COMP0120 : Numerical Optimization Assignment 3

Oluwatosin Alabi

March 2020

Question 1

1B

```
%L-BFGS method, algorithm 7.3 and 7.4 from norcedal and wright
%
%

m = 5; %pick any number between 3 and 20 should work
k = 1; %count iterations, matlab indexing starts from 1
n = length(x_k);
%
%

s_k = x_k-x_k_1;
y_k = F.df(x_k) - F.df(x_k_1);
%
%
S = zeros(n,m); % store m values of s_k
Y = zeros(n,m); % store m values of y_k
%
%

H0 = (s_k'*y_k)/(y_k'*y_k)*eye(n); %initial H
p_k = zeros(length(F.df(x_k)),1); %initialize descent direction

if (k<=m)
    % update S,Y
    S(:,k) = s_k;
    Y(:,k) = y_k;
    p_k = -1 * lbfgs(g1,S(:,1:k),Y(:,1:k),H0);
else
    S(:,1:(m-1))=S(:,2:m); % shift to the left
    Y(:,1:(m-1))=Y(:,2:m);
    S(:,m) = s_k; % at the new ones
    Y(:,m) = y_k;
    p_k = -1 * lbfgs(g1,S,Y,H0);
end

 alpha_k = ls(x_k, p_k, alpha0);

% Update x_k and f_k
x_k_1 = x_k;
x_k = x_k + alpha_k*p_k;
```

```
%%%
%%%

function H_new = lbfgs(grad,S,Y,H0)
% This function returns the approximate Hessian multiplied by the gradient, H*g
% Input
%   S:     Memory of s_i (n by k)
%   Y:     Memory of y_i (n by k)
%   g:     gradient (n by 1)
%   H0 : Initial hessian
% Output
%   Hg     the the approximate inverse Hessian multiplied by the gradient g
% Notice
% This funcion getHg_lbfgs is called by LBFGS_opt.m.
% Ref
%   Nocedal, J. (1980). "Updating Quasi-Newton Matrices with Limited Storage".
%   Wiki http://en.wikipedia.org/wiki/Limited-memory_BFGS
%   two loop recursion

    [n,k] = size(S);
    for i = 1:k
        rho(i,1) = 1/(Y(:,i)'*S(:,i));
    end

    q = zeros(n,k+1);
    alpha =zeros(k,1);
    beta =zeros(k,1);

    % step 1
    q(:,k+1) = grad;

    % loop 1
    for i = k:-1:1
        alpha(i) = rho(i)*S(:,i)'*q(:,i+1);
        q(:,i) = q(:,i+1)-alpha(i)*Y(:,i);
    end

    % Multiply by Initial Hessian
    r = H0*q(:,1);

    % loop 2
    for i = 1:k
        beta(i) = rho(i)*Y(:,i)'*r;
        r = r + S(:,i)*(alpha(i)-beta(i));
    end
    H_new=r; %approximate hessian
end
```

The implementation is more efficient because we do not explicitly store the inverse matrix H. The inverse matrix H is dense and the cost of manipulating it is very high, these costs increases when the number of data to be manipulated increases. Instead of storing H explicitly, we store a m number (between 3 and 20) of $s_i and y_i$ (i = k-m, k-m+1, k-m+2,...., k-1). The descent direction $H_k \nabla f_k$ is then calculated using inner products. These method is more memory efficient.

Question 3

All the parameters of the minimization?:
The question asks to minimize the function $f(x, y) = (x - 3y)^2 + x^4$ using the BFGS method and the SR1 method.
For the BFGS method, the parameters used are:

Maximum iterations = 200
Tolerance = 1e-10
c1 for strong wolfe condition = 1e-4
c2 for strong wolfe condition = 0.2
initial alpha = 1
, For the SR1 method, the parameter used are :
max iterations = 200
stopping tolerance = 1e-10
initial trust region radius = 0.2,
Step acceptance relative progress threshold (η) = 0.1,
overall bound on step lengths ($\hat{\Delta}$) = 1
The path taken plot and the convergence plots are shown in figure 1(a) and figure 1(b). The convergence plots where done using the MATLAB "semilogy" function.

