Unconstrained Optimization

Assignment 1:
Newton and finite difference newton methods

Lovera Maria Margherita
Rachid El Amrani

2020/21

For the reader: to visualize the Matlab scripts/functions implemented, see appendix A for the functions and appendix B for the main code.

Introduction

In this assignment we are going to solve the following unconstrained problem

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

by implementing the Newton method, first by using exact derivatives and then by using approximations for both the gradient and the Hessian matrix.

Newton method with exact derivatives

The Newton method locally approximates f(x) with a quadratic model $m_k(p)$, which is then optimized by computing its stationary points. Indeed, we obtain

$$p_k = -(\nabla^2 f(x_k))^{-1}\nabla f(x_k) \tag{1}$$

and the iterative formula is given by $x_{k+1} = x_k + \alpha p_k$, where p_k is the quantity written above. Before proceeding, we need to understand how to choose the step length α since we know that not all values are suitable. The Armijo condition is not enough to ensure reasonable progress in the alorithm we are going to build, so we decided to use a backtracking strategy for the step-length selection of our implementation. This means that we choose an initial step length $\alpha_k^{(0)}$ and, for $j \geq 0$, if the Armijo condition is satisfied than we accept $\alpha_k^{(j)}$, otherwise we reduce the value of $\alpha_k^{(j)}$ by a quantity ρ and we choose the new value obtained as our next step length.

Function newton_exact

The function *newton_exact* will implement on Matlab the Newton optimization method for a given function f, using backtracking strategy for the choice of the step-length α.

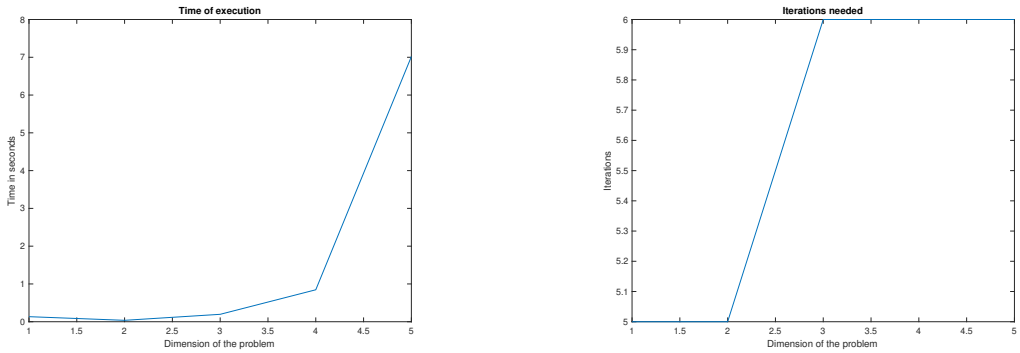
- Inputs of the function:
  - **x0**: starting point of our optimization method (n-dimensional column vector)
  - **f**: function handle variable that describes the function that we want to minimize, in our case $f : \mathbb{R}^n \to \mathbb{R}$
  - **gradf**: function handle variable that describes the gradient of f
  - **Hessf**: function handle variable that describes the Hessian matrix of f

- **alpha0**: step length of the optimization method (for the Newton method it is crucial to have $\alpha_0 = 1$ to get eventually second order convergence)
- **kmax**: maximum number of iterations allowed
- **tollgrad**: tolerance allowed, with respect to the norm of the gradient, in order to stop the method
- **c1**: factor for the Armijo condition, it must be a scalar in $(0, 1)$: we take $c_1 = 10^{-4}$ because we want the Armijo condition to be easy to satisfy
- **rho**: fixed factor used for the step length reduction in the backtracking strategy, it must be a scalar strictly less than 1
- **max_bcktrck**: maximum number of iterations allowed for the backtracking algorithm

- Outputs of the function:
  - **xk**: the last vector computed by the method
  - **fk**: the value of the function f evaluated in x_k
  - **gradfk_norm**: the norm of the gradient of $f(x_k)$
  - **k**: the index of the last iteration performed
  - **xseq**: matrix where the columns are the x_k computed during the iterations
  - **bcktrckiters**: the vector of dimension 1-by-k containing the number of backtracking iterations for each iteration of the method

Newton method with exact derivatives: implementation

We then implemented this function in order to solve our problem. Even if in the assignment the query was to solve the problem for $n = 10^4$ and $n = 10^5$, we tried more values of n in order to have a better view on the results.
For n assuming 5 different values (100, 10^3, 10^4, 10^5 and 10^6), we noticed that, as expected, the computational time increases as n increase, but despite this and the quite big dimension, the time elapsed is not superior to 8 seconds (see Figure 1a). More interesting is the plot representing the number of iterations with respect to the 5 different values of n. It is possible to observe that for $n = 10^4$ and $n = 10^5$ (i.e. the values of n that interest us, respectively the third and fourth) we just need 6 iterations in order to converge (see Figure 1b).



(a) Computational time

(b) Number of iterations

Figure 1: *newton_exact* implementation

Furthermore, we tried different starting points in order to see if there were any changes in terms of speed of convergence: we noticed that by starting from a vector of all zeros we had improvements in terms of both computational time and iterations needed with respect to a

vector of all ones and a vector with random values between 0 and 1. We also observed that the method that we implemented did not use the backtracking strategy at any iteration. This is probably due to the fact that we are applying the Newton method on exact derivatives, so it is still quite simple to converge and we do not need to use the backtracking strategy. We can confirm this also by looking at the plot representing the error with respect to n, where we consider as error the norm of the gradient taken in the last vector computed by the method (the more is close to 0, the better our method is performing). Indeed, for the first two values (i.e. 100, 1000), even if we do not have perfect convergence, the error is quite small, while for the last three values (i.e. 10^4, 10^5, 10^6) we have perfect convergence (see Figure 2).

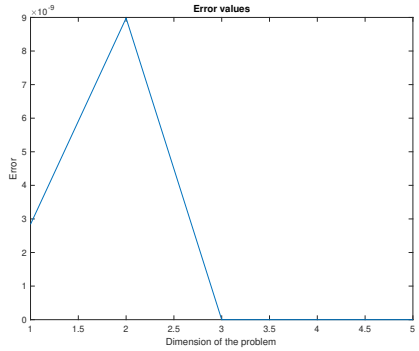


Figure 2: Gradient values for *newton_exact* implementation

Newton method with approximations

We now apply the Newton method with the only difference that, while before we had exact values for both the gradient and the Hessian of $f(x)$, now we are going to use approximation's methods in order to approximate the gradient and the Hessian of our function $f(x)$. In order to do this, we defined some functions that will allow us to do the approximations.

