Assignment in Optimization 2016

You can choose one of the following two alternatives.

- **Alternative 1**. *Nonlinear least-square fitting using the Gauss-Newton method.* The task is to solve an optimization problem numerically by an approximation of the Newton method.
- **Alternative 2**. *Quasi-Newton methods.* The task is to compare two slightly different quasi-Newton algorithms and to solve numerically several penalty/barrier problems.

The main objective of the assignment in both alternatives is to implement an optimization algorithm in Matlab, to experiment and analyze how the algorithm behaves in different examples and to do eventual final modifications to improve the performance. It means, in particular, that the cpu time used by the implementation is not of primary interest to minimize, though it should be within a reasonable bound (see examples). The project can be divided into three steps:

1. Implement the algorithm you have chosen in Matlab. The block representing your choice of a line search (Armijo? GS? Newton? - try to figure out which one is more appropriate) should be implemented and tested separately. In particular, run your line search implementation for the provided test function in order to get an idea of possible misbehaviour and to make an improvement. Pay attention to a smart structuring of the program that makes it easier for you to test different parts independently and to find eventual errors.
2. Test your program **very thoroughly** to ensure that it works correctly before you continue with the Step 3. Note that even a small implementation error may result in all your work in the next step being meaningless. Try to run many examples/initial points and find out whether the program behaves in an expected way that is consistent with the theory (for instance, fast/slow/no convergence, decreasing/increasing directions etc).
3. Use the program to solve the problems under *Things to do*. Try to observe as much different behaviour of the algorithm as possible by using many initial points, and make, if necessary, some final tuning of the program (for example, by modifying the stopping criterion, tuning parameters in the line search etc) to reduce the discovered undesirable drawbacks. Write a report.

You are encouraged to work in groups of 2–3 people. Individual work is also permitted if you prefer to work alone. One report per group. *Any copy-paste similarities between reports will lead to both groups being failed.*

Hand in your report to Andrey Ghulchak, room MH:350 (just slide it under the door if I am not there). Provide all the names of groups partners, their personal numbers and e-mails. Keep a copy for yourself. Send all relevant m-files to `ghulchak` (at) `maths.lth.se` You are likely to be asked to correct your program/complete your report as well as to eventually give oral explanations and clarifications of some details. The corrections may take several iterations before being accepted.

All corrections of the assignment must be fully accepted by **January 31, 2017**.

The files needed for the project can be found on the course homepage
http://www.maths.lth.se/matematiklth/personal/ghulchak/optimization/2016/

Alternative 1: `phi1.m phi2.m data1.m data2.m test_func.m`

Alternative 2: `grad.m rosenbrock.m test_func.m`

Some practical recommendations for both alternatives

- Do not use symbolic calculations in MATLAB. They are *extremely* slow. Everything must be done numerically.
- Use for your main MATLAB-function exactly the same name and the set and order of input parameters as it says in the project description. Think that I will need to test it for all of you, and my tests would take much longer for nonstandard names.
- Pay attention to MATLAB's warnings and fix the troubles. For example, when it complains about "division by zero" or anything similar you should find the corresponding piece of code and do something about it. Use "help" to see what a particular warning is about.
- Learn and use MATLAB's build-in debugger to track variables, values, and function returns. It does not take much time to learn it, but it will save a lot of time for you in your search for errors. The most convenient way to do debugging is from the MATLAB's editor graphical interface. Look for videos "MATLAB debugging" on `www.youtube.com`.
- The implementation (including all parts and subroutins) must be written in such a way that there is no need to make any modifications by hand when running for differerent tests or starting points.
- A good choice and a good implementation of the line search is crucial for convergence and reasonable execution time of your algorithm. To help you with that I have put some recommendations for a line search implementation below.

Some practical recommendations for a line search implementation

- Line search should never fail. A failure on the iteration, say, number 51 means the failure of the whole algorithm and makes the calculations of the previous 50 iterations useless. Design your line search strategy such that it always returns an answer (`lambda`).
- Due to the limited bit calculations one can come across functional values that are `Inf` or `NaN`. Chapter 2 does not say anything about what to do in this case. It is up to you to figure it out and to make a necessary modification to your algorithm.
- Use the format `[lambda,No_of_iterations]=linesearch(@func,x,d)` for the line search that calculates the value of λ for the function $F(\lambda) = \texttt{func}(x + \lambda d)$ and the number of preformed iterations for that. To make sure that the value of λ is acceptable, put the following checking lines at the end of your line search function

  ```
  if isnan(func(x+lambda*d)) || func(x+lambda*d)>func(x)
    error('Bad job of the line search!')
  end
  ```

  and fix your code if necessary until it is free of the error message.
- Use several test functions to test the performance. For example, the quadratic functions

  ```
  a=2;   % try a = -2 too, then a = 5 and -5, then a = 10 and -10
  func=@(x)(1-10^a*x)^2;
  ```

  with the exact minimum being equal to zero.