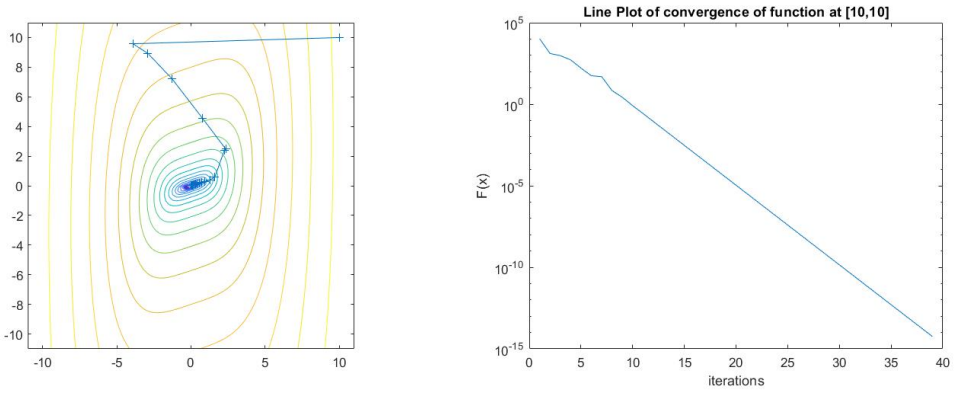

Figure 1: Contour plot BFGS (left), convergence plot BFGS (Right)

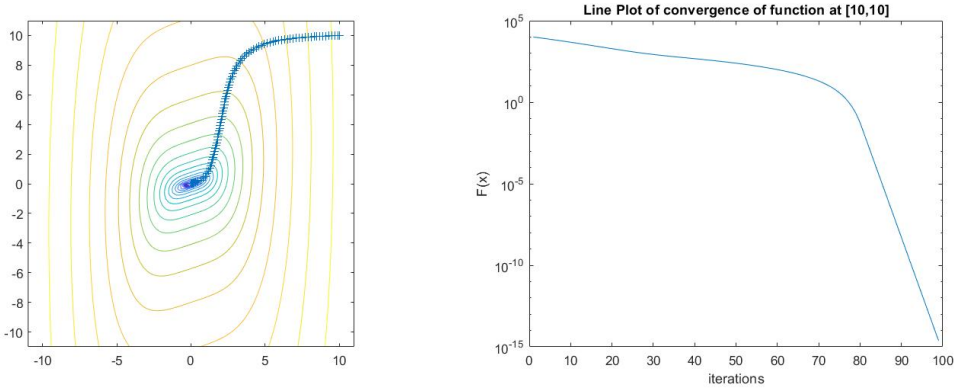

Figure 2: Contour plot SR1 (left), convergence plot SR1 (Right)

As can be seen from the convergence rates plotted on Fig. 1(a)(right), the convergence rate gotten for BFGS for this problem is linear. This conflicts with the theory which says that BFGS has a superlinear convergence.

As can be seen from the convergence rates plotted on Fig. 1(b)(right), the convergence rate gotten for SR1 for this problem is superlinear. Which agrees with the theory.

Question 3b

Fig. 3(a) and Fig. 3(b) show the error in the approximations for the the BFGS and SR1 Quasi newton method.
For BFGS, $\left\|I - H_k^{BFGS}\nabla^2 f(x_k)\right\|_2$ is calculate as H_k is an approximation of the inverse hessian, and an inverse multiplied by the original should give identity matrix. The same also goes for SR1 method where $\left\|I - \nabla^2 f(x_k)B_k^{SR}\right\|_2$ is calculated

instead as H_k^{SR} is an approximation of the Hessian directly.
It can be seen that the error in the hessian approximations for both methods goes down after some iterations