Approximation of the gradient

The function *fd_grad* will implement on Matlab the approximation of the gradient of $f(x)$ with finite differences: we allow an approximation with both the methods described in class (i.e. forward and centered).

Function fd_grad

- Inputs of the function:
  - **x**: n-dimensional column vector
  - **f**: function handle variable that describes the function f, in our case $f : \mathbb{R}^n \to \mathbb{R}$
  - **h**: step used for the finite difference computation of the gradient of $f(x)$
  - **type**: string of characters such that if type is equal to *'forward'* the function uses the forward method and if type is equal to *'centered'* the function uses the centered method (the default value will be set to *'forward'* since the centered method is slightly heavier computationally)
- Outputs of the function:
  - **gradf**: a column vector of the same size of x corresponding to the approximation of the gradient of $f(x)$ computed with finite differences

Approximation of the Hessian matrix

We discussed in class two different methods, based on two different scenarios, to approximate the Hessian of a function: the first one in which we are able to compute $\nabla f(x)$, and so $\nabla^2 f(x)$ is nothing but the Jacobian of the gradient, and the second one in which $\nabla f(x)$ is not available, so we go directly to the second finite differences to approximate $\nabla^2 f(x)$. We will implement in Matlab both of them.

Function approx_Hess

The function *approx_Hess* will implement on Matlab the approximation of the Hessian of $f(x)$ by computing the Jacobian of the $\nabla f(x)$ (i.e. fd_grad) obtained with the finite difference method.

- Inputs of the function:
  - **x**: n-dimensional column vector
  - **f**: function handle variable that describes the function f, in our case $f : \mathbb{R}^n \to \mathbb{R}$
  - **h**: step used for the finite difference computation of the Hessian of $f(x)$
  - **type**: string of characters such that if type is equal to *'forward'* the function uses the forward method to approximate $\nabla f(x)$ and if type is equal to *'centered'* the function uses the centered method (the default value will be set to *'forward'*)
- Outputs of the function:
  - **Hessf_sparse**: sparse n-by-n matrix corresponding to the approximation of the Hessian of $f(x)$

Since our Hessian matrix is diagonal, to compute the Jacobian of $\nabla f(x)$ we need just 2 evaluations of the gradient instead of n+1. Also, we do not have to bother with the symmetric approximation of $\nabla^2 f(x)$ because we are dealing with a diagonal matrix so we already have symmetry.
Notice that this function requires as input the function f and not directly the gradient of f. This is due to the fact that with this function we observed improvements in terms of reliability and convergence (see discussion on page 6, *Approximation with fd_grad and approx_Hess*). The function *approx_Hess_givengrad* instead is structured exactly as *approx_Hess* with the difference that it requires as input *gradf* (for this reason we reported it just in the Appendix A without presenting it here).

Function fd_Hess

The function *fd_Hess* will implement on Matlab the approximation of the Hessian of $f(x)$ with the centered finite difference method we discussed in class. Here it should be important to assume a regular f (i.e. $f \in C^2$) such that the Hessian matrix obtained is symmetric, but as we already stated before in this specific case the Hessian is a diagonal matrix so we don't need to make further specifications.

- Inputs of the function:
  - **x**: n-dimensional column vector
  - **f**: function handle variable that describes the function f, in our case $f : \mathbb{R}^n \to \mathbb{R}$
  - **h**: step used for the finite difference computation of the Hessian of $f(x)$
- Outputs of the function:
  - **Hessf_sparse**: sparse n-by-n matrix corresponding to the approximation of the Hessian of $f(x)$

Newton optimization general method

Now we are going to build the function *newton_opt_general* that considers both the cases in which the gradient and the Hessian are known or have to be evaluated. This means that we will be able to perform the Newton optimization method by implementing both the backtracking strategy and, if necessary, approximations for both $\nabla f(x)$ and $\nabla^2 f(x)$.

newton_opt_general

The function newton_opt_general will implement on Matlab the Newton optimization method for a given function f using the backtracking strategy for the choice of the step length α. Furthermore if the gradient and the Hessian are not given, they will be approximated. Notice also that this time we used the PCG solver to compute the descent direction since it was convenient in terms of computational time.
In order to build correctly the approximations, we implemented both *approx_Hess_givengrad* and *fd_Hess* so that if the gradient is given we can just approximate the Hessian matrix with *approx_Hess_givengrad*, if it is not given its approximation will be made with *fd_grad* and, in the case of an approximated gradient, we will allow the Hessian to be approximated with *fd_Hess* or, eventually, with a Matrix-Free implementation.

- Inputs of the function:
  - **x0**: starting point of our optimization method (n-dimensional column vector)
  - **f**: function handle variable that describes the function f, in our case $f : \mathbb{R}^n \to \mathbb{R}$
  - **gradf**: function handle variable that describes the gradient of f. If this variable is an empty vector (i.e. if it is not given), the gradient will be computed with respect to the input $'gradf\_option'$ stated below
  - **Hessf**: function handle variable that describes the Hessian of f. If this variable is an empty vector (i.e. if it is not given), the Hessian will be computed with respect to the input $'Hessf\_option'$ stated below
  - **kmax**: maximum number of iterations allowed
  - **tollgrad**: tolerance allowed, with respect to the norm of the gradient, in order to stop the method
  - **alpha0**: step length of the optimization method (for the Newton method it is crucial to have $\alpha_0 = 1$ to get eventually second order convergence)
  - **c1**: factor for the Armijo condition, it must be a scalar in $(0, 1)$: we take $c_1 = 10^{-4}$ because we want the Armijo condition to be easy to satisfy
  - **rho**: fixed factor used for the step length reduction in the backtracking strategy, it must be a scalar strictly less than 1
  - **max_bcktrck**: maximum number of iterations allowed for the backtracking algorithm
  - **gradf_option**: string of characters such that if the string is equal to *'forward'* the function uses the forward finite difference method to approximate $\nabla f(x)$ and if it is equal to *'centered'* the function uses the centered finite difference method. This variable will be used only if the value of $'gradf'$ is missing
  - **Hessf_option**: string of characters such that: if the string is equal to *'findiff'* the function uses the centered finite difference method to approximate $\nabla^2 f(x)$, if it is equal to *'approxHess_fw'* (*'approxHess_c'* respectevely) the Jacobian forward (centered respectively) finite difference approximation and if it is equal to *'matrix_free'* the function lets the algorithm solve the linear system (1) using the PCG solver instead of the backslash solver. This variable will be used only if the value of $'Hessf'$ is missing
  - **h**: step used for the finite difference computation of both the gradient and the Hessian of $f(x)$

