIMATE DERIVATIVES

In this section, we will again approach the same minimization problem with the Newton iterative method, but this time the use of derivative approximations will be made.

We will approximate the derivative using the Finite Differences formulas. We will call h the distance between the points that will allow us to calculate the derivative. The problem asks to use an h as follow:

$$h = 10^{-k}\|\hat{x}\|, \quad k = 2, 4, \cdots, 14$$

Where $\hat{x}$ is the point at which the derivatives have to be approximated. If h is large, we will have a poor approximation of the derivative. If h is small, we will have a better result but we risk to incur in cancellation phenomenon.

We know that a comfortable value of h is:

$$h = \sqrt{\varepsilon_m}\|\hat{x}\| \quad \vee \quad h = \sqrt{\varepsilon_m}$$

where ε_m represent the machine precision.

A. First attempt

Here we present the results that we obtain with a naive implementation of the basic algorithms using the finite differences.

The following problems arise:

- numerical cancellation because we go near or beyond ε_m
- increase in terms of time and cost due to the dimension

In fact, after few attempts and a brief analysis with the "Run and Time" MATLAB tool, we can see that the approximations require too much effort. For the Hessian with $n = 10^4$, we need to compute 10^8 entries with two function evaluations for each one of them. If $n = 10^5$ we have 10^{10} entries. This will creates problems not only in term of time and cost, but even for the storage required to save that matrix.

Here we need an intervention and is essential to exploit the favorable structure of our function.

B. Path towards a possible solution of the problem

To get the best performances I found two main aspect to tackle:

- take advantage of the structure of the function
- choose wisely the starting point

We will exploit the technique of the sparse Jacobian. For example if our Hessian was a tridiagonal matrix we could compute the approximation just with three perturbations in the following directions:

$$p_1 = e_1 + e_4 + e_7 + e_{10} + \ ...$$

$$p_2 = e_2 + e_5 + e_8 + e_{11} + \ ...$$

$$p_3 = e_3 + e_6 + e_8 + e_{12} + \ ...$$

Given that our Hessian is a diagonal matrix we can compute the approximation with just a single perturbation, we just need to observe that the H_f is a diagonal matrix:

$$H_f = \begin{bmatrix} \bullet & & & \\ & \bullet & & \\ & & \ddots & \\ & & & \bullet \end{bmatrix}$$

The circles represent the nonzero elements of the matrix.

1) Hessian: To calculate the Hessian matrix with forward finite differences starting from the gradient we will use the following procedure:

$$H_f \approx \frac{\nabla f(x + he_1) - \nabla f(x)}{h}$$

where e_1 is the first vector of the standard basis, and we will transform the obtained vector in a **diagonal sparse matrix**. We can exploit this formula thanks to the particular structure of the Hessian matrix.

2) Gradient: To approximate the gradient we will use the same procedure as before. We need just to rewrite the function in this way:

$$f(x) = \begin{bmatrix} \frac{1}{4}x_1^4 + \frac{1}{2}x_1^2 + x_1 \\ \frac{1}{4}x_2^4 + \frac{1}{2}x_2^2 + x_2 \\ \vdots \\ \frac{1}{4}x_n^4 + \frac{1}{2}x_n^2 + x_n \end{bmatrix} \tag{4}$$

If we write the function as written above we can exploit the single perturbation technique and compute the forward approximation of the gradient simply applying:

$$\nabla f \approx \frac{f(x + he_1) - f(x)}{h}$$

The resulting vector will be the approximation of the gradient.

We have different combination to essay. To calculate the gradient we have these following FD techniques starting from $f(x)$:

- Jacobian Forward Finite Difference
- Jacobian Centered Finite Difference
- Jacobian Backward Finite Difference

We can calculate H_f in different ways:
Starting from $f(x)$

- Jacobian Centered Finite Difference

Starting from ∇f (Jacobian of the gradiend)

- Jacobian Forward Finite Difference
- Jacobian Centered Finite Difference
- Jacobian Backward Finite Difference

V. Results

I will represent the result in form of tables. In these table you will see two acronyms: NC and MI. NC stand for numerical cancellation. I used this terms in the tables every time that the algorithm encountered a "Warning: Matrix is singular to working precision." or "Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = -1.000000e+00.". Observe that the safe threshold is:

$$\varepsilon_m = 2.2204 \times 10^{-16} \Longrightarrow h = 1.4901 \times 10^{-8} \times \|\hat{x}\|$$

And we will go further.