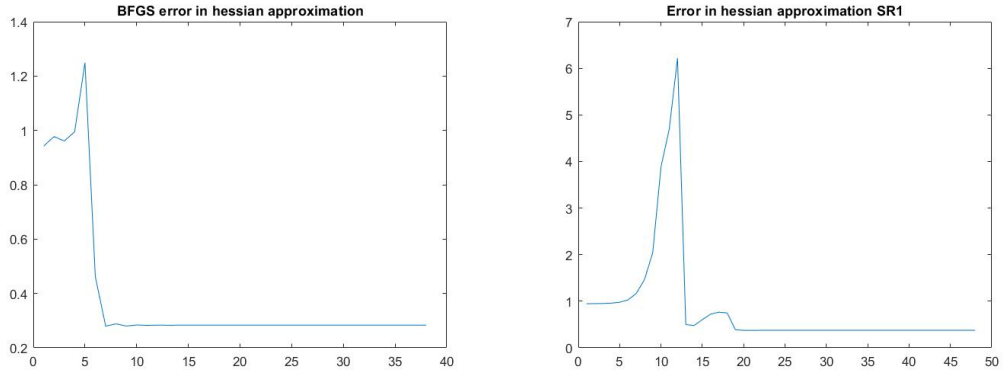

Figure 3: error in hessian approximation BFGS (left), error in hessian approximation for SR1 (Right)

Question 6

Formulate the least square problem for fitting Ψ

$$\Psi(x_1, x_2, x_3; t) = (x_1 + x_2 t^2) exp(-x_3)$$

the real numbers $x_i, i = 1, 2, 3$ are the parameters of the model and they are to be chosen to make the model $\Psi(x; t)$ fit the simulated observed data y_j. The amount of data points simulated is 200. Hence we can define the residuals

$$r_j(x) = \Psi(x; t) - y_j, j = 1, 2, ...200$$

we find the set of parameters of x that minimizes the equation

$$f(x) = \frac{1}{2} \sum_{j=1}^{m=200} \|\Psi r_j(x)\|^2$$

The jacobian of the function f is given by

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right]_{j=1,2...200, i=1,2,3} = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_{200}(x)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial r_1}{x_1} & \frac{\partial r_1}{x_2} & \frac{\partial r_1}{x_3} \\ \frac{\partial r_2}{x_1} & \frac{\partial r_2}{x_2} & \frac{\partial r_2}{x_3} \\ \vdots & \vdots & \vdots \\ \frac{\partial r_{200}}{x_1} & \frac{\partial r_{200}}{x_2} & \frac{\partial r_{200}}{x_3} \end{bmatrix}$$

where $\nabla r_j(x)^T$ is the gradient of r_j as shown above
since $r_j(x) = \Psi(x; t) - y_j, j = 1, 2, ...200$, and we know that y_j is a data point which is a scalar, hence can write a general formula for $\nabla r_(x)$ as

$$\nabla r_(x) = \nabla(\Psi(x; t) - constant) = \begin{bmatrix} exp^{-x_3 * t} & t^2 * exp^{(-x_3 * t)} & -t * x_1 * exp^{-x_3 * t} - t^3 * x_2 * exp^{-x_3 * t} \end{bmatrix}$$

Hence the jacobian J(x) can be written as

$$J(x) = \begin{bmatrix} exp^{-x_3 * t_1} & t_1^2 * exp^{(-x_3 * t_1)} & -t_1 * x_1 * exp^{-x_3 * t_1} - t_1^3 * x_2 * exp^{-x_3 * t_1} \\ exp^{-x_3 * t_2} & t_2^2 * exp^{(-x_3 * t_2)} & -t_2 * x_1 * exp^{-x_3 * t_2} - t_2^3 * x_2 * exp^{-x_3 * t_2} \\ \vdots & \vdots & \vdots \\ exp^{-x_3 * t_{200}} & t_{200}^2 * exp^{(-x_3 * t_{200})} & -t_{200} * x_1 * exp^{-x_3 * t_{200}} - t_{200}^3 * x_2 * exp^{-x_3 * t_{200}} \end{bmatrix}$$