- Outputs of the function:
  - **xk**: the last vector computed by the method
  - **fk**: the value of the function f evaluated in x_k
  - **gradfk_norm**: the norm of the gradient of $f(x_k)$
  - **k**: the index of the last iteration performed
  - **xseq**: matrix where the columns are the x_k computed during the iterations
  - **bcktrckiters**: the vector of dimension 1-by-k containing the number of backtracking iterations for each iteration of the method

Newton method with finite differences: implementation

Before proceeding we want to clarify that for the implementations executed from now on, we consider as error:

$$norm(x_N - x_k)/norm(x_k)$$

where x_N is the last vector computed by the Newton exact method and x_k is the last vector computed by the method considered.

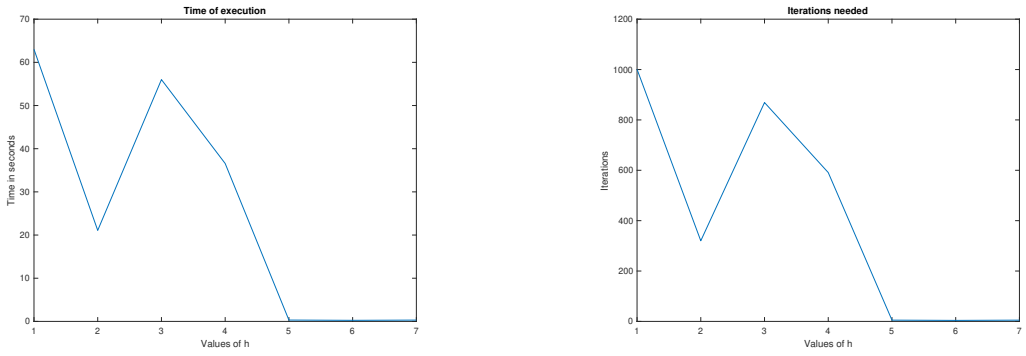
Approximation with *fd_grad* and *approx_Hess*

At first we tried to solve our problem by implementing the function *newton_exact*, approximating the gradient with the forward finite difference method and the Hessian matrix by computing the Jacobian of the approximated gradient. Let's notice that the function *approx_Hess* requires as input the function f and not directly the gradient. This means that actually with this implementation we are computing two times the gradient of f: one time in order to set it as input of *newton_exact* and the other time inside the function *approx_Hess* to use it for the computation of the Jacobian. We chose to do this because, by trying to implement the function *approx_Hess_givengrad* instead (which has as input directly the gradient), we noticed that in terms of both convergence and errors we had worse results and that we would have needed bigger dimensions in order to have good results in terms of both reliability and interpretation. We reported here the Table with the results obtained from *approx_Hess_givengrad* implementation, with $n = 1000$ and maximum number of iterations equal to 100 (it is to compare with Table 3 for the central values at page 7).

Values of h	Computational time	Number of iterations	Error value
10^{-2}	90s	101	1.50119e-1
10^{-4}	85s	101	1.58010e-3
10^{-6}	6s	7	1.58108e-5
10^{-8}	76s	101	3.59981e-6
10^{-10}	84s	101	5.14750e-1
10^{-12}	85s	101	4.73845e-1
10^{-14}	86s	101	5.04353e-1

Table 1: n = 1000 with *approx_Hess_givengrad*

Now starting with our actual analysis, we chose to modify the starting point in order to get closer to the actual solution (we selected a vector with all components equal to -0.5): this allowed us to reduce a bit the computational time elapsed. Since we are doing an approximation over another approximation, we do not expect to obtain good results from this implementation.
As requested by the assignment, we tried 7 different values for the increment h in order to understand better the behavior of the implementations performed. As a first thing we observed



(a) Computational time

(b) Number of iterations

Figure 3: *approx_Hess* implementation

that in order to try all these values of h in reasonable time we needed a quite small dimension n. So we set $n = 200$ and, by modifying h, we noticed that, as it is showed in Figure 3a, while reducing the value of h (i.e. from 10^{-2}, first value, to 10^{-14}, seventh value) we had a general reduction in terms of computational time elapsed with a peak for h = 10^{-4} and, by looking at Figure 3b, we can observe that we have an analogous behavior in terms of number of iterations. By taking a look at Table 2, containing the error values, we can see that even if the last two values of h seem to be good with respect to computational cost and number of iteration needed, actually they are not very good in terms of error.

Values of h	Error value
10^{-2}	3.61810e-1
10^{-4}	7.07010e-4
10^{-6}	7.07029e-6
10^{-8}	2.42704e-8
10^{-10}	2.38552e-6
10^{-12}	6.96500e-5
10^{-14}	1.16557e-2

Table 2: n = 200

Since we want to analyze the problem for higher dimensions and considering the observations made above (and also knowing from the theory that in general the best values for h are the central ones), in order to reduce the computational cost and improve the model in terms of convergence we decided to leave behind the first and last values of h in order to work only with the central five (i.e. 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10} and 10^{-12}). This allowed us to reduce the maximum number of iterations from 1000 to 100 (with the knowledge that for some values of h we could not reach convergence) and to increase the dimension to 1000 (see Table 3). Notice that we could have chosen a different value for the maximum number of iterations in order to allow the converge for every value but, since our goal is to increase again the dimension later on, in order to reduce significantly the computational time we chose to set it to 100.