I wrote MI when the algorithm required a number of iteration that was infeasible.

I tried a consistent number of possible combination of techniques, but I choose to represent the results of the two most significant.

In Tab.3 there are the result in term of time and iteration obtained with the simultaneous approximation of the gradient and the hessian with the Jacobian Centered Finite Difference while in Tab.4 you can see the values obtained with the actual gradient and the approximation of the Hessian with the same method of Tab.3.

The result seems consistent with the theory. If we approximate both ∇f and H_f we require more iteration in order to find the result. This is due to the fact the we approximate the Hessian with an approximation of the gradient, and usually it is not a good practice. The time required is reasonable and is proportional to the number of iteration needed and the dimension of the problem. In Tab.4 we suppose to know the actual f. This will allow us to obtain better results in term of iteration and times. Surprisingly we are able to perform even the calculation with a precision till $h = 10^{-14}$ without incurring in numerical cancellation.

TABLE III
Gradient and Hessian approximated with Jacobian Centered Finite Difference

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	24	7	6	7	NC	NC	NC
	t (ds)	4.76	1.42	1.27	1.44	-	-	-
$n = 10^5$	**iter**	MI	7	7	7	MI	NC	NC
	t (ds)	-	12.9	13.3	13.3	-	-	-

TABLE IV
Actual Gradient and Hessian approximated with Centered Finite Difference

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	15	7	6	6	6	6	7
	t(ds)	0.36	0.17	0.16	0.15	0.15	0.15	0.17
$n = 10^5$	**iter**	59	7	7	7	7	7	7
	t(ds)	11.7	1.14	1.43	1.43	1.43	1.43	1.43

References

Jorge Nocedal, Stephen J. Wright. Numerical Optimization, Springer, 1999.

Sandra Pieraccini. Lecture on finite difference applied to gradients and Jacobians. 2020.

IS THIS A KIND OF SMART STARTING ?

We can observe that if we don't start from a random point, but we choose wisely x_0 (such as a vector that has all the components with the same values):

$$x = (x_1, x_2, \cdots, x_n)^T, \quad x_i = x_j, \quad \forall i, j = 1, \cdots, n$$

The Newton method will act on this vector in a symmetric and clean way. The vector will get closer to the solution in every dimension with the same magnitude. You can use Fig.3 as a reference in two dimension and then try to imagine the same for larger spaces.

A. Gradient

We observe that the gradient is made of the same function in each one of its entries. So, with the constraint that we wisely choose our initial point, to approximate $\nabla f(x_k)$ we can perform the FD formula just once instead of n-times.

B. Hessian matrix

We observe that (3) is a diagonal matrix. So we can exploit this characteristic to improve the function that calculate the approximation of the Hessian.

$$i \neq j \Rightarrow H_f(i,j) = 0, \quad \forall i, j = 1, 2, \cdots, n$$

and the, with the wise choice of the starting point we can use the same approach that we applied to the gradient;

$$H_f(1,1) = H(2,2) = \cdots = H(n,n)$$

This will allow us to reduce the computational cost, the time and the memory space required by the algorithm.