6b

Since the problem is in least square form, f(x) can be minimized using non-linear least square methods namely the Levenberg-Marquardt method and the Gauss-Newton method.
I initialized x0 to an arbitary value, x0 = [1;1;1], which I then update continuously.
The parameter which I chose for the GN method were: max iterations = 500
stopping tolerance = 1e-10
c1 for strong wolfe condition = 1e-4
c2 for strong wolfe condition = 0.2
For the LM method, I chose parameters as: max iterations = 500
stopping tolerance = 1e-10
initial trust region radius = 2.0,
Step acceptance relative progress threshold (η) = 0.9,
overall bound on step lengths ($\hat{\Delta}$) = 2 From the Fig. 4(a) it can be seen that for this problem, Gauss-Newton algorithm

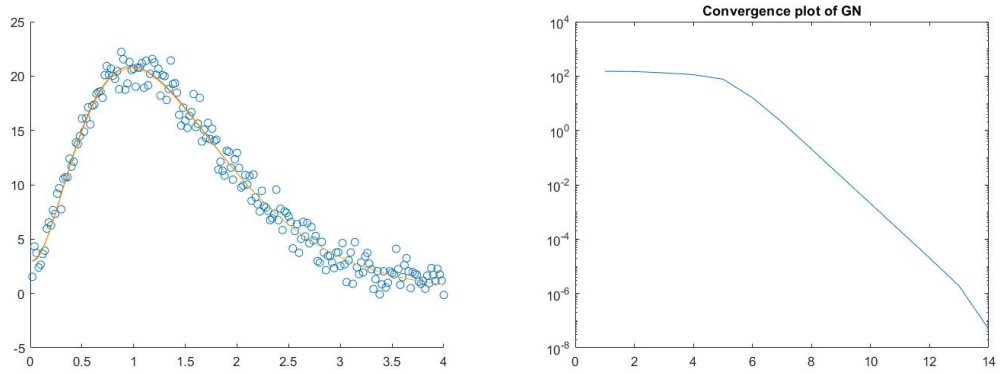

Figure 4: plot of Gauss-Newton estimated model(left), convergence plot(right)

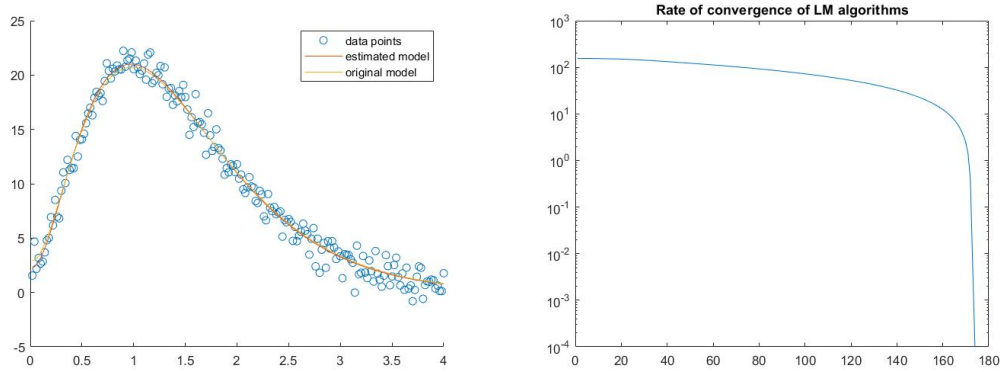

Figure 5: plot of Levenberg-Marquardt estimated model(left), convergence plot(right)

converges linearly.
. The factor $\left\|\Psi J^T J(x^*)^{-1} H(x^*)\right\|$ when evaluated was 0.0051. This is which explains the rapid convergence of the gauss newton algorithm as it converged in about 15 steps. From the Fig. 5(a) it can be seen that for this problem, Levenberg-Marquardt algorithm converges super-linearly. The factor $\left\|\Psi J^T J(x^*)^{-1} H(x^*)\right\|$ when evaluated was 0.0015 which explains the graph super-linear nature.