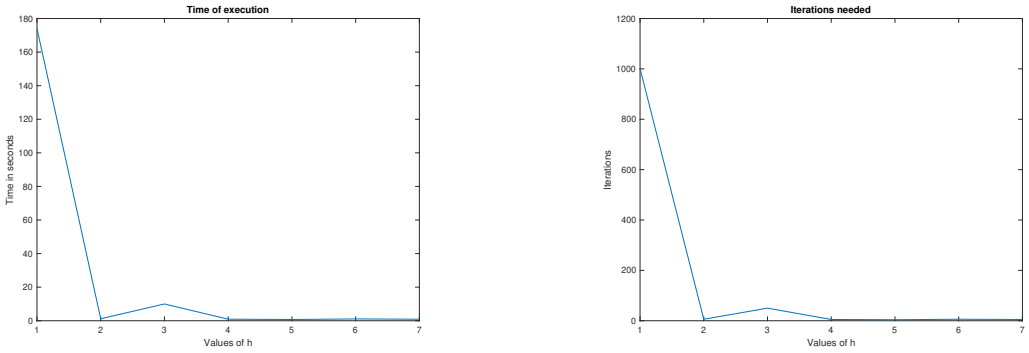
Values of h	Computational time	Number of iterations	Error value
10^{-4}	83s	101	5.69804e-1
10^{-6}	89s	101	1.43770e-5
10^{-8}	91s	101	6.93832e-6
10^{-10}	2s	3	6.60449e-6
10^{-12}	5s	6	3.95323e-4

Table 3: n = 1000

As expected, 100 iterations are not enough to guarantee convergence, but this allowed us to see that the minimum values of the error seem to be for the values of h equal to 10^{-8} and 10^{-10} respectively, even if for 10^{-8} we do not reach convergence (as we saw already in Table 2).
In order to have more precision, we should increase the dimension of the problem and, as requested by the assignment, we should see what happens for $n = 10^4$ and $n = 10^5$. But by doing some trials on Matlab with those dimensions, and as we have already understood from the previous trials, with the above application we have problems in terms of computational time: this is due to the fact that, even if we tried to improve the situation, we are approximating the Hessian matrix using an approximated gradient. We decided then to focus on other approximation methods, expecting better results.

Approximation with *fd_grad* and *fd_Hess*

This time we tried to solve our problem with an approximation of the gradient with forward finite difference method and, separately, of the Hessian matrix with the function fd_Hess (i.e. with the finite difference method). We set again the starting point equal to a vector with all components equal to -0.5: this allowed us to reduce a bit the computational time elapsed. Furthermore, we implemented the function *newton_opt_general* in order to use the backtracking strategy and to approximate both the gradient and the Hessian matrix with finite differences.



(a) Computational time

(b) Number of iterations

Figure 4: *newton_opt_general* implementation

As we did before, we started with a quite small dimension in order to understand better what happens. We chose to set a bigger starting dimension since we expect better results and by setting $n = 500$ we obtained quite interesting results (see Figure 4): in terms of both computational time and number of iterations needed it is possible to observe that all values of h seem to be quite good both in terms of computational time and number of iterations needed, a part from $h = 10^{-2}$ (see Figures 4a and 4b respectively). But before jumping to conclusions, let's have a look at the errors (see Table 3):

Values of h	Error value
10^{-2}	1.08396e-1
10^{-4}	1.11803e-3
10^{-6}	1.11803e-5
10^{-8}	8.39455e-8
10^{-10}	4.95716e-6
10^{-12}	5.07161e-4
10^{-14}	3.07494e-2

Table 4: n = 500

We can see that the two extreme values of h have a quite high error with respect to the others. So in order to increase the dimension of the problem without having too many problems in terms of computational time and iterations needed, we decided to do as we did before and pull off the extreme values of h (i.e. 10^{-2} and 10^{-14}). Then we increased the dimension to $n = 1000$ and reduced the number of iterations to 100 because our first goal is to compare the performance of this implementation with that one of the method with the implementation of *approx_Hess*.

Values of h	Computational time	Number of iterations	Error value
10^{-4}	4s	6	1.58066e-3
10^{-6}	3s	5	1.58115e-5
10^{-8}	2s	4	9.04720e-8
10^{-10}	2s	3	9.85104e-7
10^{-12}	3s	4	6.36335e-4

Table 5: n = 1000

By looking at Table 5 we can see that we have very good results both in computational time elapsed and in number of iterations needed and, in particular, by comparing it to Table 3 we can see that this implementation performs a lot better than the previous one. Furthermore, it is possible to observe that the best value for h in terms of error seems to be 10^{-8}.
So we increased again the dimension of the problem ($n = 10^4$) and reduced the number of iterations to 50 to see if something changes radically with respect to what we have already seen.

Values of h	Computational time	Number of iterations	Error value
10^{-4}	541s	10	4.99512e-3
10^{-6}	239s	5	4.99988e-5
10^{-8}	798s	16	4.79731e-7
10^{-10}	331s	5	8.81014e-6
10^{-12}	208s	2	6.87436e-4

Table 6: n = 10^4

Here we can see again that the best value for h is 10^{-8} since it has the smallest value in terms of error. As expected the computational time increased a lot with respect to before, so in order to increase the dimension to 10^5 and see the results in quite reasonable time, we decided to mantain just the three central values of h (i.e. 10^{-6}, 10^{-8} and 10^{-10}) while leaving unchanged the number of iterations.

Values of h	Computational time	Number of iterations	Error value
10^{-6}	36651s	5	1.58097e-4
10^{-8}	18221s	3	2.18185e-6
10^{-10}	15639s	3	1.14964e-4

Table 7: n = 10^5

From Table 7 we can conclude that the best choice of h for our implementation is $h = 10^{-8}$, so we will use it in order to compare the computational time and efficiency of this implementation (the best in terms of approximation) with respect to the Newton implementation with exact derivatives. It is interesting to notice once again that the computational time in this case is quite big, so the computational time increases as the dimension increases, as expected.

Approximations of the Hessian matrix with known gradient

In order to complete our assignment we decided to show what happens if we have the exact form of the gradient (i.e. known gradient) and we have to approximate the Hessian matrix.

In theory we have two methods in order to do this: with an approximation of the Hessian as the Jacobian of the gradient of f and with a Matrix-Free implementation. Since we chose to focus on finite differences methods and Jacobian computations, we chose the first of the two possibilities stated before by implementing the function *approx_Hess_givengrad*. Even if we do not know what are the results yet, we expect them to be more precise than the above approximations since the gradient is known.
We started with $n = 2000$ and maximum number of iterations equal to 100 in order to see what happened. As we can see from Table 8 we can confirm that also with this implementation the best values for h in terms of error seem to be the middle ones: 10^{-8} and 10^{-10}. Furthermore, it is clear that in this scenario we have a great improvement in terms of computational time.