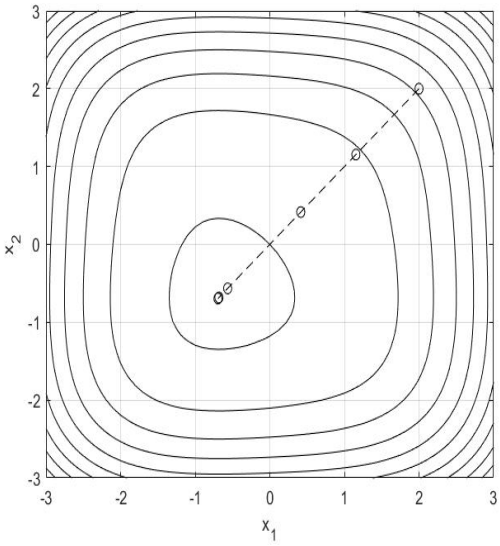

Fig. 3. Wise choice of starting point that simplify the derivatives calculation

TABLE V
TRUE GRADIENT AND HESSIAN APPROXIMATED WITH CENTERED FINITE DIFFERENCE

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	13	6	6	6	NC	NC	NC
	time (ms)	13.5	6.31	6.46	6.92	-	-	-
$n = 10^5$	**iter**	41	7	6	6	9	NC	NC
	time (ms)	403	70.5	60.2	62.8	89.0	-	-

TABLE VI
GRADIENT AND HESSIAN APPROXIMATED WITH CENTERED FINITE DIFFERENCE

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	31	7	6	6	NC	NC	NC
	time (ms)	9.93	2.26	2.69	2.12	-	-	-
$n = 10^5$	**iter**	MI	7	6	6	9	NC	NC
	time (ms)	-	25.5	20.9	20.3	28.4	-	-

OBSERVATIONS

The first test I made was with the true gradient and the centered approximation of the Hessian. You can find the result in Tab.5. As we expected, from higher values of the h parameter we encounter NC. In the second attempt I approximated both the gradient and the Hessian with the centered method of FD. On average, we decrease the time but we increase the number of iteration. This is due to the fact that the algorithm is less precise but given that the functions are optimized at the highest degree we obtain the result faster. The actual results are shown in Tab.6.

Now my target is to obtain some result for the highest values of h, so I tried to use the Jacobian of the gradient that don't require to square h in the denominator. The result in Tab.7 show us that this strategy is not useful. This is because we try to approximate the Hessian from an approximation of the gradient. The last case that I show here is the case were I keep the actual gradient and I approximate H_f with the Jacobian of the gradient. In Tab.8 we can see that we find useful result in feasible times for every h.

TABLE VII
GRADIENT APPROXIMATED WITH CENTERED FINITE DIFFERENCE AND HESSIAN WITH JACOBIAN CENTERED FINITE DIFFERENCE

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	NC	7	6	7	MI	MI	MI
	time (ms)	-	2.85	2.04	2.57	-	-	-
$n = 10^5$	**iter**	MI	7	6	6	MI	MI	MI
	time (ms)	-	21.5	22.3	24.5	-	-	-

TABLE VIII
TRUE GRADIENT AND HESSIAN APPROXIMATED WITH JACOBIAN CENTERED FINITE DIFFERENCE

k		**2**	**4**	**6**	**8**	**10**	**12**	**14**
$n = 10^4$	**iter**	16	7	6	6	6	6	6
	t (ms)	17.5	7.44	6.56	6.60	6.60	6.43	6.65
$n = 10^5$	**iter**	66	7	6	6	6	6	6
	t (ms)	478	67.5	59.1	58.4	58.1	58.7	58.8

The initialization code

Here, there are the initialization of the variables like n, the actual minimum, the starting point x. There is the definition of the funciton handle f, gradf, Hf. Then we define the values of alpha, the maximum number of iterations allowed, the tolerance for the gradient stopping criteria of the newton method.

Typeg and typeh are the variable where we store the techniques that we want to use to approximate the derivatives.

```
close all
clear
clc

n = 1e5;  % value of i
x_min = -0.682327803828019*ones(n,1); % actual minumum
x = rand(n,1);

% function handle for the function, the gradient, and
% the hessian, and reshape of f for future approximation
f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);
gradf = @(x) [x.^3 + x + 1];
Hf = @(x) sparse(1:n,1:n, 3*x.^2+1);
fg = @(x) [(1/4)*x.^4 + (1/2)*x.^2 + x];

alpha = 1;
kmax = 70;
tollgrad = 1e-8;
typeg = '';
typeh = '';
```