Alternative 1

Consider the problem of fitting the given set of data (t, y) by the function $\phi(x,t) = x_1 e^{-x_2 t} + x_3 e^{-x_4 t}$, i.e. solving the least square problem

$$\min_x f(x) \qquad \text{where} \qquad f(x) = \sum_{i=1}^{m} \left(\phi(x, t_i) - y_i\right)^2 .$$

Write the Matlab function

```
function gaussnewton(phi,t,y,start,tol,use_linesearch,printout,plotout)
```

that solves this problem by the Gauss-Newton method (See page 94 in the course book). Here `phi` denotes the function handle for the fitting function, `t,y` are the given data to be fitted, `start` is the initial point chosen by the user and `tol` is a user defined tolerance for termination. Different termination criteria are discussed on the page 107. The parameter `use_linesearch` takes the values 1 or 0 depending on whether you want to use a linesearch or not. Similarly `printout` and `plotout` can be turned on and off. Suppose now that we would like to use the function *phi*2 stored in the m-file `phi2.m` starting at $(1, 2, 3, 4)$. A command may look like

```
gaussnewton(@phi2,t,y,[1;2;3;4],0.1,1,1,1);
```

Of course, it works only if the set of data to be fitted (t, y) has already been defined, for example, by the command `[t,y]=data1`. (The file `data1.m` can be downloaded from the homepage of this course). In this example, the tolerance is chosen to be 0.1, linesearch as well as printing and plotting will be performed. The printing on the screen could look something like this (you are free to choose any reasonable output data for visual checkout that makes sense to see):

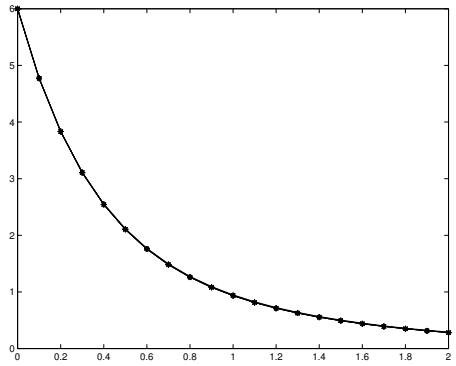
iter	x	step size	f(x)	max(abs(r))	norm(grad)	ls iters	lambda	grad'*d/norm(d)
.								
.								
.								
6	2.7811	2.0173	39.43	0.1068	1.3234	1	1.0000	-0.2993
	1.3834							
	3.2169							
	3.0136							
7	1.8615	0.9196	34.98	0.0775	1.4456	2	0.5000	-0.0910
	1.0950							
	4.1369							
	2.8589							

Nice printing is obtained by a command of the type `fprintf('%12.4f  %4.3f  %12.4f\n',a,b,c)`. To find out how it works try the following sequence of commands (see also `help fprintf`)

```
a=1.234;
b=5.678;
c=1.111;
fprintf('%12.4f  %12.1f  %12.4f\n',a,b,c);
fprintf('%12.4f  %12.1f  %12.8f\n',a,b,c);
fprintf('%12.4f  %12.1f  %32.8f\n',a,b,c);
```

Another useful command is `A\b` which solves the linear system $Ax = b$ by a least square method even if A is not quadratic, i.e. $\min \|Ax - b\|$. **Avoid solving the normal equation by `pinv`.**

If the function is stored in the m-file `func.m` you can compute its value at a point x by `func(x)`. The same can be done via the function handle, i.e. `my_f=@func` using `my_f(x)`.
Here is an example of how a possible plot of the data and the optimal function could look like



Write your line search as a separate program `linesearch(@func,x,d)` that calculates the value of λ for an approximate minimization of $F(\lambda) = \text{func}(x + \lambda d)$. Test it with `test_func.m` (from the homepage)

```
lambda_1=linesearch(@test_func,[0;0],[1;0]);
lambda_2=linesearch(@test_func,[0;0],[0;1]);
```

The functional values in both tests should be $-1 \leq F(\lambda_k) < 0$ (the smaller the better) and the search should be relatively fast (to compare with mine: $F(\lambda_k) \approx -0.9$ found in 0.006 sec).

Things to do:

- Use your program (with and without the line search) to fit `data1.m` and `data2.m` by the function `phi1.m` for *many* (≥ 20) different start points. To compare with mine: with the line search and tol=10^{-6} it needs 10–20 iterations for most starting points to converge.
- Repeat for `data1.m` and `phi2.m`.
- Repeat for `data2.m` and `phi2.m`. This time you have to figure out a strategy of how a suitable initial point could be found if you do not use the line search.