Values of h	Computational time	Number of iterations	Error value
10^{-2}	9s	101	4.68137e-6
10^{-4}	8s	101	9.00401e-7
10^{-6}	1s	4	2.23340e-11
10^{-8}	1s	4	2.43090e-13
10^{-10}	1s	4	4.90737e-13
10^{-12}	1s	4	4.31412e-12
10^{-14}	1s	4	9.52727e-11

Table 8: n = 2000

But it is still too soon to make assumptions because we first need to increase the dimension of the problem. In order to do that without spending too much time on computations we left behind the three values of h with bigger error (i.e. 10^{-2}, 10^{-4} and 10^{-14} respectively) and then increased the dimension to 10^4 in order to check which is the best value for h within the central ones (see Table 9).

Values of h	Computational time	Number of iterations	Error value
10^{-6}	6s	4	4.80977e-11
10^{-8}	6s	4	4.91387e-14
10^{-10}	7s	4	4.83252e-13
10^{-12}	7s	4	1.60498e-12

Table 9: n = 10^4

The best value here seems to be $h = 10^{-8}$ but since the computational effort is quite reasonable, we increased again the dimension to 10^5, this time reducing the maximum number of iterations to 50 for computational reasons.

Values of h	Computational time	Number of iterations	Error value
10^{-6}	13417s	51	6.31356e-11
10^{-8}	1245s	4	1.19088e-13
10^{-10}	1156s	4	4.63889e-13
10^{-12}	997s	4	4.94804e-13

Table 10: n = 10^5

We can see from the results shown in Table 10 that $h = 10^{-8}$ seems again to be the best value for our problem. Indeed, we can see that both in terms of computational time and error value it is convenient with respect to the others.
Let's notice that, as expected, with known gradient the error is significantly smaller than those cases where the gradient is unknown and so approximated.

Conclusions

Let's now compare the results of *newton_exact* implementation and *newton_opt_general* implementation, both settled with the best parameters found with the analysis. We decided to compare these two implementations since the first one is the exact one, while the second one is the best one in terms of approximation of both the gradient and the Hessian of f. We will compare the results for three values of the dimension n: $n = 10^3$, $n = 10^4$ and $n = 10^5$.

$n = 10^3$

**** NEWTON EXACT: RESULTS *****
xk: [-0.682327803946513;-0.682327803946513; ... -0.682327803946513;]
f(xk): -395.353
N. of Iterations: 5/100;
****
Elapsed time is 0.095727 seconds.
gradfk_norm_N = 8.980697969285327e-09

**** NEWTON GENERAL WITH h=10^{-8} AND FINITE DIFFERENCES: RESULTS *****
xk: [-0.682327871504884;-0.682327871504884; ... 0.682327871504884;]
f(xk): -395.353
N. of Iterations: 5/50;
****
Elapsed time is 3.419042 seconds.
err = 1.069464375825645e-07

$n = 10^4$

**** NEWTON EXACT: RESULTS *****
xk: [-0.682327803828019;-0.682327803828019; ... -0.682327803828019;]
f(xk): -3953.5304
N. of Iterations: 6/100;
****
Elapsed time is 0.200863 seconds.
gradfk_norm_N = 0

**** NEWTON GENERAL WITH h=10^{-8} AND FINITE DIFFERENCES: RESULTS *****
xk: [-0.682327904262283;-0.682327904262283; ... -0.682327904262283;]
f(xk): -3953.5304
N. of Iterations: 7/50;
****
Elapsed time is 524.653655 seconds.
err = 4.867457544313441e-07

$n = 10^5$

**** NEWTON EXACT: RESULTS *****
xk: [-0.682327803828019;-0.682327803828019; ... -0.682327803828019;]
f(xk): -39535.3045
N. of Iterations: 6/100;
****
Elapsed time is 0.946145 seconds.
gradfk_norm_N = 0

**** NEWTON GENERAL WITH h=10^{-8} AND FINITE DIFFERENCES: RESULTS *****
xk: [-0.682329269826406;-0.682329269826406; ... -0.682329269826406;]
f(xk): -39535.3045
N. of Iterations: 3/50;
****
Elapsed time is 1.8220788e+04 seconds.
err = 2.181846853599283e-06

As we can see, in terms of computational effort the results obtained with the Newton exat method are far better than those obtained with the approximations: this is what we expected,

since we are doing a comparison between exact values and approximations. But by looking at the value x_k and the value of $f(x_k)$ we can see that the approximation is quite good and we also converge with quite the same number of iterations in the two cases. So despite the great computational effort needed for the approximation, we can conclude that our function approximate quite well the solution.

Appendix A

```
function [xk, fk, gradfk_norm, k, xseq, bcktrckiters] = newton_exact(x0, f, ...
    gradf, Hessf, alpha0, kmax, tollgrad, c1, rho, max_bcktrck)

% Storage vectors
xseq = zeros(length(x0), kmax);
bcktrckiters = zeros(1, kmax);

% Initialization of the parameters
xk = x0;
k = 0;

% Evaluation of fk, gradfk and gradfk_norm
fk = f(xk);
gradfk = gradf(xk);
gradfk_norm = norm(gradfk);
% We evaluate these quantities before the while loop in order to do it just
% one time and not at every iteration

% Solution of the linear system: Hessf(xk)*pk = -gradf(xk)
pk_solver = @(x) -Hessf(x)\gradf(x);
% We compute it before the while loop so we compute it just one time and not
% at every iteration

% Function handle for the armijo condition
farmijo = @(fk, alpha, xk, pk) fk + c1*alpha*gradfk'*pk;
% We handle it before the while loop in order to do it just
% one time and not at every iteration


while k ≤ kmax && gradfk_norm ≥ tollgrad

    % Resolution of the linear system in xk
    pk = pk_solver(xk);

    % Set alpha
    alpha = alpha0;

    % Compute the new candidate point and evaluate the function in the new point
    xnew = xk + alpha*pk;
    fnew = f(xnew);

    % Initialize the number of inner iterations
    counter = 0;

    % Backtracking iterations
    while counter ≤ max_bcktrck & fnew>farmijo(fk, alpha, xk, pk)

        % Reduce the value of alpha
        alpha = rho*alpha;

        % Update x_(k+1) and f(x_(k+1)) w.r.t. the new alpha
        xnew = xk + alpha*pk;
        fnew = f(xnew);

        % Increment the counter
        counter = counter + 1;

    end

    % Update the values and computations
    xk = xnew;
    fk = fnew;
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk);

    % Increment the number of iterations performed
```

```
    k = k + 1;

    % Store current xk in the k-th column of xseq
    xseq(:, k) = xk;
    % Store the number of backtrack iterations in the k-th component of bcktrckseq
    bcktrckiters(k) = counter;

end

% "Cut" xseq and bcktrckseq to the correct size
xseq = xseq(:, 1:k);
bcktrckiters = bcktrckiters(1:k);

end
```