Iteration and time analysis

In this page there is the code I used to make the analysis for time and the number of iterations. It is basic MATLAB code.

```
% for the iteration results
for kh=2:2:14
    h = 10^(-kh);
    [xk, fk, gradfk_norm, k, xseq] = ...
            newton_method(x, fg, gradf, Hf, alpha, ...
            kmax, tollgrad, h, typeg, typeh);
end

% for time results
for kh=2:2:14
    h = 10^(-kh);
    num_iter = 5;
    times = zeros(num_iter, 1);
    for j = 1:num_iter
        tic;
        [xk, fk, gradfk_norm, k, xseq] = ...
                newton_method(x, fg, gradf, Hf, alpha, ...
                kmax, tollgrad, h, typeg, typeh);
        times(j,1) = toc;
    end
    figure(1)
    avg_time = mean(times)
    plot(kh,mean(times),'o')
    grid on
    hold on
end
```

Initialization of Newton Method

In the first part of the Newton method function I define two switch clause that allow to understand what kind of approximation we want to use.

```
function [xk, fk, gradfk_norm, k, xseq] = ...
    newton_method(x0, f, gradf, Hf, alpha, kmax, ...
    tollgrad, h, typeg, typeh)

switch typeg
    case 'Jfw'
        gradf = @(x) fd_grad(f, x, h, 'Jfw');
        disp('Gradient approximation with forward method')
    case 'Jbw'
        gradf = @(x) fd_grad(f, x, h, 'Jbw');
        disp('Gradient approximation with backward method')
    case 'Jc'
        gradf = @(x) fd_grad(f, x, h, 'Jc');
        disp('Gradient approximation with centered method')
    otherwise
end

switch typeh
    case 'Jfw'
        Hf = @(x) fd_hess(gradf, x, h, 'Jfw');
        disp('Hessian approximation with forward method')
    case 'Jbw'
        Hf = @(x) fd_hess(gradf, x, h, 'Jbw');
        disp('Hessian approximation with backward method')
    case 'Jc'
        Hf = @(x) fd_hess(gradf, x, h, 'Jc');
        disp('Hessian approximation with centered method')
    case 'c'
        Hf = @(x) fd_hess(f, x, h, 'c');
        disp('Hessian approximation with centered method no(J)')
    otherwise
end
```

Core of Newton method

Here you can find the code that perform the Newton method. There are the main steps like the computation of the descent direction and the check over the norm of the gradient in order to understand if the solution is good enough.

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

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

while k < kmax && gradfk_norm >= tollgrad

    pk = -Hf(xk)\gradf(xk);
    xk = xk + alpha * pk;

    gradfk_norm = norm(gradf(xk));
    k = k + 1;
    xseq(:, k) = xk;
end

fk = f(xk);
xseq = xseq(:, 1:k);

end
```

Gradient and Hessian function

The code for the gradient approximation:

```
function [gradfx] = fd_grad(f, x, h, type)

h = h*norm(x);

switch type
    case 'Jfw'
        gradfx = (f(x+h*(ones))-f(x))/h;
    case 'Jbw'
        gradfx = (f(x)-f(x-h*(ones)))/h;
    case 'Jc'
        gradfx = (f(x+h*(ones))-f(x-h*(ones)))/(2*h);
end

end
```

The code for the Hessian approximation:

```
function [Hessfx] = j_hess(f, x, h, type)

h = h*norm(x);

switch type
    case 'Jfw'
        Hessfx = spdiags((f(x+h*(ones))-f(x))/h, ...
                         0, length(x), length(x));
    case 'Jbw'
        Hessfx = spdiags((f(x)-f(x-h*(ones)))/h, ...
                         0, length(x), length(x));
    case 'Jc'
        Hessfx = spdiags((f(x+h*(ones))-f(x-h*(ones)))/(2*h), ...
                         0, length(x), length(x));
    case 'c'
        Hessfx = ...
        spdiags((f(x+h*(ones))-(2*f(x))+f(x-h*(ones)))/(h^2), ...
                         0, length(x), length(x));
end

end
```