Your report should contain the following

- Your program. (Send a copy of the program by email as well).
- The best least squares value and the optimal choice of parameters in all 4 cases.
- A discussion of your choice of line search and stopping criterion and how those affect the convergence.
- A discussion of consistency in the program behaviour and how you have tested it.
- A discussion of how the convergence and the solution are affected by different initial points.
- A strategy of choosing the initial point for `data2.m`.
- Few typical printed outputs confirming your conclusions.

Alternative 2

Write the Matlab function

```
function nonlinearmin(f,start,method,tol,printout)
```

minimizing a function f by use of the DFP and BFGS algorithms. (See page 82 and 89 in the course book). Here `f` denotes the function handle for the objective function, `start` is the initial point chosen by the user, `method` takes the values 'DFP' or 'BFGS'. The parameter `tol` is a user defined tolerance for termination (different termination criteria are discussed on the page 107) and the parameter `printout` is given the value 1 or 0 depending on whether you want things to be printed on the screen or not.
Suppose now that we would like to minimize the function *func* stored in the m-file `func.m` starting at $(1, 2, 3, 4)$. Then you may give the command

```
nonlinearmin(@func,[1;2;3;4],'DFP',0.1,1)
```

In this example the DFP algorithm is chosen with the tolerance = 0.1, and the results will be printed on the screen. The text on the screen could look like this (for the inner loop):

iteration	x	step size	f(x)	norm(grad)	ls iters	lambda
.						
.						
.						
6	2.7811	2.0173	0.1068	1.3234	3	0.5
	1.3834					
	3.2169					
	3.0136					
7	1.8615	0.9196	0.0775	1.4456	5	16
	1.0950					
	4.1369					
	2.8589					

Nice printing is obtained by a command of the type `fprintf('%12.4f  %4.3f  %12.4f\n',a,b,c)`. To find out how it works try the following sequence of commands.

```
a=1.234;
b=5.678;
c=1.111;
fprintf('%12.4f  %12.1f  %12.4f\n',a,b,c);
fprintf('%12.4f  %12.1f  %12.8f\n',a,b,c);
fprintf('%12.4f  %12.1f  %32.8f\n',a,b,c);
```

If the function is stored in the m-file `func.m` you can compute its value at a point x by `func(x)`. The same can be done via the function handle, i.e. `my_f=@func` using `my_f(x)`. The function handle is a convenient way to construct new functions inside another function, for example

```
a=4;
my_f=@(x)x^2+a/x;                    % my_f(2) would be = 6
my_g=@(x,y)my_f(x)+y(1)^2+y(2)^2;    % my_g(2,[1;2]) would be = 11
```

To calculate the search directions you may use the numerical differentiation in the file `grad.m`. Do not use it directly in the line search subroutine as the fixed precision in `grad.m` may not be enough there in some cases.

Write your line search as a separate program `linesearch(@func,x,d)` that calculates the value of λ for an approximate minimization of $F(\lambda) = \text{func}(x + \lambda d)$. Test it with `test_func.m` (from the homepage)

```
lambda_1=linesearch(@test_func,[0;0],[1;0]);
lambda_2=linesearch(@test_func,[0;0],[0;1]);
```

The functional values in both tests should be $-1 \leq F(\lambda_k) < 0$ (the smaller the better) and the search should be relatively fast (to compare with mine: $F(\lambda_k) \approx -0.9$ found in 0.006 sec).

Things to do:

- Use your program to minimize the function `rosenbrock.m`. Try many different initial points for both methods. To compare with mine: DFP with tol=10^{-6} needs around 50 outer iterations for the starting point $[200; 200]$.
- Use `nonlinearmin` as a subroutine in another Matlab program to solve the following problem by using a penalty function and the sequence $\mu = 0.01, 0.1, 1, 10, 100$. See page 313 and the bottom of page 314 in the course book.

  $$\min e^{x_1x_2x_3x_4x_5} \quad \text{subject to } x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 = 10, \quad x_2x_3 = 5x_4x_5, \quad x_1^3 + x_3^3 = -1$$

  A suitable initial point for $\mu = 0.01$ is $(-2, 2, 2, -1, -1)$. However other initial points may give different results. Find them and explain what happens.

  Compare the two algorithms and different tolerances.
- Apply your program to Exercises 9.3 and 9.5. Use the similar strategy for a sequence of μ_k or ϵ_k to increase accuracy of the answer. Please note the text on the top and the bottom paragraph of page 324.

Your report should contain the following

- Your program. (Send a copy of the program by email as well).
- The optimal point(s) and the optimal function value for each problem.
- A discussion of your choice of line search and stopping criterion.
- A discussion of consistency in the program behaviour and how you have tested it.
- A discussion of how different initial points affect the convergence and the solution.
- A practical comparison of these two quasi-Newton methods.
- A strategy of choosing initial points in the penalty problem.
- A discussion of how satisfactorily your algorithm solves Excercises 9.3 and 9.5.
- Few typical printed outputs confirming your conclusions.