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

% Empty gradient to fill
gradf = zeros(size(x));

% Step h of the computation
h = norm(x)*h;

% Build type input
switch type
    case 'forward'
        for i=1:length(x)
            xh = x; % just for readability
            xh(i) = xh(i) + h;
            gradf(i) = (f(xh) - f(x))/ h;
        end
    case 'centered'
        for i=1:length(x)
            xh_plus = x;
            xh_minus = x;
            xh_plus(i) = xh_plus(i) + h;
            xh_minus(i) = xh_minus(i) - h;
            gradf(i) = (f(xh_plus) - f(xh_minus))/(2 * h);
        end
    otherwise % default: 'forward'
        for i=1:length(x)
            xh = x;
            xh(i) = xh(i) + h;
            gradf(i) = (f(xh) - f(x))/ h;
        end
end
end
```

```
function [Hessf_sparse] = approx_Hess(f, x, h, type)

% Empty Hessian to fill
n = length(x);
Hessf_sparse = sparse(1, 1, 0, n, n);

% Step h
h = norm(x)*h;

% Perturbation
e = h + zeros(n,1);

% Hessian is diagonal, so we try to do just two gradient evaluations
% Different finite difference methods to approximate the gradf
switch type
    case 'forward'
        gradf = @(x) fd_grad(f, x, h, 'forward');
        gradf_xh = gradf(x+e);
```

```
        gradf_x = gradf(x);
        for i=1:n
            gradf_xh_i = gradf_xh(i);
            gradf_x_i = gradf_x(i);
            i_col_val = (gradf_xh_i - gradf_x_i)/ h;
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end

    case 'centered'
        gradf = @(x) fd_grad(f, x, h, 'centered');
        gradf_xhp = gradf(x+e);
        gradf_xhm = gradf(x-e);
        for i=1:n
            gradf_xhp_i = gradf_xhp(i);
            gradf_xhm_i = gradf_xhm(i);
            i_col_val = (gradf_xhp_i - gradf_xhm_i)/ (2*h);
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end

    otherwise % forward method as default
        gradf = @(x) fd_grad(f, x, h, 'forward');
        gradf_xh = gradf(x+e);
        gradf_x = gradf(x);
        for i=1:n
            gradf_xh_i = gradf_xh(i);
            gradf_x_i = gradf_x(i);
            i_col_val = (gradf_xh_i - gradf_x_i)/ h;
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end
end

end
```

```
function [Hessf_sparse] = approx_Hess_givengrad(gradf, x, h, type)

% Empty Hessian to fill
n = length(x);
Hessf_sparse = sparse(1, 1, 0, n, n);

% Step h
h = norm(x)*h;

% Perturbation
e = h + zeros(n,1);


switch type
    case 'forward'
        gradf_xh = gradf(x+e);
        gradf_x = gradf(x);
        for i=1:n
            gradf_xh_i = gradf_xh(i);
            gradf_x_i = gradf_x(i);
            i_col_val = (gradf_xh_i - gradf_x_i)/ h;
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end

    case 'centered'
        gradf_xhp = gradf(x+e);
        gradf_xhm = gradf(x-e);
        for i=1:n
            gradf_xhp_i = gradf_xhp(i);
            gradf_xhm_i = gradf_xhm(i);
            i_col_val = (gradf_xhp_i - gradf_xhm_i)/ (2*h);
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end

    otherwise % forward method as default
```

```
        gradf_xh = gradf(x+e);
        gradf_x = gradf(x);
        for i=1:n
            gradf_xh_i = gradf_xh(i);
            gradf_x_i = gradf_x(i);
            i_col_val = (gradf_xh_i - gradf_x_i)/ h;
            Hessf_sparse = Hessf_sparse + sparse(i, i, i_col_val, n, n);
        end
end

end
```

```
function Hessf_sparse = fd_Hess(f, x, h)

% Empty Hessian to fill
n = length(x);
Hessf_sparse = sparse(1, 1, 0, n, n);

% Step h
h = norm(x)*h;

for j = 1:n

    % Elements on the diagonal (just the diagonal because we have an Hessian ...
        diagonal matrix)
    xh_plus = x;
    xh_minus = x;
    xh_plus(j) = xh_plus(j) + h; % sum h to the j-th component of x
    xh_minus(j) = xh_minus(j) - h; % subtract h to the j-th component of x
    value = (f(xh_plus) - 2*f(x) + f(xh_minus))/(h^2);
    Hessf_sparse = Hessf_sparse + sparse(j, j, value, n, n);

end
end
```

```
function [xk, fk, gradfk_norm, k, xseq, bcktrckiters] = newton_opt_general(x0, ...
    f, ...
    gradf, Hessf, kmax, tollgrad, alpha0, c1, rho, max_bcktrck, ...
    gradf_option, Hessf_option, h)

%%%% Gradf and Hessf options

% Gradf options
if isempty(gradf) % i.e. if the gradient is not given

    switch gradf_option
        case 'forward'
            gradf = @(x) fd_grad(f, x, h, 'forward');

        case 'centered'
            gradf = @(x) fd_grad(f, x, h, 'centered');

        otherwise
            % we use the input function handle gradf
    end

    % If we have an approximated gradient and Hessf is missing, than it is
    % highly suggested to approximate Hessf just with finite differences
    % and matix free implementation
    if isempty(Hessf)
        switch Hessf_option
            case 'findiff'
                Hessf = @(x) fd_Hess(f, x, sqrt(h));

            case 'matrix_free'
                Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x))/h;
```

```
        end
    end


else % i.e. if the gradient is given
    if isempty(Hessf) % if the Hessian is not given
        switch Hessf_option % I can compute the Hessian however I want
            case 'findiff'
                Hessf = @(x) fd_Hess(f, x, sqrt(h)); % sqrt(h) in order to ...
                    avoid numerical calculation problems

            case 'approxHess_fw'
                Hessf = @(x) approx_Hess_givengrad(gradf, x, h, 'forward');

            case 'approxHess_c'
                Hessf = @(x) approx_Hess_givengrad(gradf, x, h, 'centered');

            case 'matrix_free'
                Hessf_pk = @(x, p) (gradf(x + h * p) - gradf(x)) / h;

        end
    else % if the Hessian is given
        % do nothing
    end
end

% Storage vectors
xseq = zeros(length(x0), kmax);
bcktrckiters = zeros(1, kmax);

% Initialization of the parameters
xk = x0;
k = 0;

% Evaluation of fk, gradfk and gradfk_norm
fk = f(xk);
gradfk = gradf(xk);
gradfk_norm = norm(gradfk);

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

while k ≤ kmax && gradfk_norm ≥ tollgrad

    % Resolution of the linear system in xk
    switch Hessf_option

        case 'matrix_free'
            [pk, ¬] = pcg(@(p) Hessf_pk(xk,p), -gradfk);

        otherwise
            % pk = -Hessf(xk)\gradfk;
            [pk, ¬] = pcg(Hessf(xk), -gradfk);

    end

    % Set alpha
    alpha = alpha0;

    % Compute the new candidate point and evaluate the function in the new point
    xnew = xk + alpha*pk;
    fnew = f(xnew);

    % Initialize the number of inner iterations
    counter = 0;

    % Backtracking iterations
    while counter ≤ max_bcktrck && fnew>farmijo(fk, alpha, xk, pk)

        % Reduce the value of alpha
```

```
100         alpha = rho*alpha;
101
102         % Update x_(k+1) and f(x_(k+1)) w.r.t. the new alpha
103         xnew = xk + alpha*pk;
104         fnew = f(xnew);
105
106         % Increment the counter
107         counter = counter + 1;
108
109     end
110
111     % Update the values and computations
112     xk = xnew;
113     fk = fnew;
114     gradfk = gradf(xk);
115     gradfk_norm = norm(gradfk);
116
117     % Increment the number of iterations performed
118     k = k + 1;
119
120     % Store current xk in the k-th column of xseq
121     xseq(:, k) = xk;
122     % Store the number of backtrack iterations in the k-th component of bcktrckseq
123     bcktrckiters(k) = counter;
124
125 end
126
127 % "Cut" xseq and bcktrckseq to the correct size
128 xseq = xseq(:, 1:k);
129 bcktrckiters = bcktrckiters(1:k);
130
131 end
```

Appendix B

Main Code Newton Exact

```
% Initialization of the index for the n cycle
i = 0;

% Evaluate f, gradf with function handle
% out of the loop for computational reasons
f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);
gradf = @(x) x.^3 + x + 1;

% We will solve the problem for both n = 10^4 and n = 10^5
for n = [1e2 1e3 1e4 1e5 1e6]
    i = i + 1;

    % Evaluate Hessf with function handle
    % inside the loop because it depends from n
    Hessf = @(x) spdiags(3*x.^2 +1, 0, n, n);

    % Starting point
    % x0 = rand(n,1);
    % x0 = ones(n,1);
    x0 = zeros(n,1); % less time and iterations needed

    % Parameters
    alpha0 = 1; % optimal alpha for the Newton method
    kmax = 100;
    tollgrad = 1e-8;
    c1 = 1e-4; % easier to satisfy the Armijo condition
    rho = 0.8;
    max_bcktrck = 20;

    % Function implementation
    disp('**** NEWTON EXACT: START *****')

    tic
    [xk_N, fk_N, gradfk_norm_N, k_N, xseq_N, bcktrckiters_N] = ...
        newton_exact(x0, f, gradf, Hessf, ...
    alpha0, kmax, tollgrad, c1, rho, max_bcktrck);

    disp('**** NEWTON EXACT: FINISHED *****')
    disp('**** NEWTON EXACT: RESULTS *****')
    disp('************************************')
    disp(['xk: ', mat2str(xk_N)])
    disp(['f(xk): ', num2str(fk_N)])
    disp(['N. of Iterations: ', num2str(k_N),'/',num2str(kmax), ';'])
    disp('************************************')

    % Iterations needed, time elapsed and error
    iterations_N(i) = k_N;
    time_N(i) = toc;
    err_N(i) = gradfk_norm_N;


end

%% plots
% Time elapsed
figure(1)
plot(time_N)
xlabel('Dimension of the problem')
ylabel('Time in seconds')
title('Time of execution')

% Iterations needed
figure(2)
plot(iterations_N)
```

```
xlabel('Dimension of the problem')
ylabel('Iterations')
title('Iterations needed')

% Error
figure(3)
plot(err_N)
xlabel('Dimension of the problem')
ylabel('Error')
title('Error values')
```

Main Code Newton General

```
% Finite differences with approx_Hess

% Dimension of the problem
n = 1000; % 200

% Parameters
kmax = 100; % 1000
tollgrad = 1e-8;
alpha0 = 1;
c1 = 1e-4;
rho = 0.8;
max_bcktrck = 20;

% Starting point
x0 = -0.5*ones(n,1);

% Evaluate f with function handle out of loops
f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);

% Initialization of the index for the cycle
i = 0;

% Truncate xk computed with Newton method to dimension n
xk_N = xk_N(1:n);

for h = [1e-2 1e-4 1e-6 1e-8 1e-10 1e-12 1e-14] % [1e-2 1e-4 1e-6 1e-8 1e-10 ...
    1e-12 1e-14]

    % Increment the index
    i=i+1;

    % Set gradf and Hessf
    gradf = @(x) fd_grad(f, x, h, 'forward');
    Hessf = @(x) approx_Hess_givengrad(gradf, x, h, 'forward');

    tic
    % Implement newton_exact function
    disp('**** NEWTON EXACT WITH APPROX_HESS: START *****')

    [xk_1, fk_1, gradfk_norm_1, k_1, xseq_1, bcktrckiters_1] = ...
        newton_exact(x0, f, gradf, Hessf, alpha0, kmax, tollgrad, c1, ...
        rho, max_bcktrck);

    disp('**** NEWTON EXACT WITH APPROX_HESS: FINISHED *****')
    disp('**** NEWTON EXACT WITH APPROX_HESS: RESULTS *****')
    disp('************************************')
    disp(['xk: ', mat2str(xk_1)])
    disp(['f(xk): ', num2str(fk_1)])
    disp(['N. of Iterations: ', num2str(k_1),'/',num2str(kmax), ';'])
    disp('************************************')

    % Iterations needed, time elapsed and error
    iterations_1(i) = k_1;
    time_1(i) = toc
```

```
54
55      err_1(i) = norm(xk_N-xk_1)/norm(xk_1);
56
57      % Vector with the norm of gradf at each iteration (so for each h)
58      norm_grad_1(i) = gradfk_norm_1;
59
60  end
61
62  %% Plots
63  figure(1)
64  plot(time_1)
65  xlabel('Values of h')
66  ylabel('Time in seconds')
67  title('Time of execution')
68
69  figure(2)
70  plot(iterations_1)
71  xlabel('Values of h')
72  ylabel('Iterations')
73  title('Iterations needed')
74
75  figure(3)
76  plot(err_1)
77  xlabel('Values of h')
78  ylabel('Error')
79  title('Error values')
```

```
1   % Finite differences with fd_Hess
2
3   % Dimension of the problem
4   n = 10^5; % 500-1000-10^4
5
6   % Parameters
7   kmax = 50; %1000-100
8   tollgrad = 1e-8;
9   alpha0 = 1;
10  c1 = 1e-4;
11  rho = 0.8;
12  max_bcktrck = 20;
13
14  % Starting point
15  x0 = -0.5*ones(n,1);
16
17  % Evaluate f with function handle out of loops
18  f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);
19
20  % Initialization of the index for the cycle
21  i = 0;
22
23  % Truncate xk computed with Newton method to dimension n
24  xk_N = xk_N(1:n);
25
26  for h = [1e-6 1e-8 1e-10] %[1e-2 1e-4 1e-6 1e-8 1e-10 1e-12 1e-14]
27
28      % Increment the index
29      i = i+1;
30
31      % Set the gradf and Hessf options
32      gradf_option = 'forward';
33      Hessf_option = 'findiff';
34
35      % Empty gradf and Hessf to fill
36      gradf = [];
37      Hessf = [];
38
39      tic
40      % Implement newton_opt_general function
41      disp('**** NEWTON GENERAL WITH FD_HESS: START *****')
```

```
42
43      [xk_2, fk_2, gradfk_norm_2, k_2, xseq_2, bcktrckiters_2] = ...
            newton_opt_general(x0, f, ...
44      gradf, Hessf, kmax, tollgrad, alpha0, c1, rho, max_bcktrck, ...
45      gradf_option, Hessf_option, h);
46
47      disp('**** NEWTON GENERAL WITH FD_HESS: FINISHED *****')
48      disp('**** NEWTON GENERAL WITH FD_HESS: RESULTS *****')
49      disp('************************************')
50      disp(['xk: ', mat2str(xk_2)])
51      disp(['f(xk): ', num2str(fk_2)])
52      disp(['N. of Iterations: ', num2str(k_2),'/',num2str(kmax), ';'])
53      disp('************************************')
54
55      % Iterations needed, time elapsed and error
56      iterations_2(i) = k_2;
57      time_2(i) = toc
58      err_2(i) = norm(xk_N-xk_2)/norm(xk_2);
59
60  end
61
62  %% Plots
63  figure (1)
64  plot(time_2)
65  xlabel('Values of h')
66  ylabel('Time in seconds')
67  title('Time of execution')
68
69  figure (2)
70  plot(iterations_2)
71  xlabel('Values of h')
72  ylabel('Iterations')
73  title('Iterations needed')
74
75  figure(3)
76  plot(err_2)
77  xlabel('Values of h')
78  ylabel('Error')
79  title('Error values')
```

```
1   % Known gradient, approx_Hess_givengrad approximation
2
3   % Dimension of the problem
4   n = 10^5; % 2000-10^4
5
6   % Parameters
7   kmax = 50; %100-100
8   tollgrad = 1e-8;
9   alpha0 = 1;
10  c1 = 1e-4;
11  rho = 0.8;
12  max_bcktrck = 20;
13
14  % Starting point
15  x0 = -0.5*ones(n,1);
16
17  % Evaluate f and gradf with function handle out of loops
18  f = @(x) sum((1/4)*x.^4 + (1/2)*x.^2 + x);
19  gradf = @(x) x.^3 + x + 1;
20
21  % Initialization of the index for the cycle
22  i = 0;
23
24  % Truncate xk computed with Newton method to dimension n
25  xk_N = xk_N(1:n);
26
27
28  for h = [1e-6 1e-8 1e-10 1e-12] %[1e-2 1e-4 1e-6 1e-8 1e-10 1e-12 1e-14]
```

```
29
30      % Increment the index
31      i = i+1;
32
33      % Set the gradf and Hessf options
34      gradf_option = '';
35      Hessf_option = 'approxHess_fw';
36
37      % Empty Hessf to fill
38      Hessf = [];
39
40
41      tic
42      % Implement newton_opt_general function
43      disp('**** NEWTON GENERAL WITH KNOWN GRADIENT&APPROX_HESS: START *****')
44
45      [xk_3, fk_3, gradfk_norm_3, k_3, xseq_3, bcktrckiters_3] = ...
            newton_opt_general(x0, f, ...
46      gradf, Hessf, kmax, tollgrad, alpha0, c1, rho, max_bcktrck, ...
47      gradf_option, Hessf_option, h);
48
49      disp('**** NEWTON GENERAL WITH KNOWN GRADIENT&APPROX_HESS: FINISHED *****')
50      disp('**** NEWTON GENERAL WITH KNOWN GRADIENT&APPROX_HESS: RESULTS *****')
51      disp('************************************')
52      disp(['xk: ', mat2str(xk_3)])
53      disp(['f(xk): ', num2str(fk_3)])
54      disp(['N. of Iterations: ', num2str(k_3),'/',num2str(kmax), ';'])
55      disp('************************************')
56
57      % Iterations needed, time elapsed and error
58      iterations_3(i) = k_3;
59      time_3(i) = toc
60      err_3(i) = norm(xk_N-xk_3)/norm(xk_3);
61
62  end
63
64  figure (1)
65  plot(time_3)
66  xlabel('Values of h')
67  ylabel('Time in seconds')
68  title('Time of execution')
69
70  figure (2)
71  plot(iterations_3)
72  xlabel('Values of h')
73  ylabel('Iterations')
74  title('Iterations needed')
75
76  figure(3)
77  plot(err_3)
78  xlabel('Values of h')
79  ylabel('Error')
80  title('Error values')
```